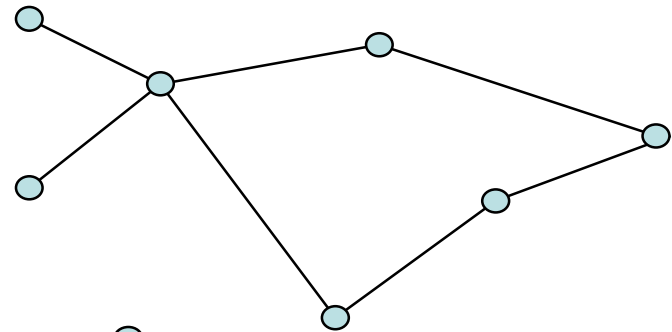


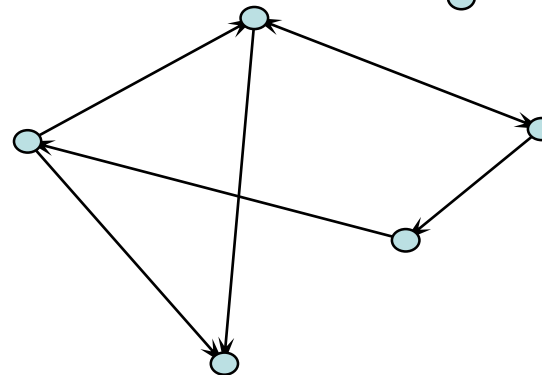
Graphen sind eine dominierende Datenstruktur in der Informatik

Viele Probleme der Informatik lassen sich durch Graphen beschreiben
und über Graphenalgorithmien lösen

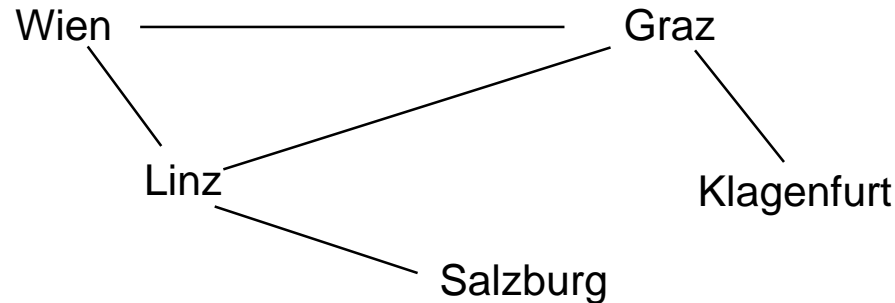
Ungerichteter Graph



Gerichteter Graph



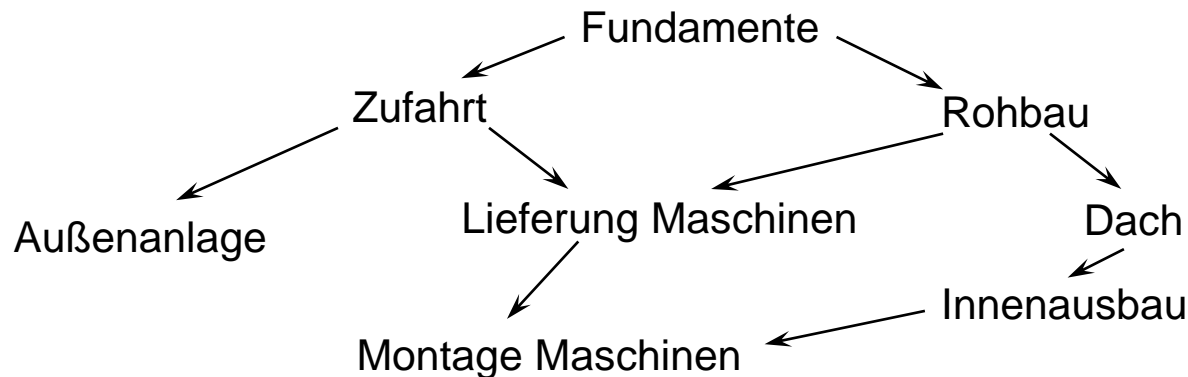
Ortsverbindungen
z.B. Züge



Ablaufbeschreibungen

z.B. Projektplanung Maschinenhalle

Kanten rep. Abhängigkeiten



6.1 Ungerichteter Graph



Ein **ungerichteter Graph** $G = (V, E)$ besteht aus einer Menge V (vertex) von **Knoten** und einer Menge E (edge) von **Kanten**, d.h.

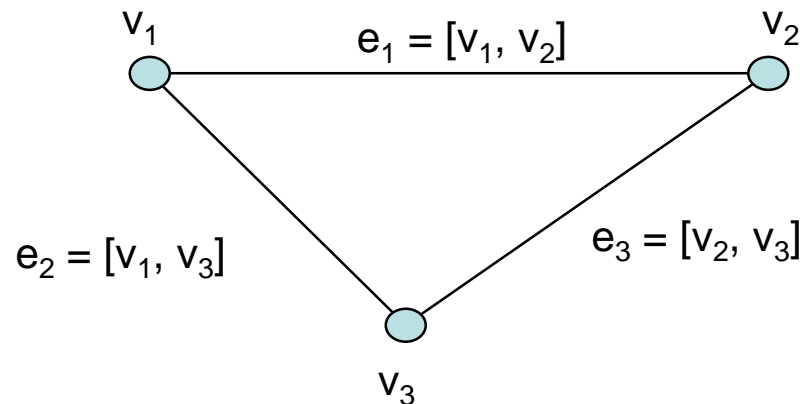
$$V = \{v_1, v_2, \dots, v_{|V|}\}, \mathbf{n} = |V|$$

$$E = \{e_1, e_2, \dots, e_{|E|}\}, \mathbf{m} = |E|$$

Eine **Kante** e ist ein ungeordnetes Paar von Knoten aus V , d.h. $e = [v_i, v_j]$ mit $v_i, v_j \in V$ und $v_i \neq v_j$. v_i und v_j sind an e **beteiligt**.

Die Anzahl der Kanten, an denen ein Knoten beteiligt ist, ist der (**Knoten-)****Grad** (*degree*) des Knotens.

$$\begin{aligned} G &= (V, E) \\ V &= \{v_1, v_2, v_3\} \\ E &= \{e_1, e_2, e_3\} \\ \text{Grad}(v_1) &= 2 \end{aligned}$$



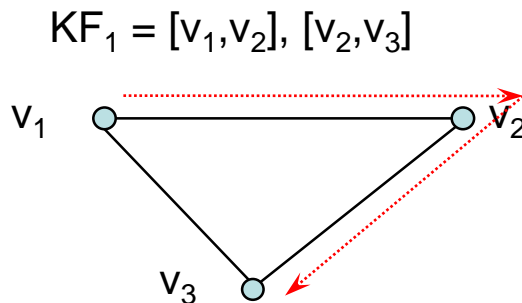
Die Anzahl der Kanten m erfüllt die folgende Bedingung:

$$0 \leq m \leq \frac{n(n-1)}{2},$$

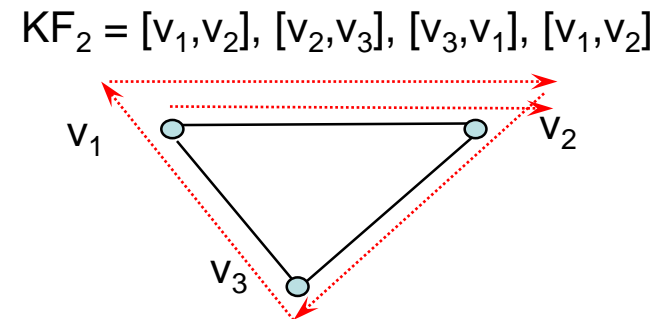
da jeder Knoten eine Kante zu jedem anderen Knoten haben kann.

Außerdem gilt: $\sum_{i=1}^n \text{degree}(v_i) = 2m$

Eine **Kantenfolge (path)** von v_1 nach v_k in einem Graphen G ist eine endliche Folge von Kanten $[v_1, v_2], [v_2, v_3], \dots, [v_{k-1}, v_k]$, wobei je 2 aufeinanderfolgende Kanten einen gemeinsamen Endpunkt haben



oder



Ein **Weg (simple path)** ist eine Kantenfolge in der alle Knoten verschieden sind (ein einzelner Knoten gilt auch als Weg) (KF_1 ist ein Weg, KF_2 ist keiner)

Die **Länge** einer Kantenfolge ist die Anzahl der Kanten auf der Kantenfolge.

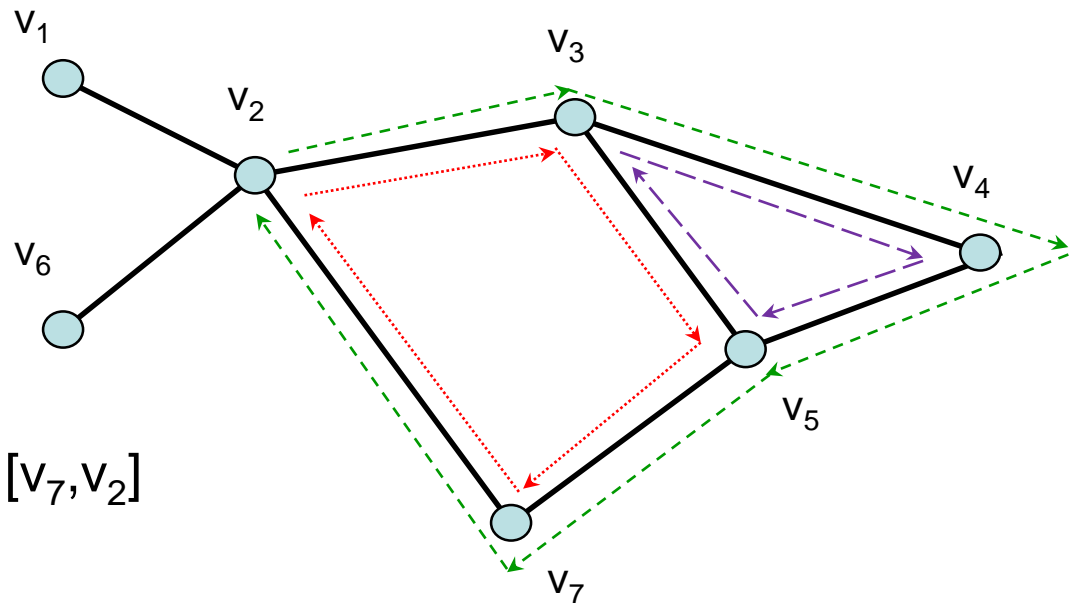
Ein **Kreis (cycle)** ist eine Kantenfolge, bei dem die Knoten v_1, v_2, \dots, v_{k-1} alle verschieden sind, $k \geq 3$ und $v_k = v_1$ gilt..

Kreise:

$[v_2, v_3], [v_3, v_5], [v_5, v_7], [v_7, v_2]$

$[v_3, v_4], [v_4, v_5], [v_5, v_3]$

$[v_2, v_3], [v_3, v_4], [v_4, v_5], [v_5, v_7], [v_7, v_2]$



Ein Graph heißt **verbunden** oder **zusammenhängend (connected)**, wenn für alle möglichen Knotenpaare v_j, v_k ein Weg existiert, der v_j mit v_k verbindet.

Ein **Baum** ist daher ein verbundener kreisloser (azyklischer) Graph.

Lemma 1: Jeder zusammenhängende, azyklische Graph G mit $n \geq 2$ Knoten hat mindestens einen Knoten mit Grad 1.

Beweis: Nimm einen beliebigen Knoten s und folge beginnend von s einem Weg P in G .

Nachdem jeder Knoten höchstens einmal von P besucht wird, wird nach höchstens $n-1$ Knoten ein Knoten v erreicht wird, der keine Kante zu einem unbesuchten Knoten hat. Hätte v eine Kante zu einem schon besuchten Knoten, gäbe es einen Kreis in G , was nicht möglich ist. Daher hat v keine Kante zu einem schon besuchten Knoten (ausser der Kante, durch die v besucht wurde) und auch keine Kante zu einem unbesuchten Knoten.

Daher hat v Grad 1.

Lemma 2: Ein Baum mit n Knoten hat genau $n - 1$ Kanten.

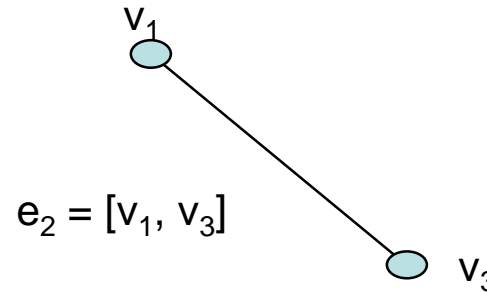
Beweis: Per Induktion über n , die Anzahl der Knoten.

$n = 1$: Jeder Graph mit 1 Knoten ist kantenlos. Daher ist auch ein Baum mit 1 Knoten kantenlos. Daher stimmt die Induktionsbehauptung, dass ein Baum mit $n = 1$ Knoten genau $n - 1 = 0$ Kanten hat.

$n > 1$: Gegeben ein Baum T mit n Knoten. Nach Lemma 1 hat T mindestens einen Knoten v , der nur an einer Kante e beteiligt ist. Entferne v vom Baum. Der neue Graph ist verbunden und kreisfrei, also wieder ein Baum, und zwar mit $n - 1$ Knoten. Wir nennen ihn T' . Nach der Induktionsannahme hat T' genau $n - 2$ Kanten. T kann aus T' erzeugt werden, indem man v und die Kante e hinzufügt. Daher hat T genau $n - 1$ Kanten.

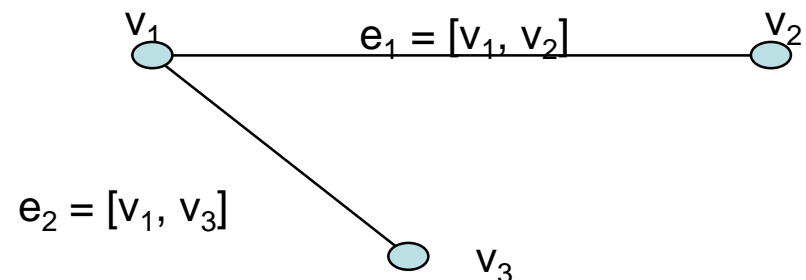
Ein Graph $G' = (V', E')$ heißt **Teilgraph** von $G = (V, E)$, wenn $V' \subseteq V$ und $E' \subseteq E$.

$$\begin{aligned} G' &= (V', E') \\ V' &= \{v_1, v_3\} \\ E' &= \{e_2\} \end{aligned}$$

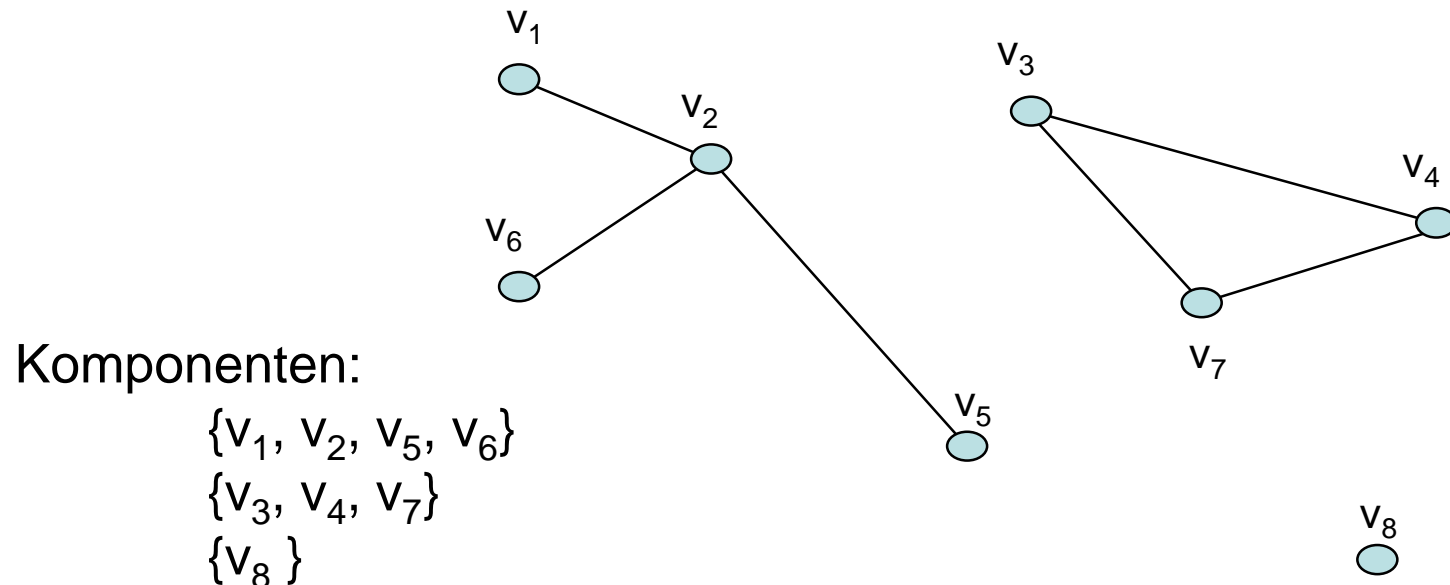


Ein Teilgraph $G'' = (V'', E'')$ ist **spannender Baum** eines zusammenhängenden Graphens $G = (V, E)$, wenn $V'' = V$ und G'' einen Baum bildet.

$$\begin{aligned} G'' &= (V'', E'') \\ V'' &= \{v_1, v_2, v_3\} \\ E'' &= \{e_1, e_2\} \end{aligned}$$



Jeder maximale, verbundene Teilgraph heißt **Komponente** des Graphen.

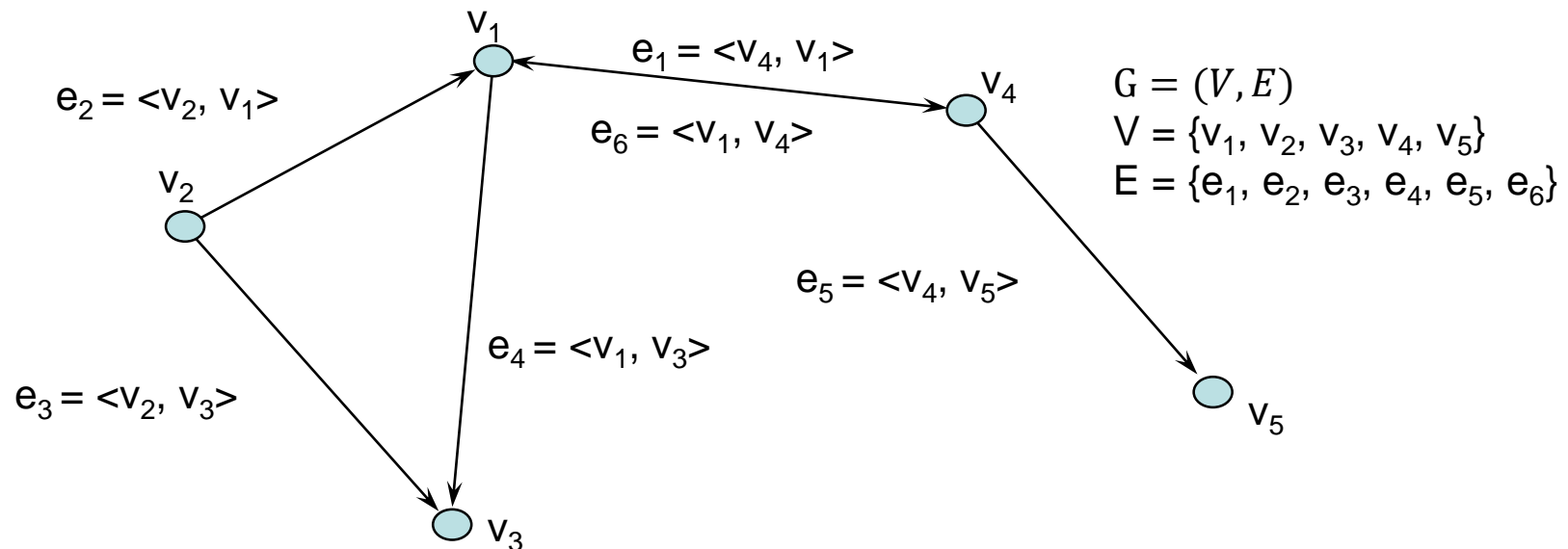


6.2 Gerichteter Graph



Ein **gerichteter Graph** $G = (V, E)$ besteht aus einer Menge V von Knoten und einer Menge E von Kanten, wobei die Kanten *geordnete* Paare $\langle v_i, v_j \rangle$ oder (v_i, v_j) von Knoten aus V sind.

