Computeralgebra (ABV 2017)
Algebraische Graphentheorie

David Ploog
Johanna Steinmeyer

Literatur. Als Quelle nur Brouwer–Haemers benutzt, die anderen beiden Texte sind aber einfacher.

A.E. Brouwers, W.H. Haemers: *Spectra of Graphs*, Springer Universitext 2012.
N. Biggs: *Algebraic Graph Theory*, Cambridge University Press 1974/1993.
R.B. Bapat: *Graphs and Matrices*, Springer Universitext 2014.

Graphentheorie

Definition: *Graph* $G = (V(G), E(G))$.

Außermathematische Motivation: elektrische Schaltkreise (Kirchhoff), Transportnetzwerke (zum Beispiel S-Bahnnetz), Programmablaufpläne, Informationsnetze (Shannon), künstliche Intelligenz (neuronale Netze), theoretische Physik (Feynman-Diagramme), chemische Moleküle.

Mathematische Motivation: Singularitäten (Coxeter-Dynkin-Diagramme), Algebra (Köcher-Darstellungen), Topologie (Graphen sind 1-Skelette von CW-Komplexen; Triangulierungen von Räumen), Graphentheorie als Teil der diskreten Mathematik.

Wichtige Beispielklassen: Pfadgraphen A_n, Zykelgraphen Z_n, vollständige Graphen K_n, vollständig bipartite Graphen $K_{n,m}$.

Aufgabe 1. Implementiere einen Datentyp **Graph** in SINGULAR.
Es gibt mehrere Lösungen, zum Beispiel

- `newstruct("Graph", "list vertices, list edges")`
- `newstruct("Graph", "int vertices, list edges")`
- `newstruct("Graph", "matrix edges")` als $2 \times n$-Matrix der Kanten.

Aufgabe 2. Implementiere eine Prozedur, die testet ob ein `Graph G` tatsächlich einen Graphen definiert. Implementiere eine Prozedur, die doppelte Ecken und Kanten in `G` entfernt.

Aufgabe 3. Implementiere Beispielklassen von Graphen: `Pfad(n), Zykel(n), Komplett(n), Bipartit(n,m)`.

Graphenkonstruktionen: *Komplementärgraph* $\overline{G}$ und *Kantengraph* $L(G)$.

Beispiel: Petersen-Graph $\overline{L(K_5)}$.

Aufgabe 4. Implementiere `Komplement(G)` und `Kantengraph(G)`.

Definition: *Wege, zusammenhängend* und *Zusammenhangskomponenten*;
Eckenabstand $d(x, y)$ für $x, y \in V(G)$;
Durchmesser $\operatorname{diam}(G) = \sup_{x,y} d(x, y)$.

Aufgabe 5. Implementiere `Nachbarn(G,x), Distanzen(G,x), Durchmesser(G)` mittels Dijkstra-Algorithmus.

Matrizen und Eigenwerte zu Graphen

Definition: *Adjazenz-Matrix* A, *Gradmatrix* D und *Laplace-Matrix* $L = D - A$.

Bemerkung: offensichtliche Verallgemeinerungen für Graphen mit Schleifen oder Mehrfachkanten und für gerichtete Graphen.

Aufgabe 6. Implementiere `AdjazenzMatrix(G)`, `LaplaceMatrix(G)`.

Wiederholung: Eigenwerte und Eigenvektoren. Selbstadjungierte Endomorphismen (zum Beispiel symmetrische reelle Matrizen) haben reelle Eigenwerte.

Weil A und L symmetrisch sind, haben $A(G)$ und $L(G)$ reelle Eigenwerte. Wir bezeichnen sie mit $\alpha_1 \geq \alpha_2 \geq \ldots \geq \alpha_n$ und $\lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$.

Aufgabe 7. Implementiere den maximalen Adjazenz-Eigenwert `AmaxEW(G)`.
Finde Graphen mit maximalem Adjazenz-Eigenwert < 2.
(Antwort: Die zusammenhängenden Graphen mit dieser Eigenschaft sind genau die ADE-Diagramme. Die zusammenhängenden Graphen mit maximalem Adjazenz-Eigenwert 2 sind die erweiterten Dynkin-Diagramme $\tilde{A}_n = Z_{n+1}, \tilde{D}_n, \tilde{E}_6, \tilde{E}_7, \tilde{E}_8$.)

Definition: *k-reguläre* Graphen.
Beispiele: Z_n 2-regulär, K_n $(n-1)$-regulär, Petersen-Graph 3-regulär.

Lemma: $\alpha_1(G) \leq d_{\max}(G)$, *der höchste Eckengrad von* G.

Beweis: Sei $v = (v_1, \ldots, v_n)^t$ ein Vektor mit $Av = \alpha_1 v$ und sei v_i der maximale Eintrag von v. Wir können $v_i > 0$ annehmen; sonst v durch $-v$ ersetzen. Dann ist $\alpha_1 v_i = (Av)_i = \sum_j A_{ij} v_j \leq \sum_j A_{ij} v_i = v_i \sum_j A_{ij} = v_i d_i \leq v_i d_{\max}$.

Aufgabe 8. Was ist der maximale Adjazenz-Eigenwert eines regulären Graphen?
(Antwort: Für G k-regulär ist $\alpha_1(G) = k$. Zunächst ist der Vektor $(1, \ldots, 1)^t$ ein Eigenvektor von $A(G)$ zum Eigenwert k. Nach dem Lemma ist $\alpha_1 \leq d_{\max} = k$.)

