---

Block 11 (Freitag 3.3.2023)

---

# 12 Graphen

VIDEO: `video10a_graphen_defs.mkv`

## 12.1 Grundlegende Definitionen

(ungerichteter/ gerichterer) Graph $G = (V, E)$: Knoten $i \in V$ und ungerichtete/gerichtete Kanten $\{i, j\} \in E \subset V^{(2)}$ bzw. $(i, j) \in E$ Hinweis: bei ungerichtetetn Graphen $\{i, j\} = \{j, i\}$

[Selbsttest]

Nennen Sie real Anwendungen von Graphen = Netzwerke

---

Weitere Definitionen:

- Anzahl der Knoten $N = |V|$ .
- Anzahl der Kanten $M = |E|$.
- $i, j \in V$ sind adjazent / benachbart falls $\{i, j\} \in E$.
- $\{i, j\}$ ist inzident zu $i$ und $j$.
- Grad $d_i$ von $i$ = Zahl der benachbarten Knoten. $i$ ist isoliert falls $d_i = 0$.

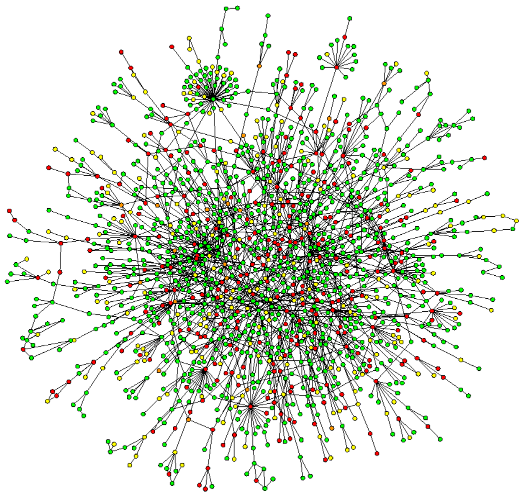
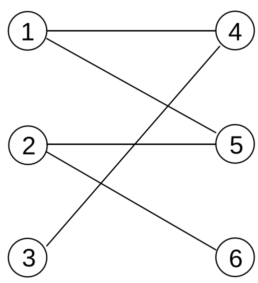


Figure 19: Protein Wechselwirkungs Netzwerk von Hefe

- Pfad $E' = \{\{i_0, i_1\}, \{i_1, i_2\}, \ldots, \{i_{l-1}, i_l\}\} \subset E$, Länge $l = |E'|$. $E'$ geht von $i_0$ nach $i_l$ und umgekehrt (Endknoten).
- $i, j$ verbunden: $\exists$ Pfad von $i$ nach $j$.
- Zusammenhangskomponente $V' \subset V$ (maximaler Größe): alle $i, j \in V'$ sind verbunden.

---

Example: Graph




Graph $G = (V, E)$ mit $V = \{1, 2, 3, 4, 5, 6\}$ und $E = \{\{1, 4\}, \{1, 5\}, \{2, 5\}, \{2, 6\}, \{3, 4\}\}$.

Knotenzahl $N = |V| = 6$, Kantenzahl $M = |E| = 5$.

Grade, z.B. $d(1) = 2$, $d(3) = 1$.

$E' = \{\{5,1\}, \{1,4\}, \{4,3\}\}$: Pfad von 5 nach 3 der Länge 3.

□

---

- Ein Graph $G' = (V', E')$ heisst Untergraph von $G$ falls $V' \subset V,\ E' \subset E$.
- Komplementärer Graph $G^C = (V, E^C)$: $E^C = V^{(2)} \setminus E = \{\{i,j\} \mid \{i,j\} \notin E\}$.

Kanten mir Orientierung:

- Ein gerichteter Graph $G = (V, E)$: $(i,j) \in E \subset V \times V$: geordnete Paare von Knoten
- gerichteter Pfad von $i_0$ nach $i_l$: $E' = \{(i_0, i_1), (i_1, i_2), \ldots, (i_{l-1}, i_l)\} \subset E$
- starke Zusammenhangskomponente $V'$: $\forall i, j \in V'$, $\exists$ ein gerichteter Pfad von $i$ nach $j$ und ein gerichteter Pfad von $j$ nach $i$.

