

# Algorithmen für Chip-Entwurfswerkzeuge

## Einführung



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Vorlesung  
WS 2018/2019

Florian Stock

Eingebettete Systeme und Anwendungen  
Technische Universität Darmstadt



- ▶ Grundlage der Vorlesung
  - ▶ *Algorithms for VLSI Design Automation*  
Sabih H. Gerez, Wiley & Sons, 1998
  - ▶ *Electronic Design Automation*  
L.-T. Wang, Y.-W. Chang & K.-T. Cheng, Morgan-Kaufmann, 2009
- ▶ Wissenschaftliche Arbeiten („Papers“)
  - ▶ Größtenteils als Download auf der Vorlesungseite verfügbar
- ▶ Wissenstiefe
  - ▶ Kein perfektes Verständnis ...
  - ▶ ... aber Überblick über das Material
    - ▶ Fragen stellen!



- ▶ 3 CP
- ▶ Normale Prüfung zum Ende der Vorlesung
- ▶ Je nach Andrang mündlich oder schriftlich  
falls mündlich, Länge ca. 30 Minuten
- ▶ Dringend empfohlen:  
Das begleitende Praktikum für 6 CP



- ▶ Geplanter Zeitplan
  - ▶ Vorlesung:  
Immer Dienstags  
(anderer Raum?)
  - ▶ Praktikum:  
Blockpraktikum, beginnt je nach Stofffortschritt  
Erwartet: Anfang in zweiter Semesterhälfte, Ende in vorlesungsfreier Zeit
- ▶ Web-Seite
  - ▶ Fachgebiets-Webseite
  - ▶ Material und Ankündigungen
- ▶ Sprechstunde
  - ▶ Offene Tür
  - ▶ Mittwochs zwischen 14:30 - 15:30 Uhr

# Fragen?



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

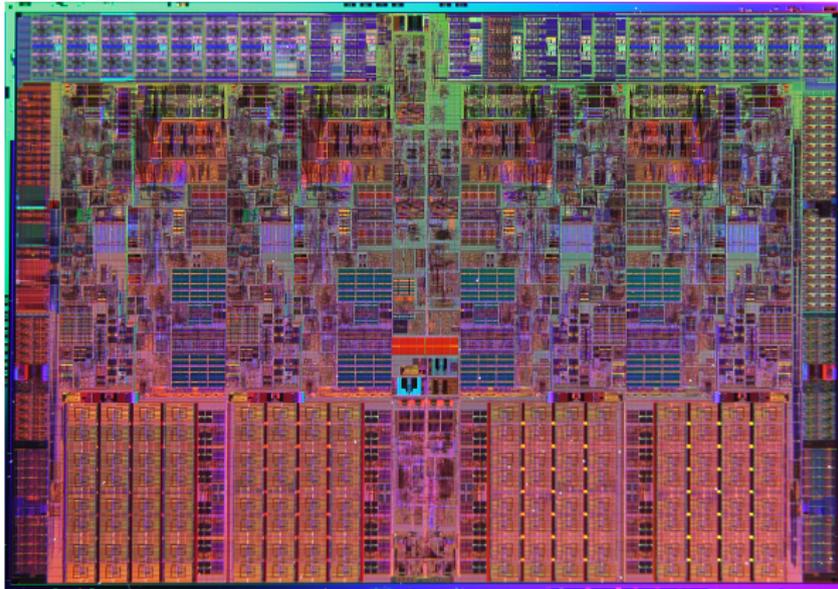
Noch Fragen zur Orga?



- ▶ VLSI Entwurf
  - ▶ Probleme
  - ▶ Bereiche
  - ▶ Tätigkeiten
  - ⇒ Werkzeuge
- ▶ Algorithmische Graphentheorie
  - ▶ Strukturen
  - ▶ Verfahren
- ▶ Mathematische Optimierungsverfahren
  - ▶ Heuristische Algorithmen
  - ▶ Exakte Algorithmen

# Ziel

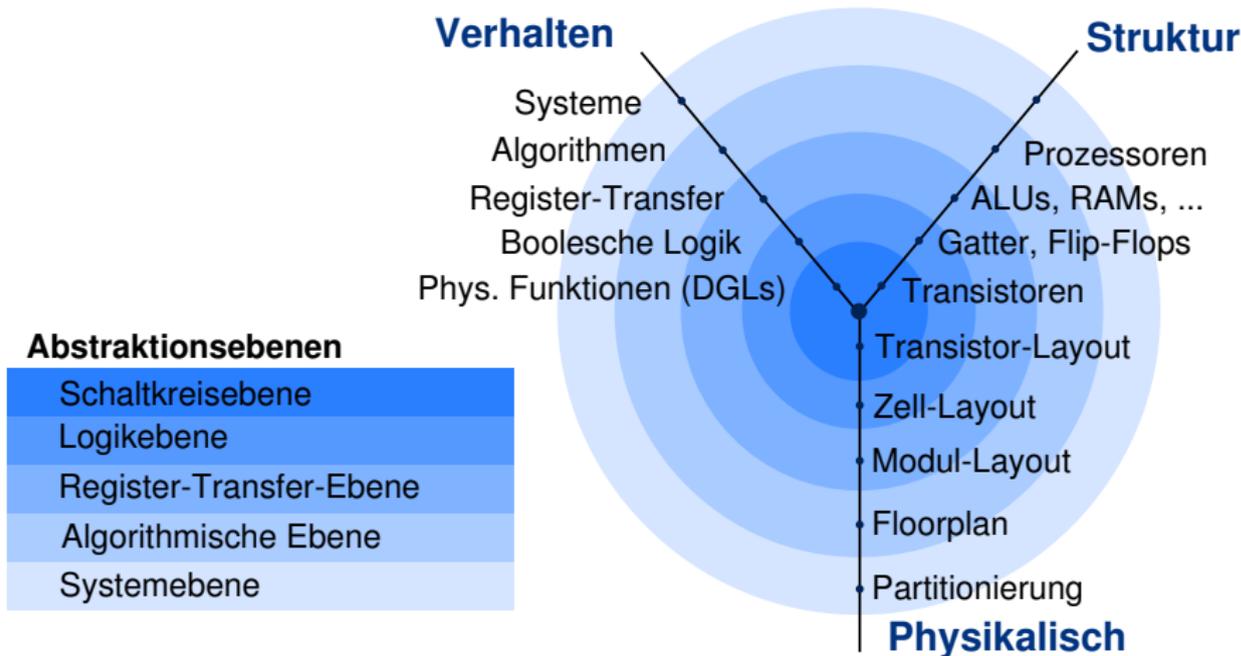
## Physikalischer Schaltkreis



Quelle: Intel (Nehalem)

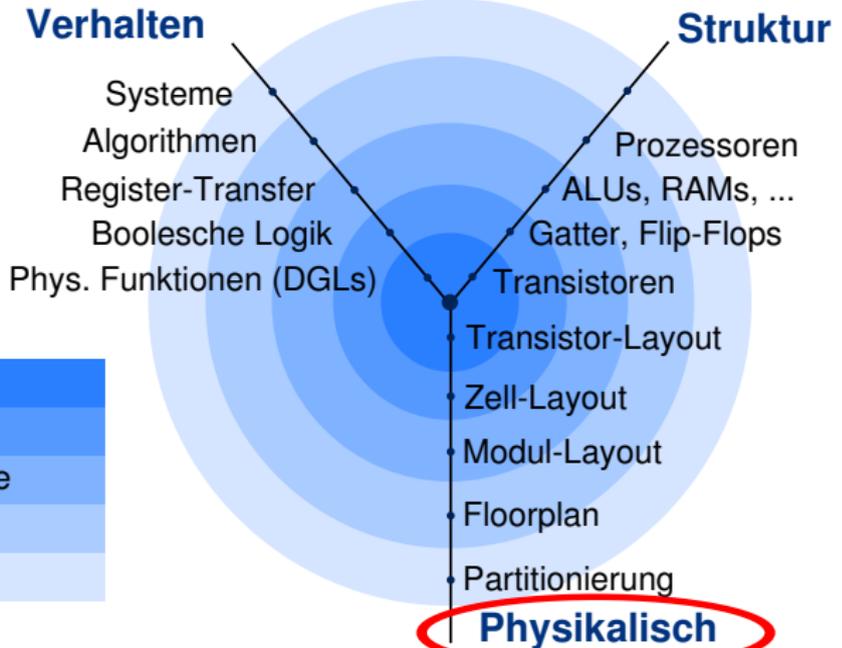


- ▶ “Implementiere eine Spezifikation in Hardware und optimiere dabei ...”
    - ▶ Fläche (min.)
    - ▶ Stromverbrauch (min.)
    - ▶ Geschwindigkeit (max. oder passend)
    - ▶ Entwurfszeit (min.)
    - ▶ Testbarkeit (max.)
  - ▶ “Alles auf einmal” ist zu komplex  
manche Ziele auch diametral
- ⇒ Aufteilen und vereinfachen
- ⇒ Qualitätseinbußen





