Einführung in Suchverfahren

Alfred Kranstedt

30.10.03

Seminar „Intelligente Algorithmen“

Was ist heute Thema?

1. Was ist ein Suchproblem?
   - Definitionen, Darstellungen etc.
2. Suchstrategien
   - Blinde Suche
   - Heuristische Suche
3. Lokale Suche und Metaheuristiken

Problemlösen durch Suche

Suche nach einer Folge von Operatoren, deren schrittweise Anwendung auf die Problemrepräsentation einen Ausdruck erzeugt, der eine Problemlösung darstellt.

Auch Entscheidungs- und Optimierungsaufgaben sind Suchprobleme!

Definition

Ein Suchproblem ist definiert durch

- Startzustand
- Beschreibung eines Zielzustandes
- Menge von Operatoren, die einen Zustand in einen anderen überführen

Eine Lösung des Problems ist gegeben entweder durch

- einen konkreten Zustand, welcher der Zielbeschreibung genügt
- oder eine Folge von Operatoren, die den Startzustand in einen Zielzustand überführen (Pfad)

Weitere Definitionen:

successor (Nachfolger): Zustand S`, der von S durch Operatoranwendungen erreichbar ist

search space (Suchraum): Alle durch Operatoranwendung vom Startzustand aus erreichbaren Zustände.

Mehrere Herangehensweisen:

Forward Reasoning: Vom Startzustand aus werden sukzessive neue Zustände bis zu einem Zielzustand generiert.

Backward Reasoning: Von einem bekannten Ziel aus werden rückwärts Vorgängerzustände bis zum Startzustand generiert.

Bidirektionale Suche: Die Suche wird vom Startzustand und einem Ziel aus begonnen in dem Bestreben, sich „in der Mitte“ zu treffen.

Im folgenden nur noch Forward Reasoning
(alles andere siehe KI-Vorlesung)

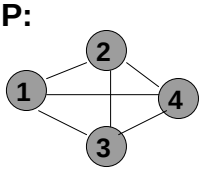
Graph-/Baumdarstellung eines Suchproblems

Knoten sind Zustände des Suchraumes

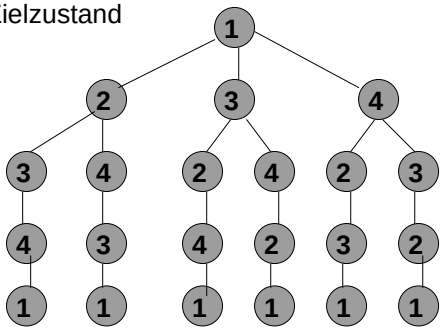
Kanten mögliche Operatoranwendungen

Lösung ist dann ein als solcher gekennzeichneter Zielzustand oder ein Pfad vom Start- zum Zielzustand

TSP:



=>



Allgemeiner Suchalgorithmus

```
function general-search(problem, queuing-fn)          returns a solution, or failure
queue <- make-queue(initial-state[problem])
loop
        if queue is empty then return failure
        else node <- remove-front(queue)
        if goal-test[problem] applied to node suceeds then return node
        else queue <- queuing-fn(queue, expand(node, operators[problem]))
endloop
```

Die Suchstrategie verbirgt sich in der Funktion `queuing-fn`, die in jedem Durchlauf die neu explorierten Zustände in die Queue einsortiert und damit auch festlegt, welcher Zustand als nächstes expandiert wird.

Suchstrategien

Kriterien zur Bewertung von Suchstrategien:

Completeness: Garantiert die Strategie das Finden einer Lösung soweit es eine gibt.

Optimality: Garantiert die Strategie das Finden der „besten“ Lösung wenn es mehrere gibt.

Time complexity: Benötigte Zeit zum Finden einer Lösung.

Space complexity: Benötigter Speicherplatz zum Finden einer Lösung.

Suchstrategien

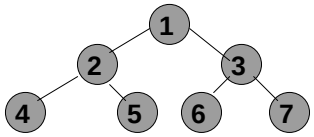
Unterschieden wird grob zwischen

Blinder (uninformierter) Suche: Brute-Force Methoden, die nach einer bestimmten Systematik den gesammten Suchraum evaluieren, bis sie eine Lösung finden.

Heuristischer (informierter) Suche: Verwendet Wissen über den Suchraum um die Suche gezielt zu lenken und so unnötige Evaluationen zu vermeiden.

Blinde Suche: Breitensuche

Ebenenweise Abarbeitung des Suchbaumes



vollständig falls alle Knoten nur endlich viele Nachfolger haben

optimal falls die Tiefe proportional der „Güte“ einer Lösung ist

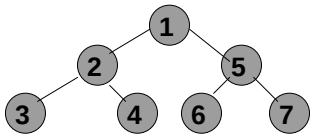
Zeitkomplexität $O(b^d)$, wobei b der maximale *branching factor* und d die Tiefe der gefundenen Lösung im Baum.

