Baumsuchalgorithmen*

Hans Jürgen Ohlbach

23. März 2018

Keywords: Bäume, Directed Acyclic Graphs (DAGs), Uninformierte Suche, Tiefensuche, Breitensuche, Iterative Tiefensuche

Empfohlene Vorkenntnisse: Rekursion, Java ist hilfreich, aber nicht notwendig

Inhaltsverzeichnis

*Dieser Text ist Teil einer Sammlung von Miniskripten zur Einführung in die Informatik. Er ist in erster Linien für Nichtinformatiker gedacht, kann aber natürlich auch als erste Einführung für Informatiker nützlich sein.

1 Einleitung

Hierarchische Strukturen die baumförmig organisiert sind, gibt es in vielen Zusammenhängen. Baumförmig bedeutet dabei, dass es eine „Wurzel“ gibt. Die Wurzel hat Nachfolger, die wiederum Nachfolger haben usw. Informatiker sprechen dann von *Knoten*, insbesondere *Wurzelknoten* und *Blattknoten*. Blattknoten haben keine Nachfolger. Ein Beispiel wären die Vorfahren eines bestimmten Menschen. Dieser Mensch wäre der „Wurzelknoten“. Seine Eltern bilden die erste Nachfolgeebene, deren Eltern die zweite Nachfolgeebene usw. Blattknoten wären in diesem Fall vermutlich Adam und Eva[1].

Eine anderes Beispiel wäre die Organisationsstruktur der Universität. Wurzelknoten ist die Universität selbst. Die erste Nachfolgeebene wären die Fakultäten. Darunter kämen die Institute, darunter wiederum die Lehrstühle usw.

In diesem Miniskript geht es um *uninformierte Suche*. Das bedeutet man sucht in einem Baum nach einem ganz bestimmten Knoten. Diesen kann man aber erst erkennen, wenn man ihn trifft. Vorher gibt es keinerlei Hinweise, wo dieser Knoten liegen könnte.

Ein Beispiel könnte sein: Paul ist rothaarig, und er versucht herauszufinden, von welchem seiner Vorfahren er die roten Haare hat. Er hat jede Menge Dokumente über seine Vorfahren, und sucht denjenigen oder diejenige, der oder die damals rote Haare gehabt hat.

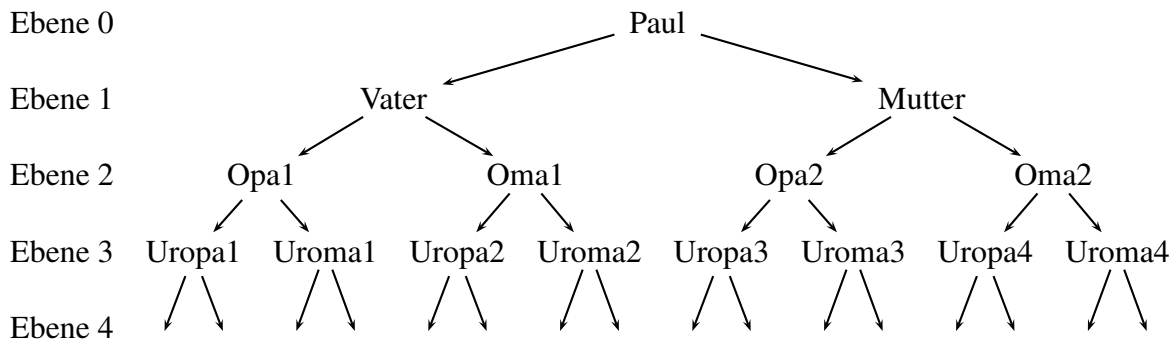
In diesem Miniskript werden drei Vorgehensweisen beschrieben, wie man einen solchen Knoten finden kann: Tiefensuche (depth first), Breitensuche (breadth first) und iterative Tiefensuche (iterative deepening).