- ▶ Synthese
  - ▶ Mehr Details durch Anwendung von Regeln
- ▶ Verifikation
  - ▶ Vergleiche Ergebnis mit Spezifikation
- ▶ Analyse
  - ▶ Untersuche Eigenschaften eines Ergebnisses
- ▶ Optimierung
  - ▶ Verbessere ein Ergebnis
- ▶ Datenverwaltung



## Abstraktionsebenen

Schaltkreisebene

Logikebene

Register-Transfer-Ebene

Algorithmische Ebene

Systemebene



- ▶ Basis: Algorithmische Grundlagen
- ▶ Layout-Synthese-Algorithmen entsprechend dem Hardware-Entwicklungsfluß  
... → Entwurf → **Layout-Synthese** → Layout-Verifikation → Fertigung → ...
- ▶ Layoutsynthese:
  1. Partitionierung
  2. Floorplanning
  3. Platzierung
  4. Verdrahtung
  5. Kompaktierung



- ▶ *Komplexitätstheorie*
- ▶ Graphen
  - ▶ Standardgraphen und Varianten
  - ▶ Datenstrukturen
  - ▶ Algorithmen
- ▶ Darstellungen von Schaltungen
- ▶ Optimierungsverfahren



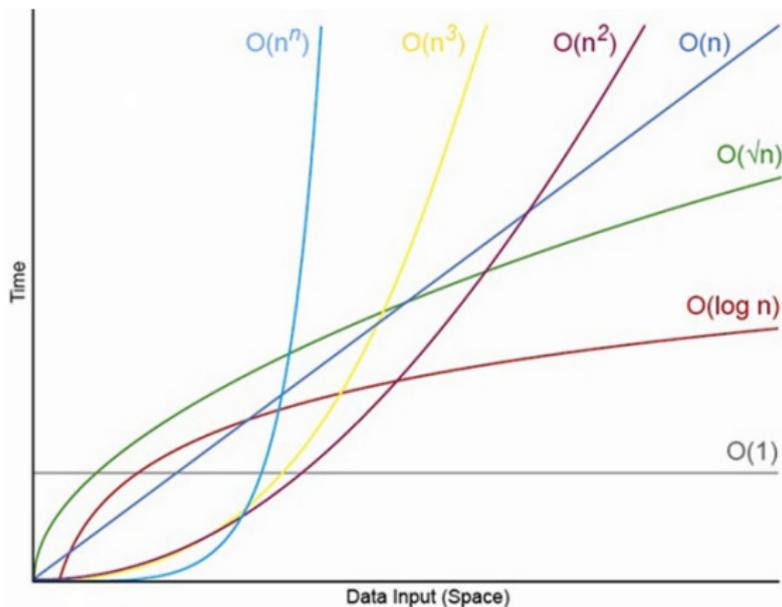
- ▶  $\mathcal{O}$  und  $\Theta$   
Siehe Grundstudium!
- ▶  $f \in \mathcal{O}(g)$   
 $f$  ist asymptotisch durch  $g$   
beschränkt
- ▶  $f \in \Theta(g)$   
 $f \in \mathcal{O}(g)$  und  $g \in \mathcal{O}(f)$
- ▶ Üblicherweise für Laufzeit benutzt  
kann aber auch für Speicher benutzt  
werden  
( $TIME \subseteq SPACE$ )

## Wichtige Ordnungen

- ▶ Exponentiell, z.B.  $2^n$
- ▶ Polynomial, z.B.  $n^3$
- ▶ Quadratisch, z.B.  $n^2$
- ▶ Superlinear, z.B.  $n \log(n)$
- ▶ Linear, z.B.  $n$
- ▶ Sublinear, z.B.  $\log(n)$ ,  $\sqrt{n}$
- ▶ Konstant, z.B. 1

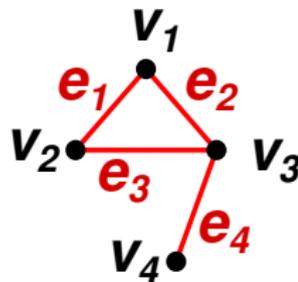
# Komplexitätsklassen

## Vergleich



Graph  $G(V, E)$

- ▶ Eine Menge  $V$  von Knoten (vertex)
- ▶ Eine Menge  $E$  von Kanten (edge)
  - ▶  $e = \{v_1, v_2\}$
  - ▶ Kante  $e$  verbindet Knoten  $v_1$  und  $v_2$
  - ▶ Kante ist ein Menge mit 2 Elementen



## Beispiel

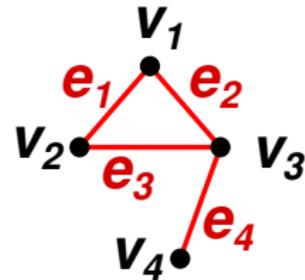
$G = (V, E)$

$V = \{v_1, v_2, v_3, v_4, v_5, v_6\}$

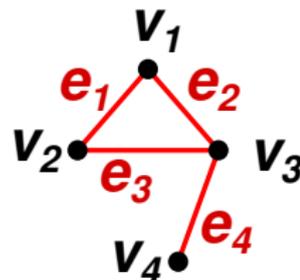
$E = \{e_1, e_2, e_3, e_4, e_5\}$  mit  $e_1 = \{v_1, v_2\}$ ,  $e_2 = \{v_1, v_3\}$ , ...



- ▶  $e = \{u, v\} \in E$ 
  - ▶  $e$  ist **inzident** mit  $u$   
(incident)
  - ▶  $e$  ist **inzident** mit  $v$   
(incident)
  - ▶  $u$  ist **adjazent** mit  $v$   
(adjacent)
- ▶ **Grad**  $g(v) = |\{e \in E \mid v \in e\}|$   
(degree)

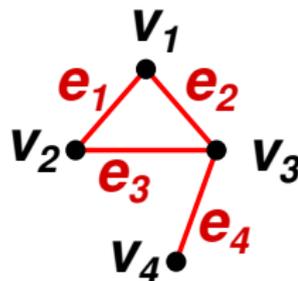


- ▶  $G(V, E)$  Graph
  - ▶ Graph  $G'(V', E')$  heißt **Teilgraph** von  $G$ , falls
    - ▶  $V' \subseteq V$ , und
    - ▶  $E' \subseteq E$ , und
    - ▶ weiterhin gilt:  
 $e = \{v_1, v_2\} \in E' \Rightarrow v_1, v_2 \in V'$
  - ▶ Anschaulich:
    - ▶ Entferne Knoten von  $G$ , und
    - ▶ Alle dazu inzidenten Kanten, und
    - ▶ Beliebige weitere Kanten
  - ▶ Werden nur die inzidenten Kanten entfernt:  
 $G'$  heißt dann (von Teilknotenmenge  $V'$ )  
**induzierter Teilgraph** oder **Untergraph**  
(induced subgraph)
- ▶  $G$  heißt **Supergraph** zu  $G'$



# Vollständigkeit und Cliques

- ▶ Komplette untereinander verbundene Knoten bilden einen **vollständigen** Graph (complete graph)  
( $|E| = \frac{|V|(|V|-1)}{2}$ )
- ▶ Induzierte Teilgraphen die vollständig sind heißen **Cliques**.
- ▶ Cliques die nicht Teilgraph von anderen Cliques sind, heißen **maximale Cliques**.



