Kapitel 3

Gruppen und Rollen

In diesem Kapitel liegt unser Augenmerk nicht auf der (strukturellen) Wichtigkeit einzelner Knoten oder Kanten, sondern auf der Zugehörigkeit von Knoten zu (strukturellen) Teilmengen. Diese können entweder durch relativ enge Beziehungen untereinander, d.h. größere Dichte der Kanten innerhalb des induzierten Teilgraphen als nach außen, oder durch Ähnlichkeit der Nachbarschaften, d.h. gleichartige Beziehungen zu gleichartigen anderen, gebildet werden. Im ersten Fall sprechen wir von *Gruppen*, im zweiten von *Rollen*.

3.1 Dichte Teilgraphen

Ein Graph ist *dicht*, wenn viele der möglichen Kanten auch tatsächlich vorhanden sind.

3.1 Definition (Dichte)
Die Dichte $\mathbb{D}(G)$ eines Multigraphen $G = (V, E)$ ist definiert durch

$$\mathbb{D}(G) = \frac{|\{(u,w) \in V \times V : (u,w) \in E, u \neq w\}|}{n(n-1)} .$$

Ein Multigraph hat damit die maximal mögliche Dichte 1, wenn jeder Knoten mit jedem anderen verbunden ist. Der schlichte ungerichtete Graph $K_n = (V, E)$ mit Knotenmenge $\{1, \ldots, n\}$ und Kantenmenge $E = \{\{v, w\} : v \neq w \in V\}$ heißt auch der *vollständige Graph* auf n Knoten.

Kommt der vollständige Graph als Teilgraph eines anderen vor, können die beteiligten Knoten als eng zusammenhaltende Gruppe angesehen werden.

3.2 Definition (Clique; Luce and Perry 1949)
In einem schlichten ungerichteten Graph $G = (V, E)$ heißt eine Knotenteilmenge $C \subseteq V$ Clique (der Größe $|C|$ bzw. $|C|$-Clique), falls der von C knoteninduzierte Teilgraph $G[C]$ vollständig ist. Eine Clique heißt maximal, wenn sie inklusionsmaximal ist. Das größte $k \in \{0, \ldots, n\}$, für das G eine k-Clique enthält, heißt Cliquenzahl $\omega(G) = k$ von G.

3.3 Satz
Die Bestimmung von $\omega(G)$ ist $\mathcal{NP}$-schwer.

■ **Beweis:** Wir zeigen, dass das Entscheidungsproblem CLIQUE (Gegeben ein Graph $G = (V, E)$ und ein $k \in \mathbb{N}$, gibt es in G eine Clique der Größe mindestens k?) $\mathcal{NP}$-vollständig ist.

CLIQUE ist klarerweise in $\mathcal{NP}$. Für die Vollständigkeit geben wir eine polynomiale Reduktion des $\mathcal{NP}$-vollständigen Problems 3SAT auf CLIQUE an. Sei als 3SAT-Instanz eine Formel $\phi = C_1 \wedge \ldots \wedge C_r$ in konjunktiver Normalform mit jeweils drei Literalen pro Klausel gegeben. Die Klauseln seien $C_i = y_{i,1} \vee y_{i,2} \vee y_{i,3}$ mit $y_{i,j} \in \{x_1, \ldots, x_n, \overline{x_1}, \ldots, \overline{x_n}\}$ für alle $i = 1, \ldots, r$. Wir konstruieren daraus einen ungerichteten Graphen $G(\phi) = (V, E)$ mit

$$\begin{aligned} V &= \{y_{i,j} \, : \, 1 \leq i \leq r, \, j = 1, 2, 3\} \\ E &= \{\{y_{i_1,j_1}, y_{i_2,j_2}\} \, ; \, 1 \leq i_1 < i_2 \leq r, \, y_{i_1,j_1} \neq \overline{y_{i_2,j_2}}\} \end{aligned}$$

und zeigen, dass ϕ genau dann erfüllbar ist, wenn es in $G(\phi)$ eine Clique der Größe r gibt.

Ist ϕ erfüllbar, dann lässt sich in jeder Klausel ein erfülltes Literal wählen. Da alle erfüllt sind, gilt für keine zwei, dass das eine Negation des anderen ist. Da sie außerdem noch aus verschiedenen Klauseln stammen, sind die zugehörigen Knoten in $G(\phi)$ vollständig verbunden.