Aufgabe 9. Finde eine graphentheoretische Interpretation von $(A(G)^l)_{ij}$.
(Antwort: $(A^l)_{ij}$ ist die Anzahl der Wege der Länge l zwischen Ecken i und j.)

Satz: *Sei G ein zusammenhängender Graph mit Durchmesser d. Dann hat G mindestens $d+1$ verschiedene Adjazenz-Eigenwerte (ebenso für Laplace-Eigenwerte).*

Beweis: Seien $\lambda_1, \ldots, \lambda_r$ die verschiedenen Eigenwerte von $A = A(G)$. Weil A diagonalisierbar ist, ist $(t - \lambda_1) \cdots (t - \lambda_r)$ das Minimalpolynom von A. Demnach $(A - \lambda_1 I) \cdots (A - \lambda_r I) = 0$, also ist A^r eine Linearkombination von $I, A, \ldots, A^{r-1}$.
Ist $d(i,j) = k$, dann gibt es nach Aufgabe 10 keine Wege der Länge $< k$ zwischen Ecken i und j, also $(A^l)_{ij} = 0$ for $l = 1, \ldots, k-1$ sowie $(A^k)_{ij} > 0$.
Seien i, j Ecken mit $d(i,j) = d$. Dann sind $0 = I_{ij} = A_{ij} = \cdots = (A^{d-1})_{ij}$ und $(A^d)_{ij} > 0$. Daraus folgt $r \geq d+1$.
Für die Laplace-Eigenwerte geht dieses Argument mit $nI - L$ anstatt A.

Aufgabe 10. Implementiere `Diskrepanz(G) := r-d-1` (nichtnegativ nach Satz).
Finde G mit positiver Diskrepanz.
(Die Anzahl der einfachen Nullstellen eines Polynoms f ist $\deg(f) - \deg(\mathrm{ggT}(f, f'))$, also hier $r = n - \deg(\mathrm{ggT}(\chi_A, \chi_A'))$. Für $n = 5, 6, 7, 8$ ist $\mathrm{Diskrepanz}(\overline{A_n}) = n - 3$.)

Visualisierung

Ausgangsfrage: wie stellt man abstrakte Graphen G gut in der Ebene dar?

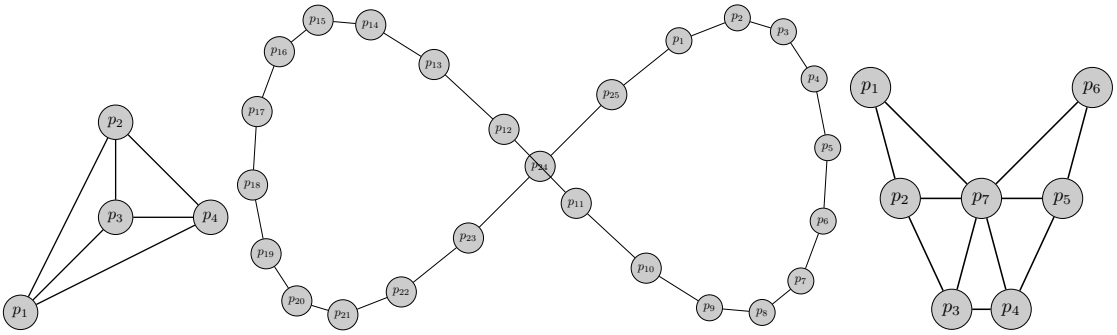
Gesucht sind also Koordinaten (x_i, y_i) für Ecken $i \in V(G)$, die den Graphen G möglichst gut repräsentieren. Eine Idee benutzt die Laplace-Matrix: seien $\lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$ die Eigenwerte von $L(G)$ und $v_1, \ldots, v_n$ ihre Eigenvektoren.
Eine Heuristik benutzt $v_2 =: (x_1, \ldots, x_n)$ und $v_3 =: (y_1, \ldots, y_n)$; also die Eigenvektoren zu den zweit- und drittkleinsten Laplace-Eigenwerten. Siehe Brouwers–Haemers §3.13.4.

Motivation für diese Heuristik: ist $\rho\colon V(G) \to \mathbb{R}^m$ eine Darstellung des Graphen, dann definieren wir die *Energie* von ρ als $\Psi(\rho) = \sum_e \|\rho(i) - \rho(j)\|^2$, wobei die Summe über alle Kanten $e = \{i, j\} \in E(G)$ läuft. Weiter sei $R \in M_{n\times m}(\mathbb{R})$ die spaltenweise Eintragung von ρ. Eine Matrizenrechnung zeigt $\Psi(\rho) = \mathrm{tr}(RLR^r)$.
Das Ziel ist es, $\Psi(\rho) = \Psi(R)$ unter geeigneten Nebenbedingungen zu minimieren. Pisanski und Shawe-Taylor benutzen $R(1, ..., 1)^t = 0$ und $RR^t = I$. Sie zeigen, dass dann die minimale Energie die Summe $\lambda_2+\lambda_3+\cdots+\lambda_{m+1}$ der ersten $m+1$ Laplace-Eigenwerte ist (dabei immer $\lambda_1 = 0$). Die Matrix R nimmt dieses Minimum an, wenn der Zeilenraum von R von Eigenvektoren zu $\lambda_2, \ldots, \lambda_{m+1}$ aufgespannt wird. Die Heuristik ergibt sich für ebene Graphendarstellungen ($m = 2$) und R besteht aus den beiden Eigenvektoren zu λ_2, λ_3.

Aufgabe 11. Implementiere die graphische Ausgabe von Punkten und Strecken in einem beliebigen Programm.

Aufgabe 12. Implementiere eine numerische Berechnung von zweitem und drittem Laplace-Eigenvektor und stelle damit Graphen graphisch dar.