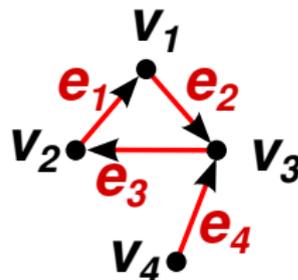


- ▶ In einfachen Graphen nicht zugelassen:
  - ▶ **Schlingen** (selfloop)  
Kante  $\{u, v\}$  mit  $u = v$
  - ▶ **Parallele Kanten**  
In **Multigraphen** erlaubt
  - ▶ Kanten  $e$  mit  $|e| \neq 2$   
In **Hypergraphen** erlaubt
- ▶ **Andere Erweiterungen:**
  - ▶ Zusätzliche Gewichte an Knoten oder Kanten  
**Gewichteter Graph** (weighted graph)
  - ▶ Kanten sind 2-Tupel (Paare) statt Menge: **Gerichteter Graph** (directed graph)

# Gerichteter Graph

## Definitionen

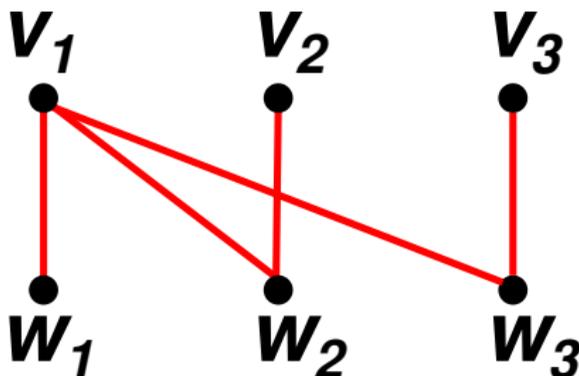
- ▶  $G(V, E)$  mit  $e = (u, v)$   $u, v \in E$ 
  - ▶  $e$  inzident von  $u$  (ausgehend)
  - ▶  $e$  inzident nach  $v$  (eingehend)
- ▶ **Außengrad** (out degree):  
Anzahl ausgehender Kanten
- ▶ **Innengrad** (in degree):  
Anzahl eingehender Kanten



# Spezielle Graphen

## Bipartite Graphen

- ▶ Kanten nur zwischen Knoten aus nicht-überlappenden Mengen
- ▶  $G = (V_1, V_2, E)$  ist **bipartiter** Graph
  - ▶  $V_1 \cap V_2 = \emptyset$
  - ▶  $E = \{\{u, w\} \mid u \in V_1 \wedge w \in V_2\}$



- ▶ Bei ungerichteten Graphen:

**Weg** Geordnete Folge von Knoten und Kanten

Beginnend und endend mit Knoten

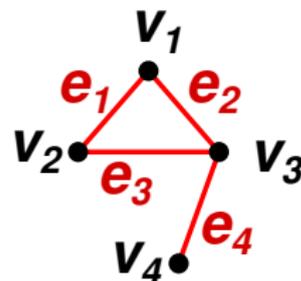
**Länge** Anzahl der Kanten

**Zyklus** Anfang = Ende

- ▶ Bei gerichteten Graphen:

**Gerichteter Weg**

**Gerichteter Zyklus**

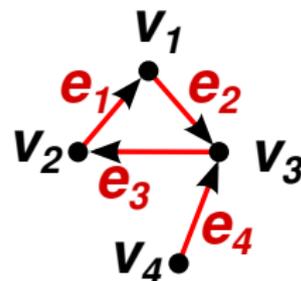


- ▶ Bei ungerichteten Graphen:

**Weg** Geordnete Folge von Knoten und Kanten  
Beginnend und endend mit Knoten

**Länge** Anzahl der Kanten

**Zyklus** Anfang = Ende



- ▶ Bei gerichteten Graphen:

**Gerichteter Weg**

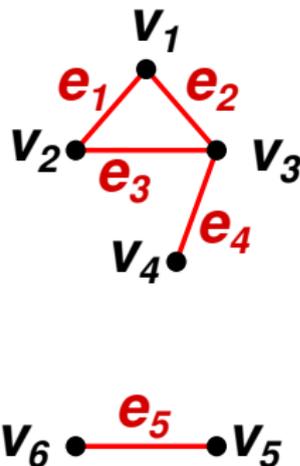
**Gerichteter Zyklus**



# Zusammenhang

## Ungerichteter Graph

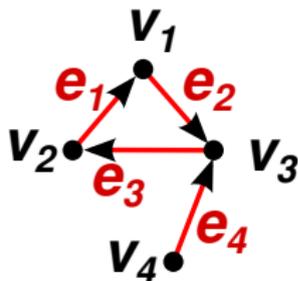
- ▶  $u$  hängt mit  $v$  zusammen,  $:\Leftrightarrow$   
Es gibt einen beide verbindenden Weg
- ▶ **Zusammenhängender** Graph:  
Alle Knoten hängen zusammen
- ▶ **Zusammenhangskomponente**  
Maximal zusammenhängende Teilgraphen



# Zusammenhang

## Gerichteter Graph

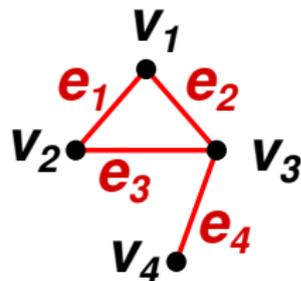
- ▶ **Starker Zusammenhang** von  $u$  und  $v$  zusammen,  $:\Leftrightarrow$   
Es gibt gerichteten Weg von  $u$  nach  $v$  und von  $v$  nach  $u$
- ▶ **Stark zusammenhängende** Komponenten:  
Alle enthaltenen Knoten hängen stark zusammen
- ▶ **Schwacher Zusammenhang**: Weg



# Datenstrukturen

## Adjazenzmatrix für Ungerichtete Graphen

- ▶ Adjazenzmatrix  $A_G$  von  $G(V, E)$ 
  - ▶  $n \times n$  Matrix mit  $n = |V|$
  - ▶  $A_{ij} = 1$  falls  $\{v_i, v_j\} \in E$ , sonst  $= 0$
  - ▶ Symmetrische Matrix
  - ▶ Statt 0 und 1 auch Gewichte möglich



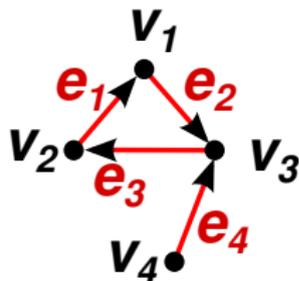
$$A_G = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$



# Datenstrukturen

## Adjazenzmatrix für Gerichtete Graphen

- ▶ Adjazenzmatrix  $A_G$  von  $G(V, E)$ 
  - ▶  $n \times n$  Matrix mit  $n = |V|$
  - ▶  $A_{ij} = 1$  falls  $(v_i, v_j) \in E$ , sonst = 0
  - ▶ Matrix nicht mehr symmetrisch
  - ▶ Statt 0 und 1 auch Gewichte möglich



$$A_G = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$



# Datenstrukturen

## Operationen auf Adjazenzmatrizen

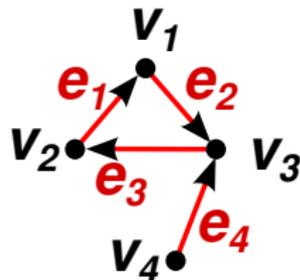


- ▶ Test, ob  $(v_i, v_j) \in E$ 
  - ▶ Nachsehen in  $A_{ij} : \mathcal{O}(1)$
- ▶ Welche  $v$  sind direkt mit  $u_i$  verbunden?
  - ▶ Zeile  $i$  durchgehen:  $\mathcal{O}(n)$
  - ▶ Ineffizient bei vielen Nullen
- ▶ Größenveränderung nur schwer möglich

# Datenstrukturen

## (Knoten-Kanten-) Inzidenzmatrix

- ▶ Inzidenzmatrix  $I_G$  von  $G(V, E)$ 
  - ▶  $m \times n$  Matrix mit  $n = |V|$ ,  $m = |E|$
  - ▶ Zeilen entsprechen Kanten, Spalten Knoten
  - ▶ Genau zwei nicht 0-Einträge pro Zeile
  - ▶ Bei ungerichteten Graphen:  
 $e_m = \{v_i, v_j\} \in E$ ,  $I_{mj} = 1$ ,  $I_{mi} = 1$
  - ▶ Bei gerichteten Graphen:  
 $e_m = (v_i, v_j) \in E$ ,  $I_{mj} = 1$ ,  $I_{mi} = -1$



$$I_G = \begin{pmatrix} 1 & -1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 \end{pmatrix}$$



- ▶ Array aus Listen
  - ▶ Knotennummer ist Index
- ▶ Listenelemente
  - ▶ Index des Zielknotens
  - ▶ Verkettung
- ▶ Test, ob  $(u, v) \in E$  unabhängig von  $n$   
abhängig vom durchschnittlichen Außengrad  $k$ :  $\mathcal{O}(k)$
- ▶ Kanten nur implizit gespeichert:  
Ggf. explizite Knoten- und Kantenmodellierung notwendig!