Speicherkomplexität ebenfalls **$O(b^d)$**, da alle Knoten der aktuellen Ebene im Speicher gehalten werden müssen.

Organisierung der Queue: first-in/first-out

Blinde Suche: Tiefensuche

Astweises Explorieren des Suchbaumes



Nur vollständig, wenn es keine unendlich tiefen Teilbäume gibt (oft nicht auszuschließen)

Nicht optimal, die erste gefundene Lösung wird ausgegeben

Zeitkomplexität $O(b^m)$ für den worst case (aber oft besser, wobei m die maximale Tiefe ist.

Speicherkomplexität O(bm), nur die Knoten auf dem Pfad und alle direkten Nachfolger

Organisierung der Queue: last-in/first-out

Wir haben also ein vollständiges und optimales aber uneffizientes Verfahren:

Breitensuche

Und wir haben ein effizienteres Verfahren welches aber nicht optimal und oft auch nicht vollständig ist:

Tiefensuche

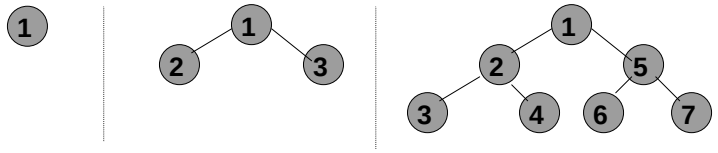
=>

Blinde Suche:

depth-limited search / iterative Deepening

depth-limited depth search verhindert das Hängenbleiben in unendlichen Teilbäumen durch die Einführung einer sinnvollen Tiefenbegrenzung. Problem: Was ist eine sinnvolle Tiefenbegrenzung?!

Iterative Deepening erhöht die Tiefenbegrenzung bei jedem Durchlauf um eins und führt eine komplette Tiefensuche durch:



ID ist vollständig und optimal mit einer Zeitkomplexität von $O(b^d)$ und einer Speicherkomplexität von O(bd) !! <- Verfahren der Wahl !!

Bsp.: für b=2, d=3: $(d+1)1+db+(d-1)b^2+...+1b^d=26$
<= 2* Kosten Breitensuche =15 (für b=10 nur 11% höhere Kosten)

Heuristische Suche

Verwendung von Wissen über den Suchraum, welches Hinweise gibt, welche Operatoranwendung am vielversprechensten ist.

Das Wissen kann organisiert werden in:

Operatorordnungsfunktion: sortiert die Operatoren danach, um wieviel sie voraussichtlich die Entfernung zum nächsten Ziel verkürzen.

Zustandsbewertungsfunktion: Bewertet die Zustände, z.B. nach ihrer vermuteten Entfernung zum nächsten Ziel (im folgenden f(n)).

Wichtig: Es handelt sich hierbei um **Schätzfunktionen**! Wüßten wir, welcher Zustand tatsächlich jeweils der beste ist, so würde sich die Suche erübrigen.

Bestensuche: Sortieren der Queue im search-alg. nach der Bewertungsfkt. (`queuing-fn` <- order *queue* by `f(n)`)

uniform cost search

Ähnlich wie Breitensuche, allerdings wird hier die Queue sortiert nach den Kosten, die zum Erreichen des jeweiligen Zustandes aufgewandt werden mußten.

Evaluationsfkt.: f(n)=g(n), wobei g(n) die benötigten Kosten zum Erreichen des Zustandes n angibt

Für g(n) = depth(n)

=> uniform cost search = Breitensuche

Vollständigkeit, Optimalität und Effizienz wie bei Breitensuche

Greedy search

Gierige Suche: Es wird immer derjenige Zustand ausgewählt, dessen vermuteter Abstand zum nächsten Ziel minimal ist.

Evaluationsfkt.: f(n)=h(n), wobei h(n) Entfernung zum Ziel schätzt.

Gesucht der kürzeste Weg von S nach Z

Vollständig (eingeschränkt), aber nicht optimal, Zeit- und Speicherkomplexität $O(b^m)$ im worst case, wobei eine gute Schätzfunktion dies aber erheblich reduzieren kann.

A*-Algorithmus

Minimierung der totalen Pfadkosten, d.h. Kombination von uniform cost search und greedy search.

Zweigeteilte Evaluationsfkt.: f(n)=g(n)+h(n)

wobei g(n) die Pfadkosten bis zum Knoten n und h(n) die geschätzten Kosten zum Ziel sind.

A* ist vollständig, optimal, wenn gilt h(n)<=h`(n), wobei h`(n) die tatsächlichen Pfadkosten zum Ziel sind (Unterschätzen der Wegkosten).

