Notizen zu Algorithmen II

Jens Ochsenmeier

18. 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.

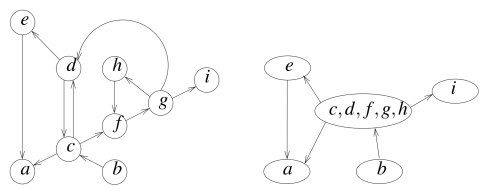

Abbildung 1.1. Gerichteter Graph G und zugehöriger Schrumpfgraph G_S.

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!

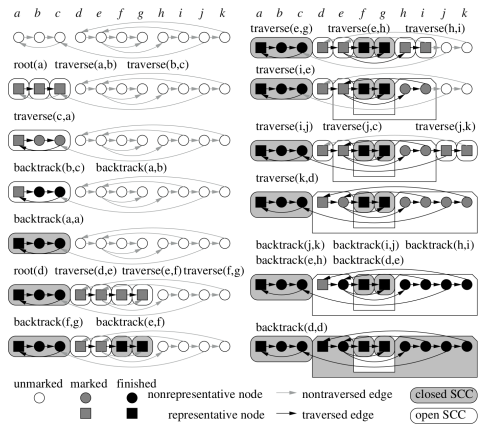

Abbildung 1.2. Kompletter Durchlauf des Algorithmus.

2

Maximale Flüsse und Matchings

2.1 Definitionen

Netzwerk

Ein **Netzwerk** ist ein gerichteter und gewichteter Graph mit zwei speziellen Knoten — einer **Quelle** und einer **Senke**.
Unterschied zwischen s, t und dem Rest der Knoten ist, dass s keine eingehenden und t keine ausgehenden Kanten hat.
Das Gewicht c_e der Kante e nennen wir die **Kapazität** der Kante. Diese muss nicht-negativ sein.

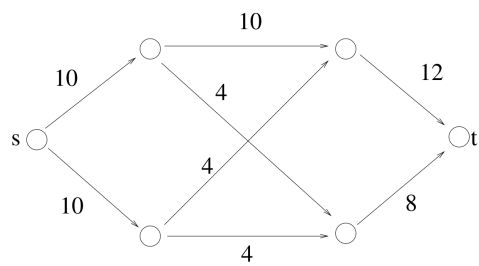

Abbildung 2.1. Beispiel für ein Netzwerk mit Quelle s (*source*) und Senke t (*sink*).

Fluss

Ein **Fluss** in einem Netzwerk ist eine Funktion f_e auf den Kanten des Netzwerks. Ein Fluss hat folgende Eigenschaften:

- $0 \leq f_e \leq c_e$ $(\forall e \in E)$
- $\forall v \in V \setminus \{s, t\}$: Summe eingehender Flüsse = Summe ausgehender Flüsse

Wir definieren außerdem den **Wert** eines Flusses f als

$$\text{val}(f) = \Sigma \text{ von } s \text{ ausgehender Fluss} \equiv \Sigma \text{ zu } t \text{ eingehender Fluss}$$

Ziel ist es in der Regel, einen Fluss in einem festgelegten Netzwerk mit *maximalem Wert* zu finden.

2.2 Cuts

Definition 2.2.1 (Cut). Ein s-t-**Cut** ist eine Partitionierung eines Graphen G in zwei Mengen S und T, sodass $s \in S$ und $t \in T$.
Die **Kapazität** eines Cuts ist

$$\sum \{c_{(u,v)} : u \in S \wedge v \in T\},$$

also die Summe der Kapazitäten aller Kanten, die durch den Cut "durchgeschnitten" werden.

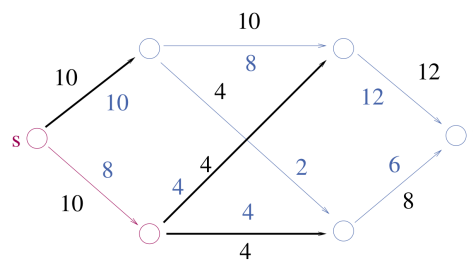

Abbildung 2.2. Die blauen Zahlen stellen einen möglichen Fluss in obigem Netzwerk dar. Die schwarzen Kanten sind ein möglicher Cut. Der Wert dieses Flusses ist 18.