- ▶ Aufgabe
  - ▶ *Besuche alle  $V$  und  $E$  von  $G(V, E)$ !*
  - ▶ Jedes Element genau einmal!
- ▶ Unterschiedliche Reihenfolgen möglich
- ▶ Weit verbreitet
  - Tiefensuche Suche von Ursprungsknoten entfernen
  - Breitensuche Erstmal angrenzende Knoten bearbeiten

# Graphen Traversierung

## Tiefensuche (DFS) – Praktisch



```
dfs(vertex v)
```

```
begin
```

```
  v.mark := 0;
```

```
  v.process();
```

```
  foreach (v,u) ∈ E do
```

```
    (v,u).process();
```

```
    if (u.mark) then dfs(u);
```

```
  ;
```

```
main()
```

```
begin
```

```
  foreach v ∈ V do
```

```
    v.mark := 1;
```

```
  foreach v ∈ V do
```

```
    if (v.mark) then dfs(v);
```

```
  ;
```

# Graphen Traversieren

## Tiefensuche (DFS) – Theoretisch

- ▶ Komplexität für DFS auf  $G(V, E)$ 
  - ▶ Jeder Knoten einmal besucht
  - ▶ Jede Kante einmal besucht $\Rightarrow \mathcal{O}(|V| + |E|)$
- ▶ Anwendungsbeispiele
  - ▶ Systematischer Graphdurchlauf
  - ▶ Finden der von einem Startknoten aus erreichbaren Knoten
    - ▶ Ersetze Schleife in `main()` durch einfachen Aufruf

# Graphen Traversierung

## Breitensuche (BFS) – Praktisch 1

bfs(vertex v)

**begin**

FIFO Q := ();

vertex u, w;

Q.shift\_in(v);

v.mark := 0;

**repeat**

    w := Q.shift\_out();

    w.process();

**foreach** (w,u) ∈ E **do**

        (w,u).process();

**if** (u.mark) **then**

            u.mark := 0;

            Q.shift\_in(u);

**until** Q == ();

main()

**begin**

**foreach** v ∈ V **do**

        v.mark := 1;

**foreach** v ∈ V **do**

**if** (v.mark) **then** bfs(v);

        ;

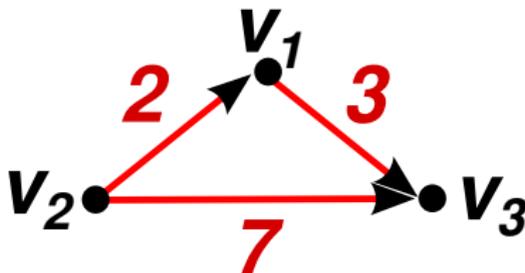
# Graphen Traversierung

## Breitensuche (BFS) – Theoretisch

- ▶ Komplexität für BFS auf  $G(V, E)$ 
  - ▶ Jeder Knoten einmal besucht
  - ▶ Jede Kante einmal besucht

⇒  $\mathcal{O}(|V| + |E|)$
- ▶ Anwendungsbeispiele
  - ▶ Systematischer Graphdurchlauf
  - ▶ Finden der von einem Startknoten aus erreichbaren Knoten
  - ▶ Besuche Knoten in Reihenfolge der Entfernung vom Startknoten

- ▶ **Aufgabe:** Bestimme den kürzesten Pfad vom Startknoten zu Zielknoten
  - ▶ Manchmal auch: zu allen anderen Knoten
- ▶ Bei ungewichteten Graphen z.B. mit BFS
  - ▶ Erweitert um Verwaltung der Pfade
- ▶ BFS nicht bei gewichteten Graphen!
  - ▶ Niedrige Anzahl von Kanten nicht immer kürzester (leichtester) Weg



# Kürzester Pfad

## Dijkstra-Algorithmus

**dijkstra**(set<vertex>  $V$ , vertex  $v_s$ , vertex  $v_t$ )

```
set<vertex> T;
```

```
vertex u, v;
```

```
 $V := V - \{v_s\}$ ;
```

```
T := {  $v_s$  };
```

```
 $v_s.dist := 0$ ;
```

```
foreach  $u \in V$  do
```

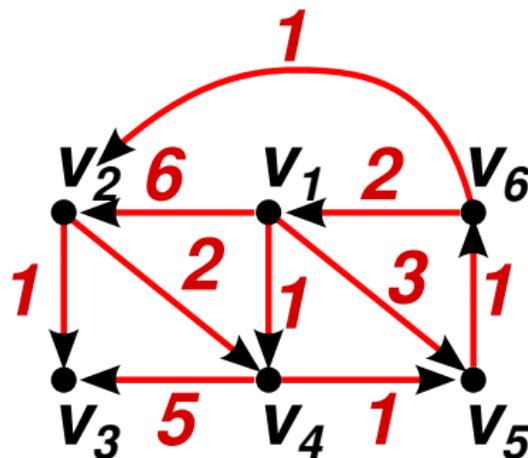
```
  if  $((v_s, u) \in E)$  then
```

```
    |  $u.dist := (v_s, u).weight$ ;
```

```
  else  $u.dist := +\infty$ ;
```

```
  ;
```

```
  ;
```





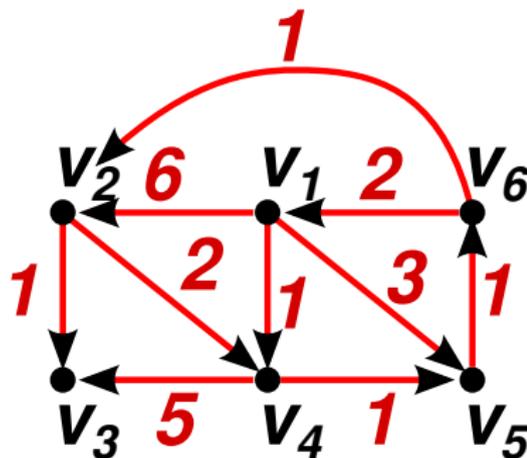
# Kürzester Pfad

## Dijkstra-Algorithmus

**dijkstra**(set<vertex>  $V$ , vertex  $v_s$ , vertex  $v_t$ )

```
set<vertex> T;  
vertex u, v;  
V := V - {v_s };  
T := { v_s };  
v_s.dist := 0;  
foreach u ∈ V do  
  if ((v_s, u) ∈ E) then  
    | u.dist := (v_s, u).weight;  
  else u.dist := +∞;  
  ;
```

⋮



Gesucht: Kürzester Pfad  $v_1 \mapsto v_3$   
 $T = \{v_1\}$       $V = \{v_2, v_3, v_4, v_5, v_6\}$   
 $v_j.dist = 0 \quad 6 \quad \infty \quad 1 \quad 3 \quad \infty$

# Kürzester Pfad

## Dijkstra-Algorithmus

dijkstra(set<vertex> V, vertex  $v_s$ , vertex  $v_t$ )