Eine Kante $e = (v_i, v_j)$ heißt **ausgehende Kante** von v_i und **eingehende Kante** von v_j .



Zwischen v_1 und v_4 existieren 2 Kanten, eine Hin- und Rückkante (auch Doppelkante).

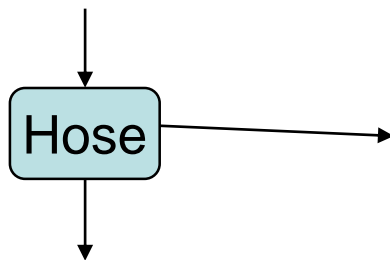
Eingangs- und Ausgangsgrad eines Knoten:

indegree(v) mit $v \in V$: $|\{v' \mid (v', v) \in E\}|$

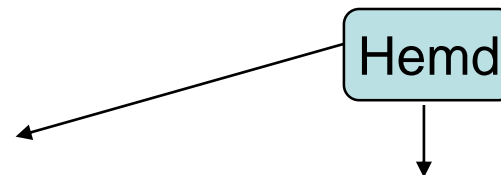
d.h. Anzahl der in v eingehenden Kanten

outdegree(v) mit $v \in V$: $|\{v' \mid (v, v') \in E\}|$

d.h. Anzahl der von v ausgehenden Kanten



indegree: 1, outdegree: 2



indegree: 0, outdegree: 2

Die Anzahl der Kanten m erfüllt die folgende Bedingung:

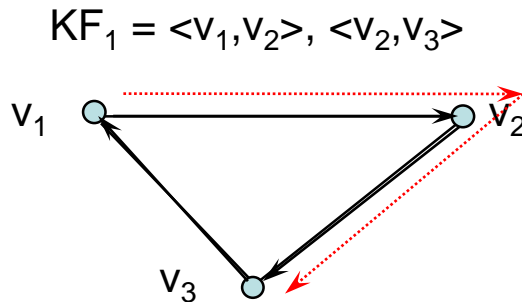
$$0 \leq m \leq n(n - 1),$$

da jeder Knoten eine Kante zu jedem anderen Knoten haben kann.

Außerdem gilt:

$$\sum_{i=1}^n \text{indegree}(v_i) = m$$
$$\sum_{i=1}^n \text{outdegree}(v_i) = m$$

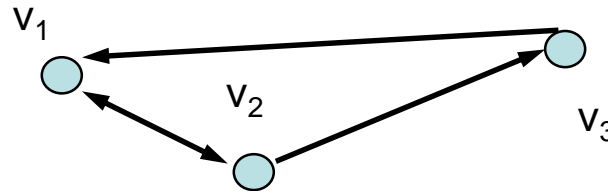
Eine **(gerichtete) Kantenfolge (directed path)** von v_1 nach v_k in einem gerichteten Graphen G ist eine endliche Folge von Kanten $\langle v_1, v_2 \rangle, \langle v_2, v_3 \rangle, \dots, \langle v_{k-1}, v_k \rangle$, wobei je 2 aufeinanderfolgende Kanten einen gemeinsamen Endpunkt bzw. Startpunkt haben



Ein **gerichteter Weg (simple directed path)** ist eine Kantenfolge in der alle Knoten verschieden sind (ein einzelner Knoten gilt auch als Weg)

Die **Länge** einer Kantenfolge ist die Anzahl der Kanten der Kantenfolge.

Ein **Kreis (cycle)** ist eine Kantenfolge, bei dem die Knoten v_1, v_2, \dots, v_{k-1} alle verschieden sind und $v_k = v_1$ gilt.



Kreise

$\langle v_1, v_2 \rangle, \langle v_2, v_1 \rangle$

$\langle v_1, v_2 \rangle, \langle v_2, v_3 \rangle, \langle v_3, v_1 \rangle$

Ein **DAG (directed acyclic graph)** ist ein gerichteter kreisloser (azyklischer) Graph.

Lemma 3a: Jeder azyklische, gerichtete Graph G hat einen Knoten mit keiner *ausgehenden* Kante.

Beweis:

Nimm einen beliebigen Knoten s und folge beginnend von s einem gerichteten Weg P in G .

Nachdem jeder Knoten höchstens einmal von P besucht wird, wird nach höchstens $n - 1$ Schritten ein Knoten v erreicht wird, der keine ausgehende Kante zu einem unbesuchten Knoten hat.

Hätte v eine ausgehende Kante zu einen schon besuchten Knoten, gäbe es einen Kreis in G , was nicht möglich ist.

Daher hat v keine ausgehende Kante in G .

Ein Knoten ohne ausgehende Kanten wird auch **Senke** genannt.

Lemma 3b: Jeder azyklische, gerichtete Graph G hat einen Knoten mit keiner *eingehenden* Kante.

Beweis:

Sei $G' = (V, E')$ der **umgekehrte** Graph von $G = (V, E)$, d.h. der Graph mit Kantenmenge $E' = \{ (v_i, v_j) \mid (v_j, v_i) \in E \}$. Da G azyklisch ist, ist G' ebenfalls azyklisch.

Nach Lemma 3a hat G' einen Knoten v ohne ausgehende Kante. Dann hat v keine eingehende Kante in G .

Ein Knoten ohne eingehende Kanten wird auch **Quelle** genannt.

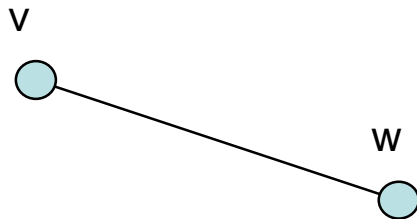
Wir unterscheiden 2 Methoden zur Speicherung von Graphen:

Adjazenzmatrix-Darstellung

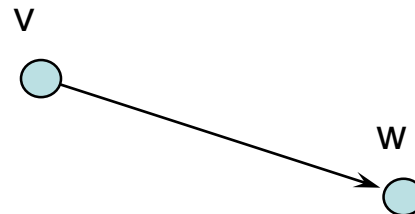
Adjazenzlisten-Darstellung

Ein Knoten w heißt **adjazent (benachbart)** zu einem Knoten v , wenn eine Kante von v nach w führt, z.B.

ungerichtete Kante $[v,w]$:
 v adjazent zu w
 w adjazent zu v



gerichtete Kante $\langle v,w \rangle$:
 w adjazent zu v
 v aber NICHT adjazent zu w

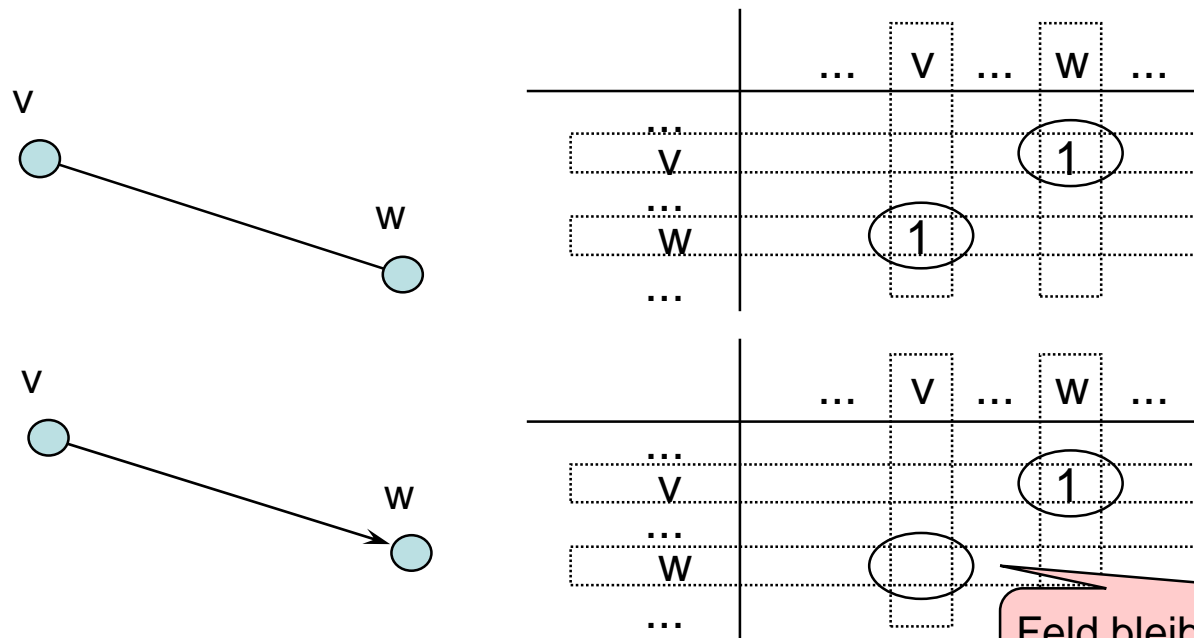


6.3.1 Adjazenzmatrix



Bei der Adjazenzmatrix-Darstellung repräsentieren die Knoten Indexwerte einer 2-dimensionalen Matrix A

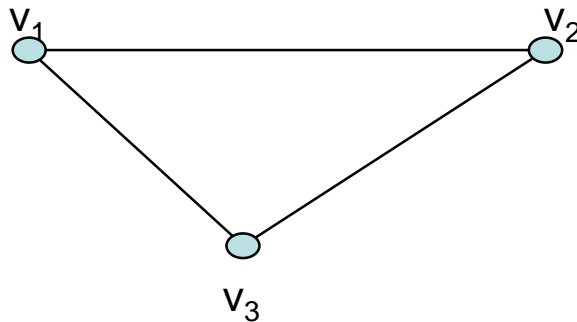
Wenn der Knoten w adjazent zum Knoten v ist, wird das Feld $A[v,w]$ in der Matrix gesetzt, z.B. 1, 'true', Wert (Kantengewicht), etc.



Bei ungerichteten Graphen ist die Matrix symmetrisch

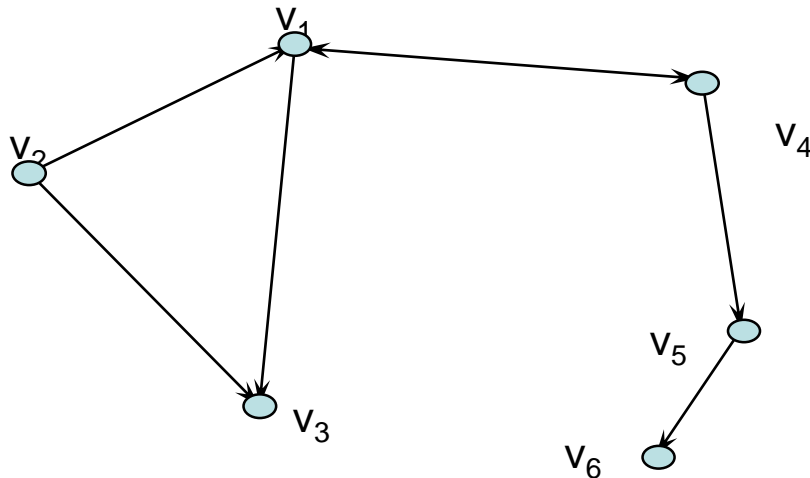
Feld bleibt leer (bzw. enthält 0), da v nicht adjazent zu w

Ungerichteter Graph



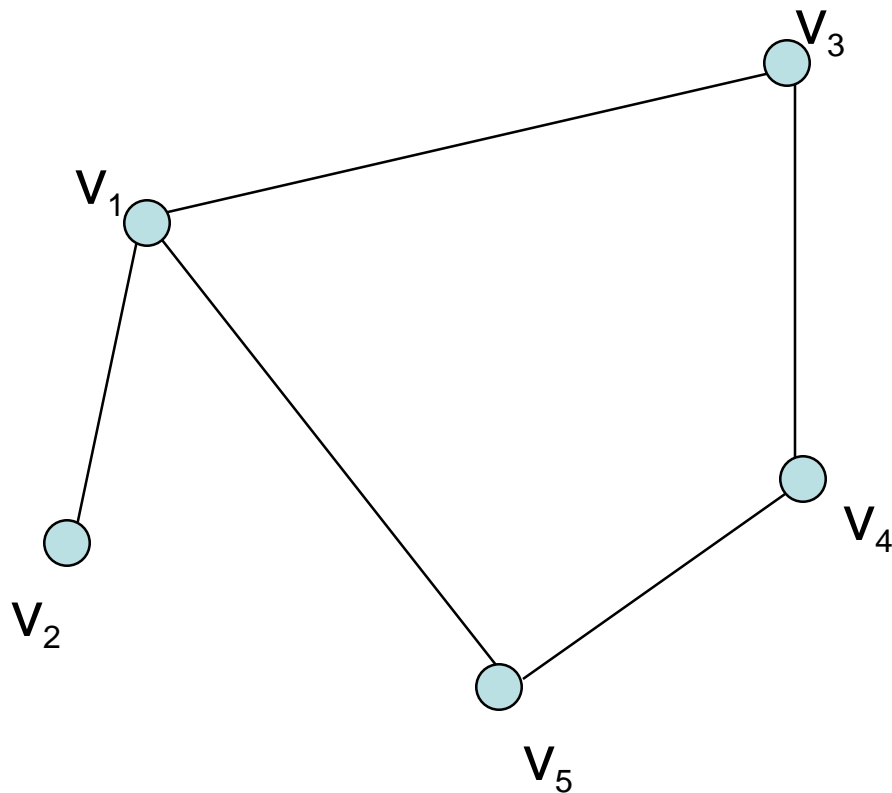
	v ₁	v ₂	v ₃
v ₁		1	1
v ₂	1		1
v ₃	1	1	

Gerichteter Graph



	v ₁	v ₂	v ₃	v ₄	v ₅	v ₆
v ₁			1	1		
v ₂	1		1			
v ₃						
v ₄	1				1	
v ₅						1
v ₆						

Ungerichteter Graph (Matrix)



	v_1	v_2	v_3	v_4	v_5
v_1					
v_2					
v_3					
v_4					
v_5					

Diagonale nicht definiert; zugleich Symmetrieachse

Ungerichteter Graph

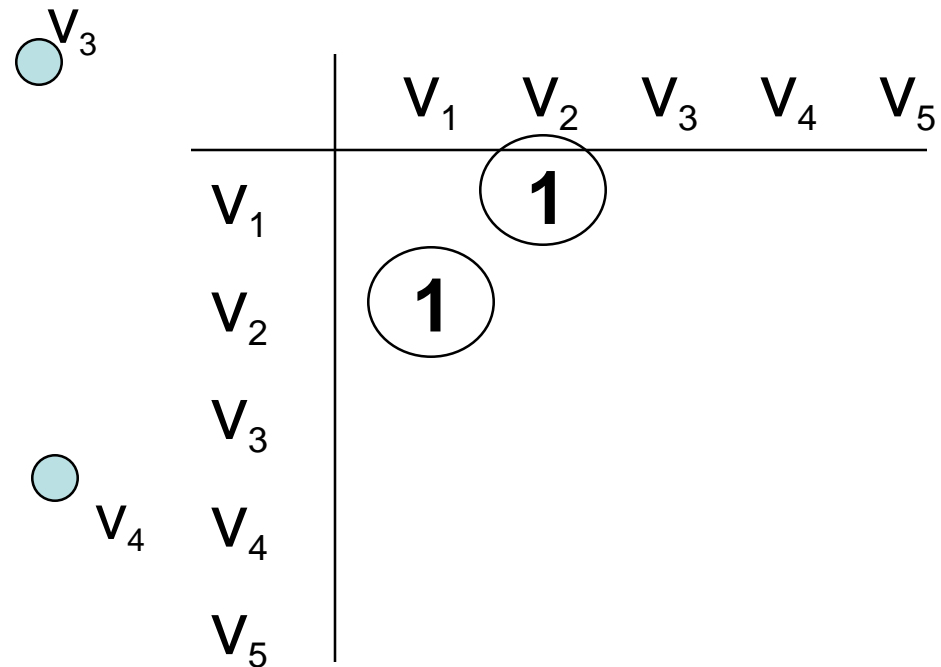
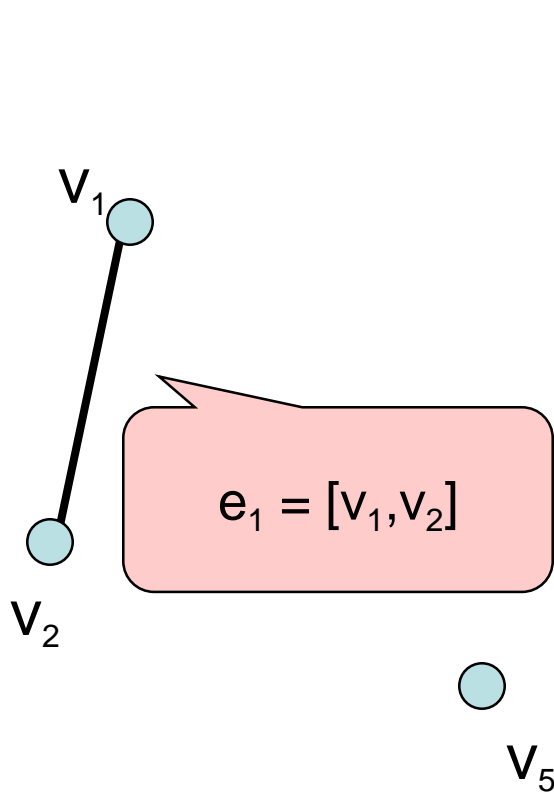
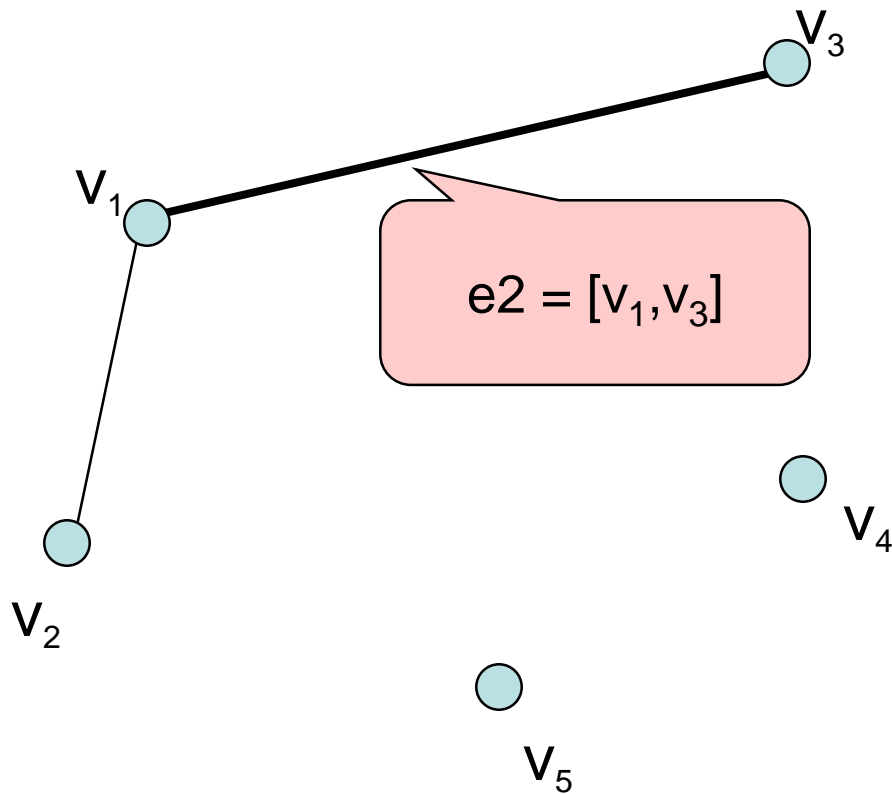


Diagram of the graph structure showing vertices v_1, v_2, v_3, v_4, v_5 and their corresponding rows in the adjacency matrix.