Hier ein paar besonders schöne Beispiele, die wir erhalten konnten:



`Komplett(4)` `Zykel(25)` `Kegel(Pfad(6))`

Perron-Frobenius-Theorie

Für reelle $n \times n$-Matrizen M schreiben wir $M \geq 0$, wenn alle Einträge nichtnegativ sind und $M > 0$, wenn alle Einträge positiv sind (analog für reelle Vektoren). Einer Matrix $M \geq 0$ ordnen wir einen gerichteten Graphen G_M zu mit Kanten $i \to j$, wenn $M_{ij} > 0$. Für symmetrisches M ist G_M ein (ungerichteter) Graph.

Definition: Eine Matrix $M \geq 0$ heißt *primitiv*, wenn $M^k > 0$ für ein $k \in \mathbb{N}$. M heißt *irreduzibel*, wenn für alle i, j ein k existiert mit $(M^k)_{ij} > 0$.

Aufgabe 13. Zeige: M irreduzibel $\iff$ G_M gerichtet zusammenhängend
$M > 0 \implies M$ primitiv $\implies M$ irreduzibel $\implies M + I$ primitiv[1].

Satz (Perron-Frobenius): *Sei M eine irreduzible Matrix. Dann existiert eine eindeutige Zahl $\theta_0 \in \mathbb{R}_{>0}$ mit diesen Eigenschaften:*

(i) Es gibt einen reellen Eigenvektor $v_0 > 0$ mit $Mv_0 = \theta_0 v_0$.
(ii) θ_0 hat algebraische und geometrische Vielfachheit 1.
(iii) Für alle Eigenwerte θ von M gilt $|\theta| \leq \theta_0$.
Ist $|\theta| = \theta_0$ und M primitiv, dann gilt $\theta = \theta_0$.
(iv) Ist $v \geq 0$ ein Eigenvektor von M, dann gilt $Mv = \theta_0 v$, also $v \in \mathbb{R}_{>0} v_0$.
(v) Für eine Matrix mit $0 \leq N \leq M$ und einem Eigenwert ν von N gilt $|\nu| \leq \theta_0$.
Dieselbe Ungleichung gilt, wenn N ein Hauptminor von M ist.
In beiden Fällen folgt aus $|\nu| = \theta_0$ schon $N = M$.

$\theta_0 = \theta_0(M)$ ist also der *Spektralradius* und heißt auch *Perron-Frobenius-Eigenwert* oder *Perron-Wurzel* von M. Wir nennen v_0 den *Frobenius-Vektor*.

Für eine präzisere Aussage und den Beweis siehe Brouwers-Haemers §2.2.

Wir zeigen (i),(iii),(v). Sei $P := (I + M)^{n-1}$. Dann $P > 0$ und $PM = MP$. Setze $B := \{v \in \mathbb{R}^n \mid v \neq \text{ und } v \geq 0\}$. Betrachte

$$\theta \colon B \to \mathbb{R}, \quad \theta(v) = \max\{t \in \mathbb{R} \mid tv \leq Mv\} = \min\{\tfrac{(Mv)_i}{v_i} \mid v_i \neq 0\}.$$

Es gilt $\theta(v) = \theta(sv)$ für $s \in \mathbb{R}_{>0}$. Außerdem ist $\theta(Pv) \geq \theta(v)$, denn $tv \leq Mv \implies P(tv) = tPv \leq PMv = MPv$. Ist v kein Eigenvektor von M, dann $\theta(Pv) > \theta(v)$. Setze $K := \{v \in B \mid \|v\| = 1\}$, das ist eine kompakte Menge. Die Funktion θ ist stetig auf $P(K)$, nimmt also auf dem Kompaktum $P(K)$ ihr Maximum an. Somit gibt es $v_0 \in P(K)$ mit

$$\theta_0 := \sup_{v \in B} \theta(v) = \sup_{v \in K} \theta(v) = \sup_{v \in P(K)} \theta(v) = \theta(v_0).$$

Damit ist v_0 ein positiver Vektor mit $Mv_0 = \theta_0 v_0$.
(iii): sei $Mv = \theta v$ mit $v = (v_1, \ldots, v_n) \neq 0$. Setze $v_+ := (|v_1|, \ldots, |v_n|)$. Aus der Dreiecksungleichung folgt $|\theta| v_+ = |\theta v| = |Mv| \leq Mv_+$, also $|\theta| \leq \theta(v_+) \leq \theta_0$.
(v): $0 \leq N \leq M$ und $Nv = \nu v$, $v \neq 0 \implies Mv_+ \geq Nv_+ \geq |\nu| v_+ \implies |\nu| \leq \theta_0$.
$|\nu| = \theta_0 \implies Mv_+ = Nv_+ = |\nu| v_+$. Aus $(M - N)v_+ = 0$ und $v_+ > 0$ folgt $M = N$.

[1] Zeigen $(I+M)^{n-1} = I + (n-1)M + \cdots + M^{n-1} > 0$. Diese Ungleichung gilt, weil G_M gerichtet zusammenhängend ist, es also gerichtete Wege von und zu allen Ecken gibt; die kürzesten dieser Wege haben Länge $< n$, kommen somit in der Summe vor.

Spektralradius von Graphen

Hier kann G ein gerichteter oder, allgemeiner, ein ungerichteter Graph sein.

Definition: *Spektralradius* $\theta_0(G) := \theta_0(A(G))$.

Sei G' ein Graph, der aus G durch Entfernen von Ecken oder Kanten entsteht. Dann ist $A(G')$ ein Hauptminor von $A(G)$ oder $0 \leq A(G') \leq A(G)$.