⋮

while ( $v_t \notin T$ ) do

$u := V.\text{findmin}(\text{dist});$

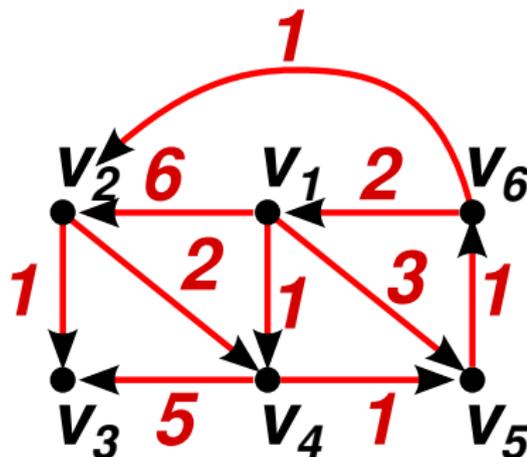
$T := T \cup \{u\};$

$V := V - \{u\};$

    foreach  $(u,v) \in E$  do

        if ( $v.\text{dist} > u.\text{dist} + (u,v).\text{weight}$ ) then

$v.\text{dist} := u.\text{dist} + (u,v).\text{weight};$



Gesucht: Kürzester Pfad  $v_1 \mapsto v_3$

$T = \{v_1\}$        $V = \{v_2, v_3, v_4, v_5, v_6\}$

$v_j.\text{dist} = 0 \quad 6 \quad \infty \quad 1 \quad 3 \quad \infty$

# Kürzester Pfad

## Dijkstra-Algorithmus

dijkstra(set<vertex> V, vertex  $v_s$ , vertex  $v_t$ )

⋮

while ( $v_t \notin T$ ) do

$u := V.\text{findmin}(\text{dist});$

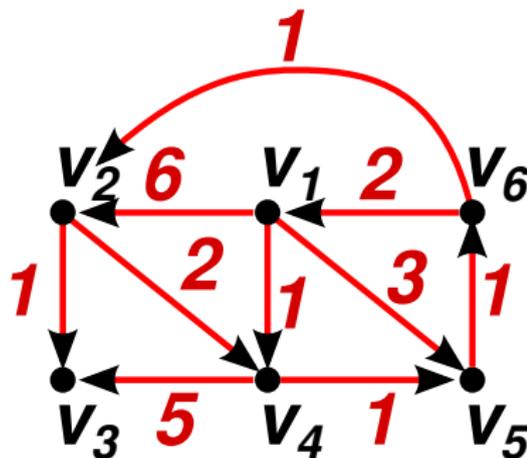
$T := T \cup \{u\};$

$V := V - \{u\};$

    foreach ( $(u,v) \in E$ ) do

        if ( $v.\text{dist} > u.\text{dist} + (u,v).\text{weight}$ ) then

$v.\text{dist} := u.\text{dist} + (u,v).\text{weight};$



Gesucht: Kürzester Pfad  $v_1 \mapsto v_3$

$T = \{v_1, v_4\}$        $V = \{v_2, v_3, v_5, v_6\}$

$v_j.\text{dist} = 0 \quad 6 \quad \infty \quad 1 \quad 3 \quad \infty$



# Kürzester Pfad

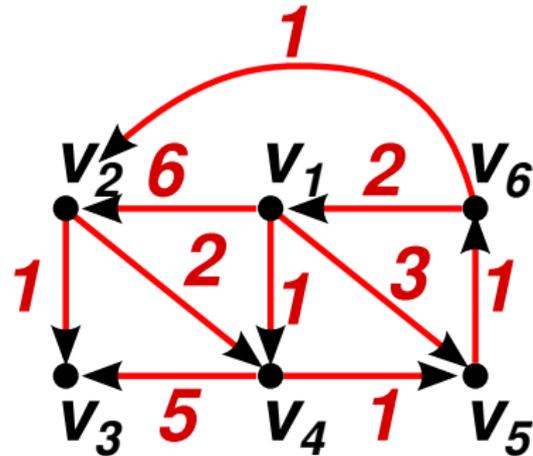
## Dijkstra-Algorithmus

dijkstra(set<vertex> V, vertex  $v_s$ , vertex  $v_t$ )

⋮

```

while ( $v_t \notin T$ ) do
  u := V.findmin(dist);
  T := T ∪ {u};
  V := V - {u};
  foreach (u,v) ∈ E do
    if ( $v.dist > u.dist + (u,v).weight$ ) then
      | v.dist := u.dist + (u,v).weight;
  
```



Gesucht: Kürzester Pfad  $v_1 \mapsto v_3$

$T = \{v_1, v_4, v_5\}$        $V = \{v_2, v_3, v_6\}$

$v_j.dist = 0 \quad 6 \quad 6 \quad 1 \quad 2 \quad \infty$

# Kürzester Pfad

## Dijkstra-Algorithmus

dijkstra(set<vertex> V, vertex  $v_s$ , vertex  $v_t$ )

⋮

while ( $v_t \notin T$ ) do

$u := V.\text{findmin}(\text{dist});$

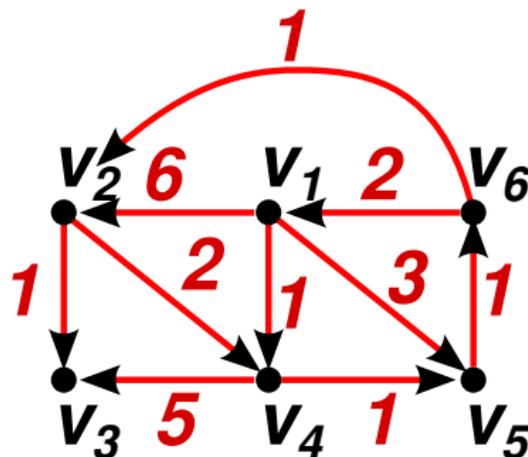
$T := T \cup \{u\};$

$V := V - \{u\};$

    foreach  $(u,v) \in E$  do

        if ( $v.\text{dist} > u.\text{dist} + (u,v).\text{weight}$ ) then

$v.\text{dist} := u.\text{dist} + (u,v).\text{weight};$



Gesucht: Kürzester Pfad  $v_1 \mapsto v_3$

$T = \{v_1, v_4, v_5\}$        $V = \{v_2, v_3, v_6\}$

$v_j.\text{dist} = 0 \quad 6 \quad 6 \quad 1 \quad 2 \quad 3$

# Kürzester Pfad

## Dijkstra-Algorithmus

dijkstra(set<vertex> V, vertex  $v_s$ , vertex  $v_t$ )

⋮

while ( $v_t \notin T$ ) do

$u := V.\text{findmin}(\text{dist});$

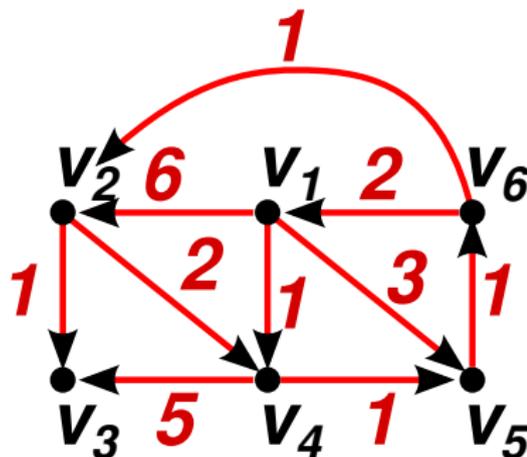
$T := T \cup \{u\};$

$V := V - \{u\};$

    foreach ( $(u,v) \in E$ ) do

        if ( $v.\text{dist} > u.\text{dist} + (u,v).\text{weight}$ ) then

$v.\text{dist} := u.\text{dist} + (u,v).\text{weight};$



Gesucht: Kürzester Pfad  $v_1 \mapsto v_3$

$T = \{v_1, v_4, v_5, v_6\}$        $V = \{v_2, v_3\}$

$v_j.\text{dist} = 0 \quad 6 \quad 6 \quad 1 \quad 2 \quad 3$

# Kürzester Pfad

## Dijkstra-Algorithmus

dijkstra(set<vertex> V, vertex  $v_s$ , vertex  $v_t$ )