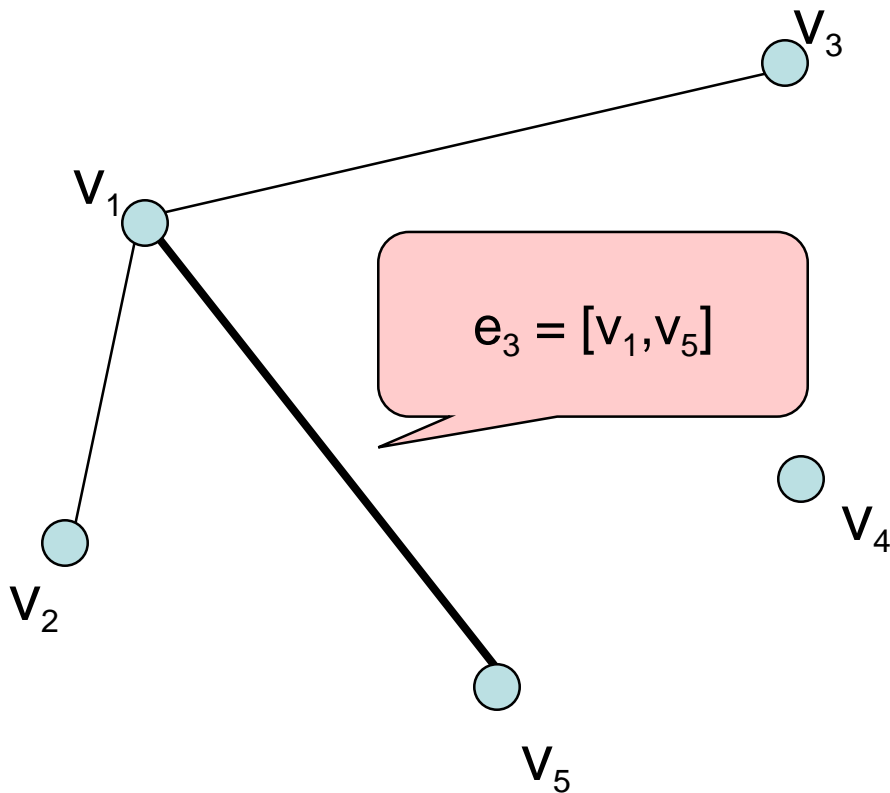
	v_1	v_2	v_3	v_4	v_5
v_1		1			
v_2	1				
v_3					
v_4					
v_5					

Ungerichteter Graph



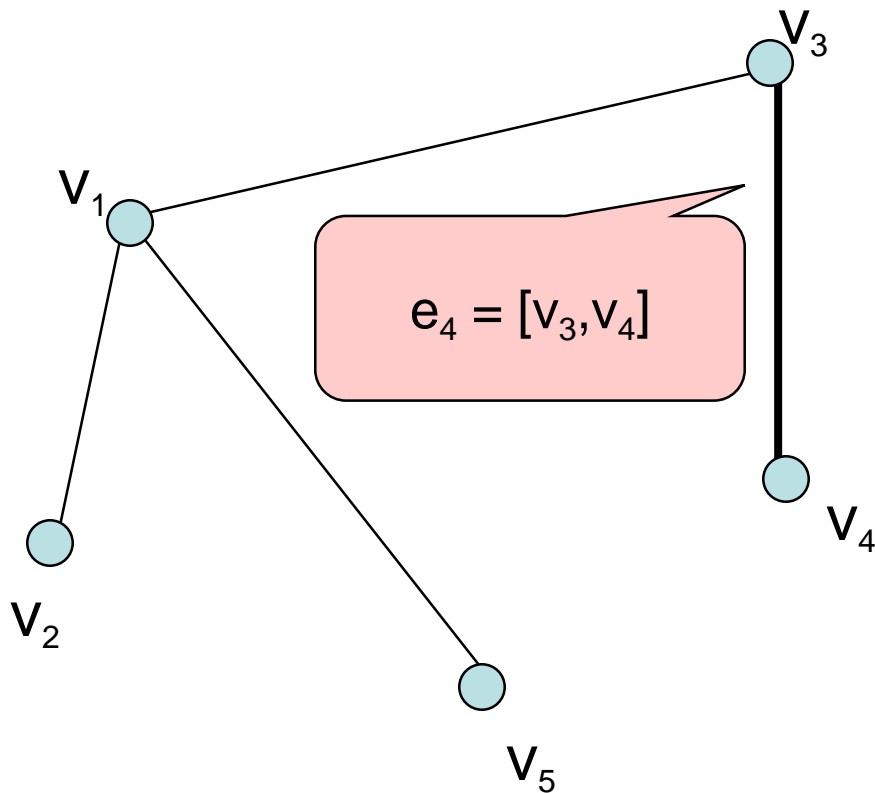
	v_1	v_2	v_3	v_4	v_5
v_1			1		
v_2	1				
v_3	1				
v_4					
v_5					

Ungerichteter Graph



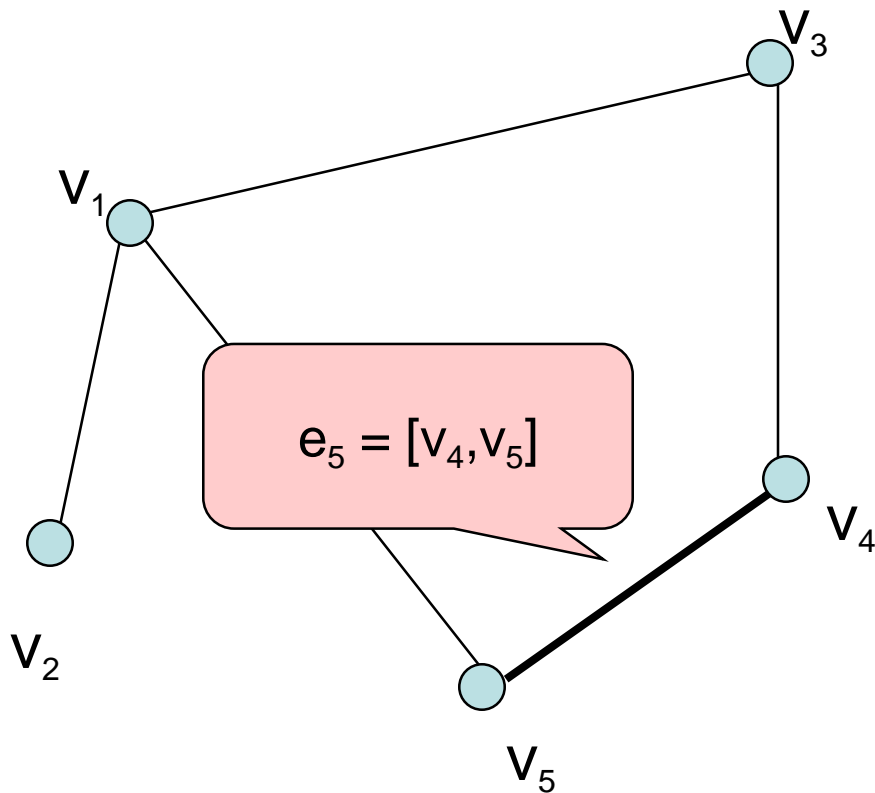
	v_1	v_2	v_3	v_4	v_5
v_1		1	1		1
v_2	1				
v_3	1				
v_4					
v_5	1				

Ungerichteter Graph



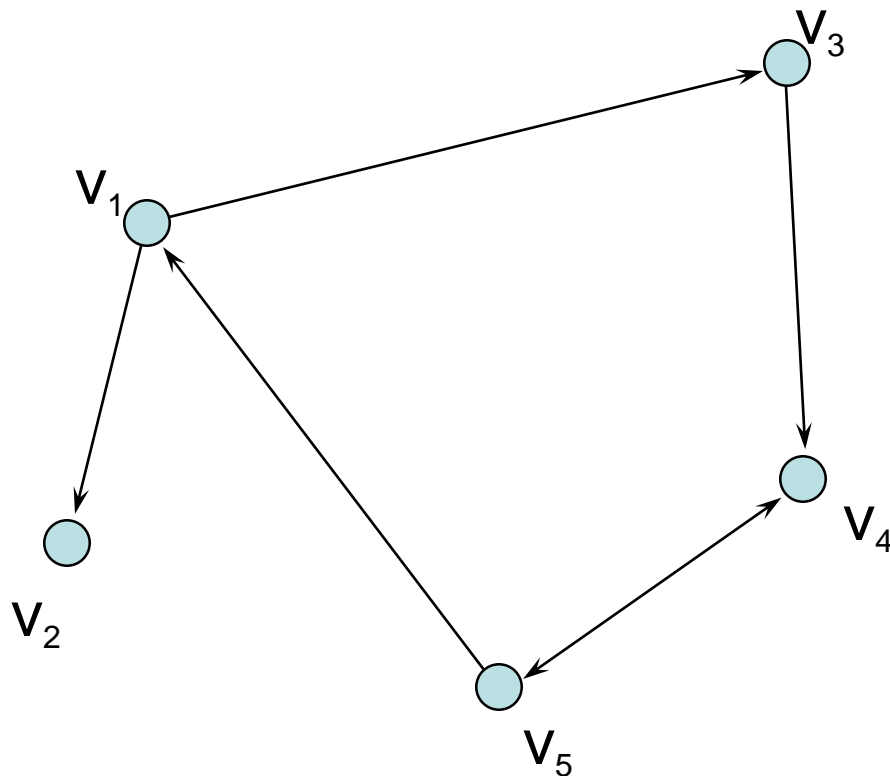
	v_1	v_2	v_3	v_4	v_5
v_1		1	1		1
v_2	1				
v_3	1			1	
v_4			1		
v_5	1				

Ungerichteter Graph



	v_1	v_2	v_3	v_4	v_5
v_1		1	1		1
v_2	1				
v_3	1			1	
v_4			1		1
v_5	1			1	

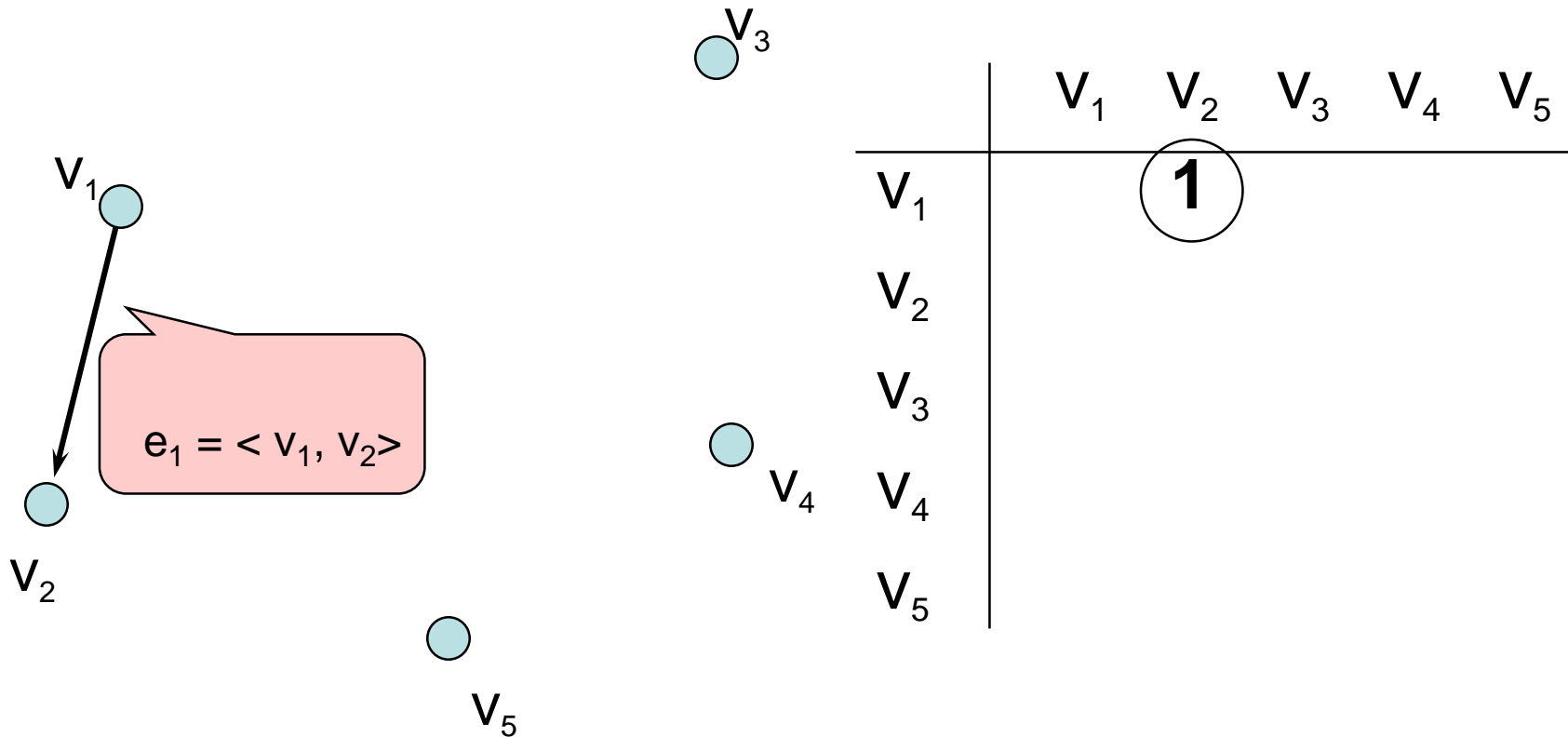
Gerichteter Graph (Matrix)



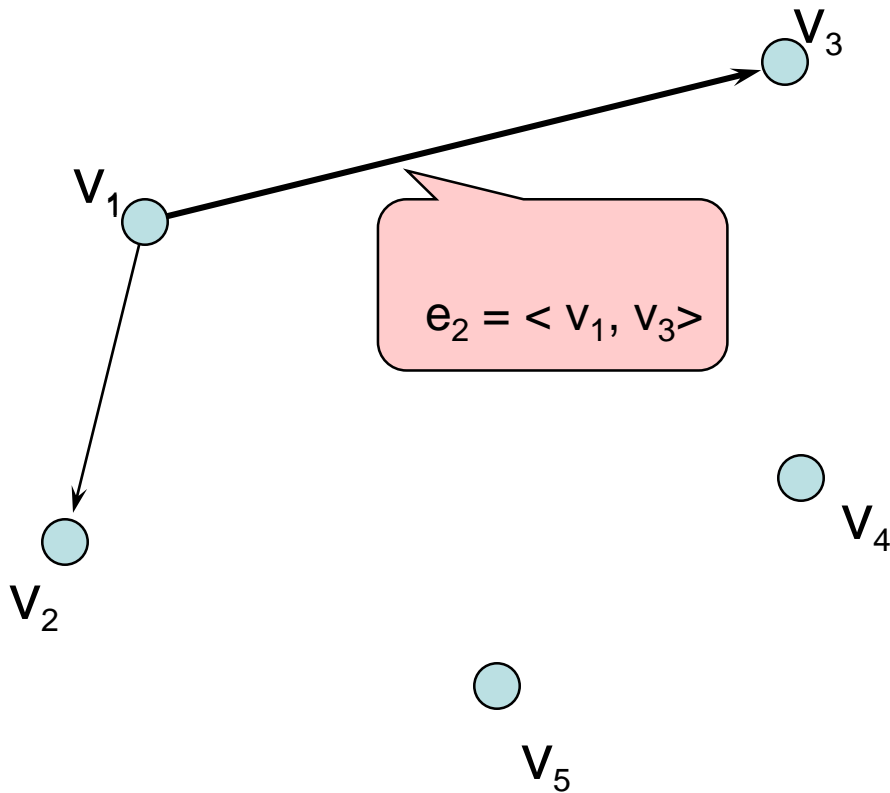
	v_1	v_2	v_3	v_4	v_5
v_1					
v_2					
v_3					
v_4					
v_5					

Verknüpfungen zu sich selbst - nicht definiert
Keine Symmetrieachse !