2 Bäume

Wie in der Einleitung beschrieben, bestehen Bäume aus Knoten. Es gibt einen *Wurzelknoten* (engl. root node), der keine Vorgängerknoten hat. Jeder Knoten kann keinen, einen oder mehrere direkte Nachfolgeknoten haben. Knoten ohne Nachfolgeknoten sind *Blattknoten*. Jeder Knoten, außer dem Wurzelknoten hat genau einen direkten *Vorgängerknoten*. Der Weg vom Wurzelknoten zu einem bestimmten Knoten bezeichnet man als *Pfad* (engl. branch). Die maximale Länge eines solchen Pfades bezeichnet man als die *Tiefe* des Baumes. Im Prinzip kann es auch Knoten geben, die unendlich viele direkte Nachfolgeknoten haben. Es kann auch unendlich lange Pfade geben. Dafür terminieren die meisten Algorithmen aber nicht.

Anders als in der Natur, wo Bäume von der Wurzel nach oben wachsen, zeichnen Informatiker Bäume meist mit er Wurzel ganz oben. Sie wachsen dann Ebene für Ebene nach unten. Die Vorfahren von Paul würde man dann so darstellen:

[1] Wenn mehrere Personen den gleichen Vorfahren haben, ist es kein Baum, sondern ein DAG (siehe Kap. 6). Das ignorieren wir im Moment mal.

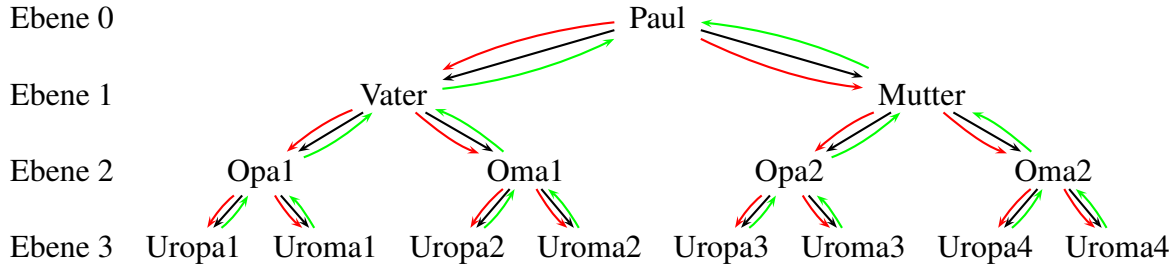


Es kann auch mehrere Wurzelknoten geben. Dann spricht man von einem *Wald*. Die Vorfahren von Paul und die Vorfahren von Maria würden als Wald dargestellt. Es kann dabei natürlich auch Knoten geben, die in mehreren Bäumen sind. Paul und Maria könnten ja denselben Vorfahren haben.

3 Tiefensuche (Depth-First Search)

Das erste Verfahren zur Suche eines bestimmten Knotens in einem Baum, welches wir betrachten, ist die Tiefensuche. Die Tiefensuche verfolgt einen Pfad bis an dessen Blattknoten, geht dann zurück zur letzten Verzweigung (*Backtracking*), verfolgt den alternativen Pfad usw. bis entweder der gesuchte Knoten gefunden wurde, oder alle Pfade abgesucht wurden.

Am besten sieht man die Vorgehensweise graphisch. Paul sucht wieder seinen rothaarigen Vorfahren, mit Tiefensuche, bis zur Ebene 3. Er sucht als erstes die rein väterliche Linie ab. Wenn der Uropa1 nicht rothaarig ist, muss er „backtracken“ zum Opa1, von dort die Uroma1 überprüfen usw. Die Pfeile in der folgen Graphik zeigen die Suchreihenfolge. Die roten Pfeile zweigen den Abstieg entlang der Pfade. Dabei wird der linkeste Knoten zuerst besucht. Die grünen Pfeile zeigen den Aufstieg (Backtracking).