⋮

while ( $v_t \notin T$ ) do

$u := V.\text{findmin}(\text{dist});$

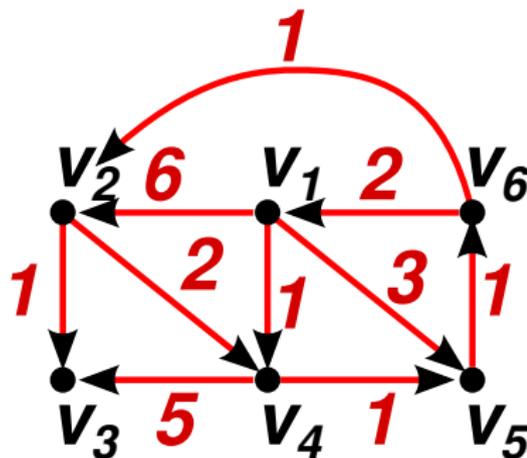
$T := T \cup \{u\};$

$V := V - \{u\};$

    foreach  $(u,v) \in E$  do

        if ( $v.\text{dist} > u.\text{dist} + (u,v).\text{weight}$ ) then

$v.\text{dist} := u.\text{dist} + (u,v).\text{weight};$



Gesucht: Kürzester Pfad  $v_1 \mapsto v_3$

$T = \{v_1, v_4, v_5, v_6\}$        $V = \{v_2, v_3\}$

$v_j.\text{dist} = 0 \quad 4 \quad 6 \quad 1 \quad 2 \quad 3$



# Kürzester Pfad

## Dijkstra-Algorithmus

dijkstra(set<vertex> V, vertex  $v_s$ , vertex  $v_t$ )

⋮

while ( $v_t \notin T$ ) do

$u := V.\text{findmin}(\text{dist});$

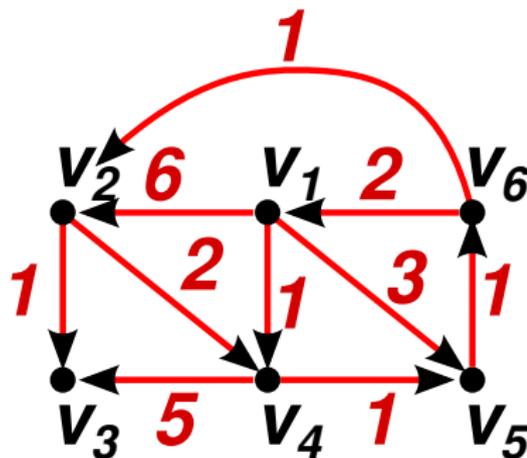
$T := T \cup \{u\};$

$V := V - \{u\};$

    foreach ( $(u,v) \in E$ ) do

        if ( $v.\text{dist} > u.\text{dist} + (u,v).\text{weight}$ ) then

$v.\text{dist} := u.\text{dist} + (u,v).\text{weight};$



Gesucht: Kürzester Pfad  $v_1 \mapsto v_3$

$T = \{v_1, v_4, v_5, v_6, v_2\}$        $V = \{v_3\}$

$v_j.\text{dist} = 0 \quad 4 \quad 5 \quad 1 \quad 2 \quad 3$

# Kürzester Pfad

## Dijkstra-Algorithmus

dijkstra(set<vertex> V, vertex  $v_s$ , vertex  $v_t$ )

⋮

while ( $v_t \notin T$ ) do

$u := V.\text{findmin}(\text{dist});$

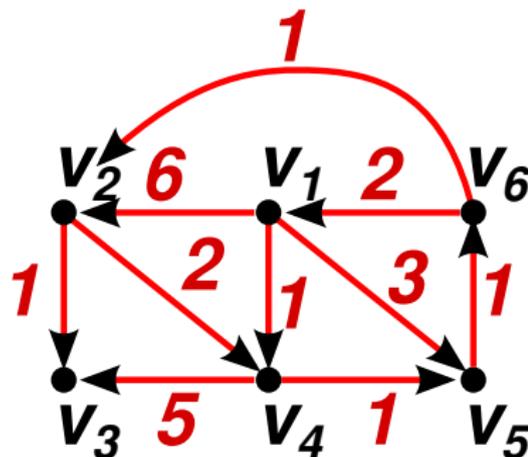
$T := T \cup \{u\};$

$V := V - \{u\};$

    foreach ( $(u,v) \in E$ ) do

        if ( $v.\text{dist} > u.\text{dist} + (u,v).\text{weight}$ ) then

$v.\text{dist} := u.\text{dist} + (u,v).\text{weight};$



Gesucht: Kürzester Pfad  $v_1 \mapsto v_3$

$T = \{v_1, v_4, v_5, v_6, v_2, v_3\}$        $T = \emptyset$

$v_j.\text{dist} = 0 \quad 4 \quad 5 \quad 1 \quad 2 \quad 3$

# Kürzester Pfad

## Dijkstra – Theoretisch



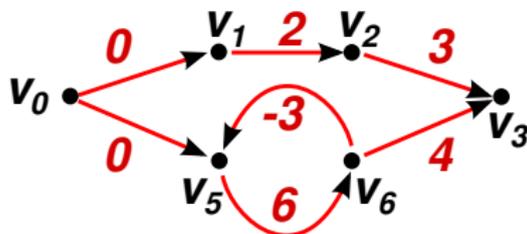
- ▶ Komplexität
  - ▶ while ( $v \notin T$ ):  $|V|$ -mal durchlaufen
    - ▶  $V.\text{findmin}(\text{dist})$ :  $\mathcal{O}(|V|)$  je Suche  
 $\Rightarrow \mathcal{O}(|V|^2)$
  - ▶ foreach  $(u, v) \in E$ :  $|E|$ -mal insgesamt
    - ▶ Einfacher Graph hat max.  $|V|^2$  Kanten  
 $\Rightarrow \mathcal{O}(|V|^2)$
  - ▶ Gesamtaufwand  $\mathcal{O}(|V|^2 + |V|^2) = \mathcal{O}(|V|^2)$
- ▶ **Wichtig:** Funktioniert nur bei Kantengewichten  $\geq 0$ !

Algorithmus abhängig vom Graphen:

Zyklusfreier Graph: OK, ähnlich zu BFS

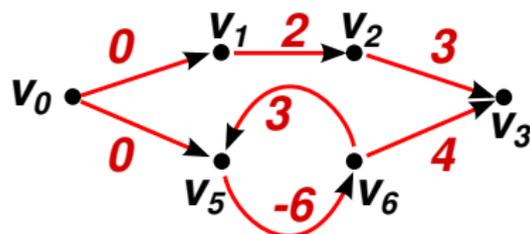
Graph mit Zyklen: Unterscheidung nach Zyklusart

Mit positivem Zyklus



Undefiniert!

Mit negativem Zyklus



OK

# Längster Weg

## Zyklenfreie Graphen (analog zu BFS)

```
main()  
begin  
  foreach  $0 \leq i < n$  do  
     $x_i := 0$   
  longestPath(G,  $v_s$ ) ;
```

Längster Pfad von  $v_s$  zu allen  
anderen Knoten

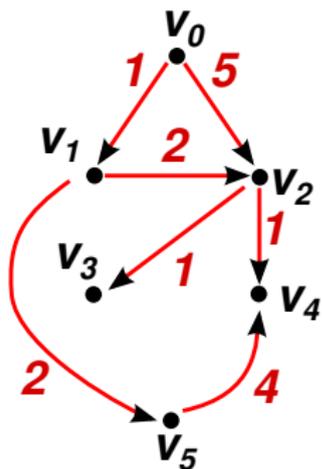
**Voraussetzung:**

Graph ist ein DAG  
(Directed Acyclic Graph)!

```
longestPath(G, vertex  $v_s$ ):  
begin  
  foreach  $v_i$  in  $V$  do  
     $p_i := v_i.inDegree()$   
  Set  $Q := \{v_s\}$ ;  
  while ( $Q \neq \emptyset$ ) do  
     $v_i := Q.pickany()$ ;  
     $Q := Q \setminus \{v_i\}$  ;  
    foreach  $(v_i, v_j) \in E$  do  
       $x_j := \max(x_j, x_i + d_{ij})$ ;  
       $p_j := p_j - 1$  ;  
      if  $p_j \leq 0$  then  
         $Q := Q \cup \{v_j\}$ 
```

# Längster Pfad DAG Beispiel

Längster Pfad von  $v_0$  zu allen anderen Knoten



Q	$p_1$	$p_2$	$p_3$	$p_4$	$p_5$	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$
Init	1	2	1	2	1	0	0	0	0	0
$\{v_0\}$	0	1	1	2	1	1	5	0	0	0
$\{v_1\}$	0	0	1	2	0	1	5	0	0	3
$\{v_2, v_5\}$	0	0	0	1	0	1	5	6	6	3
$\{v_3, v_5\}$	0	0	0	1	0	1	5	6	6	3
$\{v_5\}$	0	0	0	0	0	1	5	6	7	3
$\{v_4\}$	0	0	0	0	0	1	5	6	7	3