Enthält umgekehrt $G(\phi)$ eine Clique mit r Knoten, dann ist in jeder Klausel mindestens ein zugehöriges Literal vertreten. Die zugehörigen Variablen können alle so belegt werden, dass das Literal erfüllt ist, denn weil je zwei Knoten der Clique adjazent sind, widersprechen sich die Literale nicht. □

3.4 Bemerkung

Die Situation ist algorithmisch sogar noch unerfreulicher, weil es im Fall $\mathcal{P} \neq \mathcal{NP}$ auch keinen polynomialen Algorithmus gibt, mit dem die Cliquenzahl bis auf einen konstanten Faktor approximiert werden kann.

Das heißt aber natürlich nicht, dass sich die Cliquen eines Graphen nicht bestimmen ließen. Wir werden als nächstes sogar alle maximalen Cliquen aufzählen.

Ein *Aufzählungsalgorithmus* gibt alle Objekte einer Menge in irgendeiner Reihenfolge aus. Er arbeitet mit *Verzögerung* $\mathcal{O}(f(n))$, falls die Laufzeit vor Ausgabe des ersten Elements, zwischen je zwei Elementen und nach dem letzten Element jeweils in $\mathcal{O}(f(n))$ ist.

3.5 Satz

Algorithmus 19 zählt alle maximalen Cliquen eines ungerichten Graphen mit Verzögerung $\mathcal{O}(nm)$ und $\mathcal{O}(n+m)$ Platz auf.

■ **Beweis:** Der Algorithmus ist korrekt, denn er durchläuft einen binären Baum, dessen Knoten der Tiefe i gerade die maximalen Cliquen im von den Knoten $\{1, \ldots, i\}$ induzierten Graphen $G[\{1, \ldots, i\}]$ repräsentieren, und gibt jeweils die maximalen Cliquen in $G = G[1, \ldots, n]$ aus. Das sieht man wie folgt.

Aus einer maximalen Clique C in $G[1, \ldots, i-1]$ erhält man auf folgende Weisen eine maximale Clique in $G[1, \ldots, i]$:

1. Fall: $(C \subseteq N(i))$

Wenn alle Knoten in C zu i adjazent sind, dann ist $C \cup \{i\}$ die einzige maximale Clique in $G[1, \ldots, i]$, die C enthält.

2. Fall: $(C \not\subseteq N(i))$

Dann ist zumindest C auch maximale Clique in $G[1, \ldots, i]$. Die größte i enthaltende Clique, die in $G[1, \ldots, i]$ aus C entsteht, ist $K = (C \cap N(i)) \cup \{i\}$. Allerdings ist K nicht notwendig eine maximale Clique, und K könnte auf die gleiche Weise auch aus anderen maximalen Cliquen C' von $G[1, \ldots, i-1]$ entstehen. Im Algorithmus wird K daher nur dann als Nachfolger von C angesehen, wenn C die lexikographisch erste maximale Clique von $G[1, \ldots, i-1]$ ist, die durch Schnitt mit der Nachbarschaft von i zu einer maximalen Clique in $G[1, \ldots, i]$ führt.

Algorithmus 19: Aufzählung aller maximalen Cliquen (Tsukiyama, Ide, Ariyoshi und Shirakawa, 1977)

Eingabe : ungerichteter Graph $G = (V = \{1, \dots, n\}, E)$
Ausgabe: alle in G enthaltenen maximalen Cliquen

```
maximal(vertex set K, vertex i) begin
    // -- Ist K maximale Clique in G[1...i] ?
    for j = 1, ..., i do
        if j ∉ K and K ⊆ N(j) then return false
    return true
end

parent(vertex set K, vertex i) begin
    // -- lexikographisch erste maximale Clique
    // -- in G[1...i-1], die K-i enthält
    P ← K \ {i}
    for j = 1, ..., i − 1 do
        if P ⊆ N(j) then P ← P ∪ {j}
    return P
end

insert(vertex i) begin
    if i = n then
        gib maximale Clique C aus
    else
        if C ⊆ N(i) then
            // -- einziger Nachfolger
            C ← C ∪ {i};   insert(i + 1);   C ← C \ {i}
        else
            // -- linker Nachfolger
            insert(i + 1)

            // -- rechter Nachfolger
            K ← (C ∩ N(i)) ∪ {i}
            if maximal(K, i) and C = parent(K, i) then
                C ← K;   insert(i + 1);   C ← parent(C, i)
end

begin
    C ← ∅
    insert(1)
end
```