Markierte Graphen:
Abbildung $d: E \to \mathcal{R}$, z.B. $d(\{i,j\})$= Abstand zwischen $i$ und $j$.

VIDEO: `video10a_graphen_implement.mkv`

## 12.2 Implementierung

[Selbsttest]

Überlegen Sie sich Computerimplementierungen von Graphen
Bitte erst nach Bearbeitung der Aufgabe Weiterlesen

---

Am einfachsten: Adjazenzmatrix $\{a_{ij}\}$ mit

$$a_{ij} = \begin{cases} 1 & \exists\,\text{Kante}\,(i,j) \\ 0 & \text{sonst} \end{cases} \tag{65}$$

---

Example: Adjazenzmatrix

Für Graph aus Anfangsbeispiel:

$$\{a_{ij}\} = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

□

---

Problem: Speichbedarf $O(N^2)$ auch für dünnbesetzte Graphen.

Ausweg: Nachbarliste. Für jeden Knoten $i$: Liste mit Nachbarn (gerichtet $(i,j)$ : $j$ ist in Liste von $i$, ggf. nicht umgekehrt)

Vorteil: nur tatsächlich benötigter Speicher.

Nachteil (klein): Test ob $j$ Nachbar von $i$ dauert $O(\#\text{ Nachbarn })$ (=von Knotenzahl für die meisten Graphen im Mittel unabhängig $\rightarrow$ kein Problem)

*Bild vom Beispielgraph als Liste:*

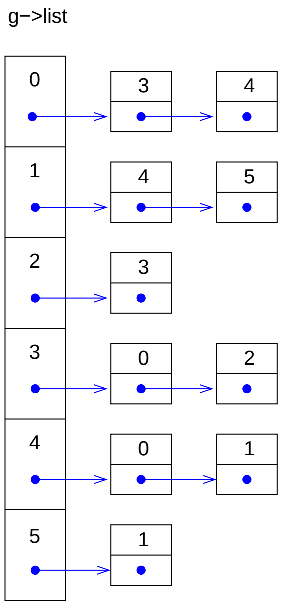



Figure 20: Nachbarlisten Darstellung des Beispiel Graphen.

Implementierung in $C$ (mit Listen aus `list.c`)

```
typedef struct
{
    elem_t *neighbors;   /* pointer to list of neighbors */
} gs_node_t;

typedef struct
{
    int             num_nodes;          /* number of nodes */
    gs_node_t           *node;          /* array of nodes */
} gs_graph_t;
```

Hinweis: hier mit fester (=maximaler) Knotenzahl.
Graph anlegen:

```
/****************** gs_create_graph() *******************/
/** Creates a graph with a fixed number of nodes and    **/
/** no edges.                                           **/
/** PARAMETERS: (*)= return-paramter                    **/
/**      num_nodes: number of nodes                     **/
/** RETURNS:                                            **/
/**     pointer to created graph                        **/
/*********************************************************/
gs_graph_t *gs_create_graph(int num_nodes)
{
    gs_graph_t *g;
    int n;

    g = (gs_graph_t *) malloc(sizeof(gs_graph_t));  /* allocate */
    g->node = (gs_node_t *) malloc(num_nodes*sizeof(gs_node_t));

    g->num_nodes = num_nodes;                      /* initialise */
    for(n=0; n<num_nodes; n++)
        g->node[n].neighbors = NULL;

    return(g);
}
```

Hinweis: Hier ungerichtete Graphen.

Kante $\{i, j\}$ einfügen:
1. Teste ob Kante bereits vorhanden
2. Erzeuge zwei Listelemente mit Knoten $i$ bzw. $j$ (Symmetrie)
3. Füge sie an Anfängen der Listen für $j$ bzw. $i$ ein.