Satz: $\theta_0(G') \leq \theta_0(G)$. *Ist G gerichtet zusammenhängend, dann* $\theta_0(G') < \theta_0(G)$.

Eine einfache Anwendung ist der Beweis der ADE-Klassifikation aus Aufgabe 8: es ist leicht zu sehen, dass $\tilde{G} := \tilde{A}_n = Z_{n+1}, \tilde{D}_n, \tilde{E}_6, \tilde{E}_7, \tilde{E}_8$ Eigenwert 2 haben mit positivem Eigenvektor. Nach Perron-Frobenius ist $\theta_0(\tilde{G}) = 2$ der Spektralradius. Die ADE-Graphen entstehen durch Weglassen einer Ecke, haben also $\theta_0(G) < 2$.
Ist umgekehrt H ein Graph mit $\theta_0(H) \leq 2$, dann kann H kein $\tilde{A}_n$ echt enthalten. Damit ist H zykelfrei, also ein Baum. Weiterhin enthält H nicht $\tilde{D}_3$, hat also keine Ecken vom Grad ≥ 4. Weil H auch kein $\tilde{D}_n$ mit $n \geq 4$ enthält, hat H höchstens eine Ecke vom Grad 3.

Definition: *Farbzahl* (auch *chromatische Zahl*) eines Graphen G.

Satz (Wilf 1967): *Für G zusammenhängend ist* $\chi(G) < \theta_0(G)$.

Beweis: Setze $m := \chi(G)$. Weil G nicht mit $m-1$ Farben eingefärbt werden kann, kann nicht $\deg(x) < m-1$ für alle Ecken x gelten; mithin existiert ein induzierter Untergraph $\Delta \subset G$, so dass $d_{\min}(\Delta) \geq m-1$. Nach Perron-Frobenius $\theta_0(G) \geq \theta_0(\Delta) \geq d_{\min}(\Delta) \geq m-1$.

* * *

Ein Ziel der algebraischen Graphentheorie ist es, klassische Grapheninvarianten durch Eigenwerte zu berechnen oder abzuschätzen. Insbesondere für große Graphen kann es leichter sein, Aussagen über das Spektrum zu treffen als über Farbzahl usw.

Hier einige weitere Aussagen dieser Art, ohne Beweis. G sei ein zusammenhängender Graph mit Adjazenz-Eigenwerten $\theta_0 = \alpha_1 \geq \ldots \geq \alpha_n$ und Laplace-Eigenwerten $0 = \lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$.

Satz: *G ist bipartit* $\iff$ *$-\alpha_1$ ist ein Eigenwert von $A(G)$.*

Satz: *G is k-regulär* $\iff$ $\alpha_1^2 + \cdots + \alpha_n^2 = kn$.

Hoffmann 1970: $\chi(G) \geq 1 - \alpha_1/\alpha_n$.

Kirchhoff 1847: $\det(L + \frac{1}{n^2}J) = \frac{1}{n}\lambda_2 \cdots \lambda_n$ *Anzahl der aufspannenden Bäume.*

Cayley 1889: *Die Anzahl der aufspannenden Bäume in K_n ist n^{n-2}.*

Ranglisten und Google PageRank

Modellieren ein Netzwerk (zum Beispiel von Personen) durch einen ungerichteten Graphen G. Wichtige Personen im Netzwerk sollten viele Verbindungen haben. Dennoch muss die Person mit den meisten Kontakten (also die Ecke maximalen Grades) nicht unbedingt am wichtigsten sein: wirklich wichtige Leute haben Kontakte zu anderen wichtigen Menschen.
Sei $I\colon V(G) \to \mathbb{R}_{\geq 0}$ eine Zuordnung, wie wichtig jede Ecke ist. Postulieren

$$I(x) = \frac{1}{\theta} \sum_{e=\{x,y\}} I(y) \qquad \text{für alle } x \in V(G);$$

die Wichtigkeit einer Ecke ist also — bis auf den Proportionalitätsfaktor $1/\theta$ — die Summe der Wichtigkeit aller Nachbarn. Dieser Ansatz ist äquivalent zu $\theta I = AI$. Mit $I \geq 0$ und $\theta > 0$ gibt es nach Perron-Frobenius für zusammenhängendes G eine eindeutige Lösung $\theta = \theta_0$ und I ist der Frobenius-Vektor von A.

* * *

Sei nun G ein gerichteter Graph mit n Ecken, zum Beispiel das Internet mit Links (dann ist $n > 10^9$). Sei A die gerichtete Adjazenz-Matrix von G und D die Diagonalmatrix der Ausgangsgrade in G. Wir nehmen an, dass G keine Ecken ohne abgehende Kanten hat; dann ist D diagonalisierbar. Seien J die nur aus Einsen bestehende $n \times n$-Matrix und $\alpha \in (0, 1)$ ein fixierter Parameter. Wir betrachten die Konvexkombination $M := \frac{1-\alpha}{n} J + \alpha D^{-1} A$. Dann haben $D^{-1}A$ und M Zeilensummen 1, und $M > 0$. Nach Perron-Frobenius gibt es einen positiven Linkseigenvektor $vM = v$ (das ist der Frobenius-Vektor für M^t; wegen $M(1, ..., 1)^t = (1, ..., 1)^t$ ist der Linksspektralradius ebenfalls 1).[2]
Normieren v als Wahrscheinlichkeitsverteilung: $\sum_x v_x = 1$. Der PageRank von x ist dann $\mathrm{PR}(x) := v_x \in [0, 1]$. Weil $vM = v$ eine Fixpunktgleichung ist, kann man v iterativ approximieren: $v^0 := (1/n, \ldots, 1/n)^t$ und $v^{i+1} := v^i M = v^0 M^i$.
Kleines α gibt schnellere Konvergenz, aber größere α bilden die Eigenschaften von G besser ab: α ist die Wahrscheinlichkeit, dass ein zufälliger Surfer einem Link auf der aktuellen Seite folgt, anstatt eine beliebige Seite anzuwählen. Man nimmt $\alpha \approx 0,85$. So entsteht eine statische Rangliste aller Webseiten, die für die SERP (search engine results page) benutzt wird, aber nur als ein Faktor von vielen.
Wegen der geringen Linkzahl (im Durchschnitt 12 pro Seite) ist M extrem dünn besetzt und Potenzen von M sind schnell berechenbar. Die Anzahl der Iterationen scheint (bei konstanter Fehlerschwelle ε) logarithmisch in n zu wachsen. Für das reale Internet macht es nur einen geringen Unterschied, ob Links als gerichtete oder ungerichtete Kanten betrachter werden.