Bei endlichen Bäumen terminiert die Tiefensuche immer. Hat der Baum aber einen unendlich langen Pfad, dann kann die Tiefensuche in diesem Pfad „versinken“ und terminiert dann nicht. Auch wenn es für einen Knoten unendlich viele direkte Nachfolgeknoten gibt (d.h. der Baum ist uneneldich breit), terminiert die Tiefensuche nicht immer.

Wie schnell das Ziel gefunden wird, hängt rein von den Daten ab. Wenn der Vater von Paul schon rothaarig ist, dann ist die Suche schnell fertig. Wenn allerdings die Mutter rothaarig wäre, dann würde Tiefensuche erst alle Vorfahren des Vaters absuchen, bis sie zur Mutter käme.

3.1 Eine Java Implementierung

In Java lässt sich ein Tiefensuchverfahren sehr elegant implementieren. Die Methode depthFirst braucht neben dem aktuellen Knoten, der getestet wird, eine Test-Funktion, die für einen Knoten entscheidet, ob er der gesuchte ist, oder nicht. Des weiteren braucht sie eine Funktion, die für einen Knoten die Menge der Nachfolgeknoten berechnet.

Eine Implementierung könnte so aussehen:

```
static <Node> Node depthFirst(Node node,
                    Function<Node, Boolean> test,
                    Function<Node, Collection<Node>> subNodes) {
   if (test.apply(node)) {return node;}
   for (Node subnode : subNodes.apply(node)) {
        Node found = depthFirst(subnode, test, subNodes);
        if (found != null) {return found;}}
   return null;}
```

Der generische Parameter `<Node>` bezeichnet die Klasse, die die Knoten repräsentiert. Das kann eine beliebige Klasse sein, auf die die Funktionen `test` und `subNodes` anwendbar sind. `test` ist eine Boolesche Funktion, die entscheidet, ob der Knoten der gesuchte ist. `subNodes` liefert für einen Knoten die Menge der Nachfolgeknoten als Collection. Die Collection kann eine beliebige Klasse sein, die das Collection Interface implementiert.

Als erstes testet die Funktion, ob der Knoten schon der richtige ist. Wenn ja, wird dieser Knoten als Ergebnis der Funktion zurückgegeben. Falls nicht, läuft eine Schleife über die von der subNodes-Funktion gelieferte Collection und ruft die depthFirst-Funktion rekursiv auf. Sobald sie eine nicht-null-Ergebnis liefert, wird die Schleife abgebrochen und der Knoten zurückgeliefert. D.h. zunächst wird depthFirst für den ersten Unterknoten aufgerufen, dort wieder für dessen ersten Unterknoten usw. Hierin steckt die Tiefensuche. Die Verwaltung der Pfade und das Backtracking übernimmt der in Java eingebaute Aufrufmechanismus für die Rekursion.

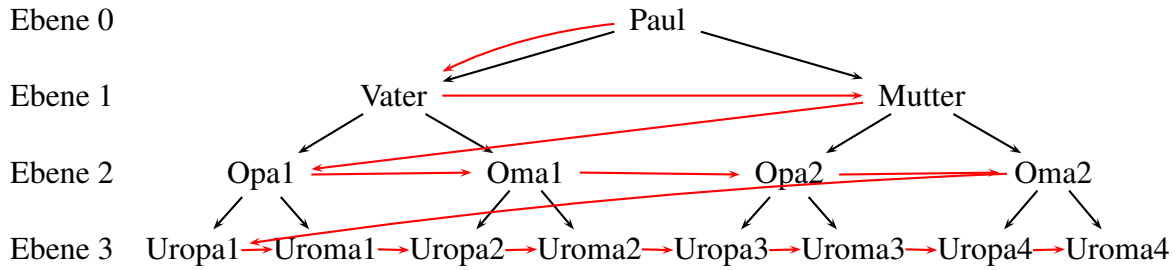
Um das zu testen, erzeugen wir einen Baum, dessen Knoten Strings aus Ziffern sind. Der Wurzelknoten ist der leere String. Jeder „String-Knoten“ hat eine zufallserzeugte Menge (eine ArrayList mit bis zu 4 Elementen) von Nachfolgeknoten, die so erzeugt sind, dass an den aktuellen String eine zufällig erzeugte Ziffer angehängt wird. Die Testfunktion sucht nach einem „String-Knoten“, der die Ziffernreihenfolge 123 enthält. Am Ende wird depthFirst aufgerufen und das Ergebnis ausgedruckt.

```
public static void main(String[] args) {
   Random rnd = new Random(); // Zufallsgenerator
   Function<String, Boolean>
      test = (nodeString -> nodeString.contains("123"));
   Function<String, Collection<String>>
      subNodes = (nodeString -> {
         ArrayList sn = new ArrayList();
         for (int i = 0; i < rnd.nextInt(5); ++i) {
               sn.add(nodeString + rnd.nextInt(5));}
         return sn;});
   String root = "";
   System.out.println(depthFirst(root, test, subNodes));}
```