Für die Laufzeit beachte, dass der Test auf Maximalität in *maximal*, die Bestimmung des Elternknotens in *parent* und die Tests und Schnitte mit der Nachbarschaft von i in $\mathcal{O}(n+m)$ Zeit möglich sind. Da die Menge C an einem inneren Knoten des impliziten Baumes nie leer ist und jeder innere Knoten mindestens einen Nachfolger hat, ist die Länge der durchlaufenen Wege von der Wurzel zum ersten Blatt, zwischen zwei Blättern und nach dem letzten Blatt jeweils durch $2n-1$ beschränkt. Da auf den Zusammenhangskomponenten von G getrennt gerechnet werden kann, ergibt sich insgesamt eine Verzögerung von $\mathcal{O}(nm)$ für den Aufzählungsalgorithmus.

Der Speicherplatzbedarf ist linear, weil nicht einmal die Menge C zwischengespeichert, sondern nach Beendigung eines rekursiven Aufrufs wiederhergestellt wird. □

3.6 Bemerkung
Algorithmus 19 gibt die Cliquen in einer unkontrollierten Reihenfolge aus. Es gibt einen anderen Aufzählungsalgorithmus, der die Cliquen bei ebenfalls $\mathcal{O}(n^3)$ Verzögerung in lexikographischer Reihenfolge aufzählt. Dieser braucht im schlechtesten Fall allerdings exponentiell viel Speicherplatz.

Kerne

Es sind zahlreiche Abschwächung der Forderung nach vollständiger Verbundenheit definiert worden. Eine wegen ihrer algorithmisch leichten Bestimmung interessante verlangt lediglich eine feste Mindestzahl von Nachbarn.

3.7 Definition (Kern)
Der k-Kern eines schlichten ungerichteten Graphen $G=(V,E)$ ist der inklusionsmaximale Teilgraph $Core_k(G) \subseteq G$ mit $\delta(Core_k(G)) \geq k$, d.h. in dem jeder Knoten mindestens Grad k hat. Der Kern, $Core(G)$, von G ist der nichtleere k-Kern mit maximalem k.

3.8 Lemma
Für alle $k,l \in \mathbb{N}_0$ ist $Core_k(G)$ eindeutig und

$$k \geq l \implies Core_k(G) \subseteq Core_l(G) \ .$$

■ **Beweis:** Gäbe es mehr als einen k-Kern, so wäre deren Vereinigung wieder ein k-Kern, da kein Knotengrad kleiner wird. Aus der Inklusionsmaximialität folgt damit die Eindeutigkeit.

Für $k \geq l$ erfüllen alle Knoten des k-Kerns die Gradbedingung an den l-Kern und sind damit höchstens weniger. □

3.9 Definition (Kernzahl)
Die Kernzahl $core(v)$ eines Knotens $v \in V$ ist das größte k, für das der Knoten im k-Kern enthalten ist.

3.10 Lemma

$$core(v) \leq |\{w \in N(v) : core(w) \geq core(v)\}|$$

■ **Beweis:** Kein Knoten $w \in N(v)$ mit $core(w) < core(v)$ liegt im $core(v)$-Kern. Es muss daher mindestens $core(v)$ viele Nachbarn von v mit Kernzahl nicht kleiner als $core(v)$ geben. □

Ein Graph G ist identisch mit seinem $\delta(G)$-Kern, und die Kernzahl eines Knotens mit minimalem Grad ist eben dieser Grad. Um den $(\delta(G)+1)$-Kern zu bestimmen, können daher zunächst alle Knoten mit minimalem Grad entfernt werden. Ein Knoten, der dann auch nur noch Grad höchstens $\delta(G)$ hat, kann ebenfalls nicht im $(\delta(G) + 1)$-Kern liegen und entfernt werden. Dieser Prozess kann fortgesetzt werden, bis alle Knoten enfernt sind oder mindestens Grad $\delta(G) + 1$ haben. Analog werden die Knoten mit höherer Kernzahl bestimmt.

