Überblick Künstliche Intelligenz

Thaís Moreira Hamasaki

20. Mai 2016

Inhaltsverzeichnis

1 Motivation

Die Hauptaufgabe von Rechnern ist, Probleme zu lösen. Oft kann man die Probleme graphisch darstellen, mit einem Start- und einem Zielknoten und die Wege zwichen den beiden. Hierfür benötigt man nicht nur eine Darstellung des aktuellen Zustandes sondern auch die Möglichkeiten, die zur Verfügung stehen, um das Ziel zu erreichen. Meistens gibt es keine einfache Lösung, denn nicht alle Probleme sind Addition von zwei positiven ganzen Zahlen. Daher benötigt man Suchverfahren. Hier beschänken wir uns auf die Depth-First Search, Breadth-First Search, Hill Climbing und Best-First Suchverfahren.

2 Suchalgorithmen

Wenn man vom Problem und der Darstellung abstrahiert, kann man die Suche nach einer Lösung in einem gerichteten Graph betrachten. Der gerichtete Graph wird nicht gegeben sondern wird durch Erzeugungsregeln abgeleitet. Der Graph kann auch unendlich groß sein und daher ist die Suche und partielle Erzeugung des Graphen somit verflochten.

Unser Suchgraph besteht hier aus Knoten, die die Zustände beschreiben, und Kanten, in Form von Funktionen, die die Nachfolgerknoten berechnet. Außerdem definieren wir eine Anfangssituation und ein Ziel.

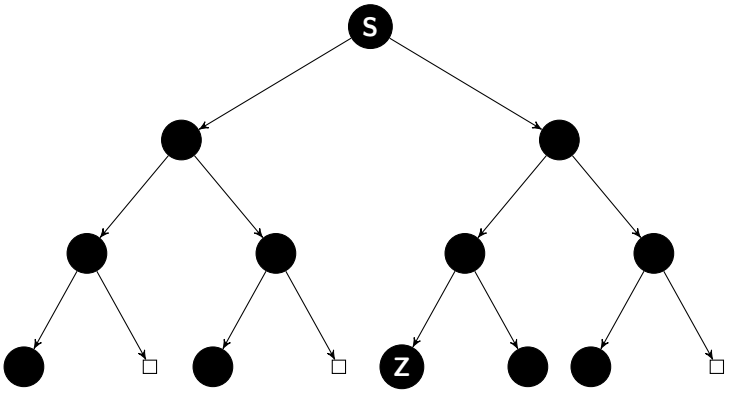

Fig1: Graphische Darstellung eines Problems S:
Startknoten, Z: Zielknoten

3 Uninformierte Suche

Wir betrachten zunächst die nicht informierte Suche. Hierbei wird bei der Suche nur den Graphen und die Nachfolgerfunktion berücksichtigt, aber keine anderen

Informationen über die Knoten, Kanten usw. dürfen verwendet werden.

Die Paramenter für unseren Graph sind

- Menge der initialen Knoten
- Menge der Zielknoten, bzw eindeutige Festlegung der Eigenschaft der Zielknoten
- Nachfolgerfunktion N

damit lässt sich folgender Algorithmus schreiben:

Algorithm 1 General-Search Algorithm

```
1:  function General-Search(problem,strategy)
2:      returns a solution, or failure
3:      initialize the search tree using the initial state of problem
4:      loop
5:          choose a leaf node for expansion according to strategy
6:          if there are no candidates for expansion then failure
7:          end if
8:          if the node contains a goal state then the corresponding solution
9:          else expand the node and add the resulting nodes to the search tree
10:         end if
11:     end loop
12: end function
```

Wir betrachten im folgenden Varianten der blinden Suche, die insbesondere das Wählen des Knotens eindeutig durchführen.

3.1 Depth-First Search

Die Depth-First Suche verwendet eine Liste von Knoten und wählt als nächsten zu betrachtenden Knoten dieser Liste. Außerdem werden neue Nachfolger vorne in die Liste eingefügt, was zur Charakteristik führt, dass zunächst in der Tiefe gesucht wird.

Algorithm 2 Depth-First Search Algorithm

```
1: function Depth-First-Search(problem)
2:     returns a solution, or failure
3:     function General-Search(problem,Enqueue-At-Front)
4:     end function
5: end function
```

Die Depth-First Suche eignet sich damit besonders für Suchbäume, deren Äste eine vertretbare Länge nicht überschreiten.