Gerichteter Graph

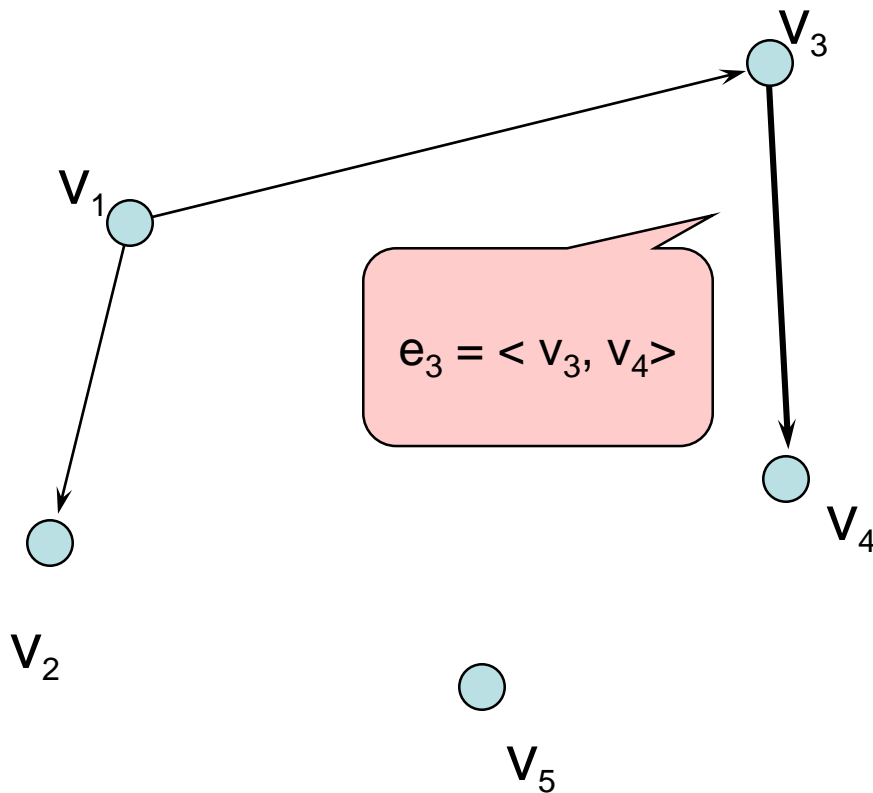


Gerichteter Graph



	v_1	v_2	v_3	v_4	v_5
v_1		1	1		
v_2					
v_3					
v_4					
v_5					

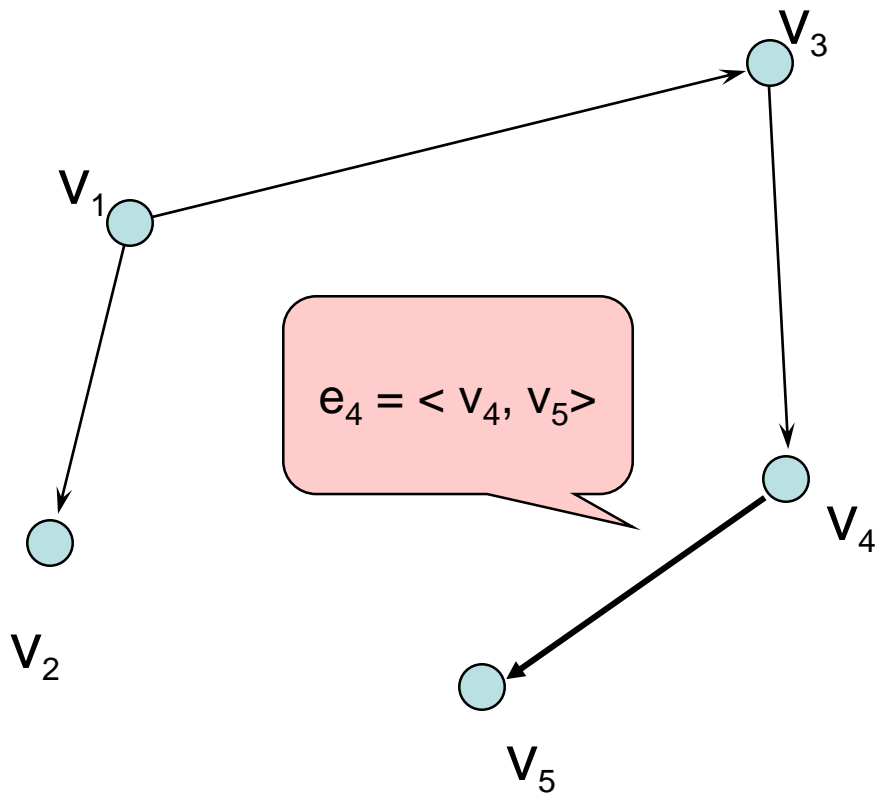
Gerichteter Graph



	v_1	v_2	v_3	v_4	v_5
v_1		1	1		
v_2					
v_3					
v_4					
v_5					

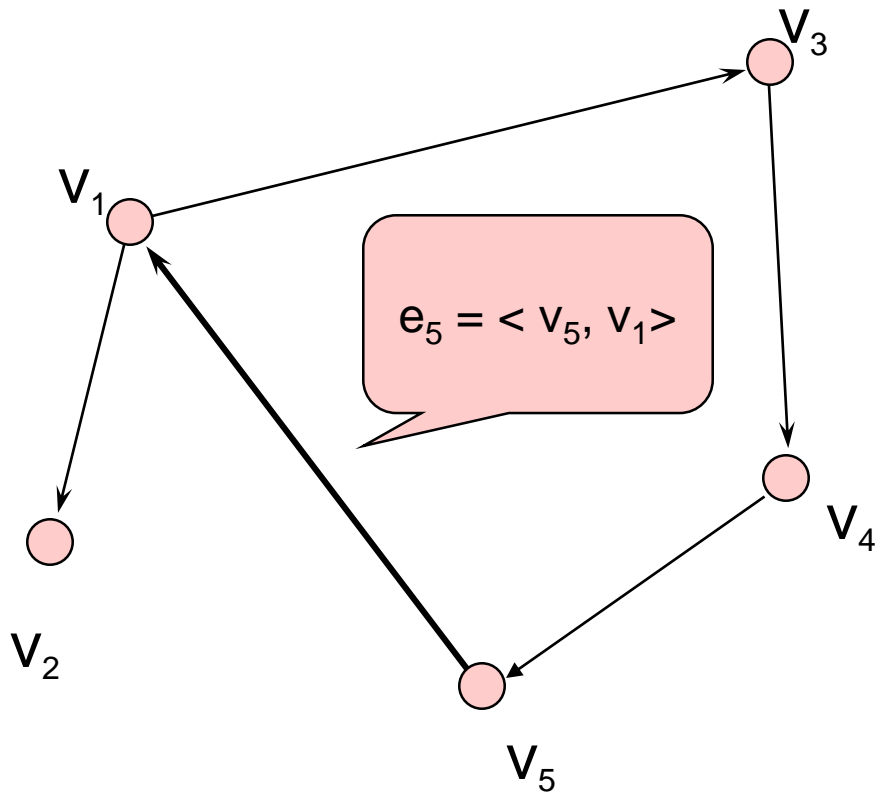
1

Gerichteter Graph



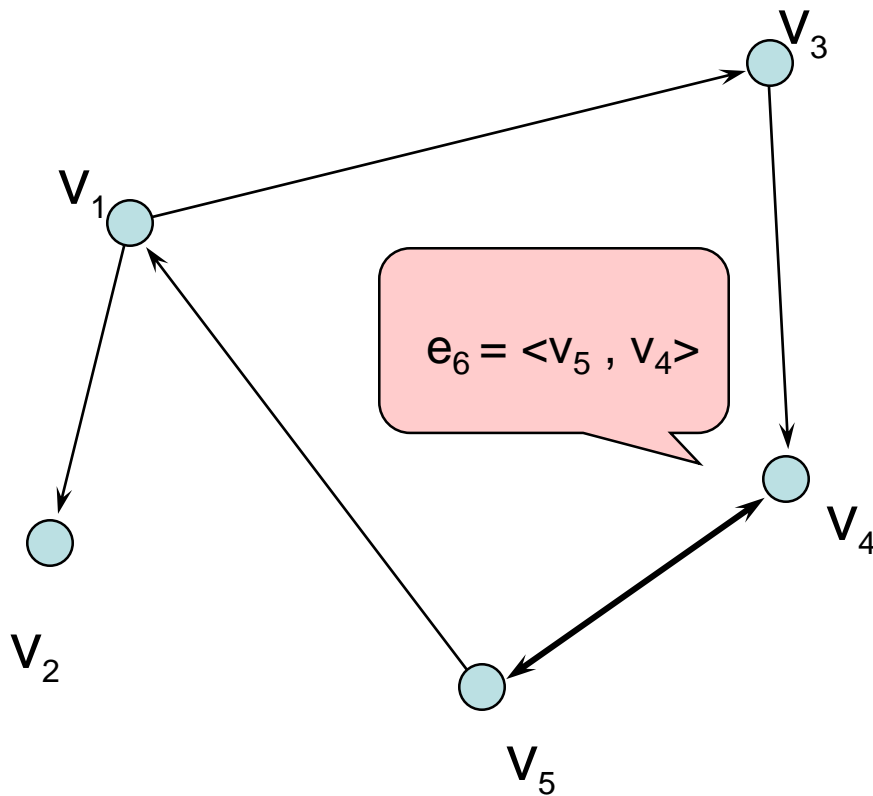
	v_1	v_2	v_3	v_4	v_5
v_1		1	1		
v_2					
v_3				1	
v_4					1
v_5					

Gerichteter Graph



	v_1	v_2	v_3	v_4	v_5
v_1		1	1		
v_2					
v_3				1	
v_4					1
v_5	1				

Gerichteter Graph

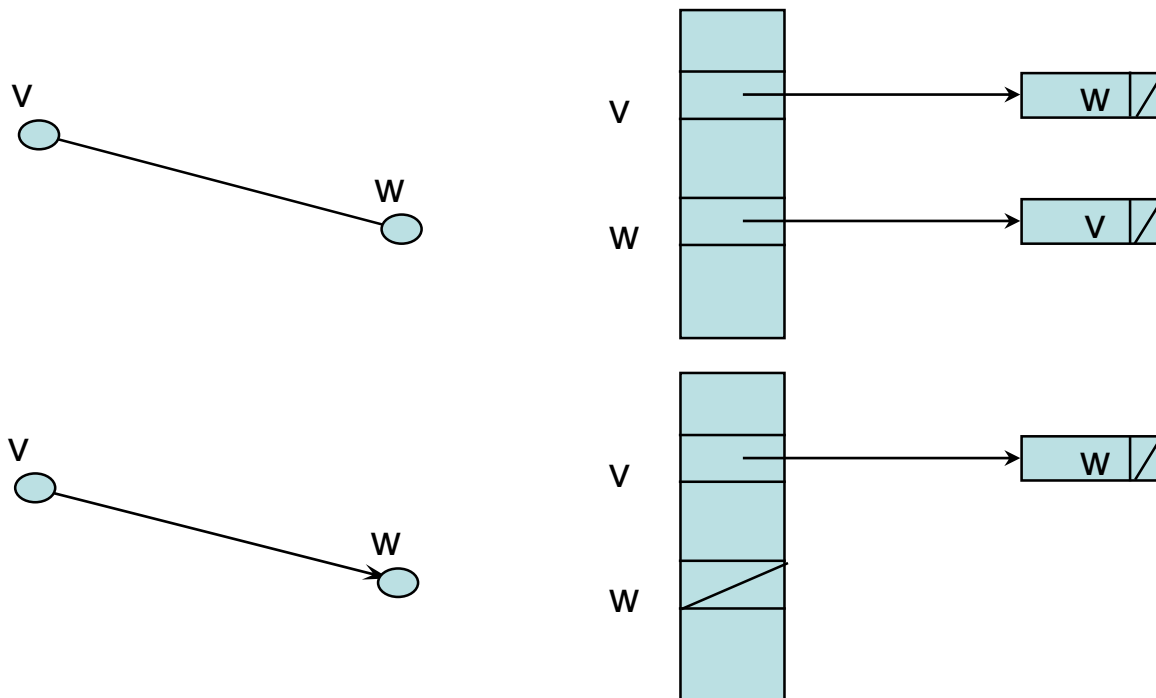


	v_1	v_2	v_3	v_4	v_5
v_1		1	1		
v_2					
v_3				1	
v_4					1
v_5	1			1	

- + gut für kleine Graphen oder Graphen mit vielen Kanten
- + Überprüfung von Adjazenzeigenschaft: $O(1)$
- + manche Algorithmen einfacher
- quadratischer Speicheraufwand: $O(|V|^2)$
- schlecht geeignet für Traversierung, Rechenaufwand: $O(|V|^2)$

Bei der Adjazenzlistendarstellung werden für jeden Knoten alle adjazenten Knoten in einer linearen Liste gespeichert

Somit werden nur die auftretenden Kanten vermerkt

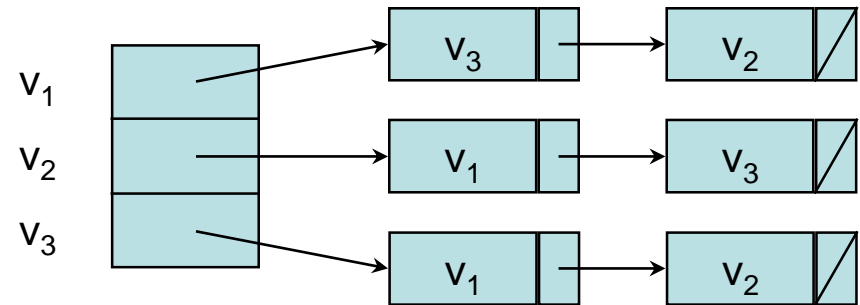
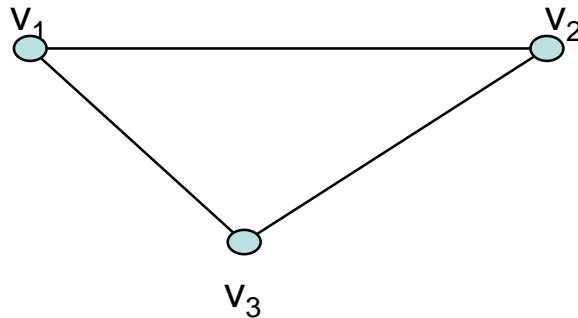


Mögliche C++ Datenstruktur:

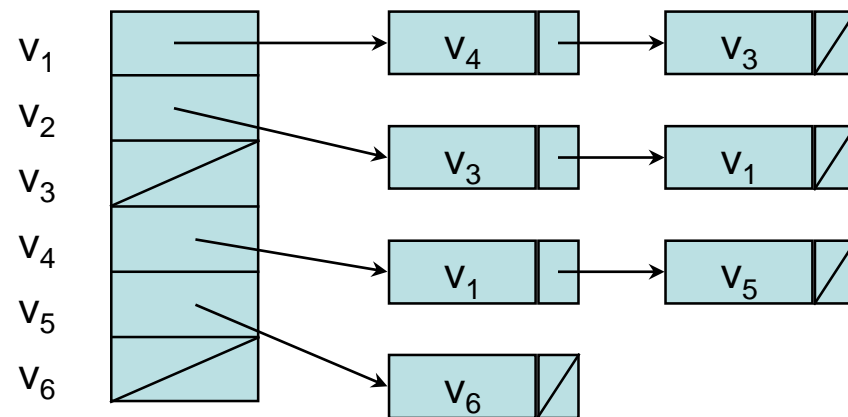
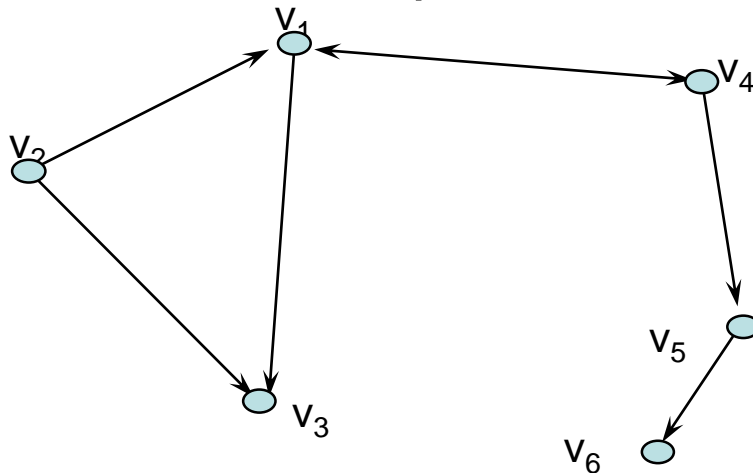
```
class node { public: int v; node *next; }
```

```
node *adjliste[maxV]; // Adjazenzliste (maxV: maximale Anzahl von Knoten)
```

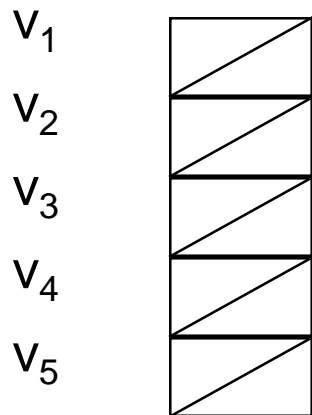
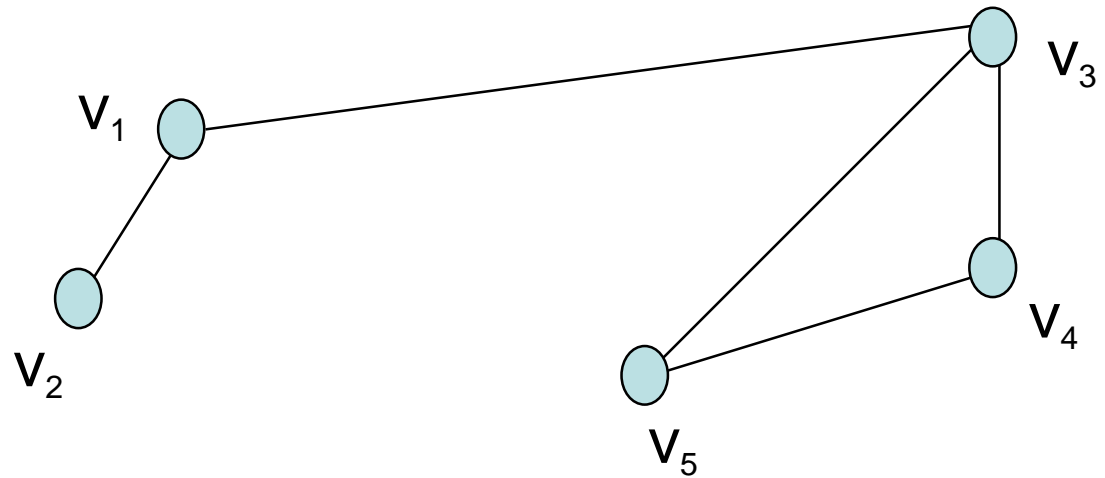
Ungerichteter Graph



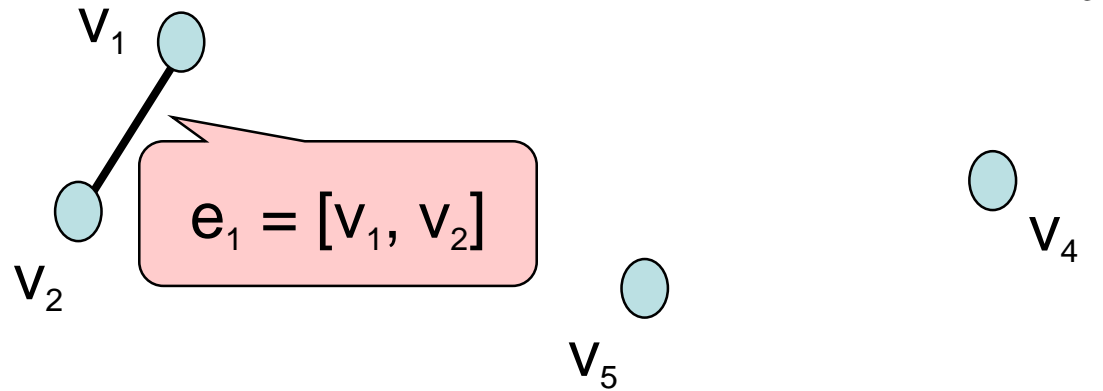
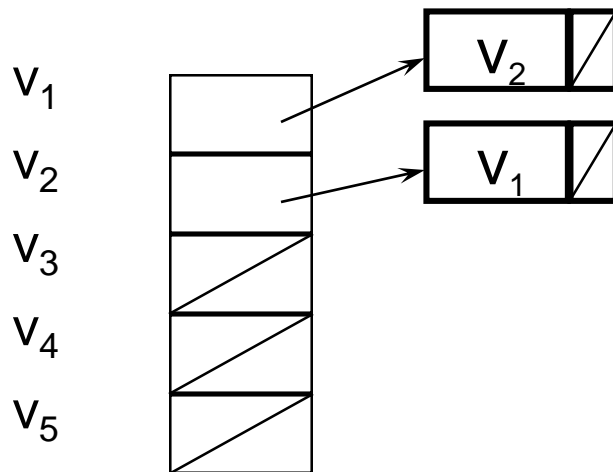
Gerichteter Graph