Es gilt folgender extrem praktischer Satz:

Satz 2.2.2 (Fluss-Cut-Dualität). Der Wert eines maximalen s-t-Flusses ist die minimale Kapazität eines s-t-Cuts.

Wie können wir nun diesen Satz nutzen, um einen maximalen Fluss zu finden? Eine Möglichkeit ist *Lineare Programmierung*, aber es gibt bessere Lösungen.

2.3 Pfade augmentieren

Idee ist folgende:

1. Wähle einen s-t-Pfad, der noch Kapazität übrig hat.
2. Sättige die Pfadkante mit der kleinsten Restkapazität.
3. Korrigiere die Kapazitäten aller anderen Kanten mithilfe des *Residualgraphen* und starte wieder von vorne.

Residualgraph

Ist ein Netzwerk $G = (V, E, c)$ mit Fluss f gegeben, so erhalten wir den **Residualgraphen** $G_f = (V, E_f, c^f)$. Dabei gilt für jedes $e \in E$:

$$\begin{cases} e \in E_f \text{ mit } c_e^f = c_e - f(e) & \text{falls } f(e) < c(e) \\ e^{\text{rev}} \in E_f \text{ mit } c_{e^{\text{rev}}}^f = f(e) & \text{falls } f(e) > 0 \end{cases}$$

Dabei ist für $e = (u, v) \in E$ die Kante $e^{\text{rev}} = (v, u)$.
Die Kanten in die "normale" Richtung sind also diejenigen, die noch Restkapazität haben; die neue Kapazität ist die Restkapazität.
Die Kanten in umgekehrte Richtung sind die Kanten, wo der Fluss > 0 ist (das heißt insbesondere, dass auch beide Fälle eintreten können). Das Gewicht dieser Kanten entspricht dem Fluss der Kanten in normale Richtung.

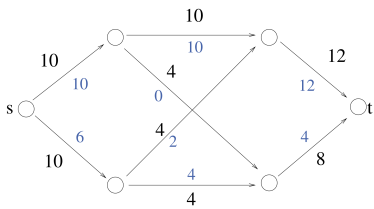

Abbildung 2.3. Nochmal der Fluss von oben.

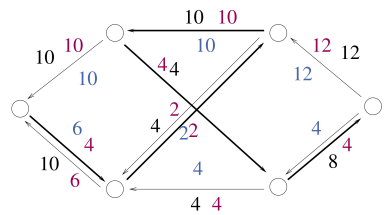

Abbildung 2.4. Der zu nebenstehendem Fluss gehörende Residualgraph. Schwarz die Kapazität, Blau der Fluss, Rot die residuale Kapazität.

Wir suchen jetzt nach einem s-t-Pfad p, sodass jede Kante residuale Kapazität $c_e^f \neq 0$ hat:

```
Δf := min_{e∈p} c_e^f
foreach (u, v) ∈ p do
  if (u, v) ∈ E then f_(u,v) += Δf
  else f(v, u) −= Δf
```

Wir können nun den **Ford-Fulkerson-Algorithmus** implementieren:

```
FFMaxFlow(G = (V, E), s, t, c : E → ℕ) : E → ℕ
  f := 0
  while ∃ path p = (s, ..., t) in G_f do
    augment f along p
  return f
```

Die Zeitkomplexität ist $O(m \cdot \text{val}(f))$, da bei jedem Durchgang der Fluss um mindestens 1 erhöht wird und jedes Mal DFS $O(m)$ ausgeführt werden muss.

Problem — Blocking Flows

Es gibt Netzwerke, in denen der Ford-Fulkerson-Algorithmus tatsächlich die längstmögliche Laufzeit braucht, obwohl die Lösung sehr trivial ist. Grund dafür sind sogenannte **Blocking Flows**.
f_b ist ein *blocking flow* in H, falls für jeden s-t-Pfad p gilt:

$$\exists\, e \in p : f_b(e) = c(e).$$

Das bedeutet, dass jeder s-t-Pfad eine vollständig ausgelastete Kante beinhaltet.

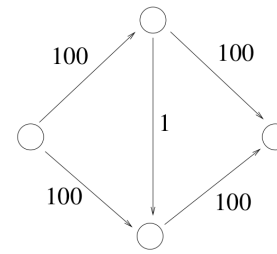

Abbildung 2.5. In diesem Netzwerk könnte der Ford-Fulkerson-Algorithmus stets einen Pfad über die mittlere Kante augmentieren und würde deswegen 200 Schritte brauchen.