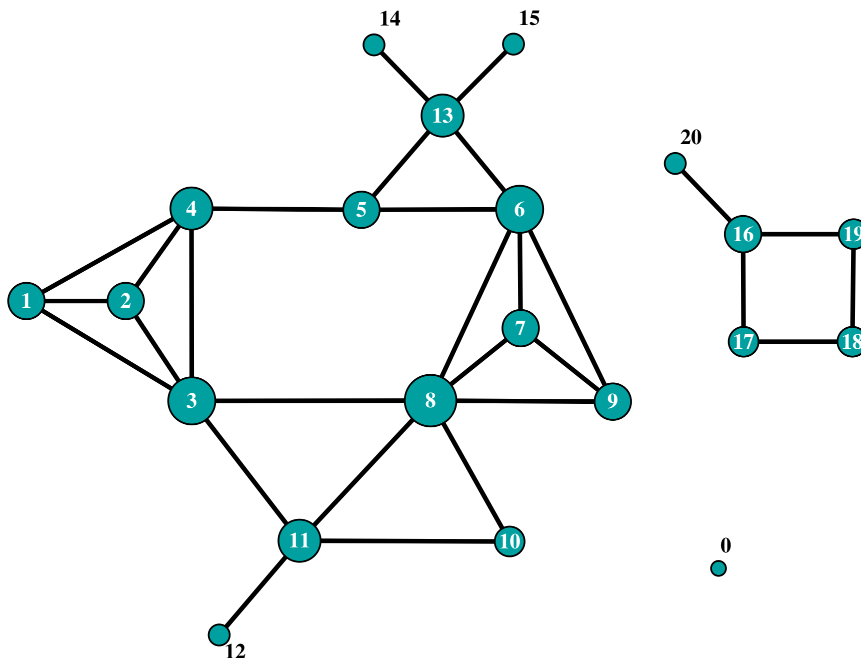
Algorithmus 20: Kernzahlen (Batagelj und Zaveršnik 1999)

```
Eingabe : schlichter ungerichteter Graph G = (V, E)
Daten   : Array vert (Knoten geordnet nach Grad)
          Knotenarray pos (Position in Array)
          Array bucket (Δ(G) + 1 Behälter, initialisiert mit 0)
Ausgabe: Knotenarray c (Kernzahlen)

foreach v ∈ V do
    c[v] ← d_G(v)
    inkrementiere bucket[d_G(v)]
first ← 0
for d = 0, ..., Δ(G) do
    size ← bucket[d]
    bucket[d] ← first
    first ← first + size
foreach v ∈ V do
    pos[v] ← bucket[d_G(v)]
    vert[pos[v]] ← v
    inkrementiere bucket[d_G(v)]
for d = Δ(G), ..., 1 do bucket[d] ← bucket[d − 1]
bucket[0] ← 0

for i = 0, ..., n − 1 do
    v ← vert[i]
    foreach w ∈ N_G(v) do
        if c[w] > c[v] then
            u ← vert[bucket[c[w]]]
            if u ≠ w then
                pos[u] ← pos[w]
                pos[w] ← bucket[c[w]]
                vert[pos[u]] ← u
                vert[pos[w]] ← w
            inkrementiere bucket[c[w]]
            dekrementiere c[w]
```

3.11 Beispiel



c:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	3	3	5	4	3	5	3	6	3	2	4	1	4	1	1	3	2	2	2	1

bucket: (nach 1. Schleife)

0	1	2	3	4	5	6
1	4	4	6	3	2	1

bucket: (nach 2. und wieder nach 4. Schleife)

0	1	2	3	4	5	6
0	1	5	9	15	18	20

bucket: (nach 3. Schleife)

0	1	2	3	4	5	6
1	5	9	15	18	20	21

vert: (nach 4. Schleife, Doppelstriche markieren Behältergrenzen)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	12	14	15	20	10	17	18	19	1	2	5	7	9	16	4	11	13	3	6	8

Im zweiten Teil werden nun alle Knoten in der Reihenfolge ihrer Kernzahlen abgearbeitet. In den Tabellen sind $vert(i)$ *und* $c(vert(i))$ *jeweils nach dem* i*-ten Durchlauf aufgeführt. Die Behältergrenzen (bucket) sind durch horizontale Trennlininen gekennzeichnet.*

i	*vert*	*c*
→ 0	**0**	**0**
1	12	1
2	14	1
3	15	1
4	20	1
5	10	2
6	17	2
7	18	2
8	19	2
9	1	3
10	2	3
11	5	3
12	7	3
13	9	3
14	16	3
15	4	4
16	11	4
17	13	4
18	3	5
19	6	5
20	8	6

