Graph: Transitiver Abschluss

Herbert Schiemann <h.schiemann@herbaer.de>

Inhaltsverzeichnis

Wohin von hier aus?

Wir sitzen auf einem Knoten in einem endlichen gerichteten Graphen und fragen uns: Wo kommen wir hin? Wir suchen nicht nur die unmittelbar folgenden Knoten (Kinder), sondern auch deren folgenden Knoten und alle weiteren erreichbaren Ziele.

Sie sehen im Folgenden Programmier-Esperanto, eine leicht verständliche Pseudo-Programmiersprache. Unter einem Knoten verstehe ich etwa Folgendes:

```
type Node {
   Set<Node> children
}
```

Wir arbeiten mit zwei Eimern (`Set<Node>`). Die Kinder unseres Knotens kommen erst einmal in den zweiten Eimer `found`. Dann packen wir alle Knoten aus dem zweiten Eimer in den ersten Eimer `reachable`, und immer, wenn wir einen Knoten in den ersten Eimer packen, packen wir alle seine Kinder in den zweiten Eimer:

```
type  Nodeset  Set<Node>
Nodeset find_reachable_nodes (Node node) {
   Nodeset reachable  = empty_set
   Nodeset found      = node.children
   Node    nn
   while ( not found.is_empty() ) {
      nn = found.draw ()
      if (not reachable.contains (nn)) {
         found.unite (nn.children)
         reachable.insert (nn)
      }
   }
   return reachable
}
```

Natürlich würde es reichen, nur die Kinder in den Eimer `found` zu packen, die noch nicht im Eimer `reachable` sind. Wir prüfen aber schon, wenn wir einen Knoten in den Eimer `reachable` packen. Ob die vorgeschaltete Prüfung wirklich die "Performance" steigert, kann man nicht generell sagen. Es kommt darauf an :-)

Die Details stecken in den Funktionen `found.draw` und `found.unite`. Auch wenn hier im Esperanto `found` und `reachable` beide Nodesets sind, können sie ganz unterschiedlich implementiert sein. `found` kann eine Liste sein, und `found.draw` immer das erste Element der Liste entfernen. Wenn `found.unite`(*nn*) die Elemente von *nn* an das Ende der Liste anfügt, dann ergibt sich eine Rasterfahndung ("Breitensuche"). Wenn `found.unite` die neuen Elemente am Anfang der Liste einfügt, ergibt sich eine Tiefbohrung ("Tiefensuche"). Wenn `found.draw` immer das mittlere Element der Liste herausnimmt, ergibt sich ein systematisches Stochern. `found.draw` kann auch pseudozufällig ein Element herausnehmen. Hier gibt es keinen Grund, warum eine Variante gegenüber einer anderen Variante systematisch zu bevorzugen sei.

Die Datentypbezeichnung Set legt nahe, dass jedes Objekt nur einmal enthalten sein kann. Darauf kommt es hier aber nicht an. Wenn es so ist und der damit verbundene Test nicht zu aufwendig ist, schön; wenn nicht, dann nicht. Vielleicht ist eine einfache Liste die effektivste Implementation. Kiss.

Was wir wissen ...

... das wissen wir.

Wenn wir einmal die Knoten aufgesammelt haben, die von einem bestimmten Knoten aus erreichbar sind, dann können wir das Ergebnis merken und nutzen, wenn wir später wissen wollen, welche Knoten von einem anderen Knoten aus erreichbar sind. Dazu erweitern wir den Datentyp "Node":

```
type Node {
   Set<Node>          children
   optional Set<Node> reachable
}
```

Unser Programm sieht jetzt so aus:

```
type  Nodeset  Set<Node>
Nodeset find_reachable_nodes (Node node) {
   if (not defined node.reachable) {
      Nodeset reachable  = empty_set
      Nodeset found      = node.children
      Node    nn
      while ( not found.is_empty() ) {
         nn = found.draw ()
         if (not reachable.contains (nn)) {
            reachable.insert (nn)
            if (defined nn.reachable) {
               reachable.unite (nn.reachable)
            }
            else if (nn != node) {
               found.unite (nn.children)
            }
         }
      }
      node.reachable = reachable
   }
   return node.reachable
}
```

Wenn die erreichbaren Knoten noch nicht bekannt sind (`defined node.reachable`), dann ermitteln wir sie und speichern das Ergebnis als `node.reachable`. Wenn von einem hinzuzufügenden Knoten die erreichbaren Knoten bekannt sind (`defined nn.reachable`), dann packen wir diese in den zweiten Eimer (`reachable.unite (nn.reachable)`) und brauchen uns um dessen Kinder nicht zu kümmern. Andernfalls packen wir die Kinder des neuen Knotens in den ersten Eimer (`found.unite (nn.children)`), wenn der neue Knoten nicht "unser" Knoten ist (`nn != node`). Die Kinder "unseres" Knotens sind nämlich schon "im Eimer". Im ersten Programm habe ich auf diese letzte Prüfung verzichtet. Diese Version funktionierte auch ohne die Prüfung, aber wenn schon eine Bedingung geprüft wird, dann macht die zweite Prüfung nicht mehr so viel aus. Ob die zweite Prüfung die "Performance" steigert (wahrscheinlich) oder eher herabsetzt, hängt von der Struktur des Graphen ab.