2.4 Dinic-Algorithmus

Der **Dinic-Algorithmus** sieht so aus:

```
DinitzMaxFlow(G = (V, E), s, t, c : E → ℕ) : E → ℕ
  f := 0
  while ∃ path p = (s, ..., t) in G_f do
    d = G_f.reverseBFS(t) : V → ℕ
    L_f := (V, {(u, v) ∈ E_f : d(v) = d(u) − 1})   // layer graph
    find blocking flow f_b in L_f
    augment f += f_b
  return f
```

Die rot gekennzeichneten Funktionen/Objekte müssen wir noch klären.
Der Ablauf lässt sich folgendermaßen zusammenfassen:

1. Berechne Distanz-Labels (Abstand zur Senke) für alle Knoten des Graphen.
   (Rückwärtsgerichtete Breitensuche von t aus)
2. Stelle auf Basis der Distanz-Labels den Layer-Graph auf.
3. Suche auf dem Layer-Graph einen blockierenden Fluss zwischen s und t.
4. Führe den gefundenen blockierenden Fluss auf dem Residualgraph aus und aktualisiere diesen entsprechend.
5. Konstruiere den aktualisierten Layer-Graphen und gehe zu Schritt 3.
6. Breche ab, sobald kein weiterer Fluss im Layer-Graph existiert.

Abstandsfunktion

Die Abstandsfunktion d gibt den Abstand eines Knotens zur Senke t an.

Layer-Graph

Den **Layer-Graph** eines Netzwerks erhält man, indem man alle Kanten aus dem Residualgraphen entfernt, die nicht von einer Schicht in die vorherige führen. Es werden also alle Kanten

- innerhalb einer Schicht und
- zwischen Schicht i und $i + k$ ($k \in \mathbb{N}$)

entfernt. Formal ist das

$$L_f = (V, \{(u, v) \in E_f : d(v) = d(u) - 1\}).$$

Die Laufzeit des Dinic-Algorithmus ist damit in $O(mn^2)$.

2.5 Matchings

Ein **Matching** in einem ungerichteten Graphen $G = (V, E)$ ist eine Teilmenge der Kanten, sodass es keine Kanten gibt, die einen gemeinsamen Knoten berühren.
Ein Matching ist *maximal*, wenn es keine Kante gibt, die man aus E zum Matching hinzufügen könnte, ohne die Matching-Anforderung zu verletzen.
Ein Matching hat *maximale Kardinalität*, wenn es kein Matching auf G gibt, dass mehr Knoten abdeckt.
Wir werden Matchings zuerst einmal verwenden, um Flüsse zu berechnen.

2.6 Preflow-Push

Nachteil der pfadaugmentierenden Algorithmen von vorhin ist, dass man Pfade mit hoher Kapazität sehr häufig durchgehen muss aufgrund von späteren Kanten mit geringer Kapazität.
Preflow-Push-Algorithmen lösen dieses Problem.

Definition 2.6.1 (Preflow). Ein **Preflow** ist ein Fluss f, bei dem die Summe der eingehenden Flüsse für einen Knoten höher sein darf als die Summe der ausgehenden Flüsse. Die Differenz dieser Summen nennen wir den **Exzess** des Knotens:

$$\text{excess}(v) := \underbrace{\sum_{(u,v)\in E} f_{(u,v)}}_{\text{inflow}} - \underbrace{\sum_{(v,w)\in E} f_{(v,w)}}_{\text{outflow}} \geq 0.$$

Wir nennen einen Knoten $v \in V \setminus \{s, t\}$ **aktiv**, falls $\text{excess}(v) > 0$.

Wir definieren nun die push-Funktion:

```
PUSH(e = (v, w), δ)
  assert δ > 0
  assert excess(v) ≥ δ
  assert residual capacity of e ≥ δ
  excess(v) −= δ
  excess(w) += δ
  if e is reverse edge then f(reverse(e)) −= δ
  else f(e) += δ
```

Wir nennen einen Push

- **saturierend**, falls $\delta = c_e^f$,
- **nicht-saturierend**, falls $\delta < c_e^f$.