Diese Verfahren sind auf beliebige Graphen anwendbar und werden eingesetzt in: Bibliothekswissenschaften (Referenzen, ISI impact factor; Eigenvektormethode um 1977 in diesem Bereich beschrieben), soziale Netzwerke (z.B. Twitter), Neurowissenschaften (Neuronen), Biologie (Proteinnetze), Semantik (Unterscheidungen von Wortbedeutungen), web crawler.

[2] Sei M eine irreduzible Matrix mit nichtnegativen Links- und Rechtseigenvektoren $wM = \lambda w$ und $Mv = \theta v$. Aus den Eigenwertgleichungen folgt $\theta wv = wMv = \lambda wv$. Mit B wie im Beweis von Perron-Frobenius und $v \in B, w = w_0$ bzw. $v = v_0, w \in B$ folgt $wv \neq 0$, also $\lambda = \theta (= \theta_0 = \lambda_0)$.

Prozeduren zur Arbeit mit Listen

```
proc ElementVon(list l, int a){
    for (int i=1; i<=size(l); i++){ if(l[i]==a){ return(1); } }
    return(0);
}

// Gibt das Minimum einer Liste von nichtnegativen Zahlen aus.
proc MinIndex(list h){
    int Min=1;
    for(int i=1; i<=size(h); i++){ if( h[i]<h[Min] ){ Min = i; } }
    return(Min);
}

// Gibt den Index in I des minimalen positiven Eintrags in einer Liste d von integers
proc Mini(list d, list I){
    int m; int i;
    for(i=1; i<=size(I); i++){ if(d[I[i]]>=0){ m=i; break; } }
    if(m==0){ return(m); }
    for(i=1; i<=size(I); i++){
        if(d[I[i]]<d[I[m]] && d[I[i]]>=0){ m=i; }
    }
    return(m);
}

// Gibt das Maximum einer Liste von nichtnegativen ganzen Zahlen aus.
proc Maximum(list h){
    int Max=0;
    for(int i=1;i<=size(h);i++){
        if( h[i]==-1 ){ return(-1); }
        if( h[i]>Max ){Max = h[i];}
    }
    return(Max);
}

// Gibt das Minimum einer Liste von nichtnegativen Zahlen aus.
proc Minimum(list h){
    int Min=0;
    for(int i=1;i<=size(h);i++){
        if( h[i]<Min ){Min = h[i];}
    }
    return(Min);
}


// Gibt das Maximum einer Liste von reellen Zahlen aus.
proc MaximumR(list h){
    number Max=0;
    for(int i=1;i<=size(h);i++){
        if( h[i]>Max ){Max = h[i];}
    }
    return(Max);
}
```

Datentypen Graph

```
newstruct("Graph", "list vertices, list edges");

proc PrettyPrint(Graph G){
    list v=G.vertices; list e=G.edges;
    string sv = string(size(v)) + " Ecken: ";
    for (int i=1; i<size(v); i++){  sv = sv + string(v[i]) + ","; }
    sv = sv + string(v[size(v)]);
    print(sv);
    string se = string(size(e) div 2) + " Kanten: ";
```

```
    if (size(e) == 0) { se = se + "keine"; }
    else {
        for (int j=1; j<size(e) div 2; j++){
            se = se + string(e[2*j-1]) + "," + string(e[2*j]) + " ; ";
        }
        se = se + string(e[size(e)-1]) + "," + string(e[size(e)]);

    }
    print(se);
}

// Überprüft, ob G wirklich einen Graphen definiert
proc wirklichGraph(Graph G){
    list v=G.vertices; list e=G.edges;
    int i; int j=1;
    if(size(e) mod 2 == 1){ return("Kantenvektor ungerader Länge!"); }
    for(i=1; i<=size(e); i++){
            if(!ElementVon(v,e[i]){
                return("Kanten benutzen undefinierte Ecke!");
            }
    }
    return("OK");
}

// Verschönert den Graphen:
// - löscht Doppelecken
// - sortiert Kanten in sich, z.B. (1,2) statt (2,1)
// - sortiert Ecken der Größe nach
proc Entdoppelt(Graph G){
    list v=G.vertices; list e=G.edges;
    int i,j,x;
    // Schritt 1: Kanten ausrichten
    for(i=1; i<=size(e) div 2; i++){
        if(e[2*i-1]>e[2*i]){
            x=e[2*i-1];
            e[2*i-1]=e[2*i];
            e[2*i]=x;
        }
    }
    // Schritt 2: Doppelte Ecken streichen
    G.edges = e;
    for(i=1; i<size(v); i++){
        for(j=i+1; j<=size(v); j++){
            if (v[i]==v[j]){ v = delete(v,j); j--;}
        }
    }
    // Schritt 3: Ecken sortieren
    list h = v;
    for (i=1; i<=size(v); i++){
        j=MinIndex(h); v[i] = h[j]; h = delete(h,j);
    }
    G.vertices = v;
    // Schritt 4: Doppelte Kanten streichen
    for(i=1; i<size(e) div 2; i++){
        for(j=i+1; j<=size(e) div 2; j++){
            if (e[2*i-1]==e[2*j-1] && e[2*i]==e[2*j]){
                e = delete(e,2*j); j--; e = delete(e,2*j-1);
            }
        }
    }
    G.edges = e;
    return(G);
}
```