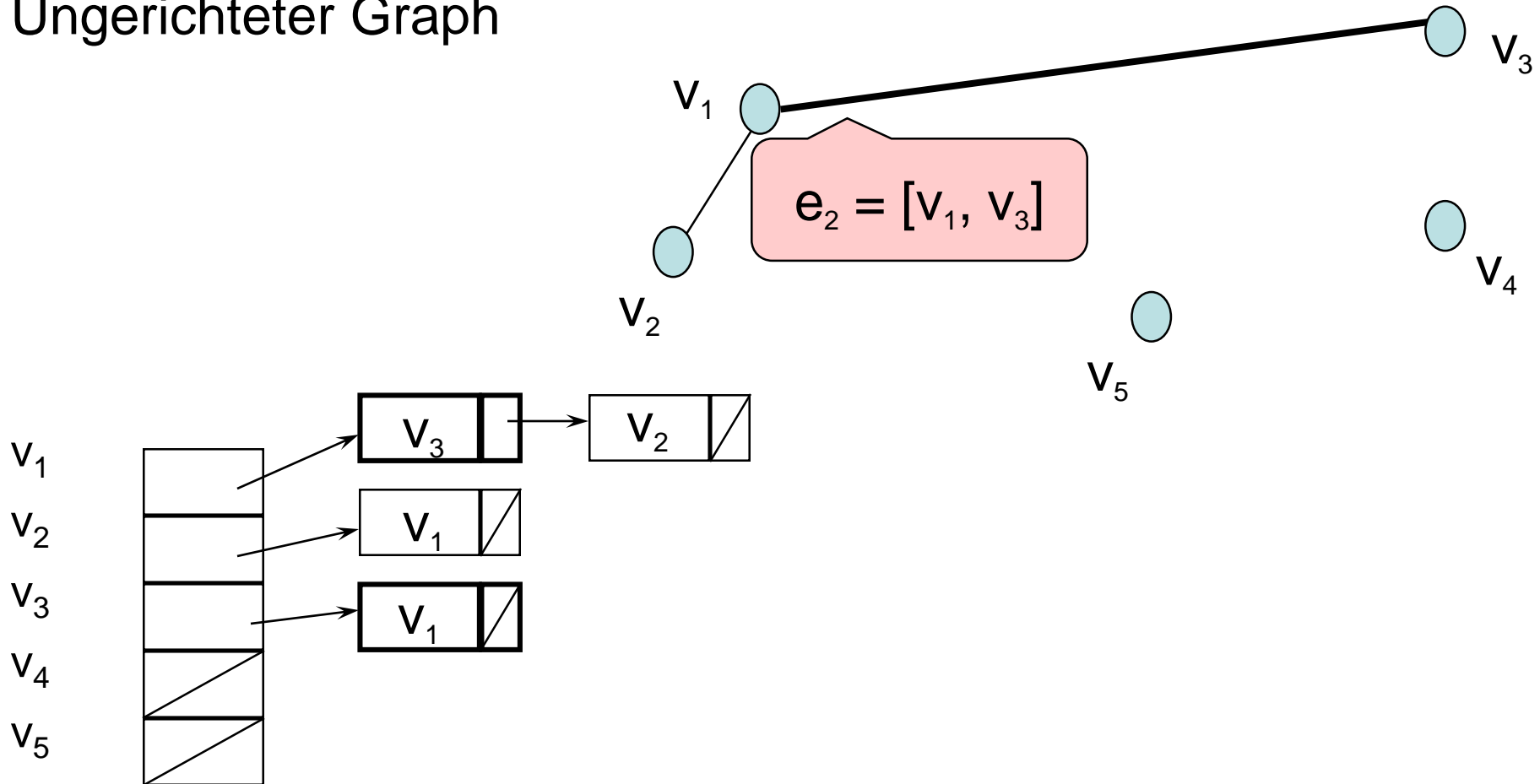
Ungerichteter Graph (Liste)



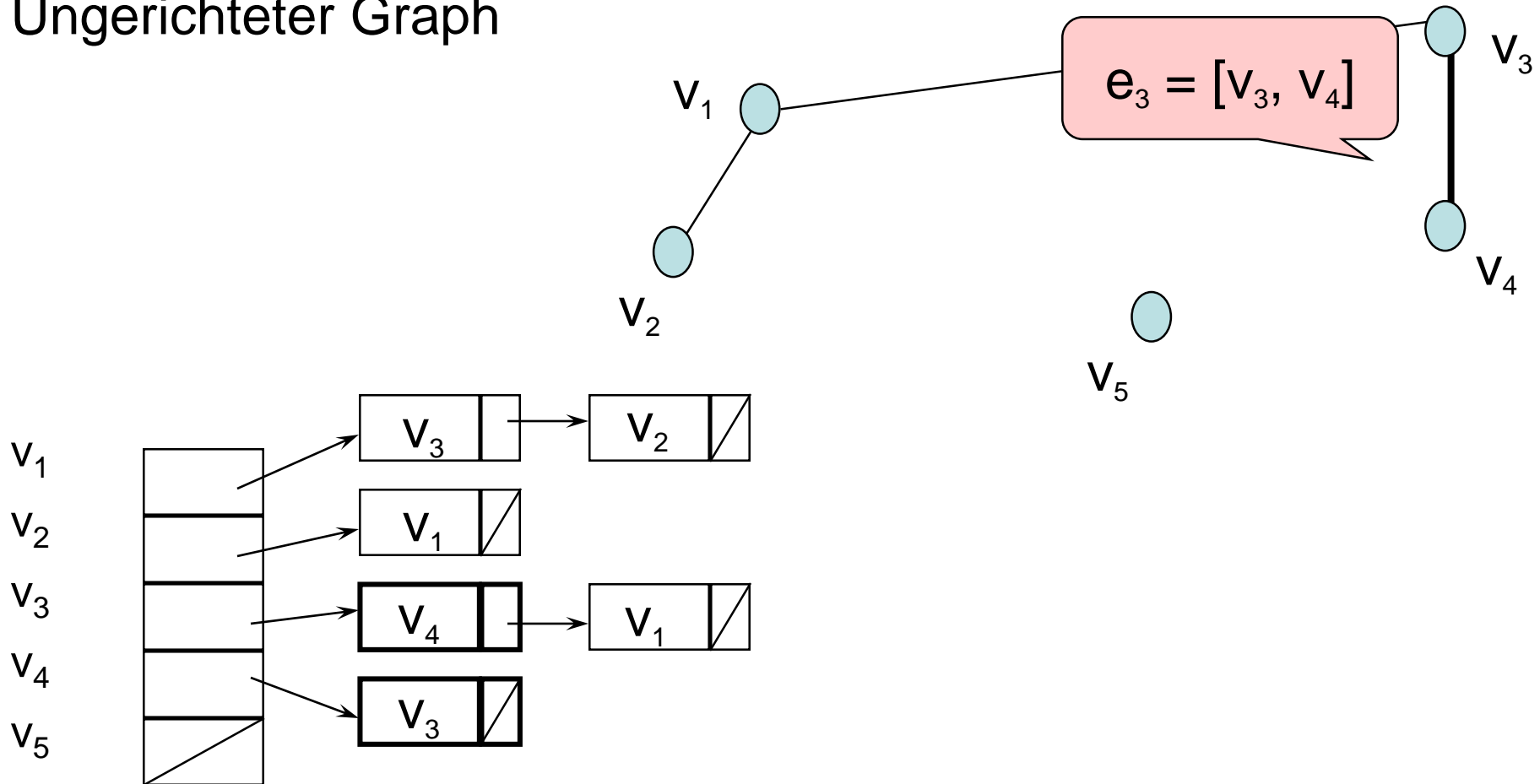
Ungerichteter Graph



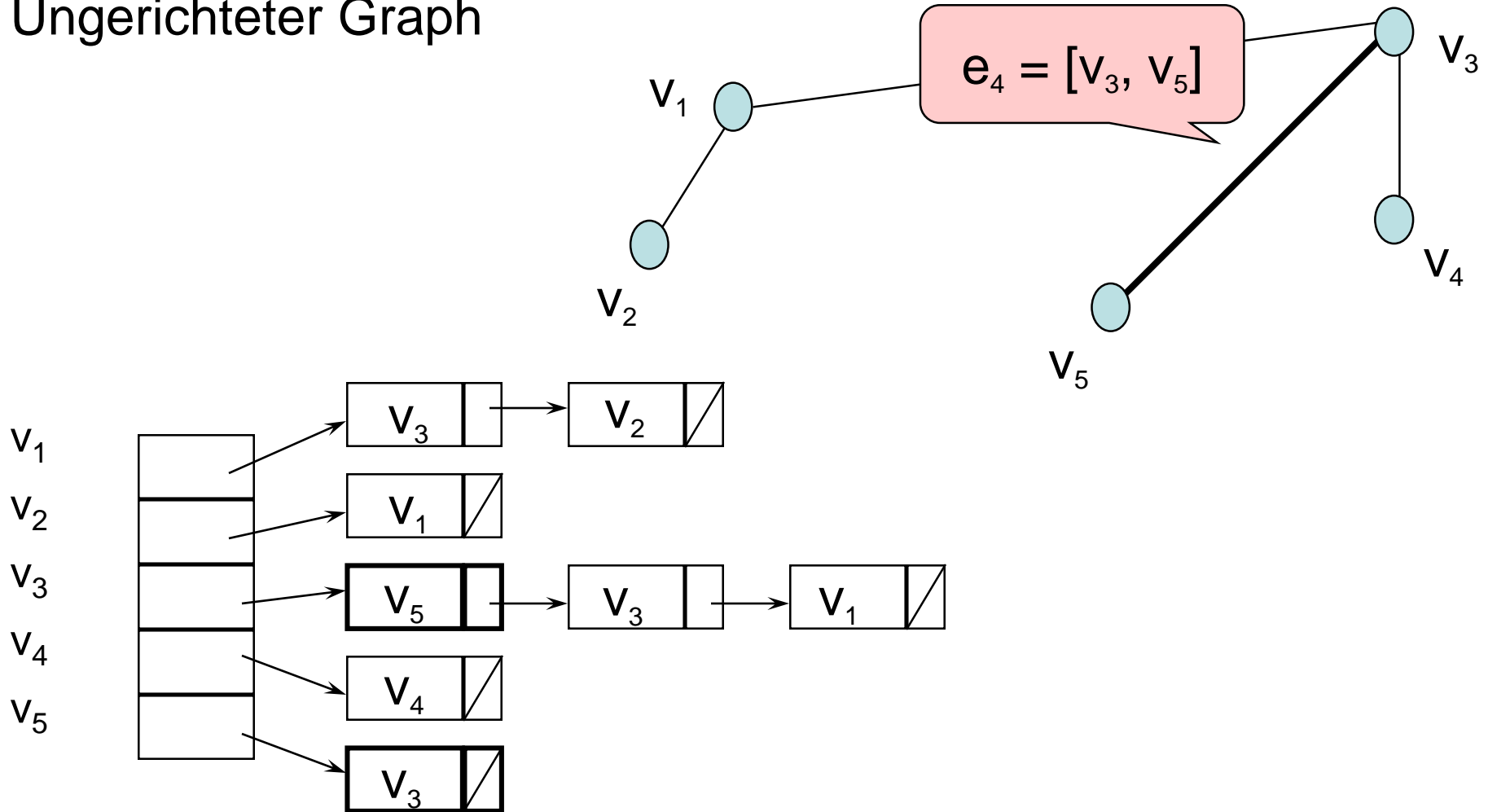
Ungerichteter Graph



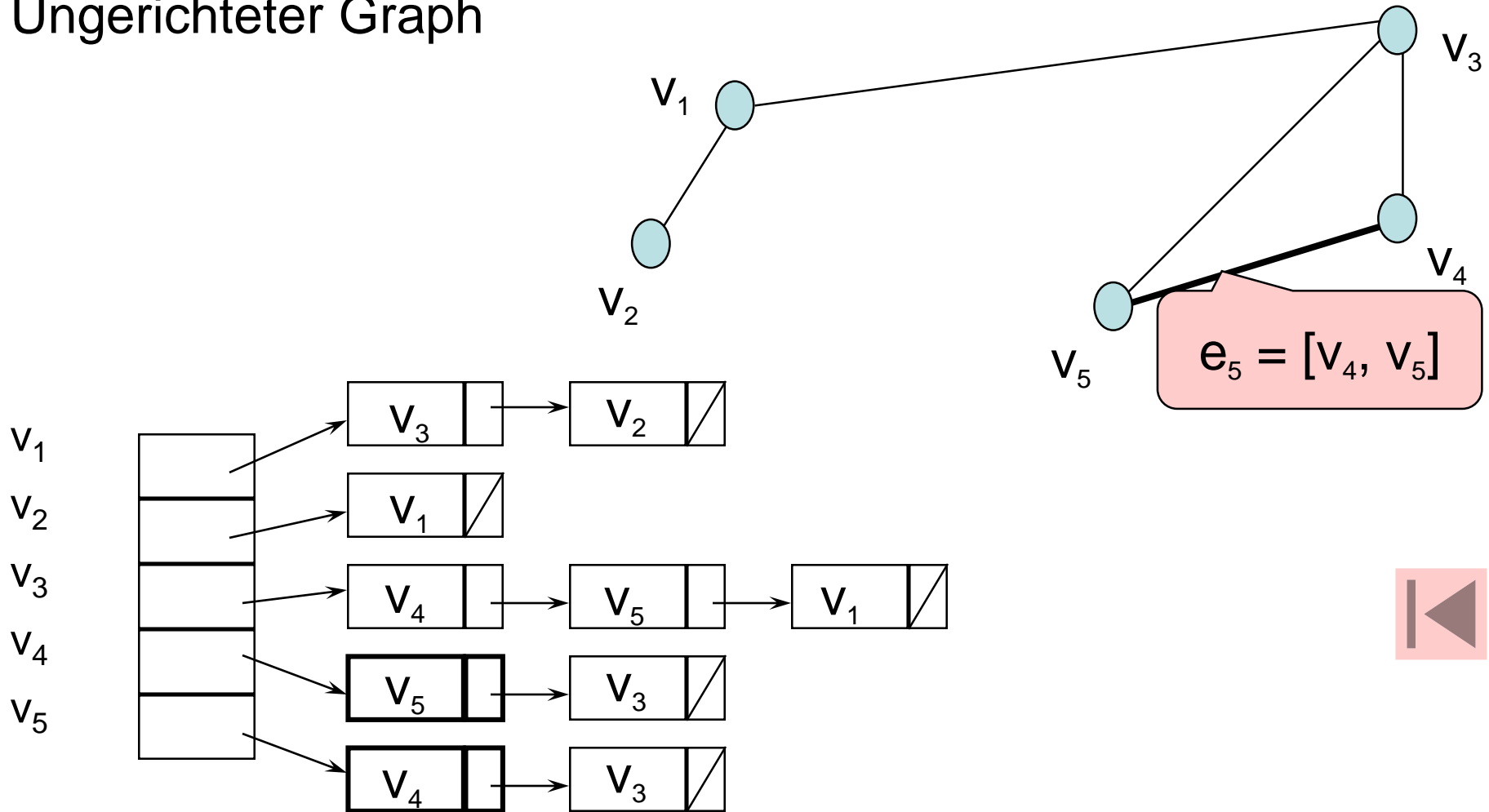
Ungerichteter Graph



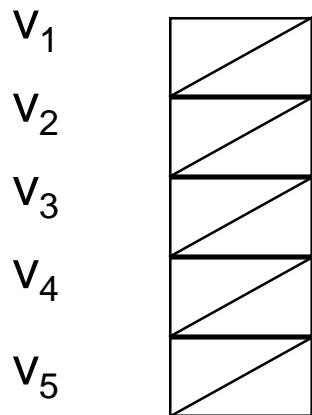
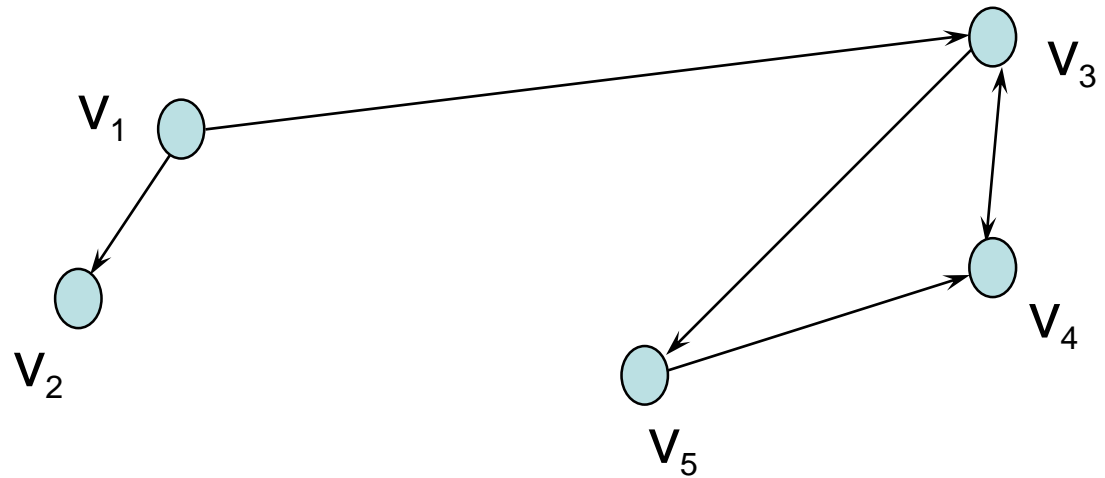
Ungerichteter Graph



