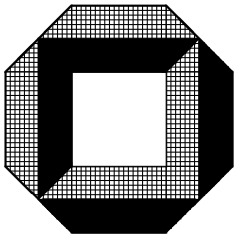
Proseminar

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

Humorale Immunantwort

Clemens Lode

Inhaltsverzeichnis

Kapitel 1

Humorale Immunantwort

1.1 Einleitung

Blabla

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*)

1.3 Zusammenfassung

1.3.1 Allgemeine Suchalgorithmen

Im ersten Teil wurden uns 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.

1.3.2 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 Ausfuhrung, 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, dass 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, egab 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] Russel S., Norvig P. 2003: Seite 74 ,,Time and memory requirements"

[3] Russel S., Norvig P. 2003: Seite 99 ,,triangle inequality"

[4] Russel S., Norvig P. 2003: Seite 101 ,,A* is optimally efficient"

[5] Russel S., Norvig P. 2003: Seite 104 ,,SMA* is optimally efficient"

[6] Russel S., Norvig P. 2003: Seite 105 ,,8-puzzle"

[7] Russel S., Norvig P. 2003: Seite 107 ,,Comparison of the search costs"

[8] Russel S., Norvig P. 2003: Seite 108 ,,Manhattan-distance"

[9] Russel S., Norvig P. 2003: Seite 108 ,,ABSOLVER"

[10] Russel S., Norvig P. 2003: Seite 150 ,,Min-conflicts"

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

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

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

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

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