Notizen zu Algorithmen II

Jens Ochsenmeier

16. Februar 2018

Inhaltsverzeichnis

1

Anwendungen von DFS

1.1 Starke Zusammenhangskomponenten

Zusammenhangskomponenten in einem ungerichteten Graph sind Teilgraphen, in denen es zwischen je zwei beliebigen Knoten einen Pfad gibt.

In gerichteten Graphen sind **starke Zusammenhangskomponenten** Teilgraphen $G \subseteq H$, in denen ebenfalls gilt: für jedes Knotenpaar $u, v \in G$ gibt es einen u-v-Pfad *und* einen v-u-Pfad.

Insbesondere werden starke Zusammenhangskomponenten durch Zyklen erzeugt (dann kann man einfach im Kreis laufen von einem Knoten zum anderen). Das bedeutet im Umkehrschluss, dass der **Schrumpfgraph** — das ist der Graph, den man erhält, indem man jede starke Zusammenhangskomponente als einen Knoten zusammenfasst — zyklenfrei ist.

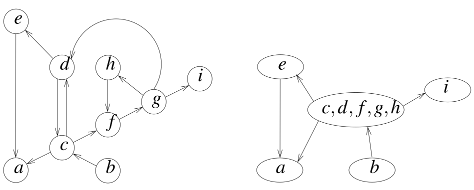

Abbildung 1.1. Gerichteter Graph G und zugehöriger Schrumpfgraph G_S.

Tiefensuchschema

Um später SCCs ermitteln zu können konstruieren wir ein Tiefensuchschema:

```
unmark all nodes
init()
foreach s ∈ V do
  if s is not marked then
    mark s
    root(s)
    DFS(s, s)

DFS(u, v : Node)
  foreach (v, w) ∈ E do
    if w is marked then
      traverseNonTreeEdge(v, w)
    else
      traverseTreeEdge(v, w)
      mark w
      DFS(v, w)
  backtrack(u, v)
```

Dieses Tiefensuchschema kann auf unterschiedliche Graphtraversierungsprobleme angepasst werden.
Wir verwenden nun zwei Arrays zum Zwischenspeichern unserer Resultate:

- `oNodes` speichert die bereits besuchten Knoten,
- `oReps` speichert die Repräsentanten der einzelnen SCCs.

Beim Durchlaufen des Graphen werden die Knoten mit `dfsNum` inkrementell durchnummeriert.
Außerdem gibt es drei Invarianten, die wir im Folgenden nicht verletzen dürfen:

1. Kanten von abgeschlossenen Knoten gehen zu abgeschlossenen Knoten
2. Offene Komponenten $S_1, \ldots, S_k$ bilden einen Pfad in G_C^s.
3. Repräsentanten partitionieren die offenen Komponenten bezüglich ihrer `dfsNum`.

Für das Finden von SCCs brauchen wir folgende Implementierungen für die rot gekennzeichneten Prozesuren:

- **`root(s)`**:

  ```
  oReps.push(s)
  oNodes.push(s)
  ```

  Hierdurch wird eine neue offene Komponente gebildet und s als besucht gekennzeichnet.

- **`traverseTreeEdge(v,w)`**:

```
oReps.push(w)
oNodes.push(w)
```

Hier wird $\{w\}$ als neue offene Komponente angelegt.

- **`traverseNonTreeEdge(v,w)`**:

```
if w ∈ oNodes then
  while w.dfsNum < oReps.top.dfsNum do oReps.pop
```

Ist $w \notin$ oNodes ist w abgeschlossen und die Kante somit uninteressant. Ist w allerdings in oNodes, so werden die auf dem Kreis befindlichen SCCs kollabiert.

- **`backtrack(u,v)`**:

```
if v ≡ oReps.top then
  oReps.pop
  repeat
    w := oNodes.pop
    component[w] := v
  until w = v
```

Damit haben wir alles was wir brauchen, um die Suche nach SCCs durchführen zu können. Wir kriegen sie sogar in $O(m + n)$, also in Linearzeit, hin!

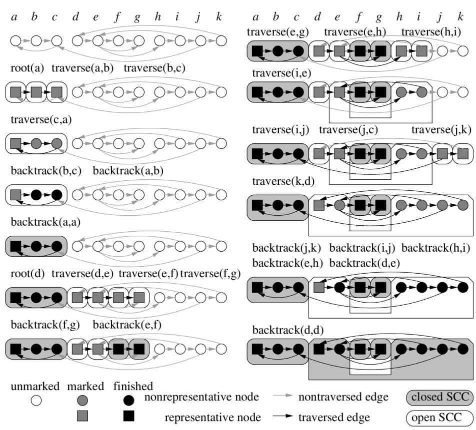

Abbildung 1.2. Kompletter Durchlauf des Algorithmus.