i	*vert*	*c*
0	**0**	**0**
→ 1	**12**	**1**
2	14	1
3	15	1
4	20	1
5	10	2
6	17	2
7	18	2
8	19	2
9	1	3
10	2	3
11	5	3
12	7	3
13	9	3
14	16	3
15	11	3
16	13	4
17	4	4
18	3	5
19	6	5
20	8	6

i	*vert*	*c*
0	**0**	**0**
1	**12**	**1**
→ 2	**14**	**1**
3	15	1
4	20	1
5	10	2
6	17	2
7	18	2
8	19	2
9	1	3
10	2	3
11	5	3
12	7	3
13	9	3
14	16	3
15	11	3
16	13	3
17	4	4
18	3	5
19	6	5
20	8	6

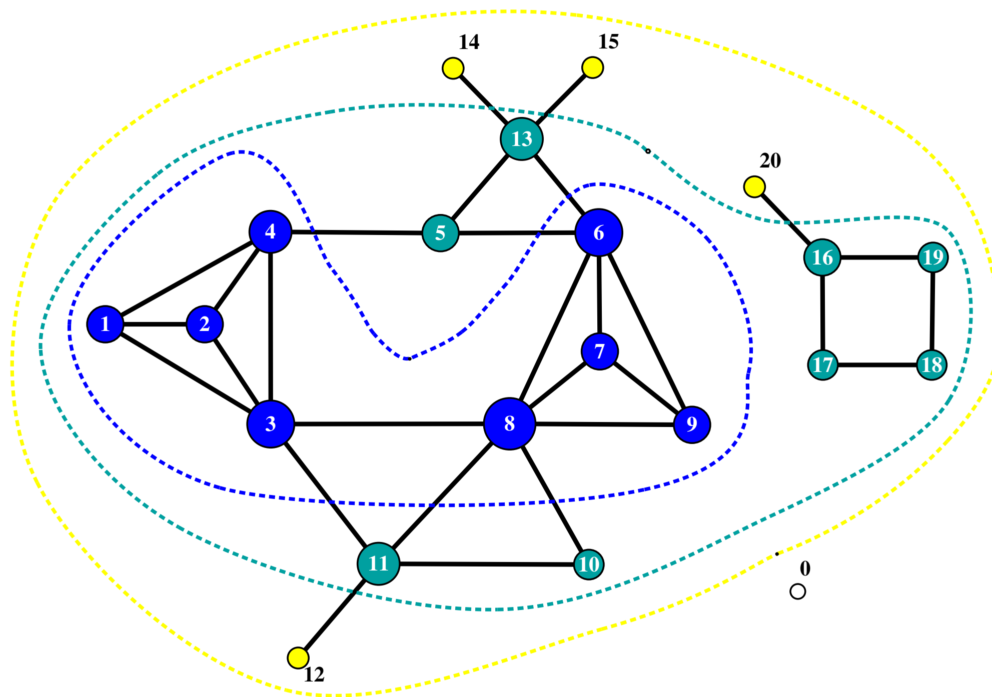
i	*vert*	*c*
0	**0**	**0**
1	**12**	**1**
2	**14**	**1**
→ 3	**15**	**1**
4	20	1
5	10	2
6	17	2
7	18	2
8	19	2
9	13	2
10	2	3
11	5	3
12	7	3
13	9	3
14	16	3
15	11	3
16	1	3
17	4	4
18	3	5
19	6	5
20	8	6

i	*vert*	*c*
0	**0**	**0**
1	**12**	**1**
2	**14**	**1**
3	**15**	**1**
→ 4	**20**	**1**
5	10	2
6	17	2
7	18	2
8	19	2
9	13	2
10	16	2
11	5	3
12	7	3
13	9	3
14	2	3
15	11	3
16	1	3
17	4	4
18	3	5
19	6	5
20	8	6

i	*vert*	*c*
0	**0**	**0**
1	**12**	**1**
2	**14**	**1**
3	**15**	**1**
4	**20**	**1**
→ 5	**10**	**2**
6	17	2
7	18	2
8	19	2
9	13	2
10	16	2
11	11	2
12	7	3
13	9	3
14	2	3
15	5	3
16	1	3
17	4	4
18	3	5
19	6	5
20	8	5

...

i	*vert*	*c*
0	**0**	**0**
1	**12**	**1**
2	**14**	**1**
3	**15**	**1**
4	**20**	**1**
5	**10**	**2**
6	**17**	**2**
7	**18**	**2**
8	**19**	**2**
9	**13**	**2**
10	**16**	**2**
11	**11**	**2**
12	**5**	**2**
13	**9**	**3**
14	**2**	**3**
15	**7**	**3**
16	**1**	**3**
17	**4**	**3**
18	**6**	**3**
19	**8**	**3**
→ 20	**3**	**3**



3.12 Bemerkung

k-Kerne sind nicht notwendig zusammenhängend.