In C:

```
/****************** gs_insert_edge() *******************/
/** Insert undirected edge (from,to) into graph        **/
/** if the edge does not exists so far.                **/
/** PARAMETERS: (*)= return-paramter                   **/
/**            g: graph                                **/
/**      from, to: id of nodes                         **/
/** RETURNS:                                           **/
/**     (nothing)                                      **/
/********************************************************/
void gs_insert_edge(gs_graph_t *g, int from, int to)
{
    elem_t *elem1, *elem2;

    if(search_info(g->node[from].neighbors, to)!=NULL)/* edge exists? */
        return;

    elem1 = create_element(to);         /* create neighbor for 'from' */
    g->node[from].neighbors =
        insert_element(g->node[from].neighbors, elem1, NULL);
    elem2 = create_element(from);         /* create neighbor for 'to' */
    g->node[to].neighbors =
        insert_element(g->node[to].neighbors, elem2, NULL);
}
```

Beispielfunktion: Berechung des Knotengrads:

```
/*********************** gs_degree() *******************/
/** Calculates number of neighbors of node             **/
/** PARAMETERS: (*)= return-paramter                   **/
/**            g: graph                                **/
/**            n: node                                 **/
/** RETURNS:                                           **/
/**     degree                                         **/
/********************************************************/
int gs_degree(gs_graph_t *g, int n)
{
  int degree;
  elem_t *neighb;

  if(n >= g->num_nodes)
    return(0);
  degree = 0;
  neighb = g->node[n].neighbors;
```

```
  while(neighb != NULL)      /* go through all neighbors */
  {
    degree++;
    neighb = neighb->next;
  }
  return(degree);
}
```

Analog: Überprüfen ob Kante exisitiert, Kante löschen.

Für markierte Graphen:
Erweiterung der Listenelemente um `distance`, dass $d(\{i,j\})$ speichert.
VIDEO: `video10a_graphen_skalenfrei.mkv`

## 12.3 Skalenfreie Graphen

Für viele Graphen: Matthäus Prinzip: "Wer hat, dem wird gegeben´´
Bsp: Häufig zitierte Artikel sind sehr bekannt, daher werden sie noch häufiger zitiert.

Erzeugen solcher Graphen zufällig mit preferential attachment.

- Füge neue Knoten $i$ einem nach dem anderen hinzu, $t_i \in [0, N]$: relativer Zeitpunkt zu dem $i$ hinzugefügt wird.
- Verbinde jeden neuen Knoten $i$ mit mit $m$ alten $j$, dabei Wahrscheinlichkeit $j$ zu wählen ist $\sim d_j(t)$ (Grad zu dem Zeitpunkt $t$)

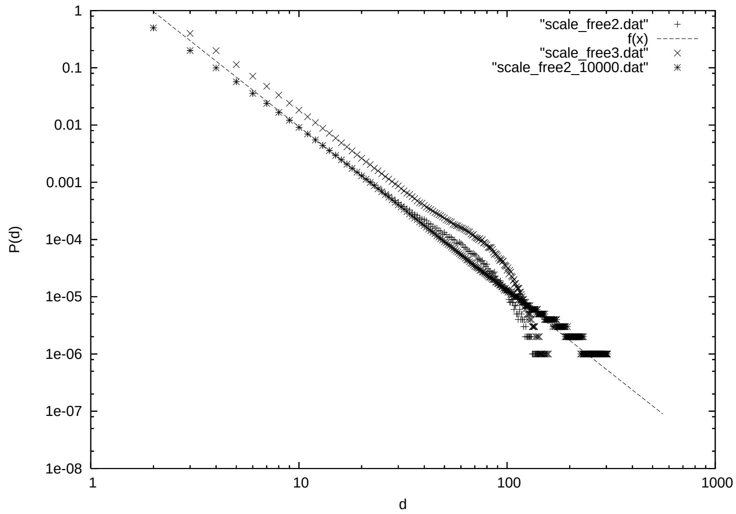
C Realisierung:

```
void gs_preferential_attachment(gs_graph_t *g, int m)
{
  int t, n1, n2;
  int *pick;            /* array which holds for each edge {n1,n2} */
                        /* the numbers n1 and n2. Used for picking */
                       /* nodes proportional to its current degree */
  int num_pick;              /* number of entries in 'pick' so far */
  int max_pick;                       /* maximum number of entries */

  if(g->num_nodes < m+1)
  {
     printf("graph too small to have at least %d edges per node!\n", m);
     exit(1);
  }
  max_pick = 2*m*g->num_nodes- m*(m+1);
  pick = (int *) malloc(max_pick*sizeof(int));
  num_pick=0;
  for(n1=0; n1<m+1; n1++) /* start: complete subgraph of m+1 nodes */
    for(n2=n1+1; n2<m+1; n2++)
    {
      gs_insert_edge(g, n1, n2);
      pick[num_pick++] = n1;
      pick[num_pick++] = n2;
    }

  for(n1=m+1; n1<g->num_nodes; n1++)            /* add other nodes */
  {
    t=0;
    while(t<m)                                   /* insert m edges */
    {
      do
        n2 = (int) pick[(int) floor(drand48()*num_pick)];
      while(n2==n1);              /* chose pair of different nodes */
      if(!gs_edge_exists(g, n1, n2))
      {
        gs_insert_edge(g, n1, n2);
        pick[num_pick++] = n1;
        pick[num_pick++] = n2;
        t++;
      }
    }
  }
  free(pick);
}
```