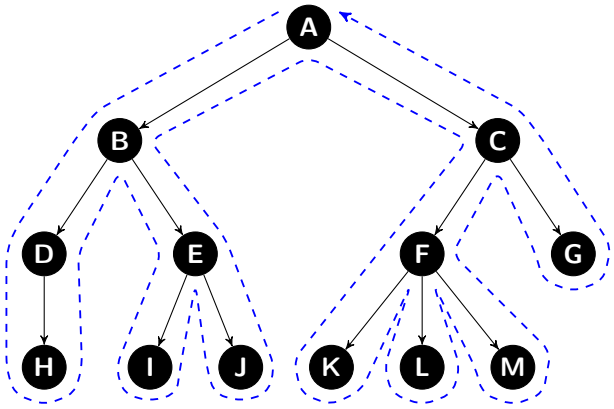

Fig2: Graphische Darstellung der Depth-First Suche

Eine mögliche Implementierung der Depth-First Suche in Prolog ist:

```
1  loesung(Start,Ziel):-
2    weg(Start,Ziel,[],Pfad),
3    write('Pfad:␣'),   % Aufruf der Tiefensuche
4    write(Pfad).
5
6  % weg(Startknoten, Zielknoten, Liste der
7  %          besuchten Knoten, Ergebnispfad)
8
9  weg(Knoten,Knoten,Liste,Pfad):-
10   reverse(Liste,Pfad).
11
12 weg(Start,Ziel,Liste,Pfad):-
13   kante(Start,Knoten),
14   not(member(Knoten,Liste)),
15   weg(Knoten,Ziel,[Knoten|Liste],Pfad).
```

Und in Haskell:

```
1  dfs ::(a -> Bool) -- Zieltest (goal)
2    -> (a -> [a]) -- Nachfolgerfunktion (succ)
3    -> [a] -- Startknoten
4    -> Maybe (a, [a]) -- Ergebnis: Just (Zielknoten,Pfad)
5                       -- oder Nothing
6    dfs goal succ stack =
7      -- Einfuegen der Anfangspfade, dann mit iterieren mit go
8    go [(k,[k]) | k <- stack]
9  where
10   go [] = Nothing -- Alles abgesucht, nichts gefunden
11   go ((k,p):r)
12     | goal k = Just (k,p) -- Ziel gefunden
13     | otherwise = go ([( k  , k  :p) |  k   <- succ k] ++ r)
```

Die explizite Speicherung der Pfade sowohl in Haskell als auch in Prolog kostet nicht viel Effizienz. Man kann den Algorithmus auch in imperativen Sprachen implementieren, indem jeder Eintrag einen Knoten mit Zeiger auf den nächsten Nachbarknoten zugeordnet wird. Die Komplexität des Verfahrens (worst-case) für die Zeit ist entsprechend der Anzahl der besuchten Knoten exponentiell in der Tiefe und für den Speicherplatz linear bei fester Verzweigungsrate.

Das Problem bei der Depth-First Suche ist, dass der Algorithmus nicht vollständig ist. Wenn der Suchgraph unendlich groß ist, kann die Suche am Ziel vorbeilaufen und für immer im unendlichen langen Pfad laufen.

Um dies zu verbeugen kann man eine Tiefenbeschränkung (k) einbauen. Wenn die vorgegebene Tiefenschrank (k) überschriten wird, werden keine Nachfolger dieser Knoten mehr erzeugt. Die Depth-First Suche findet in diesem Fall jeden Zielknoten, der höchstens Tiefe k hat.

Wenn der gerichtete Graph kein Baum ist, kann man auch die schon untersuchten Knoten in einer Hash-Tabelle speichern und somit werden die Knoten nicht nochmal untersucht sondern redundant weiterverfolgt.

Hierfür ist der Platzbedarf entsprechend der Anzahl der besuchten Knoten und die Berechnungszeit gleich $n*log(n)$ mit n = Anzahl der untersuchten Knoten.

(a) Backtracking

Die Arbeitsweise der Depth-First Suche ist: Wenn Knoten K keine Nachfolger hat (oder eine Tiefenbeschränkung) überschritten wurde, dann mache weiter mit dem nächsten Bruder von K. Ist der Pfad nicht erfolgreich, führt die Depth-First

Suche Backtracking durch. Dies wird als chronologisches Zurücksetzen bezeichnet.