3.13 Satz

Der Algorithmus bestimmt die Kernzahlen in Zeit $\mathcal{O}(n+m)$.

■ **Beweis:** Im ersten Teil werden die Knoten mittels Bucket-Sort entsprechend ihrer Knotengrade sortiert. Das geschieht wie folgt:

1. Nach Durchlauf der ersten Schleife enthält das Ausgabearray c als obere Schranke für die Kernzahl den Knotengrad, und das Array $bucket(0, \ldots, \Delta(G))$ für jeden möglichen Knotengrad die Anzahl der Knoten dieses Grades.

2. Das Array $vert(0, \ldots, n-1)$ soll später die sortierte Knotenfolge enthalten. In der zweiten Schleife werden dazu die jeweils ersten Positionen von Knoten eines jeden Grades durch Präfixsummenbildung bestimmt.

3. In der dritten Schleife wird jeder Knoten an den Beginn des noch freien Abschnitts vor der ersten Position von Knoten höheren Grades gesetzt. Die Knoten sind damit sortiert, aber die ersten Positionen sind nun die des jeweils nächsten Behälters.

4. Diese Verschiebung wird in der vierten Schleife rückgängig gemacht.

Für alle $v \in V$ gilt dann also $c(v) = d_G(v)$, $v = vert(pos(v))$ und mit der Vereinbarung $bucket(\Delta(G) + 1) = n$ auch

$$bucket(d_G(v)) \leq pos(v) < bucket(d_G(v) + 1) \ .$$

Sei $v_0, \ldots, v_{n-1}$ die Reihenfolge, in der die Knoten im zweiten Teil betrachtet werden (da nur hinter dem Laufindex umsortiert wird, wird kein Knoten zweimal betrachtet). Ferner seien nach Ausführung des i-ten Schleifendurchlaufs $b_{i+1} = bucket(c(v_i) + 1)$, wobei wir wieder $bucket(\Delta(G) + 1) = n$ setzen, und H_i der von den Knoten mit $pos(v) > i$ induzierte Graph. Dann gelten die folgenden Invarianten.

1. Für alle $v \in V$ gilt $v = vert(pos(v))$ und $bucket(c(v)) \leq pos(v) < bucket(c(v) + 1)$.

2. Für alle $w \in V$ mit $pos(w) \geq b_{i+1}$ gilt $c(w) = d_{H_i}(w)$.

3. Für alle $v \in V$ mit $pos(v) < b_{i+1}$ gilt $c(v) = core(v)$.

Für die erste Invariante ergeben sich nur dann Änderungen, wenn zwei Knoten u und w vertauscht werden. Während der erste Teil der Aussage explizit wiederhergestellt wird, bleibt der zweite erhalten, weil w mit der reduzierten Abschätzung der Kernzahl durch den Tausch mit dem ersten Element des Behälters und Verschiebung von dessen Anfang um eins nach hinten in den vorhergehenden Behälter wechselt.

Die zweite Invariante gilt, weil alle Knoten w ab b_{i+1} nach der ersten Invariante $c(w) > c(v_i)$ haben und daher bei Entfernen von v_i ihr Grad richtig angepasst wird. Die dritte Invariante folgt aus den ersten beiden und am Ende der Schleife folgt aus ihr die Korrektheit des Algorithmus'.

Die ersten vier Schleifen werden höchstens n mal durchlaufen und jeder Durchlauf benötigt konstant viel Zeit. In der letzten Schleife werden für jeden Knoten alle inzidenten Kanten betrachtet. Da alle anderen Operationen konstant viel Zeit benötigen, folgt mit dem Handschlaglemma, dass die Laufzeit in $\mathcal{O}(n + m)$ ist. □

3.2 Rollen

Anstelle von Zusammengehörigkeit werden wir nun Ähnlichkeit als Partitionskriterium verwenden.