Beispiele

```
// Pfad(n) liefert den Pfad-Graphen mit n Ecken
proc Pfad(int n){
    Graph A;
    list v,e;
    for(int i=1; i<=n;i++){
        v[i]=i;
        if (i != n){ e[2*i-1]=i; e[2*i]=i+1; }
    }
    A.vertices=v; A.edges=e;
    return(A);
}

// Zykel(n) liefert den Zykel-Graphen mit n Ecken
proc Zykel(int n){
    Graph A = Pfad(n);
    A.edges[2*n-1]=n; A.edges[2*n]=1;
    return(A);
}

// Kegel(G) erweitert G um eine Ecke, die mit allen Ecken von G verbunden wird.
proc Kegel(G){
    list v = G.vertices; list e = G.edges;
    int n = size(v); int m = size(e);
    v[n+1] = n+1;
    for (int i=1; i<=n; i++){ e[m+2*i-1]=i; e[m+2*i]=n+1; }
    Graph H;
    H.vertices = v; H.edges = e;
    return(H);
}

// Komplett(n) liefert den vollständigen Graphen mit n Ecken.
proc Komplett(int n){
    if (n==1) { return(Pfad(1)); }
    else { return(Kegel(Komplett(n-1))); }
}

// Bipartit(n,m) vollständiger bipartiter Graph mit n+m Ecken
proc Bipartit(int n, int m){
    int i,j; Graph A;
    list e,v;
    for(i=1; i<=n; i++){ v[i]=i; }
    for(i=1; i<=m; i++){ v[n+i]=-i; }
    for(i=1; i<=n; i++){
        for(j=1; j<=m; j++){ e[size(e)+1]=i; e[size(e)+1]=-j; }
    }
    e=e[2..size(e)];
    A.vertices=v; A.edges=e;
    return(A);
}

// Liefert den Komplementärgraphen.
// Nimmt an, dass Kanten geordnet sind (1,2, nicht 2,1)
proc Komplement(Graph G){
    Graph H;
    list v = G.vertices; list e = G.edges; list ee;
    int i,j,k, l; int n = size(v);
    H.vertices = v;
    for (i=1; i<n; i++){
        for (j=i+1; j<=n; j++){
            k=1; l=0;
            for (k=1; k<=size(e) div 2; k++){
                if (e[2*k-1]==v[i] && e[2*k]==v[j]){ l=1; }
            }
            if ( l==0 ){ ee = ee + list(v[i]) + list(v[j]); }
        }
    }
```

```
    H.edges = ee;
    return(H);
}

// Gibt den Kantengraph L(G) von G aus.
proc Kantengraph(Graph G){
    list EG = G.edges;
    int m = size(EG) div 2;
    Graph L; list VL, EL;
    int i,j;
    for (i=1; i<=m; i++) { VL = VL + list(i); }
    for (i=1; i<m; i++){
        for (j=i+1;j<=m;j++){
            if (   EG[2*i-1] == EG[2*j-1] || EG[2*i] == EG[2*j-1] ||
                   EG[2*i-1] == EG[2*j]   || EG[2*i] == EG[2*j] )
               { EL = EL + list(VL[i]) + list(VL[j]); }
        }
    }
    L.vertices = VL; L.edges = EL;
    return(L);
}

// Eine schöne Definition des Petersen-Graphen
Graph Petersen = Komplement(Kantengraph(Komplett(5)));
```

Nachbarn und Abstände

```
// Gibt die Liste der Indizes aller Nachbarn von G.vertices[a].
proc Nachbarn(Graph G, int a){
    list e = G.edges; list v = G.vertices;
    if (a > size(v)){
        ERROR(print("Die angegebene Ecke existiert nicht."));
    }
    list N; int j; int k;
    for (int i=1; i<=size(e); i++){
        if ( e[i]==v[a] ){
            // Hier ist v[a] Teil der Kante; je nach
            // Parität nehmen wir Vorgänger oder Nachfolger.
            for (j=1;j<=size(v);j++){
                k=i mod 2; k=i+2*k-1;
                if ( v[j]==e[k] ){ N = N + list(j); }
            }
        }
    }
    return(N);
}

// Dijkstra: Gibt die Liste der Distanzen der Ecken zu der a-ten Ecke aus.
proc Distanzen(Graph G, int a){
    list e = G.edges; list v = G.vertices; int u; list nachbarn;
    if (a > size(v)){
        return("Die angegebene Ecke existiert nicht.");
    }
    list dist; list unvisited; list reached;
    for(int i=1; i<=size(v); i++){
        dist[i]=-1; unvisited[i]=i;
    }
    dist[a]=0;
    while(size(unvisited)!=0){
        if(Mini(dist, unvisited)==0){return(dist);}
        u=Mini(dist, unvisited);
        nachbarn=Nachbarn(G, unvisited[u]);
        for(i=1; i<=size(nachbarn); i++){
            if(dist[unvisited[u]]+1<dist[nachbarn[i]] ||  dist[nachbarn[i]]==-1){
                dist[nachbarn[i]]=dist[unvisited[u]]+1;
            }
```

```
        }
        unvisited=delete(unvisited,u);
    }
    return(dist);
}

// Berechnet den Durchmesser eines zusammenhängenden Graphen
proc Diam(Graph G){
    list v = G.vertices; list h;
    for(int i=1;i<=size(v);i++){ h = h+Distanzen(G,i); }
return(Maximum(h));
}

// Berechnet r-diam(G)-1, dabei ist r die Anzahl der verschiedenen
// Adjazenz-Eigenwerte. Immer Diskrepanz >= 0.
// Diskrepanz(Komplement(Pfad(n))) = n-3
// Setzt voraus, dass G zusammenhängend ist.
proc Diskrepanz(Graph G){
    ring hilfsring=0,x,dp;
    matrix A=Adjazenz(G);
    poly f=charpoly(A);
    return(size(G.vertices)-deg(gcd(f,diff(f,x)))-Diam(G)-1) ;
}
```