Blattknoten sind die „String-Knoten“, wo der Zufallsgenerator bei `i < rnd.nextInt(5)` die 0 ausgibt. Der Baum wäre nur endlich, wenn der Zufallsgenerator irgendwann genügend viele 0 nacheinander ausgeben würde. Da das kaum der Fall ist, ist der Baum im Prinzip unendlich. Ein Sonderfall ist allerdings, wenn der Zufallsgenerator schon für den Wurzelknoten die 0 ausgibt. Dann besteht der Baum nur aus dem Wurzelknoten. Wenn das nicht der Fall ist wird durch den Zufallsgenerator irgendwann ein Knoten erzeugt, wo die 123 drin vorkommt.

4 Breitensuche (Breadth-First Search)

Die Breitensuche durchsucht den Baum Ebene für Ebene. Sie kann also nicht, wie bei der Tiefensuche, in einem unendlich langen Pfad „versinken“. Das folgende Bild illustriert die Breitensuche wieder für unseren rothaarigen Freund Paul, der einen rothaarigen Vorfahren sucht.



4.1 Eine Java Implementierung

Breitensuche ist nicht ganz so einfach zu implementieren wie Tiefensuche. Wenn man in einer Ebene k sucht, dann muss man sich bei einem Knoten N der Ebene k, die Nachfolgeknoten von N aus der Ebene k+1 merken, damit man, wenn die Ebene k abgesucht wurde, die Nachfolgeknoten auf Ebene k+1 bereit hat. In dem obigen Beispiel von Paul, der seinen rothaarigen Vorfahren sucht, bedeutet das, dass er sich, wenn er den Vater getestet hat, Opa1 und Oma1 merken muss, weil er nach dem Test von Mutter dort weiter suchen muss.

In der Implementierung unten geschieht das durch eine Schlange (`queue`). Zu Beginn kommt als einziges Element der Wurzelknoten in die Schlange. Dann gibt es eine Schleife, in der man der Schlange das erste Element entnimmt, testet, und dann dessen Nachfolgeelemente hinten an die Schlange anhängt.

Für Paul mit seinen Vorfahren würde sich die Schlange folgendermaßen entwickeln:
Paul
Vater, Mutter
Mutter, Opa1, Oma1
Opa1, Oma1, Opa2, Oma2
Oma1, Opa2, Oma2, Uropa1, Uroma1
Opa2, Oma2, Uropa1, Uroma1, Uropa2, Uroma2
Oma2, Uropa1, Uroma1, Uropa2, Uroma2, Uropa3, Uroma3
Uropa1, Uroma1, Uropa2, Uroma2, Uropa3, Uroma3, Uropa4, Uroma4
Ab jetzt verkürzt sich die Schlange nur noch.

