Tiefensuche

Stapel und die Grundidee der Tiefensuche

Die Idee der *Tiefensuche* (depth first search) ist einfach. Hat ein Knoten, den man besucht, noch unentdeckte Nachbarn, so geht man zum ersten solchen Nachbarn, den man findet, und von dort wieder in die 'Tiefe' zu einem noch unentdeckten Nachbarn des Nachbarn, falls es ihn gibt. Das macht man rekursiv, bis man nicht mehr in die Tiefe gehen kann. Dann geht man solange zurück (backtracking–Schritte), bis wieder eine Kante in die Tiefe geht, bzw. alles Erreichbare besucht wurde.

Die Farben haben wieder dieselbe Bedeutung wie beim BFS, ebenso die π–Zeiger, die die entstehende Baumstruktur (genauer den aufspannenden Wald) implizit speichern. Der DFS berechnet aber nicht die Abstände vom Startknoten.

Man kann aber jedem Knoten ein Zeitintervall zuordnen, indem er 'aktiv' ist, was für verschiedene Anwendungen interessant ist. Dazu läßt man eine globale Uhr `t` mitlaufen, und merkt sich für jeden Knoten u den Zeitpunkt $d[u]$, bei dem u entdeckt wird und den Zeitpunkt $f[u]$, bei dem u erledigt ist, da der Algorithmus alles in der 'Tiefe' unter u gesehen hat. Daher gilt für beliebige zwei Knoten im Graphen, dass entweder eines der Zeitintervalle voll im anderen enthalten ist, oder beide disjunkt sind.

Die geeignete Datenstruktur, um DFS zu implementieren, ist ein *Stapel (Kellerspeicher, Stack)*. Der Stack ist eine Datenstruktur, die das LIFO–Prinzip (*last–in–first–out*) umsetzt. Das heißt, die zu speichernden Daten (hier die grau werdenden Knoten) sind linear angeordnet und man kann ein neuen Eintrag nur als neues "oberstes" Element in den Stack einfügen (mit einer `push`–Operation) bzw. das oberste Element entfernen (mit einer `pop`–Operation, also beim DFS wenn der Knoten schwarz wird). Als weitere Funktionalität bietet ein Stack die Abfrage nach seiner Größe (`size`–Operation), die Boolesche Anfrage `isEmpty` und die Ausgabe des obersten Eintrages (`top`–Operation, ohne den Eintrag zu entfernen).
In der folgenden Implementierung wird der Stack nur implizit angelegt, um die Rekursionsaufrufe zu verwalten, d.h. der Pseudocode kommt ohne die genannten Operationen aus, aber zur Veranschaulichung des Ablaufs kann die Verwendung eines Stapels sehr hilfreich sein.

Pseudocode und ein Beispiel

Wie beim BFS hängen die entstehenden DFS–Bäume von der Reihenfolge in den Adjazenzlisten ab. Verschiedene Reihenfolgen können zu nichtisomorhen DFS–Bäumen führen. Die Tiefensuche wird auch oft für gerichtete Graphen verwendet, d.h. man besucht dann alle Knoten, die vom Startknoten über einen gerichteten Weg erreichbar sind. In diesem Fall beinhaltet $Adj[u]$ alle Knoten v mit $(u,v) \in E$. Der folgende Pseudocode ist für gerichtete und ungerichtete Graphen geeignet. Man beachte, dass in gerichteten Graph die berechneten π-Zeiger die Gegenrichtung der Kante anzeigt, über die ein Knoten erreicht wurde. Der Algorithmus verzichtet auf die Spezifizierung eines speziellen Startknotens und verwendet den ersten Knoten aus der Gesamtliste als solchen.

Tiefensuche `DFS(G)`:

```
t = 0
for all u ∈ V(G)
      Farbe[u] = weiss
      π[u] = NIL
for all u ∈ V(G)
if (Farbe[u] == weiss) then DFS-visit(u)
```

```
procedure DFS-visit(u)
Farbe[u] = grau
t = t+1
d[u] = t
for all v ∈ Adj[u]
      (if Farbe[v] == weiss) then
                π[v] = u
                DFS-visit(v)
Farbe[u] = schwarz
t = t+1
f[u] = t
```

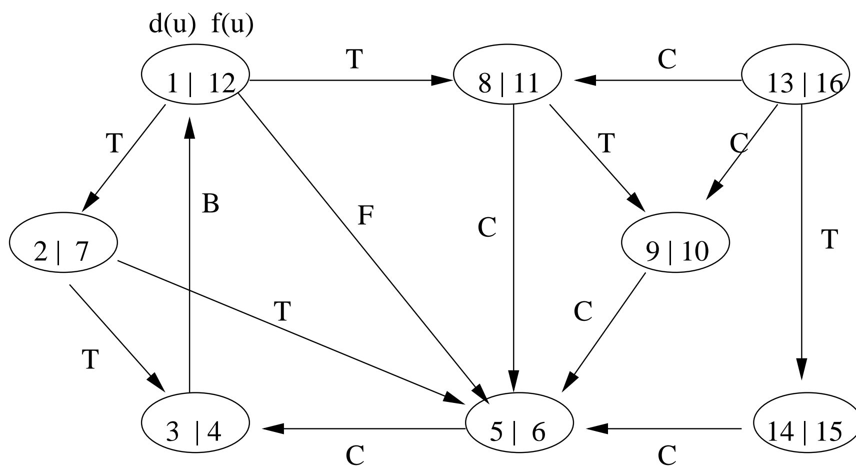
Anmerkungen:

1) Ist der Graph in Adjazenzlistenform gegeben, so läuft DFS ebenfalls in Zeit $O(|V|+|E|)$.