- ▶ Nur mit *negativen* Zyklen
- ▶ Erkenne positive Zyklen
- ▶ Aber lokalisere sie nicht

```
foreach  $0 \leq i < n$  do
   $x_i := -\infty$ 
 $x_0 := 0$  ; loop_count = 0 ;
repeat
  is_modified := false ;
  longestPath( $G_f$ ) ;
  foreach  $(v_i, v_j) \in E_b$  do
    if  $x_j < (x_i + d_{ij})$  then
       $x_j := x_i + d_{ij}$  ;
      is_modified := true ;
  if  $(++loop\_count > |E_b| \ \&\& \ is\_modified)$ 
  then
    error("positive cycle!");
until ! is_modified;
```

Idee: Zyklen auftrennen

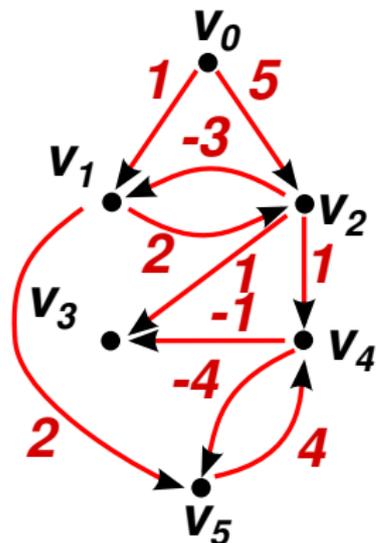
- ▶ Kantenmenge  $E$  aufteilen in  $E_f$  und  $E_b$  mit  
 $E = E_f \cup E_b$ ,  $E_f \cap E_b = \emptyset$   
und  $E_f \gg E_b$

⇒ Teilgraph  $G_f(V, E_f)$

- ▶ Löse LongestPath( $G_f$ )
- ▶ Korrigiere für entfernte  $E_b$  (Zyklen schließen)
- ▶ Jedes  $e_b \in E_b$  max.  $1 \times$  im Pfad  
⇒ stabilisiert sich in  $|E_b|$
- ▶ Wenn nicht  
⇒ überbeschränkt

# Längster Pfad

## Liao-Wong Beispiel



Schritt	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$
Init	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
Vor 1	1	5	6	7	3
Zurück 2	2	5	6	7	3
Vor 2	2	5	6	8	4
Zurück 2	2	5	7	8	4
Vor 3	2	5	7	8	4
Zurück 3	2	5	7	8	4

- ▶ Verbesserung:  $\text{longestPath}(G_f)$  bemerkt Änderung
- ▶  $\mathcal{O}(|E_f| \times |E_b|)$   
d.h. besonders gut, falls  $|E_b| \ll |E_f|$

# Längster Pfad

## Bellman-Ford

```
foreach  $0 \leq i < n$  do
   $x_i := -\infty$ 
 $x_0 := 0$  ; loop_count = 0 ;
 $S_1 := \{v_0\}$  ;  $S_2 := \emptyset$  ;
while loop_count  $\leq n$  &&  $S_1 \neq \emptyset$  do
  foreach  $v_i \in S_1$  do
    foreach  $(v_i, v_j) \in E$  do
      if  $x_j < (x_i + d_{ij})$  then
         $x_j := x_i + d_{ij}$  ;
         $S_2 := S_2 \cup \{v_j\}$  ;
     $S_1 := S_2$  ;  $S_2 := \emptyset$  ;
    ++ loop_count ;
  if loop_count  $> n$  then error("positive cycle!");
```

Idee: Zwei Wellenfronten

$S_1$  aktuelle

$S_2$  nächste Iteration

▶ Vergleichbar azyklischem LP

aber mehrere Durchläufe

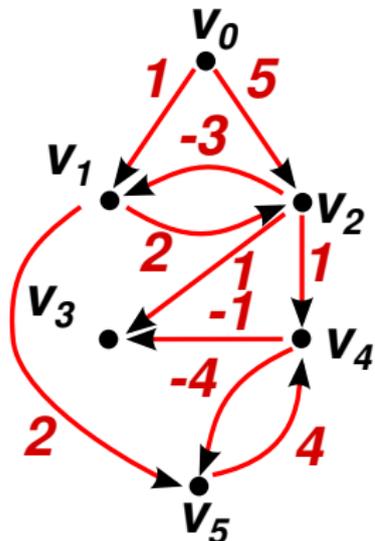
▶ In  $k$ -ter Iteration LP durch  $k - 1$  Knoten

⇒ Zyklendetektion  
LP  $> n$  Knoten ⇒ Zyklus!

▶  $\mathcal{O}(n^3)$ , avg.  $\mathcal{O}(n^{1.5})$

# Längster Pfad

## Bellman-Ford Beispiel



$S_1$	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$
	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
$\{V_0\}$	1	5	$-\infty$	$-\infty$	$-\infty$
$\{V_1, V_2\}$	2	5	6	6	3
$\{V_1, V_3, V_4, V_5\}$	2	5	6	7	4
$\{V_4, V_5\}$	2	5	6	8	4
$\{V_4\}$	2	5	7	8	4
$\{V_3\}$	2	5	7	8	4

# Pfadalgorithmen

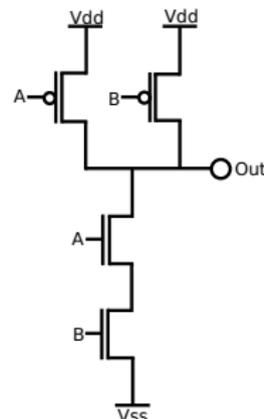
## Übersicht



- ▶  $LP \leftrightarrow SP$  bei Multiplikation der Gewichte mit  $-1$
- ▶ Gerichtete zyklensfreie Graphen (DAG)  
SP und LP lösbar in linearer Zeit
- ▶ Gerichtete Graphen mit Zyklen
  - ▶ Alle Gewichte positiv  
 $\Rightarrow$  SP in P, LP ist NP-vollständig
  - ▶ Alle Gewichte negativ  
 $\Rightarrow$  LP in P, SP ist NP-vollständig
  - ▶ Keine positiven Zyklen: LP in P
  - ▶ Keine negativen Zyklen: SP in P
  - ▶ Sonst: NP-Vollständig

# Sichten

## Schematisch und Transistorlayout



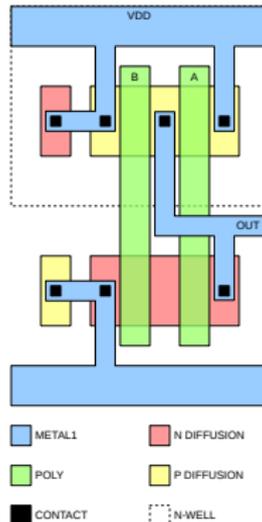
Bildquelle: Wikimedia Commons

Schematisches Schaltsymbol

Transistorlayout

# Sichten

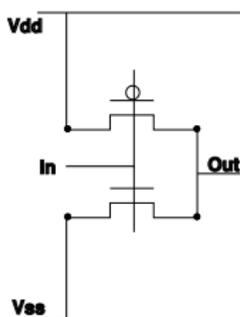
## Physikalisches/Geometrisches/Masken Layout