Level-Funktion

Idee ist nun, von s aus Richtung t zu pushen. Dazu benötigen wir eine Approximation $d(v)$ der BFS-Distanz von v zu t in G_f.
Wir können nun den Preflow-Push-Algorithmus implementieren:

```
GENERICPREFLOWPUSH(G = (V, E), f)
  forall e = (s, v) ∈ E do push(e, c(e))
  d(s) := n,  d(v) = 0 for all other nodes
  while ∃ v ∈ V \ {s, t} : excess(v) > 0 do
    if ∃ e = (v, w) ∈ E_f : D(w) < d(v) then  // eligible edge
      choose some δ ≤ min{excess(v), c_e^f}
      push(e, δ)  // no new steep edges
    else d(v)++  // relabel; no new steep edges
```

Dieser generische Algorithmus hat eine Laufzeit von $O(n^2 m)$.

Highest Level Preflow Push

Wählt man immer die aktiven Knoten aus, die $d(v)$ maximieren, so kann man die Laufzeit auf $O(n^2\sqrt{m})$ drücken.
Mit heuristischen Mitteln wie agressivem lokalen Relabeling lässt sich der practical case weiter reduzieren.

3

Externe Algorithmen

Greift man auf sehr große Datenbestände zu, so sind diese in der Regel auf Sekundärspeicher wie Platte oder Band gespeichert, weil diese sehr günstig sind. Problem ist, dass der Zugriff auf den Speicher sehr viel Zeit benötigt.

Externe Algorithmen nehmen an, dass die *interne Arbeit*, also die Rechenarbeit, kostenlos ist. Minimiert werden soll die Anzahl an Zugriffen auf den Sekundärspeicher.

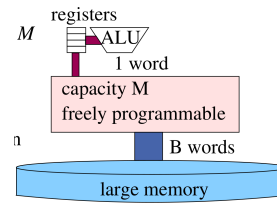

Abbildung 3.1. Sekundärspeichermodell mit schnellem internen Speicher der Größe M und einem beliebig großem externen Speicher, der über einen B Wörter breiten Bus angebunden ist.

3.1 Mehrwegemischen

Als nichttriviales Beispiel werden wir **Mehrwegemischen** betrachten:

```
multiwayMerge(a_1,...,a_k, c:File of Element)
  for i := 1 to k do x_i := a_i.readElement
  for j := 1 to Σ_{i=1}^{k} |a_i| do
    find i ∈ 1...k that minimizes x_i    // no IOs, O(log k) time
    c.writeElement(x_i)
    x_i := a_i.readElement
```

Der Aufwand beträgt

- **I/O**: a_i lesen $\approx \frac{|a_i|}{B}$, c schreiben $\approx \sum_{i=1}^{k} \frac{|a_i|}{B}$
  $\Rightarrow$ Insgesamt $\leq\approx 2\frac{\sum_{i=1}^{k}|a_i|}{B}$
  Bedigungung: brauchen $k+1$ Pufferblöcke.
- **Interne Arbeit** durch Benutzung einer Prioritätsliste:

$$O\left(\log k \sum_{i=1}^{k} |a_i|\right)$$

Sortieren durch Mehrwegemischen

Wir können Mehrwegemischen verwenden, um zu sortieren:

1. Sortiere $\left\lceil \frac{n}{M} \right\rceil$ runs mit je M Elementen.
2. Mische jeweils $\frac{M}{B}$ runs, bis nur noch ein run übrig ist.

Die Anzahl an I/Os setzt sich aus $2\frac{n}{B}$ I/Os für das sortieren, $2\frac{n}{B}$ I/Os für das Mischen und $\left\lceil \log_{M/B} \frac{n}{M} \right\rceil$ Mischphasen zusammen, insgesamt also

$$\text{sort}(n) \cong \frac{2n}{B}\left(1 + \left\lceil \log_{M/B} \frac{n}{M} \right\rceil\right) \text{ I/Os.}$$

Die interne Arbeit setzt sich zusmamen aus

- run formation: $O(n \log M)$,
- Zugriffe auf die Prioritätsliste pro Phase: $O\left(n \log \frac{M}{B}\right)$,
- Anzahl Phasen: $\left\lceil \log_{M/B} \frac{n}{M} \right\rceil$

zusammen, insgesamt

$$O\left(n \log M + n \log \frac{M}{B} \cdot \left\lceil \log_{M/B} \frac{n}{M} \right\rceil\right) = O(n \log n).$$