Die Java-Implementierung könnte so aussehen.

```
static <Node> Node breadthFirst(Node node,
                    Function<Node, Boolean> test,
                    Function<Node, Collection<Node>> subNodes) {
   ArrayDeque<Node> queue = new ArrayDeque<Node>();
   queue.add(node);
   Node subnode;
   while((subnode = queue.pollFirst()) != null) {
       if(test.apply(subnode)) {return subnode;}
       queue.addAll(subNodes.apply(subnode));}
   return null;}
```

(`pollFist()` entfernt das erste Element der Schlange und liefert es zurück.
`addAll(collection)` hängt alle Elemente der collection hinten an die Schlange an.)

Die Argumente für die Methode sind die gleichen wie bei depthFirst. Nur die Suchreihenfolge ist anders. Es ist dabei auch keine Rekursion nötig. Testen kann man die Methode auf die gleiche Weise wie die depthFirst-Methode. Man muss nur den Aufruf in der main-Methode oben durch breadthFirst ersetzen.

5 Iterative Tiefensuche (Iterative Deepening)

Die Breitensuche hat den Nachteil, dass man die Schlange *auf Vorrat* mit den Elementen der nächsten Ebene des Baumes auffüllen muss, egal ob sie tatsächlich gebraucht werden, oder nicht. Bei einfachen Beispielen ist das kein Problem. Es gibt aber Beispiele, wo die Knoten des Baumes umfangreiche Datenstrukturen sind, und daher das Erzeugen der nächsten Ebene des Baumes *auf Vorrat* sehr teuer sein kann. Tiefensuche würde das vermeiden, da immer nur ein Element der nächsten Ebene gebraucht wird. Allerdings kann Tiefensuche in unendlichen Pfaden versinken.

Ein Ausweg ist, Tiefensuche zu machen, aber nur bis zu einer bestimmten Ebene k. Wird der Knoten dann nicht gefunden, startet man die Tiefensuche von vorne, diesmal bis zur Ebene k+1, und dann bis k+2 usw. Das erscheint extrem aufwendig, da ja die oberen Ebenen mehrfach abgesucht werden.

Man kann sich jedoch folgendes überlegen: hat jeder Knoten des Baumes mindestens 2 Nachfolgeknoten, dann ist der Aufwand, die Ebene $k+1$ mit seinen mindestens 2^{k+1} Knoten allein abzusuchen, größer als alle Knoten $\Sigma_{i=0}^{k} 2^k = 2^{k+1} - 1$ bis zur Ebene k abzusuchen. Der Mehraufwand für die wiederholte Suche in den oberen Ebene ist also nicht so schlimm.

5.1 Eine Java Implementierung

Für die Java-Implementierung trennen wir die Suchmethoden in zwei einzelne Methoden. Die erste macht die Tiefensuche, allerdings nur bis zu einer gegebenen Tiefe `maxLevel`. Die zweite Methode ruft die erste Methode auf, und erhöht dabei `maxLevel` so lange, bis der gesuchte Knoten gefunden wurde.

```
static <Node> Node limitedDepthFirst(Node node,
                    Function<Node, Boolean> test,
                    Function<Node, Collection<Node>> subNodes,
                    int level, int maxLevel) {
  if (test.apply(node)) {return node;}
  if(level == maxLevel) {return null;}
  for (Node subnode : subNodes.apply(node)) {
       Node found = limitedDepthFirst(subnode, test, subNodes,
                                      level+1,maxLevel);
       if (found != null) {return found;}}
  return null;}

static <Node> Node iterativeDeepening(Node node,
                    Function<Node, Boolean> test,
                    Function<Node, Collection<Node>> subNodes) {
  for(int maxLevel = 1;;++maxLevel) {
      Node found = limitedDepthFirst(node,test,subNodes,0,maxLevel);
      if(found != null) {return found;}}}
```

Diese Implementierung hat noch einen Haken. Falls der Baum endlich ist, aber den Knoten nicht enthält, wird `maxLevel` trotzdem unbegrenzt weiter erhöht. Was fehlt ist ein Test, der feststellt, welche Ebene überhaupt noch Knoten enthält, und damit `maxLevel` begrenzt.