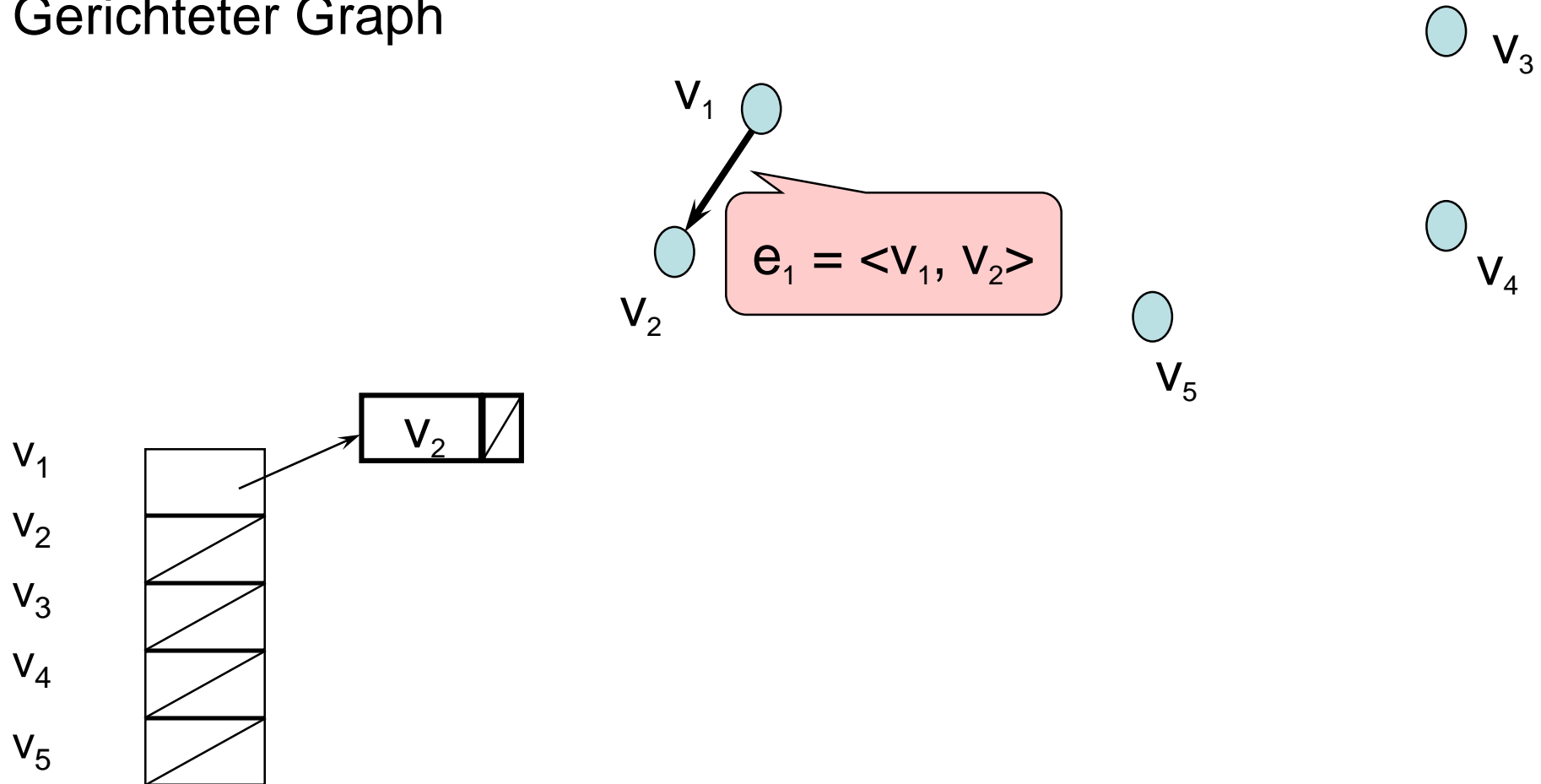
Ungerichteter Graph



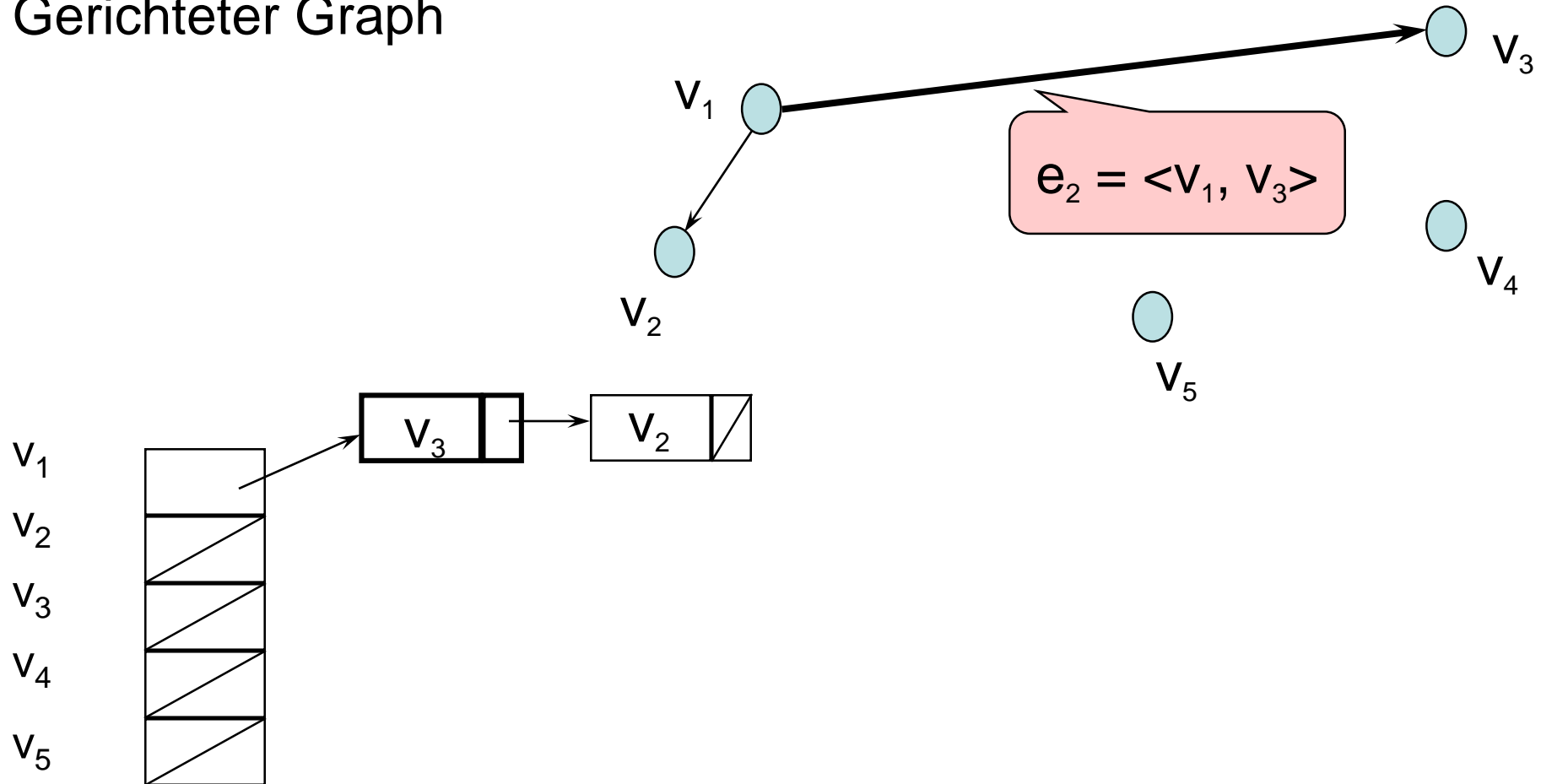
Gerichteter Graph (Liste)



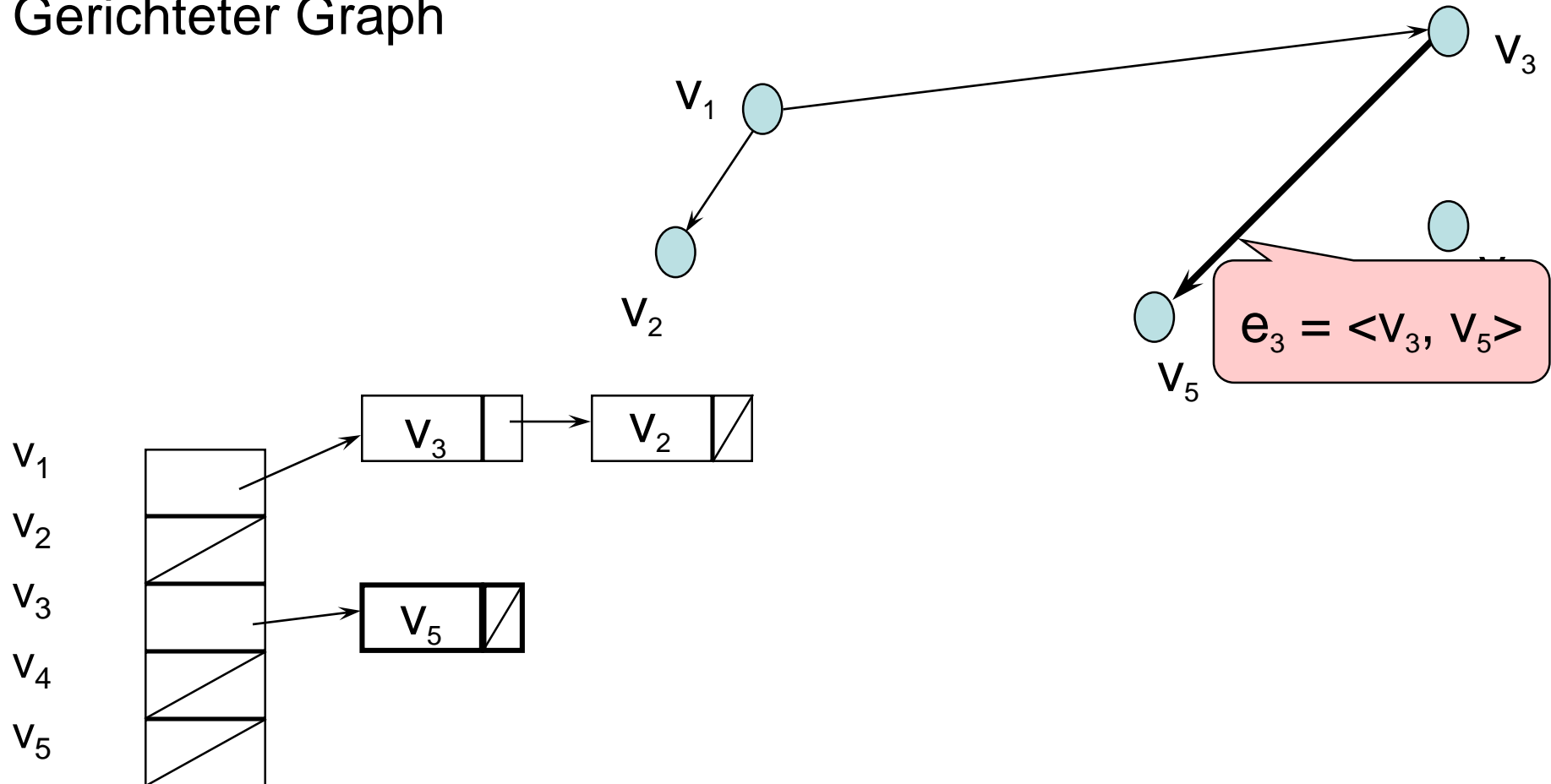
Gerichteter Graph



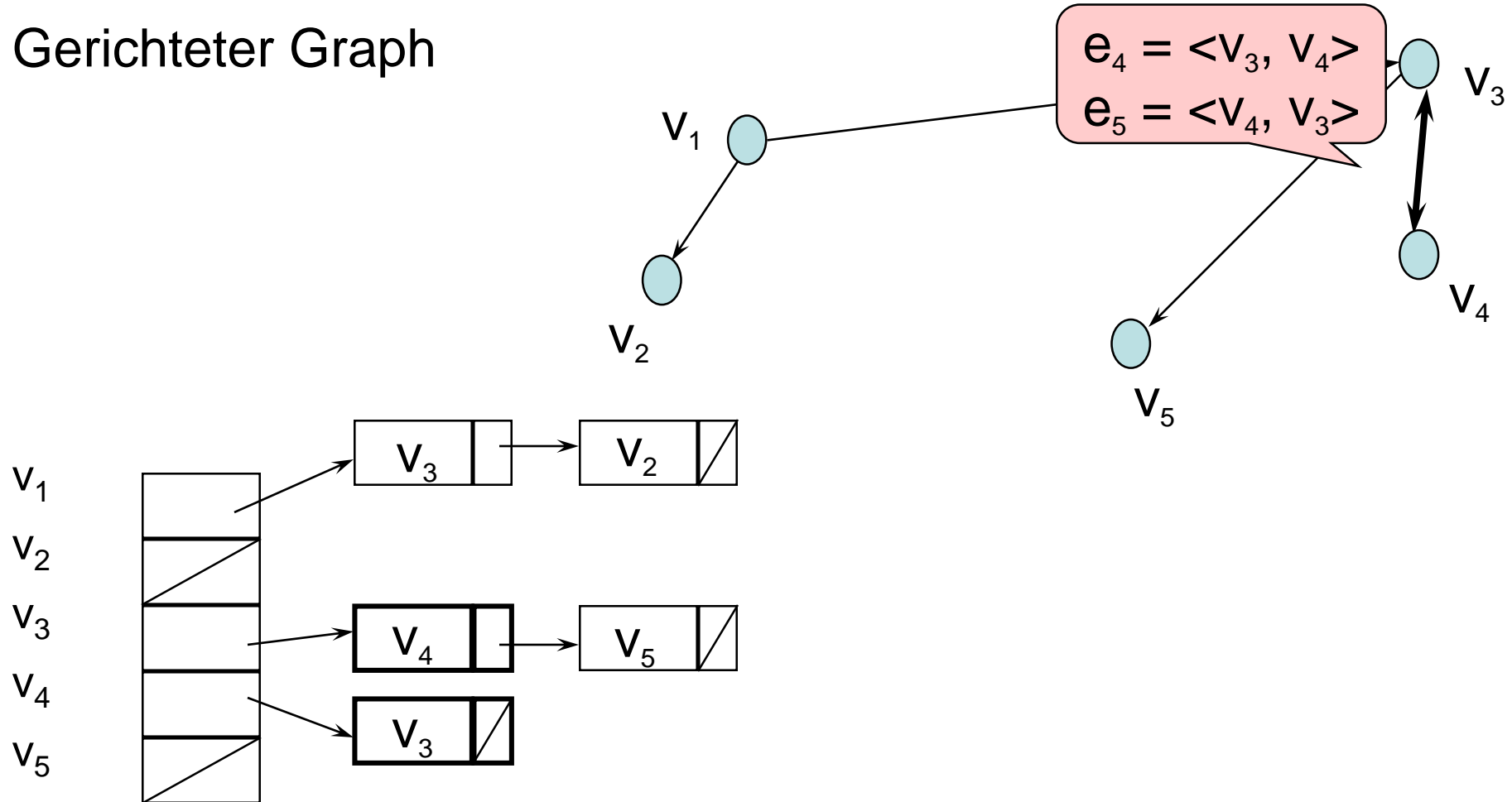
Gerichteter Graph



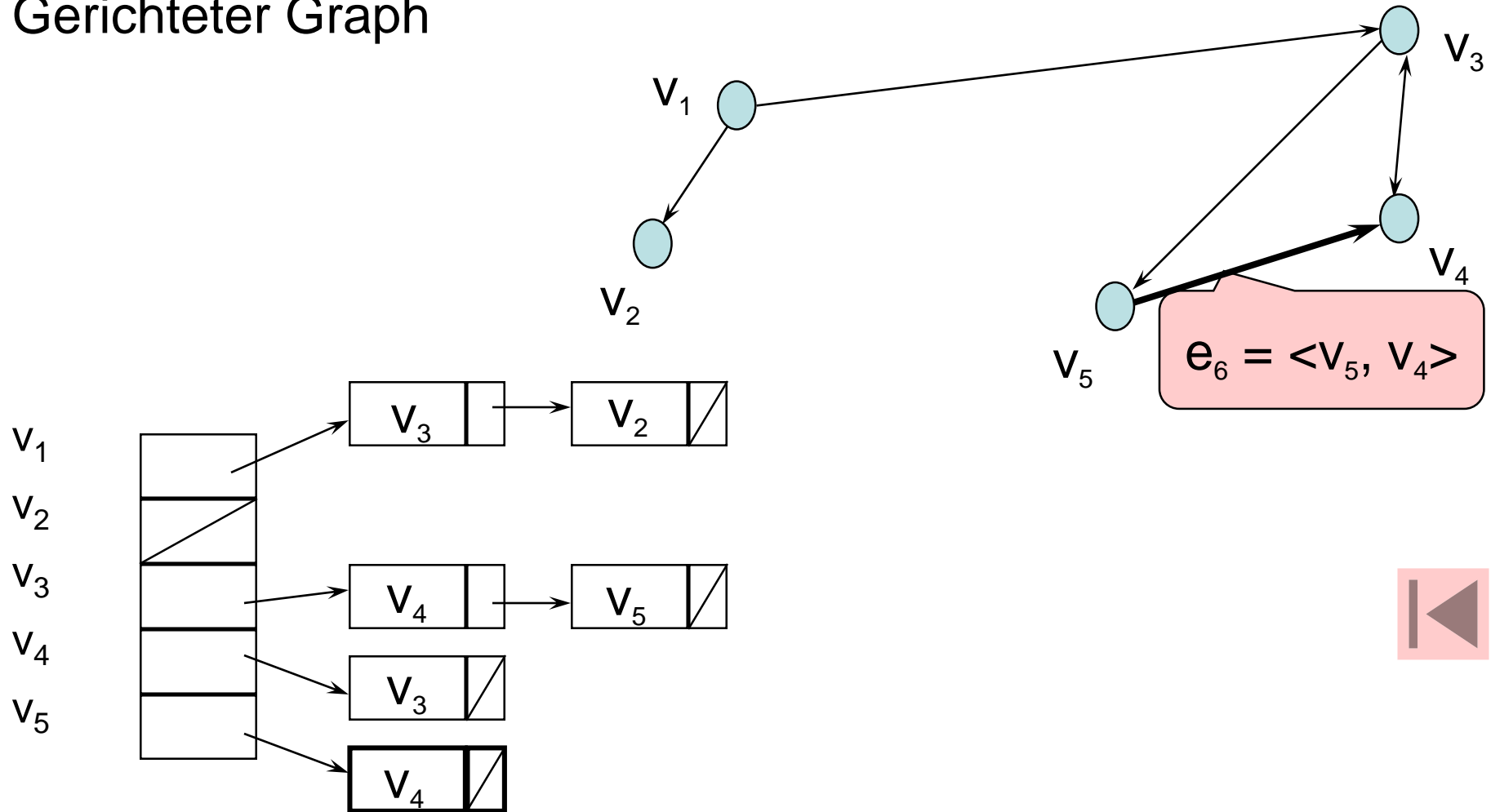
Gerichteter Graph



Gerichteter Graph



Gerichteter Graph



- + linearer Speicheraufwand: $O(|V|+|E|)$
- + gut für Traversierung, Rechenaufwand: $O(|V|+|E|)$
- Überprüfung der Adjazenzeigenschaft: $O(|V|)$
- manche Algorithmen komplexer

Problem: Ausgehend von einer binären Beziehung (z.B. „muss erledigt sein, bevor man weitermachen kann mit“) von Elementen ist zu klären, ob es eine Reihenfolge der Elemente gibt, ohne eine der Beziehungen zu verletzen.

Beispiel:

Professor Bumstead kleidet sich an

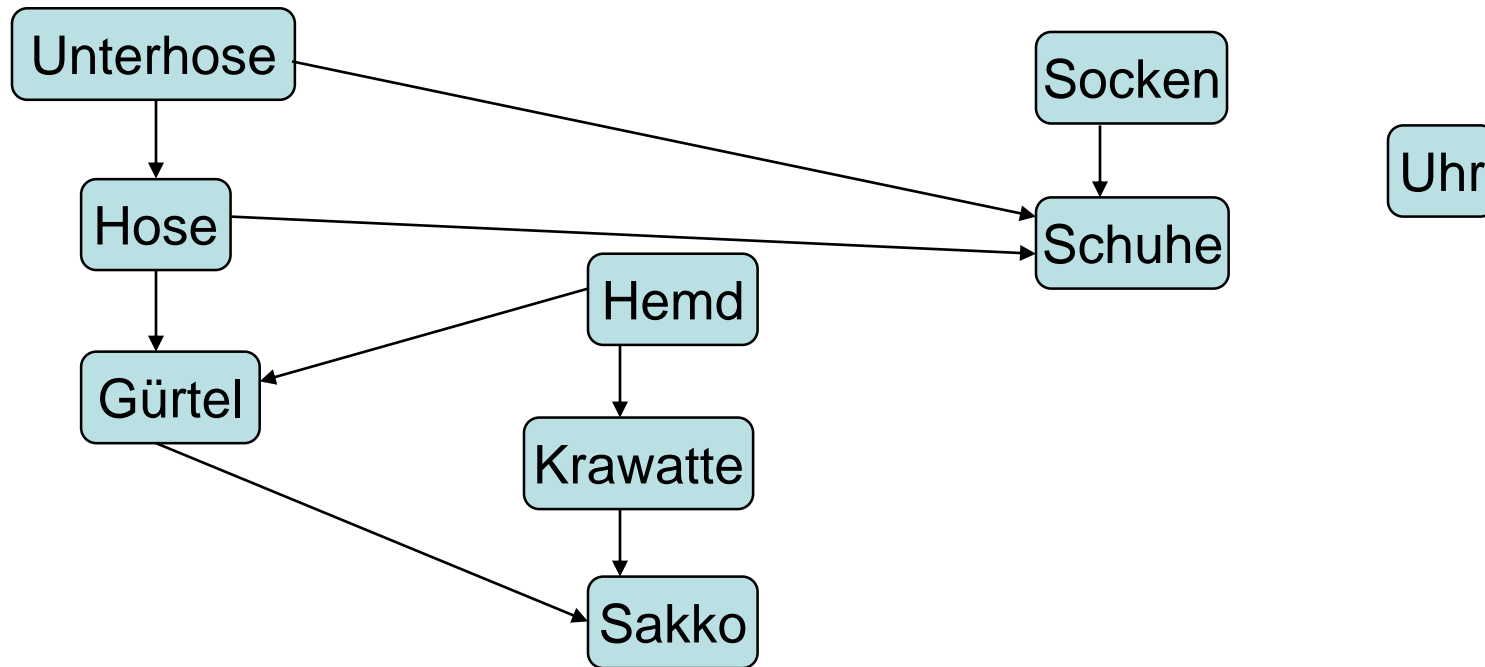
Kleidungsstücke sind: Unterhose, Socken, Schuhe, Hosen, Gürtel, Hemd, Krawatte, Sakko, Uhr

Beziehung: Kleidungsstück A muss vor B angezogen werden

Problem: Finde eine Ankleidereihenfolge damit sich Prof. Bumstead anziehen kann.