Die Effizienz ist abhängig von der Güte von h wobei gilt: h1 führt zu effizienteren Ergebnissen als h2 wenn h1(n)>h2(n) für alle n, also die Kosten knapper unterschätzt.

Lokale Suche

Der **Suchraum** entspricht hier dem **Lösungsraum** (klassisch für Optimierungsaufgaben). Auch für kontinuierliche Lösungsräume geeignet.

Geeignet für Probleme mit vielen bekannten Lösungen, in denen die optimale gefunden werden muß. TSP klassisches Bsp.: Lösungsraum: Alle Permutationen der Städteliste.

Prinzip:

1. Wähle beliebige Lsg. aus dem Lösungsraum und ermittle ihre Güte
2. Verändere diese Lsg. lokal (in einer zu definierenden Nachbarschaft) und stelle fest, ob das zu einer Verbesserung führt
3. Wenn ja wiederhole das so lange bis keine Verbesserung mehr möglich ist

Nachbarschaftsdefinition

Nachbarschaften entstehen durch vorab definierte „Austausche“ oder „Verschiebungen“, die eine zulässige Lösung $x \in X$ in $x` = m(x) \in X$, $m(x) \in M(x)$ überführen.

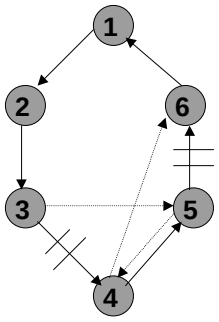
Dann kann man eine Nachbarschaft N definieren als Abbildung $N: X \rightarrow P(X)$ mit $N(x) = \{x` = m(x) : m(x) \in M(x)\}$.

Eine Nachbarschaft N heißt symmetrisch, wenn gilt: $m(m(x))=x$.

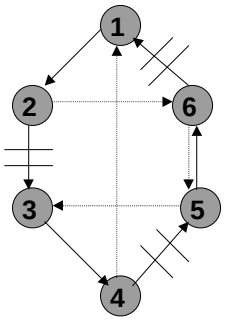
Ist X endlich und N(x) symmetrisch, so kann N durch einen (gerichteten) Graphen dargestellt werden.

Beispiele für Nachbarschaften beim TSP

2-opt-Tausch



3-opt-Tausch



Suchheuristiken

Grundlegende Suchstrategien:

First-Accept: Bestimme den ersten verbesserten Nachbarn
$x` \in N(x)$, sofern einer existiert.

Best-Accept: Bestimme stets den besten Nachbarn
$x` \in \arg\min \{f(y) : y \in N(x)\}$

Hill-Climbing / Gradientenabstieg: Implementierung der Best-Accept Strategie. Hill-Climbing versucht den Gewinn zu maximieren, wählt also de Weg des stärksten Anstiegs, Gradientenabstieg versucht die Kosten zu minimieren, wählt den Weg des stärksten Abstiegs.

Problem: Hängenbleiben in lokalen Extrema und zielloses Herumirren auf Plateaus

Metaheuristiken

Eine **Metaheuristik** ist ein allgemein anwendbares Verfahren, um zugrundeliegende, problemspezifische Heuristiken (z.B. ein lokales Suchverfahren) in erfolgversprechende Regionen des Suchraumes zu leiten.

Einordnung der im Folgenden vorgestellten Verfahren I

Beispiele für Metaheuristiken auf lokaler Suche:

Tabu Search versucht unnötige Wege bei der Optimierung zu vermeiden

Randomisierung (Simulated Annealing, Monte Carlo) soll ein Hängenbleiben in lokalen Minima verhindern

Evolutionäre Ansätze (und auch „Ants“) parallelisieren die Suche, beginnend an unterschiedlichen Stellen im Suchraum, konkurrieren die lokalen Suchstränge

Memetische Alg. verbinden mehrere Metaheuristiken

Einordnung der im Folgenden vorgestellten Verfahren II

Branch-and-Bound Methoden sind verwandt mit heuristischen Verfahren der **Zustandsraumsuche** wie dem **A*-Algorithmus**

Ebenso gibt es Verbindungen zum **dynamischen Programmieren**. Kumar und Kanal (1988) postulieren eine „grand unifikation“ von Heuristischer Suche, Branch-and-Bound und Dynamischer Programmierung zur **CDP** (composite decision process)

Lineares Programmieren wiederum steht in Verbindung mit der hier unterschlagenen **constraint satisfaction search**:
Die Zustände im Suchraum werden durch einen Satz von Variablen repräsentiert, deren Belegungen verschiedenen Constraints genügen müssen, um eine Lösung darzustellen. Haben diese Constraints die Form linearer Ungleichungen sind wir bei der linearen Programmierung.