Auch diese Methode kann man mit der obigen main-Methode testen, indem man statt `depthFirst` die Methode `iterativeDeepening` aufruft.

6 Directed Acyclic Graphs (DAGs)

Eine erste Verallgemeinerung von Bäumen sind sog. *Directed Acyclic Graphs* (oder DAGs). Diese sind auch Bäume, wo es für jeden Knoten eine Reihe von Unterknoten gibt. Der Unterschied ist, dass die verschiedenen Pfade auch wieder zusammen laufen können. In unserem Vorfahren-Beispiel könnte es ja sein, dass der Uropa des Vaters und der Uropa der Mutter die gleiche Person ist.

Für DAGs funktionieren die Suchverfahren im Prinzip genauso wie für Bäume, können aber optimiert werden. Ist in unserem Beispiel der Uropa des Vaters von Paul und der Uropa der Mutter tatsächlich die gleiche Person, dann braucht man dessen Vorfahren ja nur einmal abzusuchen.

Um zu verhindern, dass die Nachfolgeknoten eines Knotens mehrfach abgesucht werden, muss man den Knoten beim ersten mal markieren. Kommt die Suche später über einen anderen Pfad an denselben Knoten, kann sie an der Markierung erkennen, dass hier nicht weiter gesucht werden muss. Für die Implementierung bedeutet das aber, dass die Knotenklasse (`Node`) eine Möglichkeit bieten muss, die Markierung zu setzen und auszulesen.

7 Ausblick

Wir haben jetzt nur uninformierte Suche betrachtet. Das bedeutet, man erkennt das Ziel erst dann, wenn es erreicht ist. Vorher gibt es keinerlei Information, wo das Ziel liegen könnte. Es gibt jedoch andere Situationen, wo mehr Information vorhanden ist, um die Suche zu steuern:

Bäume für Objekte mit einer Ordnungsrelation: z.B. eine alphabetisch sortierte Telefonliste. Hier bieten bestimmte Baumtypen besondere Vorteile, um die Verwaltung und die Suche möglichst effizient zu machen.

Graphen mit Kantengewichten: Hierbei haben die Kanten zwischen den Knoten Gewichte. Die Knoten könnten z.B. Straßenkreuzungen sein, und die Kanten die Straßen dazwischen, mit Angaben über deren Länge. Das Problem ist dann, den kürzesten Weg zwischen zwei Knoten zu suchen. Dafür eignet sich der Dijkstra-Algorithmus. Kann man darüber hinaus noch den Abstand zum Zielknoten schätzen (z.B. Luftlinie), dann hilft der A*-Algorithmus.

Suche nach Variablenbelegungen in Constraint Netzen: Ein Beispiel ist das n-Farben Problem in Graphen: kann man die Knoten in einem gegebenen Graphen mit n Farben einfärben, so dass Nachbarknoten nicht dieselbe Farbe haben? Hier sind die Knotennamen die Variablen, und die möglichen Farben deren mögliche Werte. Gesucht ist eine Belegung der Variablen mit Werten, so dass die gegeben Bedingungen (Constraints) eingehalten werden.

Hill-Climbing: Dabei hat man eine Struktur, in der man Punkte lokalisieren kann, und für jeden Punkt ergibt sich ein Wert. Einfachstes Beispiel wäre eine Gebirgslandschaft. Die 2-D Koordinaten erlauben die Lokalisierung der Punkte, und die Höhe über NN wäre der Wert. Gesucht ist jetzt der Punkt mit dem höchsten Wert. Im Gebirge wären das die Koordinaten des höchsten Berges.

Algorithmen für diese Art von Problemen sind z:B.: Steepest Ascent, Simulated Annealing, Genetische Algorithmen.

Stichwortverzeichnis