Simulation ($m = 2$, $N = 100, 1000, 10000$ Mittelung über 10000 Graphen): Verteilung des Grads $d_i$




[Selbsttest]

Welche funktionale Abhängigkeit sehen Sie?

---

Näherungsweise Rechnung (Vernachlässigung, dass zu Anfang vollständiger Teilgraph da ist, Übergang kontinuierliche Zeit):
Wahrscheinlichkeit, dass Knoten $i$ vor der Zeit $K$ hinzugefügt wird:

$$P(t_i < K) = \frac{K}{N} \tag{66}$$

Wahrscheinlichkeit, dass ein Knoten $j$ in einem Schritt als Nachbar ausgewählt wird (zur Zeit $t$: insgesamt $mt$ Kanten):

$$\Pi_j = \frac{d_j}{\sum_l d_l} = \frac{d_j}{2\,m\,t} \tag{67}$$

(Mittlere) Rate mit der sich Grad eines Knoten erhöht ($m$ Versuche pro Zeitschritt)

$$\frac{\partial\, d_j}{\partial\, t} = m\Pi_j \overset{(67)}{=} \frac{d_j}{2\,t}$$

$$\Rightarrow\ d_j(t) = m(t/t_j)^{0.5} \quad (\text{mit}\ d_j(t_j) = m) \tag{68}$$

Wahrscheinlichkeit, dass Knoten $j$ am Ende ($t = N$) einen Grad kleiner als $d$ hat:

$$\begin{aligned} P\left(d_j(N) < d\right) &\overset{(68)}{=} P\left(t_j > \frac{m^2 N}{d^2}\right) \\ &= 1 - P\left(t_j \leq \frac{m^2 N}{d^2}\right) \\ &\overset{(66)}{=} 1 - \frac{m^2}{d^2} \end{aligned}$$

$\rightarrow$ Verteilungsfunktion

$$p(d) = \frac{\partial P\left(d_j < d\right)}{\partial d} = \frac{2m^2}{d^3} \tag{69}$$

Weitere Graphentypen:

- Preferential attachment mit $\Pi_j \sim k_0 + d_j \rightarrow (k_0 > -m)\ \gamma = -3 - k_0/m$
- Kleine-Welt Graphen (“six degrees of separation´´)
- Erdös-Rhenyi Zufallsgraphen (Übungen)
- Regelmäßige Gitter
- Zufallsgraphen mit fester Nachbarzahl (“Bethe-Gitter´´)

# References

[1] A. M. Ferrenberg, D. P. Landau, and Y. J. Wong. Monte Carlo simulations: Hidden errors from ”good” random number generators. *Phys. Rev. Lett.*, 69:3382, 1992.

[2] B.J.T. Morgan. *Elements of Simulation*. Cambridge University Press, Cambridge, 1984.

[3] Alexander K. Hartmann. *Practical Guide to Computer Simulations*. World Scientific, Singapore, 2009.

[4] D. Stauffer and A. Aharony. *Perkolationstheorie*. Wiley-VCH, Weinheim, 1995.

[5] A. Dhar. Heat conduction in a one-dimensional gas of elastically colliding particles of unequal masses. *Phys. Rev. Lett.*, 86:3554, 2001.

[6] P. Grassberger, W. Nadler, and Lei Yang. Heat conduction and entropy production in a one-dimensional hard-particle gas. *Phys. Rev. Lett.*, 89:180601, 2002.

[7] L.E. Reichl. *A Modern Course in Statistical Physics*. John Wiley & Sons, New York, 1998.

[8] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A.H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *J. Chem. Phys.*, 21:1087, 1953.