Gerichtete Graphen und PageRank

```
// Alle Matrizen-Prozeduren setzen Eckenliste 1,...,n voraus.
// Gerichtete Graphen

proc diAdjazenz(Graph G){
    list e=G.edges; list v=G.vertices; int n=size(v);
    matrix A[n][n];
    for(int i=1; i<=size(e) div 2; i++){
        A[e[2*i-1],e[2*i]]=1;
    }
    return(A);
}

proc OutGradmatrix(Graph G){
    int n=size(G.vertices);
    list e=G.edges;
    matrix M[n][n];
    for(int i=1; i<=size(e) div 2; i++){
        M[e[2*i-1],e[2*i-1]]=M[e[2*i-1],e[2*i-1]]+1;
    }
    return(M);
}

proc PageRank(Graph G, number a, number eps){
    int n=size(G.vertices);
    matrix A=Adjazenz(G);
    matrix D=Gradmatrix(G);
    matrix S=inverse(D)*A;
    matrix M=(1-a)/number(n)*Einsmatrix(n,n)+a*S;
    matrix v[1][n]=1/number(n)*Einsmatrix(1,n);
    matrix u[1][n]=v*M;
    while(number(((u-v)*Einsmatrix(n,1))[1,1])>eps){
        v=u;
        u=v*M;
    }
    return(u);
}

proc Einsmatrix( m, n){
    matrix J[m][n]; int j;
    for(int i=1; i<=m; i++){
```

```
        for(j=1; j<=n; j++){
            J[i,j]=1;
        }
    }
    return(J);
}

proc Laplace(Graph G){
    return(Gradmatrix(G)-Adjazenz(G));
}

proc AmaxEW(Graph G){
    bigint b = bigint(10)^100; number t = 1/b;
    matrix A=Adjazenz(G);
    number max=MaximumR(qrds(A,t,t,t)[1]);
    return(max);
}
```

Eigenvektoren in Singular

```
ring Q=0,x,dp;

// Berechnet die Eigenwerte und -vektoren einer symmetrischen Matrix m
// algebraisch in Singular mit fortgesetzten Körpererweiterungen und
// schreibt numerische Approximationen der zwei Laplace-Eigenvektoren
// von m in die Datei "outfile.txt".
proc Eigenvektoren(matrix m){
    write(":w outfile.txt","Matrix");
    write(":a outfile.txt",print(m));

    if(deg(eigenvals(m)[1])<=1){
        matrix j=matrix(jordanbasis(m)[1]);
        matrix e=matrix(eigenvals(m)[1]);
        matrix V=matrix(eigenvals(m)[2]);
        list EV=list(e)+list(V)+list(j);
    }
    else{
        def B=basering;
            poly Char=charpoly(m);
            def r=roots(Char);
        setring r;
            matrix m=fetch(B,m);
            matrix J=matrix(jordanbasis(m)[1]);   J=subst(J,a,x);
            matrix E=matrix(eigenvals(m)[1]);     E=subst(E,a,x);
            matrix V=matrix(eigenvals(m)[2]);     poly mini=poly(subst(minpoly,a,x));
        setring B;
            poly mini=fetch(r,mini);
            def S=solve(mini);
            matrix J=fetch(r,J);
            matrix E=fetch(r,E);
            matrix V=fetch(r,V);
        setring S;
            number b=SOL[1];
        ring R=real,x,dp;
            number b=fetch(S,b);
            matrix J=fetch(B,J);
            matrix E=fetch(B,E);
            matrix V=fetch(B,V);
            matrix j=subst(J,x,b);
            matrix e=subst(E,x,b);
            keepring R;
        ring RR=(real,10),x,dp;
            matrix j=fetch(R,j);
            matrix e=fetch(R,e);
            matrix V=fetch(R,V);
            keepring RR;
```

```
            list EV=list(e)+list(V)+list(j);

    }
    write(":a outfile.txt","Eigenwerte");
    write(":a outfile.txt",print(e));
    write(":a outfile.txt","Vielfachheiten");
    write(":a outfile.txt",print(transpose(V)));
    write(":a outfile.txt","Eigenvektoren");
    write(":a outfile.txt",print(j));
    return(EV);
}
```