Für alle Knoten

Wir wollen für alle Knoten eines endlichen gerichteten Graphen die von dort erreichbaren Knoten ermitteln. Es liegt nahe, das Programm rekursiv aufzurufen:

```
type  Nodeset  Set<Node>

// so geht es nicht
Nodeset find_reachable_nodes (Node node) {
   if (not defined node.reachable) {
      Nodeset reachable  = empty_set
      Node    nn
      for nn in node.children {
         if (not reachable.contains (nn)) {
            reachable.insert (nn)
            if (nn != node) {
               find_reachable_nodes (nn)
               reachable.unite (nn.reachable)
```

```
            }
         }
      }
      node.reachable = reachable
   }
   return node.reachable
}
```

Solcher Quelltext entsteht nachts um halb Eins. (Nachts um halb Eins. Ein Mann tastet sich um die Litfassäule herum - "Buh, eingemauert.")

Damit wir nicht in einer Schleife um die Litfassäule hängen bleiben, sperren wir unseren Knoten und nehmen wieder den zweiten Eimer:

```
type Node {
   Set<Node>          children
   optional Set<Node> reachable
   logical            locked
}
type  Nodeset  Set<Node>

Nodeset find_reachable_nodes (Node node) {
   if (not defined node.reachable) {
      Nodeset reachable  = empty_set
      Nodeset found      = node.children
      Node    nn
      node.locked = true
      while ( not found.is_empty() ) {
         nn = found.draw ()
         if (not reachable.contains (nn)) {
            reachable.insert (nn)
            if (not nn.locked) {
               find_reachable_nodes (nn)
               reachable.unite (nn.reachable)
            }
            else if (nn != node) {
               found.unite (nn.children)
            }
         }
      }
      node.locked    = false
      node.reachable = reachable
   }
   return node.reachable
}
```

So funktioniert es.

Wir wissen mehr

Während wir die erreichbaren Knoten aufsammeln, zeigen sich die wachsenden Erkenntnisse im "Füllstand" der Eimer `reachable` und `found`. Es ist Vergeudung, den Zwischenstand der Erkenntnis nicht zu nutzen, wenn wir zum Ausgangsknoten zurückkommen. Binden wir deshalb unsere beiden Eimer sofort an den Knoten! Die Sperre brauchen wir dann nicht

```
type Node {
   Set<Node>          children
   optional Set<Node> reachable
   optional Set<Node> found
}
type  Nodeset  Set<Node>

Nodeset find_reachable_nodes (Node node) {
   if (not defined node.reachable) {
      node.reachable  = empty_set
      node.found      = node.children
      Node    nn
      while ( not node.found.is_empty() ) {
         nn = node.found.choose ()
```

```
         if (not reachable.contains (nn)) {
            find_reachable_nodes (nn)
            reachable.unite (nn.reachable)
            if (defined nn.found) {
               node.found.unite (nn.found)
            }
            node.reachable.insert (nn)
         }
         node.found.remove (nn)
      }
      delete node.found
   }
   return node.reachable
}
```

`node.reachable` ist jetzt selbst die Sperre gegen Endlosschleifen. Wenn beim Aufruf von `find_reachable_nodes (node)` `node.reachable` definiert ist, dann muss jeder Kindknoten von *node* oder eines Knotens in `node.reachable` in `node.reachable` oder `node.found` enthalten sein. Es funktioniert nicht, wenn ich den Knoten `nn` vor dem Aufruf von `find_reachable_nodes (nn)` aus `node.found` herausnehme: der Knoten `nn` würde als möglicher erreichbarer Knoten möglicherweise unterschlagen. Also wähle ich erst `nn` aus (`nn = node.found.choose ()`) und entferne ihn später (`node.found.remove (nn)`).

Schlusswort

Vielleicht ist es sinnvoll, alle Äquivalenzklassen zyklisch erreichbarer Knoten zu bilden, und dann den "transitiven Abschluss" des Baums der Äquivalenzklassen.

Eine interessante Variation des Programms ist es, wenn die Funktion `find_reachable_nodes` bei jedem Aufruf nur eine begrenzte Zahl von Elementen aus dem Eimer `found` herausnimmt. In diesem Fall braucht man wieder eine Sperre zum Schutz gegen Endlosschleifen.

Der "echte" Anwendungsfall, der diesen Artikel inspirierte: Aus einem XML-Schema soll Code generiert werden, der Daten aus einem XML-Dokument in eine Datenstruktur liest und umgekehrt eine Datenstruktur als XML-Dokument ausgibt. Dabei stellt sich die Frage, welche Element-Inhaltsmodelle als Nachkommen eines Element-Inhaltsmodell vorkommen können.