Es gibt auch Suchprobleme, bei denen kein Backtracking erforderlich ist (greedy Verfahren ist möglich) zum Beispiel, wenn jeder Knoten noch einen Zielknoten als Nachfolger hat. Die Depth-First Suche reicht dann aus, wenn die Tiefe der Zielknoten in alle Richtungen unterhalb jedes Knotens beschränkt ist. Anderenfalls reicht Depth-First Suche nicht aus, da ein Ast immer ausgesucht werden kann, indem der Zielknoten weiter weg ist.

3.2 Breadth-First Search

Die Breadth-First Suche verwendet auch eine Liste von Knoten wie bei der Depth-First Suche. Der Unterschied hierbei ist, dass die neuen Nachfolger hinten in die Liste eingefügt werden, was zur Charakteristik führt, dass zunächst in der Breite gesucht wird.

Algorithm 3 Breadth-First Search Algorithm

```
1: function BREADTH-FIRST-SEARCH(problem)
2:     returns a solution, or failure
3:     function GENERAL-SEARCH(problem,Enqueue-At-End)
4:     end function
5: end function
```

Die Breadth-First Suche untersucht somit ausgehend von Startknoten alle Nachbarknoten. Ist der Zielknoten noch nicht erreicht, werden alle Nachbarknoten der bisher untersuchten Knoten betrachtet.

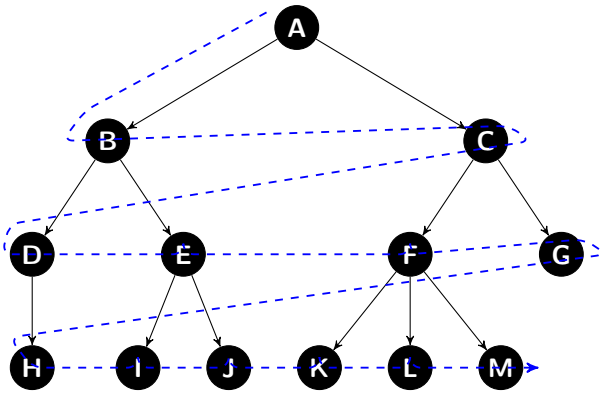

Fig3: Graphische Darstellung der Breadth-First Suche

Eine mögliche Implementierung der Breadth-First Suche in Prolog ist:

```
loesung(Start,Ziel):-
         weg([Start],Ziel,[],Pfad),% Aufruf der Breitensuche,
         %Besonderheit: Startknoten wird als Liste   bergeben
         write('Pfad:␣'),
         write(Pfad).

% weg(Startknotenliste, Zielknoten, Liste der besuchten
                     %Knoten, Ergebnispfad)

weg(Startliste,Ziel,_,Pfad):-
         Startliste=[Ziel|_],
         reverse(Startliste,Pfad).

weg(Startliste,Ziel,Liste,Pfad):-
         Startliste=[Start|_],
         findall([Knoten|Startliste],
            (kante(Start,Knoten),
             not(member(Knoten,Startliste))),
            Knotenliste),
         append(Liste,Knotenliste,Listeneu),
         Listeneu=[PfadN|RestPfad],
         weg(PfadN,Ziel,RestPfad,Pfad).
```

Und in Haskell:

```
bfs goal succ start =
go [(k,[k]) | k <- start] -- Pfade erzeugen
where
go [] = Nothing -- nichts gefunden
go rs =
case filter (goal . fst) rs of -- ein Zielknoten enthalten?
-- Nein, mache weiter mit allen Nachfolgern
[] -> go [( k  , k  :p) | (k,p) <- rs,  k   <- succ k]
-- Ja, dann stoppe:
(r:rs) -> Just r
```

Die Vor- und Nachteile der Breadth-First Suche lässt sich wie folgt zusammenfassen:

- Die Breadth-First Suche ist vollständig. Das bedeutet, wenn es eine Lösung gibt, wird sie mit Sicherheit gefunden.
- Der kürzeste Weg wird gefunden, dafür wird aber viel Speicherplatz benötigt.

- Es werden Knoten besucht, die für den optimalen Pfad nicht besucht werden müssten. Damit wird auch der Zeitaufwand erhöht. Dies beträgt nämlich $n + n * log(n)$, für n = Anzahl der Knoten.