Visualisierung

Zu einem ungerichteten, zusammenhängenden Graphen G berechnet die Singular-Prozedur `Bildchen` die Laplace-Eigenvektoren v_i berechnet und gibt v_2, v_3 spaltenweise als `vertices.csv` aus (die logische Kanten stehen in `edges.csv`).

```
// Gibt zu einem ungerichteten Graphen G drei Dateien aus:
// edges.csv    -- eine integer-Matrix mit zwei Spalten (G.edges)
// outfile.txt  -- Laplace-Matrix von G, Eigenwerte, Vielfachheiten, Eigenvektoren
// vertices.csv -- eine real-Matrix mit zwei Spalten (v2,v3)
proc Bildchen(Graph G){
    matrix m=Laplace(G);
    if(size(G.edges)!=0){
        matrix Edges[1][2];
        Edges[1,1]=G.edges[1]; Edges[1,2]=G.edges[2];
        write(":w edges.csv",print(Edges));
        for(int i=2; i<=size(G.edges) div 2; i++){
            Edges[1,1]=G.edges[2*i-1]; Edges[1,2]=G.edges[2*i];
            write(":a edges.csv",print(Edges));
        }
    }
    else{
        write(":w edges.csv","");
    }
    write(":w outfile.txt","Matrix");
    write(":a outfile.txt",print(m));

    if(deg(eigenvals(m)[1])<=1){
        matrix j=matrix(jordanbasis(m)[1]);
        matrix e=matrix(eigenvals(m)[1]);
        matrix V=matrix(eigenvals(m)[2]);
        list EV=list(e)+list(V)+list(j);
    }
    else{
        def B=basering;
            poly Char=charpoly(m);
            def r=roots(Char);
        setring r;
            matrix m=fetch(B,m);
            matrix J=matrix(jordanbasis(m)[1]);    J=subst(J,a,x);
            matrix E=matrix(eigenvals(m)[1]);      E=subst(E,a,x);
            matrix V=matrix(eigenvals(m)[2]);      poly mini=poly(subst(minpoly,a,x));
        setring B;
            poly mini=fetch(r,mini);
            def S=solve(mini);
            matrix J=fetch(r,J);
            matrix E=fetch(r,E);
            matrix V=fetch(r,V);
        setring S;
            number b=SOL[1];
        ring R=real,x,dp;
            number b=fetch(S,b);
            matrix J=fetch(B,J);
            matrix E=fetch(B,E);
            matrix V=fetch(B,V);
```

```
            matrix j=subst(J,x,b);
            matrix e=subst(E,x,b);
            keepring R;
        ring RR=(real,1000),x,dp;
            matrix j=fetch(R,j);
            matrix e=fetch(R,e);
            matrix V=fetch(R,V);
            keepring RR;
            list EV=list(e)+list(V)+list(j);
    }
    write(":a outfile.txt", "Eigenwerte");
    write(":a outfile.txt", print(e));
    write(":a outfile.txt", "Vielfachheiten");
    write(":a outfile.txt", print(transpose(V)));
    write(":a outfile.txt", "Eigenvektoren");
    write(":a outfile.txt",  print(j));
    matrix koordinaten[ncols(j)][2];
    matrix v1[ncols(j)][1]; matrix v2[ncols(j)][1];
    list Min=Minimum(e); int k;
    for(i=1; i<Min[1]; i++){ k=k+int(V[i,1]); }
    v1=j[1..ncols(j),k+1];
    if(Min[1]==Min[2]){ v2=j[1..ncols(j),k+int(V[Min[1],1])]; }
    else{ k=0;
        for(i=1; i<Min[2]; i++){ k=k+int(V[i,1]); }
        v2=j[1..ncols(j),k+1];
    }
    koordinaten[1..ncols(j),1]= v1;
    koordinaten[1..ncols(j),2]= v2;
    write(":w vertices.csv",print(koordinaten));
    setring Q; keepring Q;
    //return(EV);
}

// Gibt die Indizes der zwei minimalen positiven Einträge einer 1xn Matrix aus
// (dabei das Minimum zweifach, falls es nur einen positiven Eintrag gibt).
proc Minimum(matrix M){
    int Min=1; int Min2=1; int i;
    for(i=1; i<=ncols(M); i++){
        if(M[1,i]>0){Min=i; break;}
    }
    for(i=i+1; i<=ncols(M); i++){
        if(M[1,i]<M[1,Min] &&  M[1,i]>0){Min = i;}
    }
    for(i=1; i<=ncols(M); i++){
        if(M[1,i]>0 && i!=Min){Min2=i; break;}
    }
    for(i=i+1; i<=ncols(M); i++){
        if(M[1,i]<M[1,Min] &&  M[1,i]>0 && i!=Min){Min2 = i;}
    }
    if(M[1,Min]==0){Min=ncols(M);}
    if(M[1,Min2]==0){Min2=Min;}
    return(Min,Min2);
}
```

TikZ und LaTeX:

```
\documentclass{article}
\usepackage{tikz,csvsimple}
\usepackage[active,tightpage]{preview}
\PreviewEnvironment{tikzpicture}
\edef\i{1}
\begin{document}
\begin{tikzpicture}
\csvreader[no head]{vertices.csv}{1=\first, 2=\second}%
    {\draw node[draw, color=black, shape=circle, minimum size=1pt, fill= black!20]
```

```
            (p_{\i}) at (2*\first,2*\second) {$p_{\i}$};
                        \pgfmathparse{int(\i+1)}
                        \let\i\pgfmathresult
                        \par}
\csvreader[no head]{edges.csv}{1=\first, 2=\second}%
    {\draw[thick] (p_{\first}) -- (p_{\second});
                        \par}
\end{tikzpicture}
\end{document}
```