3.14 Definition (Strukturelle Äquivalenz)
Eine Äquivalenzrelation $\sim \subset V \times V$ *auf der Knotenmenge eines Multigraphen* $G = (V, E)$ *heißt* <u>*strukturelle Äquivalenz*</u>*, falls für alle* $v \sim w$ *gilt*

$$\begin{array}{rccc} & (x,v) \in_k E & \Longrightarrow & (x,w) \in_k E \\ \textit{und} & (v,x) \in_k E & \Longrightarrow & (w,x) \in_k E\ . \end{array}$$

Im Allgemeinen ist diese Definition viel zu streng, um nützlich zu sein. Von den zahlreichen Abschwächungen des strukturellen Äquivalenzbegriffs betrachten wir hier nur den am weitesten verbreiteten.

3.15 Definition (Reguläre Äquivalenz)
Eine Äquivalenzrelation $\approx \subset V \times V$ *auf der Knotenmenge eines Multigraphen* $G = (V, E)$ *heißt* <u>*(reguläre) Äquivalenz*</u>*, falls für alle* $v \approx w$ *gilt*

$$\begin{array}{rccc} & x \in N^-(v) & \Longrightarrow & \textit{es ex. ein } y \in N^-(w) \textit{ mit } x \approx y \\ \textit{und} & x \in N^+(v) & \Longrightarrow & \textit{es ex. ein } y \in N^+(w) \textit{ mit } x \approx y\ . \end{array}$$

Eine reguläre Äquivalenz definiert eine Partition der Knotenmenge in Teilmengen aus äquivalenten Knoten. Diese Teilmengen können wir als *Rollen* auffassen. Wir zeigen zunächst, dass die Menge der möglichen Rollenzuweisungen eine wohlgeordnete Struktur hat.

Erinnerung: Eine binäre Relation $\preceq \subseteq P \times P$ heißt *partielle Ordnung*, falls sie reflexiv, antisymmetrisch und transitiv ist. Das Paar $(P, \preceq)$ ist dann eine *partiell geordnete Menge*. Zu $X \subseteq P$ ist $y \in P$ eine *obere* (*untere*) *Schranke*, falls $x \preceq y$ ($y \preceq x$) für alle $x \in X$. Eine obere (untere) Schranke $y \in P$ von $X \subseteq P$ heißt *Supremum* (*Infimum*), falls für alle oberen (unteren) Schranken $z \in P$ von X gilt $y \preceq z$ ($z \preceq y$). Falls ein Supremum oder Infimum existiert, so ist es eindeutig. Eine partiell geordnete Menge $(P, \preceq)$ ist ein *Verband*, falls Infimum und Supremum für alle $X \subseteq P$ existieren.

Für den Nachweis, dass eine partiell geordnete Menge ein Verband ist, reicht es aus, für alle Teilmengen die Existenz des Supremums zu zeigen.

3.16 Lemma
Eine partiell geordnete Menge $(P, \preceq)$ ist ein Verband, falls für jedes $X \subseteq P$ das Supremum existiert.

■ **Beweis:** Zu einer beliebigen Teilmenge $X \subseteq P$ erhält man das Infimum als Supremum von $\{y \in P \,:\, \text{für alle } x \in X \text{ gilt } y \preceq x\}$. □

3.17 Satz
Die regulären Äquivalenzen eines Multigraphen bilden einen Verband.

■ **Beweis:** Sei $\mathcal{R}$ die Menge der regulären Äquivalenzen eines Multigraphen $G = (V, E)$. Wir definieren auf $\mathcal{R}$ eine partielle Ordnung $\preceq$ durch $\approx_1 \preceq \approx_2 \iff \approx_1 \subseteq \approx_2$. Damit bedeutet $\approx_1 \preceq \approx_2$, dass $\approx_1$ *feiner* als $\approx_2$ und $\approx_2$ *gröber* als $\approx_1$ ist. Wir zeigen, dass $(\mathcal{R}, \preceq)$ ein Verband ist.

Wegen des vorstehenden Lemmas und weil mit der Knotenmenge auch die Menge der regulären Äquivalenzen endlich ist, reicht es zu zeigen, dass für je zwei reguläre Äquivalenzen $\approx_1, \approx_2 \in \mathcal{R}$ das Supremum existiert. Sei $\approx$ die transitive Hülle der Vereinigung von $\approx_1$ und $\approx_2$. Da es sicher keine kleinere Menge gibt, die als Supremum in Frage kommt, bleibt zu zeigen, dass $\approx$ regulär ist.