2) Man kann die Kanten eines gerichteten Graphen nach ihrer Rolle bei einer DFS-Durchmusterung klassifizieren. Wir unterscheiden:

- Tree-Kanten (T–Kanten): von grauen nach weißen Knoten
- Back–Kanten (B–Kanten): von grau nach grau
- Forward–Kanten (F–Kanten): von grau nach schwarz und zwar von Vorfahren zu Nachkommen im DFS–Baum
- Cross–Kanten (C–Kanten): alle restlichen Graphkanten

Im folgenden Beispiel ist der Ablauf der Tiefensuche durch die Zeitintervalle und die Kantenklassifikation illustriert:



DFS–Suche. Zeitintervalle der Knoten und Kantenklassifikation

Die Zeitintervalle haben eine sehr schöne Eigenschaft. Liegt Knoten v unter Knoten u, so ist $(d[v], f[v]) \subset (d[u], f[u])$. Also Knoten, die tiefer liegen werden später entdeckt, sind aber eher fertig. Liegt v nicht unter u und u nicht unter v, so sind die Intervalle disjunkt.

Gerichtete azyklische Graphen und topologisches Sortieren

Definition: Ein *gerichteter, azyklischer Graph (DAG)* ist gerichteter Graph ohne gerichtete Kreise.

DAG's spielen in der Informatik an verschiedenster Stelle eine Rolle, zum Beispiel bei der Vererbungshierarchie in Java.
Für eine andere häufig vorkommende Anwendung stelle man sich eine Menge von Jobs vor, die linear zu ordnen sind. Dabei gibt es Einschränkungen (Constraints) derart, dass ein Job a vor einem anderen Job b bearbeitet werden muß, repräsentiert durch eine gerichtete Kante von a nach b. Wenn diese Einschränkungen nicht in sich widersprüchlich sind, beschreiben die Kanten einen DAG. Gesucht ist eine lineare Ordnung (eine sogenannte *topologische Sortierung*), die alle diese Constraints berücksichtigt.

Definition: Eine topologische Sortierung eines gerichteten Graphen ist eine Nummerierung seiner Knoten derart, dass aus $(u,v) \in E$ folgt $u \leq v$ in der Nummerierung.

Offensichtlich muss der gerichtete Graph ein DAG sein, um eine topologische Sortierung zuzulassen. Das es dann aber immer geht, überlegt man sich wie folgt.

Satz: Jeder DAG hat eine Quelle (Knoten mit Ingrad 0) und eine Senke (Ausgrad 0).

Beweis: Wir wählen einen beliebigen Knoten v_0 und testen, ob er Quelle ist. Falls nicht, betrachten wir eine Kante (v_1, v_0) und testen, ob v_1 Quelle ist u.s.w. Angenommen, wir haben einen gerichteten Weg $v_i \to \ldots \to v_0$ von Nichtquellen schon gefunden. Dann gilt $v_{i+1} \notin \{v_0, \ldots, v_i\}$, denn sonst gäbe es einen gerichteten Kreis. Dieser Prozess muss abbrechen mit einer gefundenen Quelle, denn der Graph ist endlich.
Eine Senke findet man analog, indem man ausgehende Kanten verfolgt. □

Diesen Satz kann man auch algorithmisch umsetzen: Suche eine Senke, diese bekommt die größte Nummer in der topologischen Sortierung. Entferne die Senke und die eingehenden Kanten. Dies liefert einen neuen DAG und man iteriert.
Das ist zwar einfach aber auch nicht besonders effizient, wenn der Startknoten bei der Suche nach einer Senke immer ungünstig gewählt ist. Es geht besser mit einer Tiefensuche:

Fakt: Ein gerichteter Graph ist ein DAG genau dann, wenn sein DFS–Durchmustern keine Back–Kanten produziert.

Beweis:
($\Rightarrow$): Angenommen, es gäbe eine Back–Kante (u,v). Das heißt, v ist Vorfahre von u im dfs–Wald. Damit gibt es einen gerichteten Weg von v nach u und (u,v) schließt einen gerichteten Kreis.
($\Leftarrow$): Angenommen, G enthält einen gerichteten Kreis c. Sei v der erste Knoten in c, der beim DFS entdeckt wird und sei (u,v) Kante in c. Es gibt also zum Zeitpunkt $d[v]$ einen gerichteten Weg vom grauen Knoten v zum weißen Knoten u, der nur weiße Knoten benutzt.
u ist Nachkomme von v im DFS–Wald, also $f[u] < f[v]$ und (u,v) ist Back–Kante. □

Basierend auf dieser Einsicht kann man topologisches Sortieren wie folgt realisieren: Führe ein DFS für den DAG durch; wenn immer ein Knoten schwarz wird, gib ihn aus.

Behauptung: Dies ergibt ein topologisch invers sortierte Knotenfolge.

Beweis: Zu zeigen, wenn $(u,v) \in E$, dann ist $f(v) \leq f(u)$. Wir betrachten den Moment, wenn die Kante (u,v) vom DFS untersucht wird. Zu diesem Zeitpunkt ist u grau. v kann nicht grau sein wegen obigen Fakts. Also bestehen nur die beiden Möglichkeiten, v ist weiß oder schwarz. Aber in beiden Fällen ist offensichtlich $f(v)$ kleiner als $f(u)$, denn weiße Knoten, die unter u liegen, sind "eher" fertig als u und schwarze Knoten sind ja schon völlig abgearbeitet. □