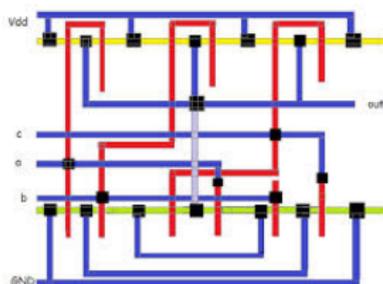
Bildquelle: Wikimedia Commons

# Sichten

## Symbolisches Layout



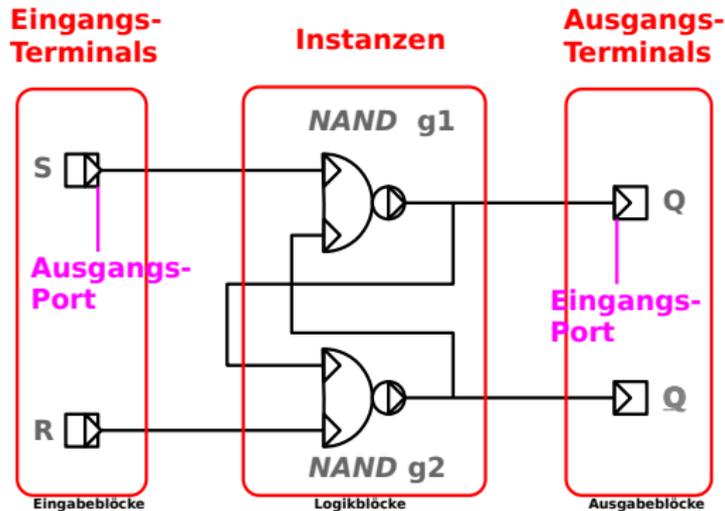
Symbolisches Layout



Stick-Diagramm

- ▶ Kein vollständiges Layout
- ▶ Keine absoluten geometrischen Angaben
- ▶ Nicht notwendige physikalische Angaben fehlen komplett (z.B. n- und p-Wellen)
- ▶ *Symbole* für Elemente wie Transistoren oder Kontakte
- ▶ Länge, Breite, Layer noch variabel

# Darstellungen von Schaltungen



# Zelle und Master-Zelle



```
class cell_master {  
    String name;  
    truth_table func;  
    Rect extent;  
    set<port_master> ins , outs ;  
    ...  
}
```

```
class cell {  
    cell_master master;  
    String name;  
    set<port> ins , outs ;  
    ...  
}
```



```
class port_master {  
    String name;  
    Point location;  
    ...  
}  
  
class port {  
    port_master master;  
    String id;  
    cell parent;  
    net connects;  
    ...  
}
```



```
class net {  
    String name;  
    set<port> joined;  
}
```

**Instanz oder Zelle** ▶ Ein Auftreten einer *Master-Zelle*  
▶ Speichert instanzspezifische Eigenschaften

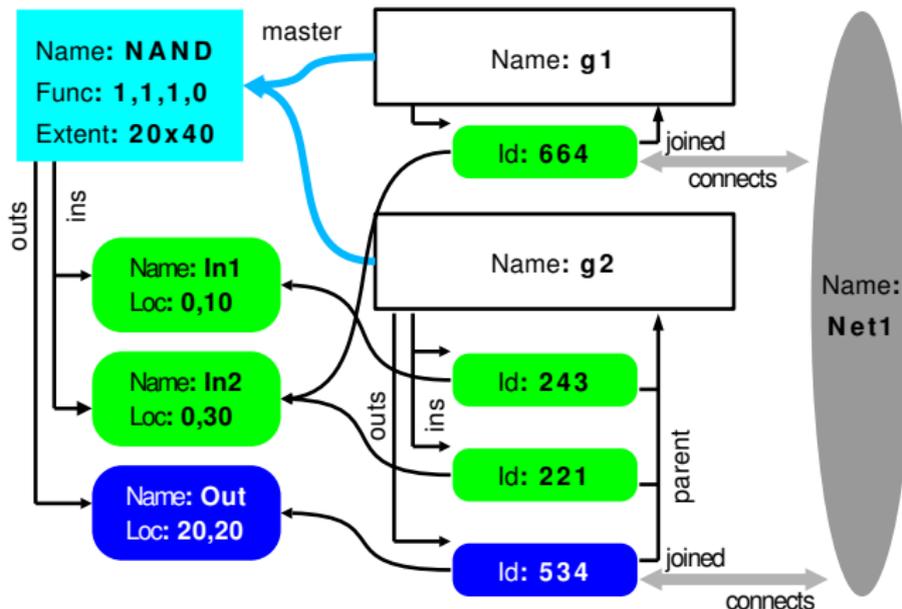
**Master-Zelle** Speichert Eigenschaften aller Instanzen

**Netz** Verbindung von mehreren Ports

**Port** ▶ Anschlusspunkt von Leitung an Zelle  
▶ I.d.R nicht untereinander austauschbar  
▶ Hierarchie: Terminals werden zu Ports

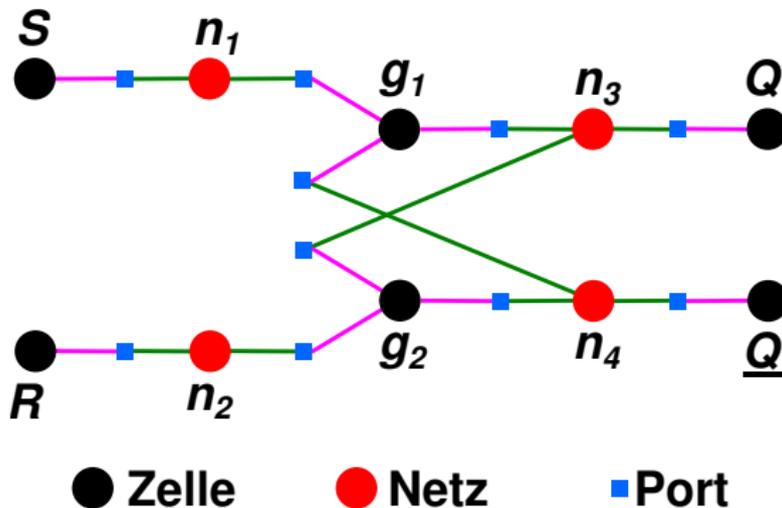
# Schaltungsdarstellung

## Beispiel



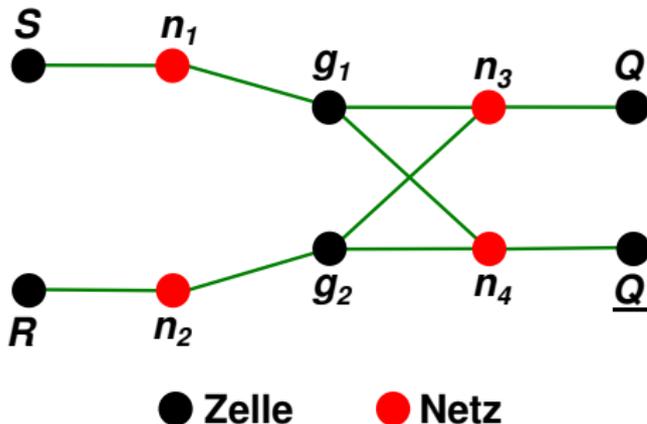
# Schaltungsdarstellung – Graphmodellierung

## Tripartiter Graph



# Schaltungsdarstellung – Graphmodellierung

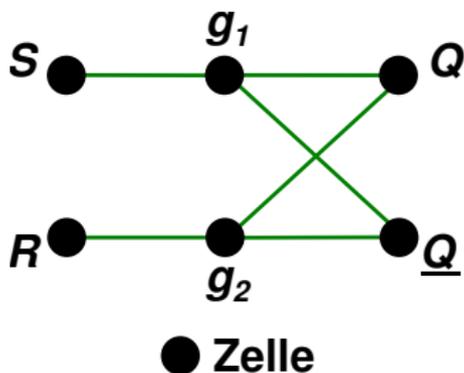
## Bipartiter Graph



- ▶ Weniger Details
- ▶ Verschmelzt Ports mit Zellen
- ▶ Äquivalent zu Hypergraph

# Schaltungsdarstellung

## Graphmodellierung



- ▶ Netze nicht mehr explizit modelliert
- ▶ Zellen an Netzen bilden jetzt Clique

# Schaltungsdarstellungen

## Übersicht

- ▶ Zelle-Port-Netz-Modell
- ▶ Tripartiter Graph
- ▶ Bipartiter Graph
- ▶ Clique-Modell



ungenauer

## Für Problem passendes Modell wählen

- ▶ Mehr Daten nicht immer besser

## Konvertierungsroutinen bereitstellen

- ▶ Nur in ungenauere Darstellung möglich
- ▶ Buchführung über Herkunft von Daten



- ▶ VLSI
  - ▶ Entwurfsbereiche
  - ▶ Tätigkeiten
  - ▶ Werkzeuge
- ▶ Hierarchie und Abstraktion
- ▶ Graphentheorie
  - ▶ Konzepte und Begriffe
  - ▶ Datenstrukturen
  - ▶ Algorithmen: DFS, BFS, SP, LP
- ▶ Schaltungsdarstellung