3.3 Reverse Search

Bei der Rückwärtssuche wird von einem (bekannten) Zielknoten ausgegangen in die Richtung des Anfangsknoten.

Voraussetzungen für die Rückwärtssuche sind:

- Der Zielknoten kann explizit angeben werden(nicht nur eine Eigenschaft, die der Zielknoten erfüllen muss).
- Von jedem Knoten ist es möglich, die direkten Vorgänger zu berechnen.

Reverse Suche ist aber nur besser als Vorwärtssuche, wenn die Verzweigungsrate in Rückwärtsrichtung kleiner ist als die Verzweigungsrate in Vorwärtsrichtung. Das Problem an der Reverse Suche ist oft, dass die Vorgängerfunktion schwer zu finden ist.

4 Informierte Suche

Die Algorithmen, die bis jetzt betrachtet wurden, erzeugen neuen Zuständen und vergleichen sie immer wieder mit dem Zielzustand, bis sie übereinstimmen. Diese Strategie funktioniert, ist aber oft sehr kostenaufwendig. Wir betrachten im folgenden heuristischen Algorithmen.

Die Suche wird als „$heuristische$„ oder „$informierte$„ bezeichnet, wenn man zusätzlich zu den Knoten und zu der Datenstruktur eine Schätzfunktion gibt, die als Schätzung des Abstands zum Zielknoten interpretieren werden kann. Der Zielknoten sollte ein Maximum bzw. Minimum der Schätzfunktion sein.

Eine Heuristik ist dann eine Methode, die oft ihren Zweck erreicht, aber nicht mit Sicherheit. Man spricht von heuristischer Suche, wenn die Schätzfunktion in vielen praktisch brauchbaren Fällen die richtige Richtung zu einem Ziel angibt, aber möglicherweise manchmal versagt.

Wir suchen dann das Maximum (oder Mininum) einer Knotenfunktion auf einem gerichteten Graphen.

4.1 Hill Climbing

Der Hill Climbing Suchalgorithm ist auch als Gradientenaufstieg bekannt, da die Suche immer in Richtung der Vergrößerung einer Funktion läuft.

Parameter, die notwendig für den Algorithmus sind:

- die Menge der initialen Knoten,
- die Nachfolgerfunktion,
- die Schätzfunktion,
- und der Zieltest.

Der Algorithmus verhält sich wie die Depth-First Suche, wobei der Nachfolger, der zu expandieren ist, nach der Schätzfunktion ausgesucht wird.

Algorithm 4 Hill Climbing Algorithm

```
 1: function HILL-CLIMBING(problem)
 2:     returns a solution state
Input:
    problem                                              ▷ Ein Problem
    current                                              ▷ Ein Knoten
    next                                                 ▷ Ein Knoten
 3:     current ← Make − Node(Initial − State[problem])
 4:     loop
 5:         next ← größter Wert aus current
 6:         if VALUE[next] < VALUE[current] then return current
 7:         end if
 8:         current ← next
 9:     end loop
10: end function
```

Das Problem bei dem Hill Climbing Algorithmus ist, dass er bei lokalen Maxima abbricht. Es ist möglich, in dem Hill-Clinbing Algorithmus Zyklen einzubauen, damit Backtracking durchgeführt wird, aber er bleibt eine zeitlang in den Maxima hängen.

Wenn mehrere Nachfolger die gleiche Schätzwerte haben, ist der Weg, den ausgesucht werden soll, nicht eindeutig.

Eine mögliche Implementierung der Hill Climbing Suche in Haskell ist:

```
1  hillclimbing cmp heuristic goal successor start =
2  let -- sortiere die Startknoten
3  list = map (\k -> (k,[k])) (sortByHeuristic start)
4  in go list []
5  where
6  go ((k,path):r) mem
7  | goal k
8  = Just (k,path) -- Zielknoten erreicht
9  | otherwise =
10 let -- Berechne die Nachfolger (nur neue Knoten)
11 nf = (successor k) \\ mem
12 -- Sortiere die Nachfolger entsprechend der Heuristik
13  l   = map (\k -> (k,k:path)) (sortByHeuristic nf)
14 in go ( l   ++ r) (k:mem)
15 sortByHeuristic = sortBy (\a b -> cmp (heuristic a)
16         (heuristic b))
17
18 --cmp ist die compare-Funktion und ermoeglicht zu
19                  --maximieren oder zu minimieren
```

4.2 Best-First-Search

Die Best-First-Suche wählt immer den Knoten aus, der den besten Wert bzgl. der Schätzfunktion hat. Der Knoten wird dann überprüft und, wenn der nicht das Ziel ist, expandiert.