Binäre Beziehung zwischen Elementen ist gerichtete Kante zwischen entsprechenden Knoten

Prof. Bumstead



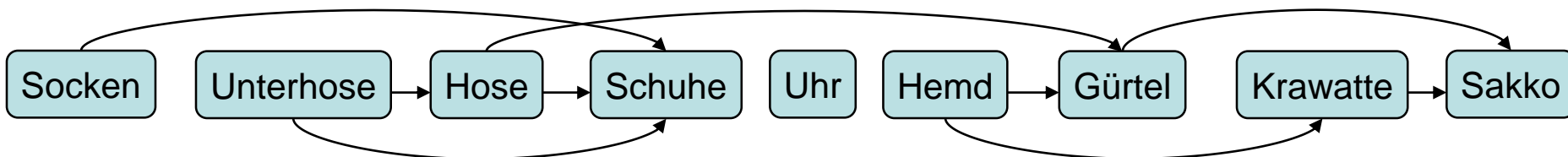
Eine **Topologische Ordnung** eines gerichteten, azyklischen Graphs G (**directed acyclic graph, DAG**) ist eine lineare Anordnung aller Knoten, sodass u vor v in der Anordnung steht, wenn G eine Kante $\langle u, v \rangle$ enthält

Falls der Graph nicht azyklisch ist, gibt es keine topologische Ordnung

Eine **Topologische Nummerierung** eines DAGs G ist eine Funktion $f: V \rightarrow \{1, \dots, |V|\}$ sodass (1) $f(u) \neq f(v)$ wenn $u \neq v$ und (2) $f(u) < f(v)$ wenn $u \neq v$ und es einen Weg von u nach v in G gibt.

Topologische Nummerierung impliziert topologische Ordnung

Topologische Ordnung: Professor Bumstead



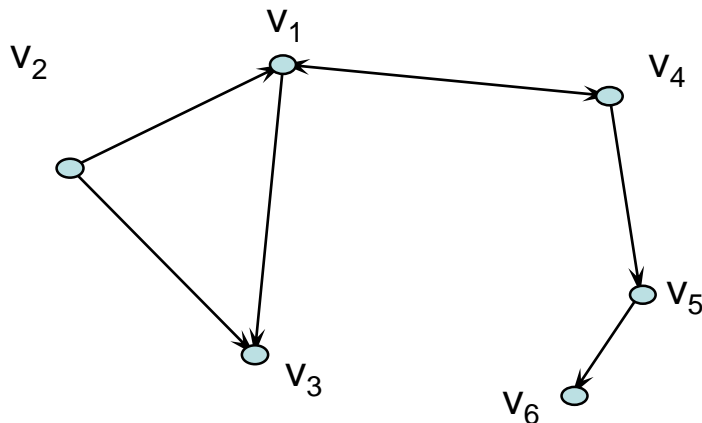
Topologische Nummerierung:

1. $lfdNr = 0$
2. Teste ob es einen Knoten v ohne eingehender Kante gibt.
Falls nein, gehe zu 6.
/* Wenn es keinen solchen Knoten gibt, dann ist der Graph nach Lemma 3b nicht azyklisch. Daher gibt es keine topologische Ordnung und der Algorithmus kann anhalten. */
3. Ordne den Knoten v in der topologischen Ordnung an, d.h. erhöhe $lfdNr$ um 1 und gib v Nummer $lfdNr$.
4. Lösche v aus dem Graphen
5. Weiter bei 2.
6. Falls es noch Knoten gibt, gib Fehler “no topologic order exists” aus

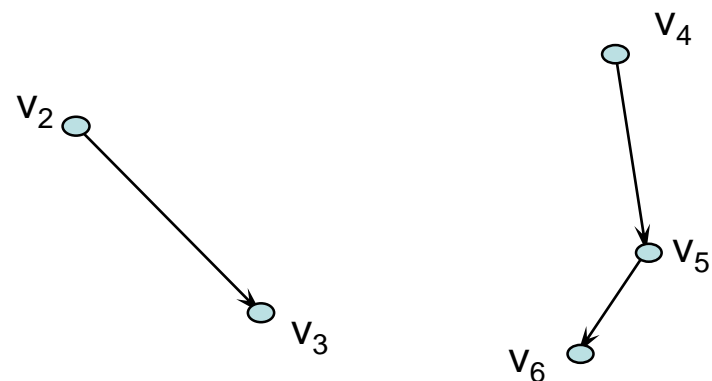
Ein Graph $G' = (V', E')$ heißt **induzierter Teilgraph (induzierter Untergraph, induced subgraph)** von $G = (V, E)$, wenn $V' \subseteq V$ und $E' = E \cap \{V' \times V'\}$.

Der von $V' \subseteq V$ induzierte Teilgraph von $G = (V, E)$ wird auch kurz als $G[V']$ geschrieben.

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



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



```
lfdNr = 0;
while ( $\exists v \in V: \text{indegree}(v) = 0$ ) {
    lfdNr = lfdNr + 1;
    N[v] = lfdNr;
    G = G[V \setminus \{v\}];           // löscht v aus G
}
if (V = {}) {
    return N;                       // G ist azyklisch
} else {
    raise error;                     // es existiert keine top. Ordnung
}
```



```
lfdNr = 0;

// indegree(v) :  $O(n)$  Zeit pro Knoten
//       $\rightarrow O(n^2)$  Zeit pro Schleifeniteration
//      n Iterationen  $\rightarrow O(n^3)$  Zeit insgesamt

while (  $\exists v \in V$ : indegree(v) = 0 ) {
    lfdNr = lfdNr + 1;
    N[v] = lfdNr;
    G = G[V \setminus \{v\}]; // löscht v aus G
    //  $O(n)$  Zeit pro Iteration  $\rightarrow$  n Iterationen  $\rightarrow O(n^2)$  Zeit insgesamt
}

if (V = {}) { return N; } // G ist azyklisch
else { raise error; } // es existiert keine top. Ordnung
```


Implementierung mit *Adjazenzmatrixdarstellung*:

- Test am Anfang der while-Schleife dauert Zeit $O(n^2)$.
 - Entfernen von einem Knoten dauert Zeit $O(n)$.
 - Höchstens n Iterationen der while-Schleife
- insgesamt $O(n^3)$ Laufzeit

Implementierung mit *Adjazenzmatrixdarstellung* und zusätzlicher Datenstruktur:

NEU: speichere Eingangsgrad jedes Knotens in Array **IND**

- **Initialisierung** von **IND** vor der while-Schleife: Berechne **IND** mithilfe der Adjazenzmatrix A , indem man die Einträge der Spalte von jedem Knoten w summiert und sie in **IND**[w] speichert $\rightarrow O(n^2)$ Zeit einmalig
 - **Test am Anfang der while-Schleife** läuft über **IND** und sucht nach dem ersten Knoten v mit **IND**[v] = 0 $\rightarrow O(n)$ Zeit pro Iteration
 - **Entfernen von einem Knoten v :** Setze alle Einträge in der Zeile von v in der Adjazenzmatrix auf 0 und für jeden positiven Eintrag $A[v,w]$ reduziere **IND**[w] um 1. Setze **IND**[v] auf -1 (um zu zeigen, dass v nicht mehr existiert) $\rightarrow O(n)$ Zeit pro Iteration
 - Höchstens n Iterationen der while-Schleife
- \rightarrow insgesamt $O(n^2)$ Laufzeit

Implementierung mit *Adjazenzlistendarstellung* und zusätzlicher Datenstruktur:

Speichere Eingangsgrad jedes Knotens in Array **IND**.

- **Initialisierung** von **IND** vor der while-Schleife: Initialisiere **IND** mit 0 für jeden Knoten. Traversiere all Kanten, d.h. alle Adjazenzlisten und erhöhe **IND**[v] um 1 für jede Kante $\langle u, v \rangle$ $\rightarrow O(m + n)$ Zeit
 - **Test am Anfang der while-Schleife** läuft über **IND** und sucht nach dem ersten Knoten v mit **IND**[v] = 0 $\rightarrow O(n)$ Zeit pro Iteration
 - **Entfernen von einem Knoten v**: Iteriere über die Adjazenzliste von v und für jeden Knoten w reduziere **IND**[w] um 1. (Optional: Setze danach **adjliste**[v] auf NIL um v zu entfernen.)
 $\rightarrow O(outdegree(v)) \subseteq O(n)$ Zeit pro Iteration
 - Höchstens n Iterationen der while-Schleife
- \rightarrow insgesamt $O(m + n + n^2) = O(n^2)$ Laufzeit

Implementierung mit *Adjazenzlistendarstellung* und zusätzlichen Datenstrukturen:

Speichere Eingangsgrad jedes Knotens in Array **IND**.

NEU: Speichere Knoten mit Eingangsgrad 0 in einer Liste **SRC**.

- **Initialisierung** von **SRC** und **IND** vor der while-Schleife: Initialisiere **IND** mit 0 für jeden Knoten. Traversiere all Kanten, d.h. alle Adjazenzlisten und erhöhe **IND**[v] um 1 für jede Kante $\langle u, v \rangle$. **Iteriere über alle Knoten und speichere jeden Knoten v mit $\text{IND}[v] = 0$ in SRC** $\rightarrow O(m + n)$ Zeit
- **Test am Anfang der while-Schleife** **prüft ob SRC leer ist und entnimmt ansonsten einen beliebigen Knoten v** $\rightarrow O(1)$ Zeit pro Iteration
- **Entfernen von einem Knoten v:** Iteriere über die Adjazenzliste von v und für jeden Knoten w reduziere **IND**[w] um 1. **Falls $\text{IND}[w]$ dadurch auf 0 reduziert wird, füge w der Liste SRC hinzu.**
 $\rightarrow O(\text{outdegree}(v))$ Zeit pro Iteration
- **In jeder Iteration der while-Schleife wird genau ein Knoten bearbeitet**

\rightarrow insgesamt $O(m + n + \sum_{v \in V} (1 + \text{outdegree}(v))) = O(n + m)$

Unter dem **Traversieren** eines Graphen versteht man das systematische und vollständige Besuchen aller Knoten des Graphen

Es lassen sich prinzipiell 2 Ansätze unterscheiden:

Tiefensuche, dfs

depth-first search - Traversierung

Breitensuche, bfs

breadth-first search - Traversierung

Über diese beiden Ansätzen lassen sich fast alle wichtigen Problemstellungen auf Graphen lösen, z.B.

Suche einen Weg vom Knoten v nach w ?

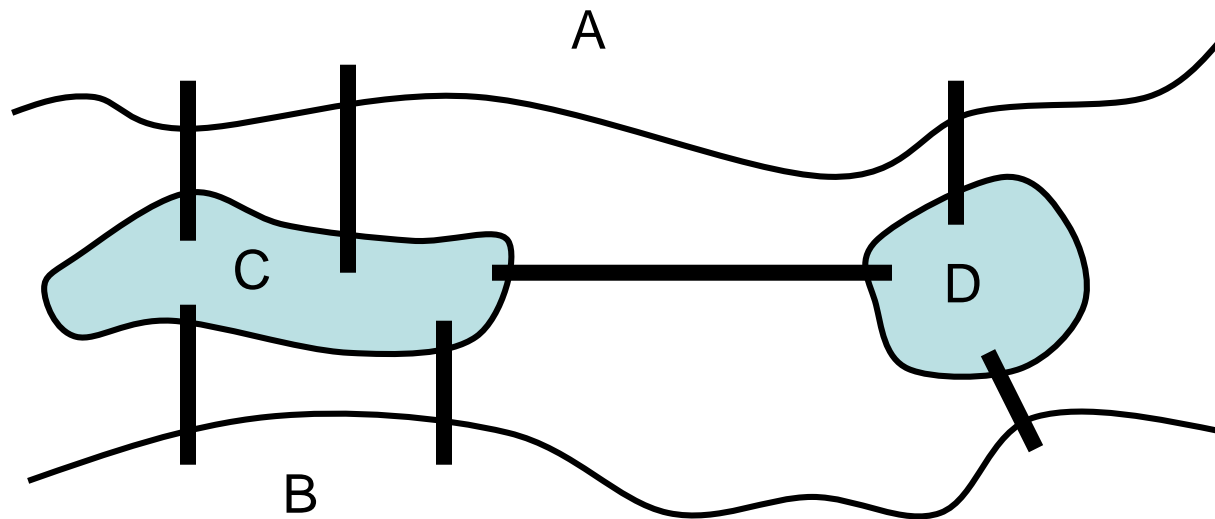
Besitzt der Graph einen Zyklus?

Finde alle Komponenten?

...

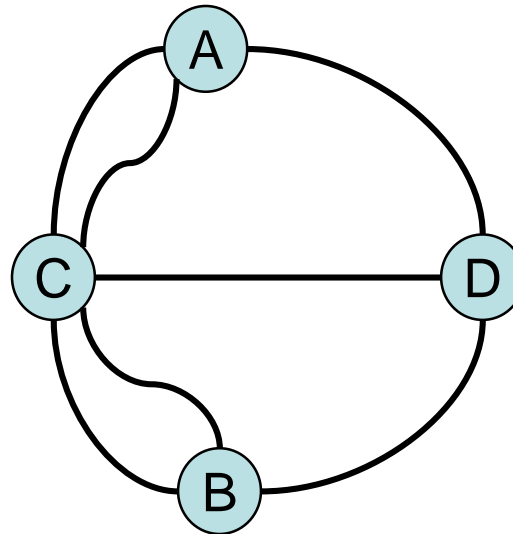
Leonhard Euler, 1736

Die Stadt Königsberg (Kaliningrad) liegt an den Ufern und auf 2 Inseln des Flusses Pregel und ist durch 7 Brücken verbunden



Frage: Gibt es einen Weg auf dem ich die Stadt besuchen kann, alle Brücken genau einmal überquere und an den Anfangspunkt zurückkehre?

Frage: Gibt es einen Kreis im Graphen, der alle Kanten genau einmal enthält?



Euler löste das Problem, indem er bewies, dass so ein Kreis genau dann möglich ist, wenn der Graph zusammenhängend und der Knotengrad aller Knoten gerade ist

Solche Graphen werden **Eulersche Graphen** genannt.

Für Kaliningrad gilt dies offensichtlich nicht.

Eulersche Graphen werden als das erste gelöste Problem der Graphentheorie angesehen

Satz: Ein Graph ist Eulersch *genau dann*, wenn er zusammenhängend ist und alle Knoten geraden Grad haben.

Hinrichtung (→): Wenn ein Graph Eulersch ist, ist er zusammenhängend und alle Knoten haben geraden Grad.

Jeder Knoten muss genauso oft betreten wie verlassen werden → gerader Grad

Alle Kanten sind teil desselben Kreises → Graph ist zusammenhängend

Rückrichtung (←): Wenn ein Graph zusammenhängend ist und alle Knoten geraden Grad haben, ist er Eulersch.

Beweis durch Induktion:

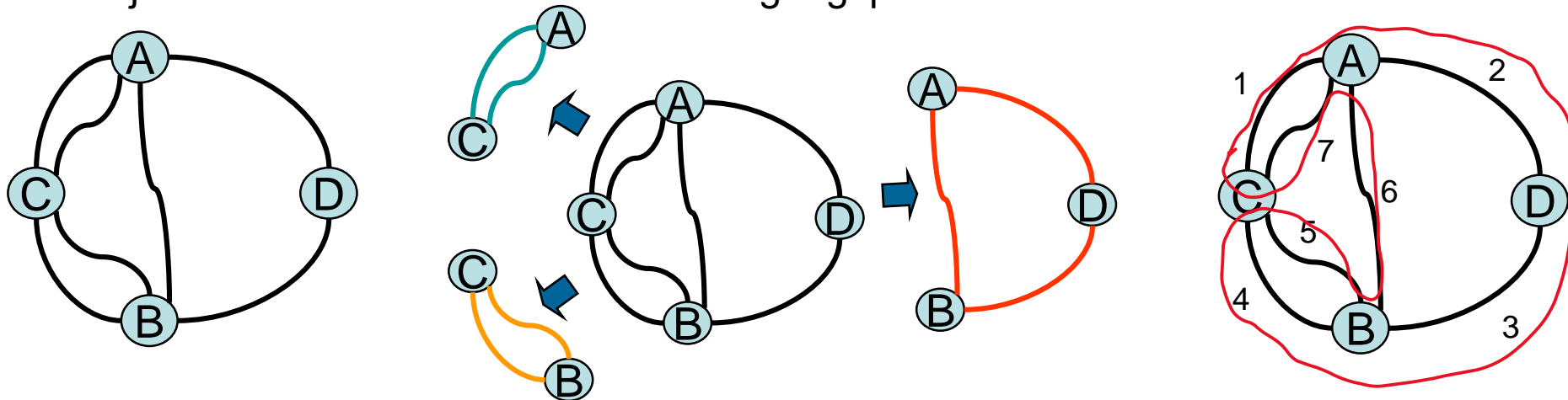
Induktionsanfang: Ein leerer Graph mit 0 Knoten und 0 Kanten hat trivialerweise einen (leeren) Kreis, der alle Kanten genau einmal enthält.

Hypothese: Ein zusammenhängender Graph mit $< m$ Kanten, in dem alle Knoten geraden Grad besitzen, ist Eulersch.

Induktionsschritt: Sei G ein zusammenhängender Graph mit $m > 0$ Kanten, in dem alle Knoten geraden Grad besitzen. Entferne einen Kreis P (gerader Knotengrad, gerade Anzahl von Kanten) aus G . Im verbleibenden Graphen G' haben wieder alle Knoten geraden Grad, aber...

Problem: G' könnte in Komponenten G'_1, G'_2, \dots, G'_k zerfallen, wobei aber jede Komponente die Bedingungen der Hypothese erfüllt. Die Anwendung ergibt also k Kreise P_1, P_2, \dots, P_k , die jeweils alle Kanten von G'_1, G'_2, \dots, G'_k enthalten.

Lösung: Kombiniere P_1, P_2, \dots, P_k und $P_{k+1} := P$ zu einem Gesamtkreis wie folgt: Beginne bei einem beliebigen Knoten im Kreis P_1 und folge P_1 solange, bis man zu einem Knoten v_i kommt, der auch zu G'_i ($i \neq 1$) gehört. Von dort verfolge P_i zu einem Knoten v_j , der auch zu G'_j ($1 \neq j \neq i$) gehört, usw. Wenn es keinen Knoten gibt, der zu einem noch nicht gesehenen Kreis gehört, folge dem jeweils aktuellen Kreis bis zum Ausgangspunkt...



Wie kombiniert man die Kreise systematisch? Wie traversiert man einen Graphen? → nächstes Thema!

Idee

Wir interpretieren den Graphen als Labyrinth, wobei die **Kanten Wege** und die **Knoten Kreuzungen** darstellen.

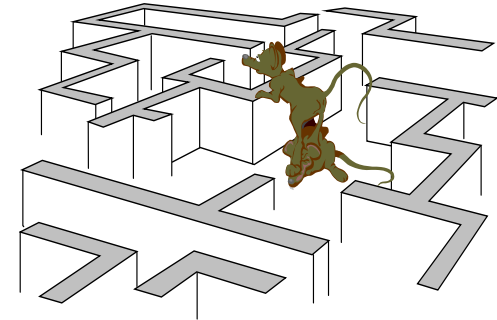
Beim depth-first search Ansatz versuchen wir einen möglichst langen neuen Weg zurückzulegen.

Wenn wir keinen neuen Weg mehr finden, gehen wir zurück und versuchen den nächsten, noch nicht besuchten Weg.