Seien dazu $u \approx w$ und $x \in N^-(u)$ für beliebige $u, v, x \in V$. Wir müssen ein $z \in N^-(w)$ mit $x \approx z$ finden. Nach Konstruktion von $\approx$ (Vereinigung und transitive Hülle) existiert eine Folge von Knoten $u = v_1, \ldots, v_k = w \in V$ so, dass für alle $i = 1, \ldots, k-1$ gilt $v_i \approx_j v_{i+1}$ für jeweils mindestens ein $j \in \{1, 2\}$. Weil $\approx_1$ und $\approx_2$ regulär sind, gibt es dann aber auch eine Folge $x = z_1, \ldots, z_k$ mit $z_i \in N^-(v_i)$ und $z_i \approx_j z_{i+1}$ für alle $i = 1, \ldots, k$ und jeweils mindestens ein $j \in \{1, 2\}$. Das gesuchte z ist z_k. Analog findet man zu $x \in N^+(u)$ ein $z \in N^+(w)$ mit $x \approx z$. □

Da $(\mathcal{R}, \preceq)$ ein Verband ist, existiert zu jeder Partition der Knoten eine gröbste reguläre Äquivalenz, die diese verfeinert.

3.18 Definition
Das reguläre Innere einer Partition ist das Supremum aller regulären Äquivalenzen, welche die Partition verfeinern.

Ein naheliegender Ansatz zur Berechnung des regulären Inneren besteht darin, in der gegebenen Partition alle Paare von Knoten derselben Teilmenge daraufhin zu testen, ob sie regulär äquivalent sind, und gegebenfalls die

Menge zu teilen. Das Verfahren wird auch CATREGE genannt und häufig in Programmen für die Analyse von sozialen Netzwerken verwendet.

Algorithmus 21: Reguläres Inneres (Borgatti und Everett 1993)

Eingabe : Multigraph $G = (V = \{1, \dots, n\}, E)$
Partition $P : V \to \{1, \dots, n\}$ von V
Ausgabe: reguläres Inneres von P

```
repeat
    for v ∈ V do Q[v] ← v
    stable ← true
    for v = 2, ..., n do
        for u = 1 ..., v − 1 do
            if P[u] = P[v] then
                if {P[x] : x ∈ N⁻(u)} = {P[x] : x ∈ N⁻(v)}
                and {P[x] : x ∈ N⁺(u)} = {P[x] : x ∈ N⁺(v)} then
                    Q[v] ← Q[u]
                else
                    stable ← false
    P ← Q
until stable
```

3.19 Satz
Algorithmus 21 berechnet das reguläre Innere der Eingabepartition in $\mathcal{O}(n^3)$ Zeit und $\mathcal{O}(n + m)$ Platz.

■ **Beweis:** Im Algorithmus werden alle Paare von Knoten daraufhin überprüft, ob sie bisher als äquivalent eingestuft waren und bezüglich der aktuellen Partition die Bedingung für reguläre Äquivalenz erfüllen.

Falls ja, werden die Partitionsmengen in P jeweils durch den Knoten mit dem kleinsten Index repräsentiert. Andernfalls gelten die beiden Knoten in der nächsten Runde nicht mehr als äquivalent. Da die Partition auf diese Weise immer nur verfeinert wird, geraten Knoten nur dann in verschiedene Äquivalenzklassen, wenn sie in keiner Verfeinerung der Eingabepartition regulär äquivalent sind.

Für die Laufzeit beachte, dass die Anzahl der Äquivalenzklassen höchstens $n-1$ mal größer werden kann. In jedem Durchlauf der äußeren Schleife werden

alle $\binom{n}{2}$ Knotenpaare betrachtet. Der Vergleich der Nachbarschaften erfordert den Durchlauf durch alle Adjazenzlisten und kann in amortisiert $\mathcal{O}(m)$ realisiert werden, sodass wir insgesamt eine Laufzeit von $\mathcal{O}(n^3)$ erhalten. □

3.20 Bemerkung

Mit einem effizienteren Verfahren von Tarjan und Paige (1987), das vor allem im Bereich Programmanalyse bekannt ist, kann das reguläre Innere in Laufzeit $\mathcal{O}(m \log n)$ berechnet werden.