Algorithm 5 Best-First-Search Algorithm

1: **function** BEST-FIRST-SEARCH(problem,EVAL-FN)
2: **returns** a solution sequence

Input:

$problem$ ▷ Ein Problem
$Eval - Fn$ ▷ Eine Evaluationsfunktion

3: $Queueing - Fn \leftarrow$ a function that orders nodes by EVAL-FN
4: **return** GENERAL-SEARCH(problem, Queueing-Fn)
5: **end function**

Eine Implementierung der Best-First Suche in Haskell ist:

```
1  bestFirstSearchMitSharing
2  cmp heuristic goal successor start =
3  let -- sortiere die Startknoten
4  list = sortByHeuristic (map (\k -> (k,[k])) (start))
5  in go list []
6  where
7  go ((k,path):r) mem
8  | goal k    = Just (k,path) -- Zielknoten erreicht
9  | otherwise =
10 let -- Berechne die Nachfolger und nehme nur neue Knoten
11 nf = (successor k) \\ mem
12 -- aktualisiere Pfade
13 l' = map (\k -> (k,k:path)) nf
14 -- Sortiere alle Knoten nach der Heuristik
15 l'' = sortByHeuristic (l' ++ r)
16 in go l'' (k:mem)
17 sortByHeuristic =
18 sortBy (\(a,_) (b,_)-> cmp (heuristic a) (heuristic b))
```

Die Best-First-Suche ist mit der Depth-First Suche eng verbunden. Der Unterschied ist, dass bei der Best-First Suche den nächsten Knoten, der überprüft und expandiert wird, mithilfe der Schätzfunktion ausgesucht wird. Dabei werden alle Knoten im Stack bewertet und dadurch können lokale Maxima schnell verlassen werden.

Wie bei der Depth-First Suche ist die Best-First Suche hier nicht vollständig und die Platzkosten steigern exponentiell in der Tiefe.

5 Fazit & Bewertung

Zunächst wurden das Suchproblem und der Begriff „*Suche*„ erläutert. Die nicht informierte Suchverfahren wurden präsentiert und es wurde gezeigt, dass ein General Search Algorithmus alle Probleme löst, wenn die richtige Strategie ausgesucht wird.

Tiefensuche bedeutet, dass man beginnend im Startknoten so weit wie möglich entlang der bestehenden Kanten in die Tiefe geht, ehe man zurückläuft und dann in bislang unbesuchte Teilbäume absteigt.

Breitensuche bedeutet, dass man beginnend im Startknoten alle direkt verbundenen Knoten besucht, bevor die nächst tiefere Ebene überprüft wird.

Die Vollständigkeit und die Komplexität der gängigen uninformierten Suchalgorithmen wurden erläutert.

Dabei ist klar geworden, dass die Zeit- und Speicherplatzkosten hoch sind.

Um die Kosten zu minimieren werden die bekannte Suchverfahren weiterentwickelt. Daraus folgte zum Beispiel die Best-First Suche. Diese ist eine General Search, wobei nur die minimale Anzahl der Knoten berücksichtigt wird.
Solche Algorithmen, wobei die Kosten minimiert werden, nennen wir **greedy Algorithmen**. Der Algorithmus ist aber immer noch nicht optimal.

6 Quellen und Literatur

Literatur

[1] Stuart J. Russel and Peter Norvig, "Artificial Intelligence - A Modern Approach", Pretince Hall, Englewood Cliffs, New Jersey 2010

[2] Ivan Bratko, "Prolog Programming for Artificial Intelligence", Addison-Wesley Yugoslavia 1990

[3] R. Schenk and R.P. Abelson, "Scripts, Plans and Knowledge", International Joint Conference on Artificial Intelligence, 1975

[4] F.C. Pereira and S.M.Schieber, "Prolog and Natural-Language Analysis", CSLI, 1987

[5] F.M. Donini, M. Lenzerini and others, "The complexity of concept languages", Inf. Comput., 1997

[6] I. Wegener, "Highlights aus der Informatik", Springer, Berlin, 1996

[MIT] http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-034-artificial-intelligence-fall-2010/, Zugriff 28.04.2016