Wie finden wir den nächsten, noch nicht besuchten Weg?

Wenn wir zu einer unmarkierten Kreuzung kommen, markieren wir sie (im Labyrinth durch einen Stein) und merken uns alle möglichen Wegalternativen.

Kommen wir zu einer bereits markierten Kreuzung, gehen wir *denselben* Weg wieder solange zurück bis wir auf eine Kreuzung mit unbesuchter Wegalternative stoßen. Dies ist der nächste, noch nicht besuchte Weg.



Analogie

Buch lesen, Detailinformation suchen, Entscheidungsbaum,
Auswahlkriterien, etc.

2 Ansätze

Rekursiv ohne explizite (weitere) Datenstruktur

Iterativ (nicht-rekursiv) mit Stack

Kann auf gerichteten und ungerichteten Graphen angewandt
werden.

Hier: ungerichtet (gerichtet funktioniert analog)

Rekursiver Algorithmus

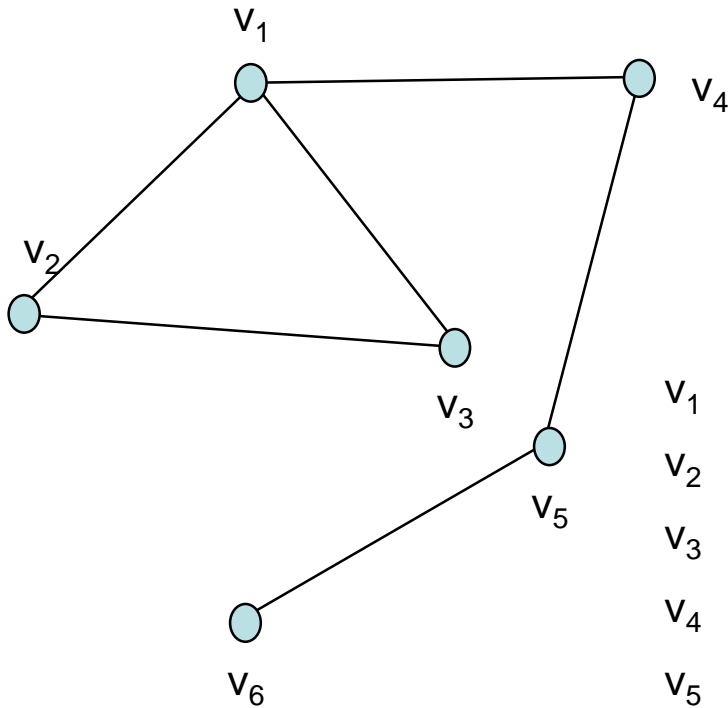
zu besuchende Knoten werden (automatisch) am Rekursionsstack vermerkt.

“Zuerst in die Tiefe, danach in die Breite gehen”

```
void besuche-dfs ( Knoten x ) {  
    markiere Knoten x mit 'besucht';  
    for ( jeden zu x adjazenten Knoten v )  
        if ( v ist noch unbesucht )  
            besuche-dfs ( v );  
}
```

Rekursiver Ansatz

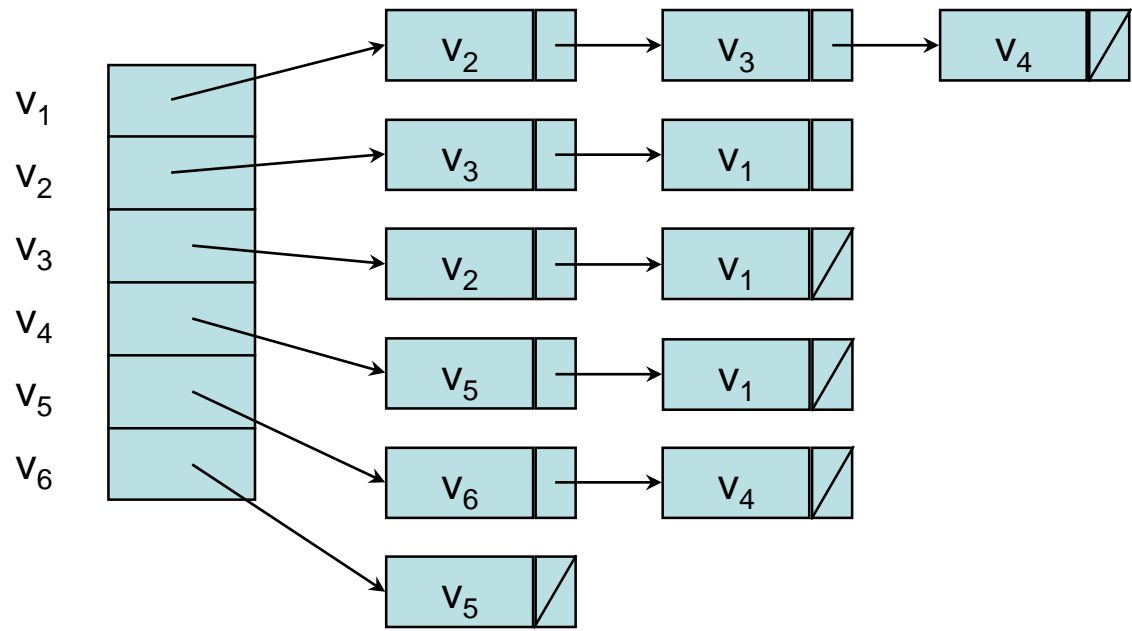
Markierung: Ein Feld mit den Knotenbezeichnungen als Index, initialisiert mit 'unbesucht' (keine Steine).

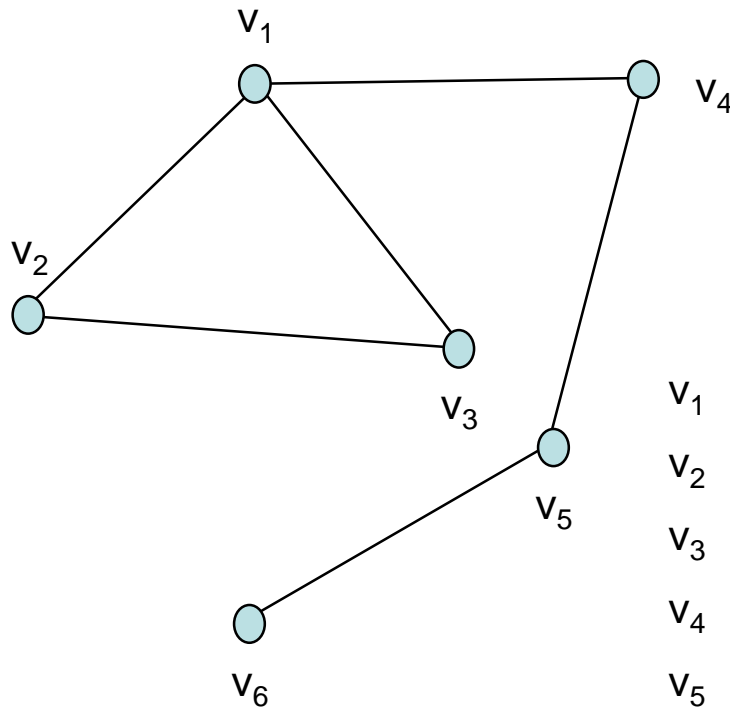


Markierung

v ₁	v ₂	v ₃	v ₄	v ₅	v ₆
u	u	u	u	u	u

Adjazenzliste

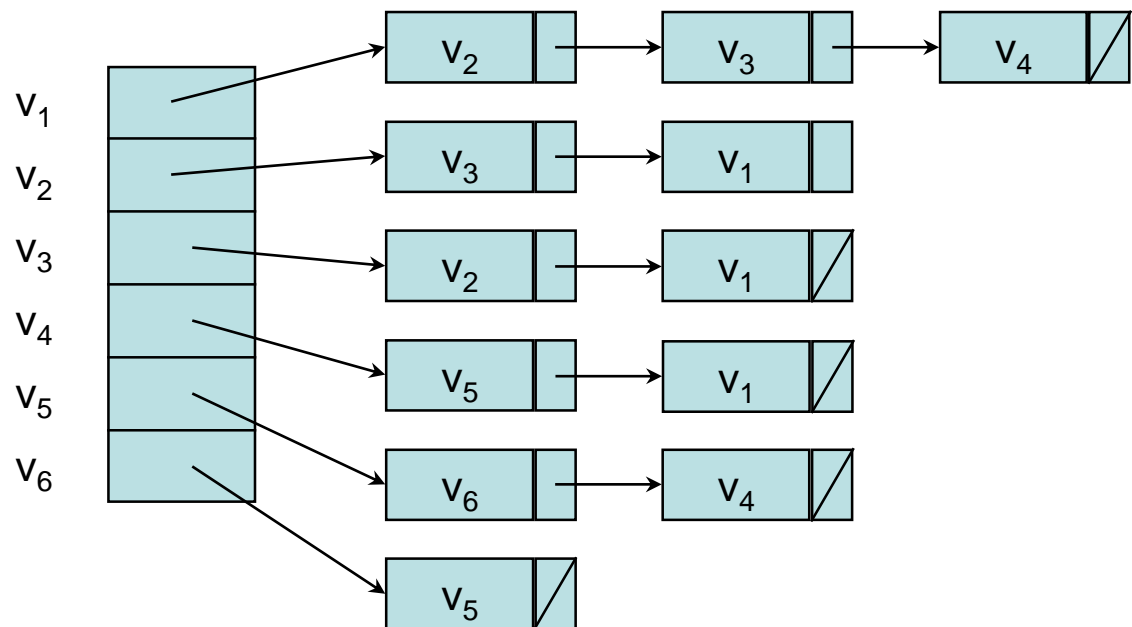




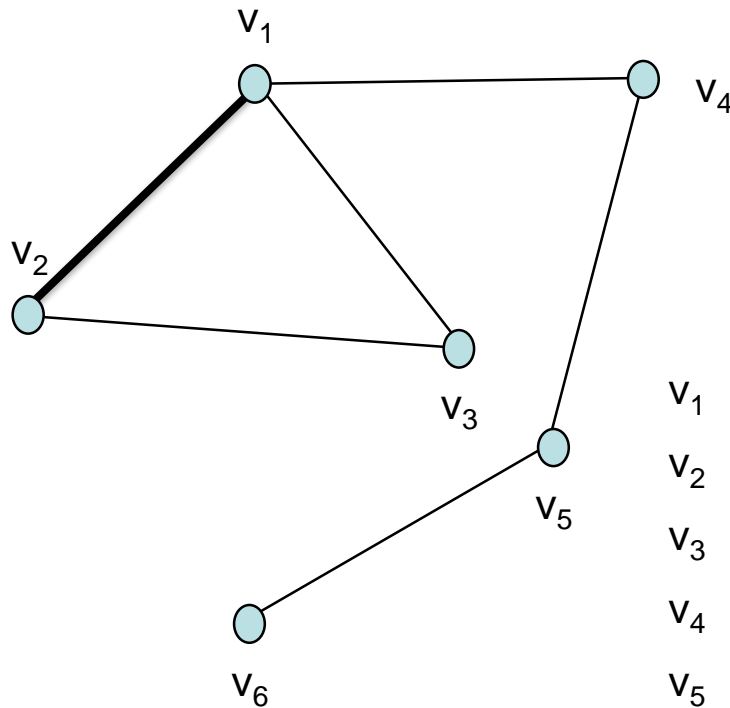
Markierung

v_1	v_2	v_3	v_4	v_5	v_6
b	u	u	u	u	u

Adjazenzliste



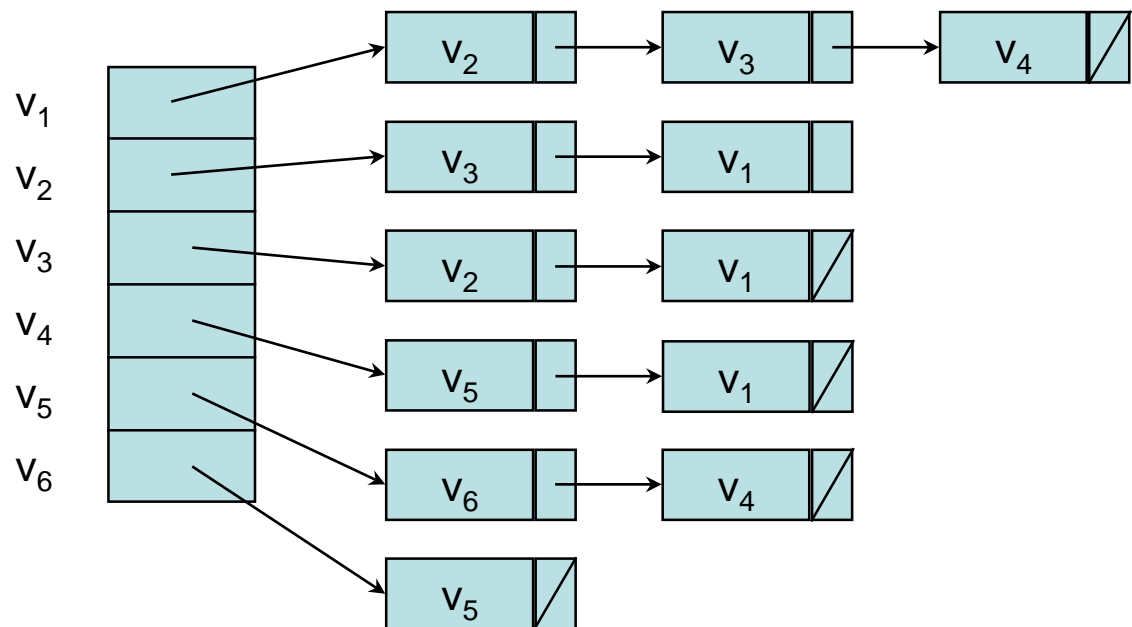
besuche-dfs (Knoten v_1)



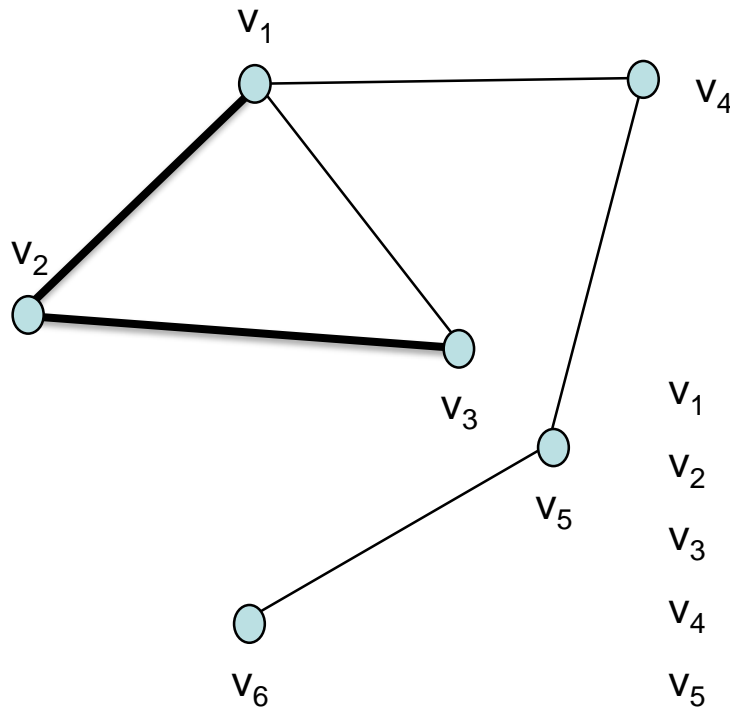
Markierung

v_1	v_2	v_3	v_4	v_5	v_6
b	b	u	u	u	u

Adjazenzliste



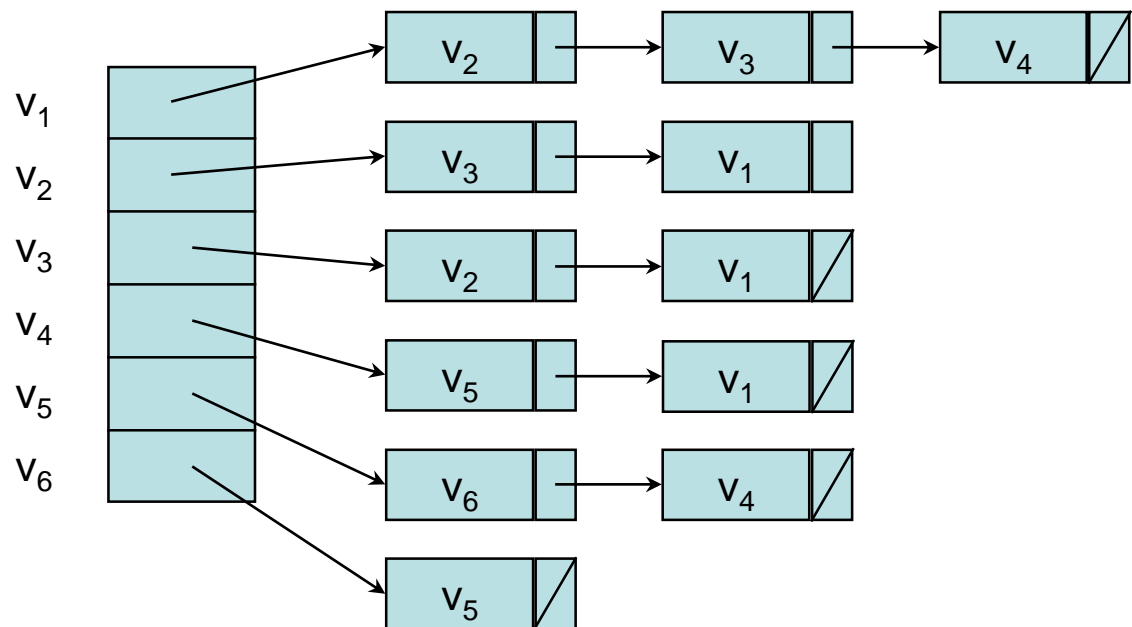
besuche-dfs (Knoten v_2)



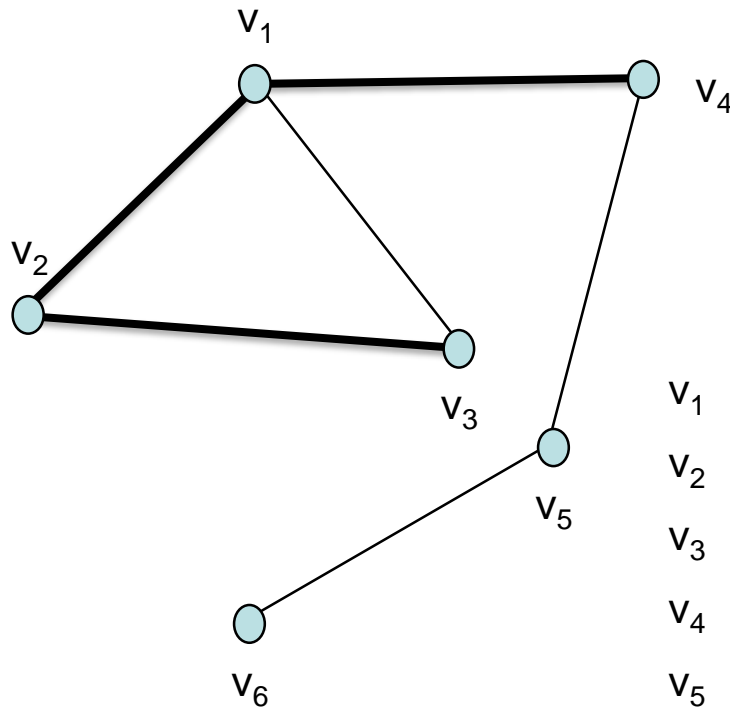
Markierung

v_1	v_2	v_3	v_4	v_5	v_6
b	b	b	u	u	u

Adjazenzliste



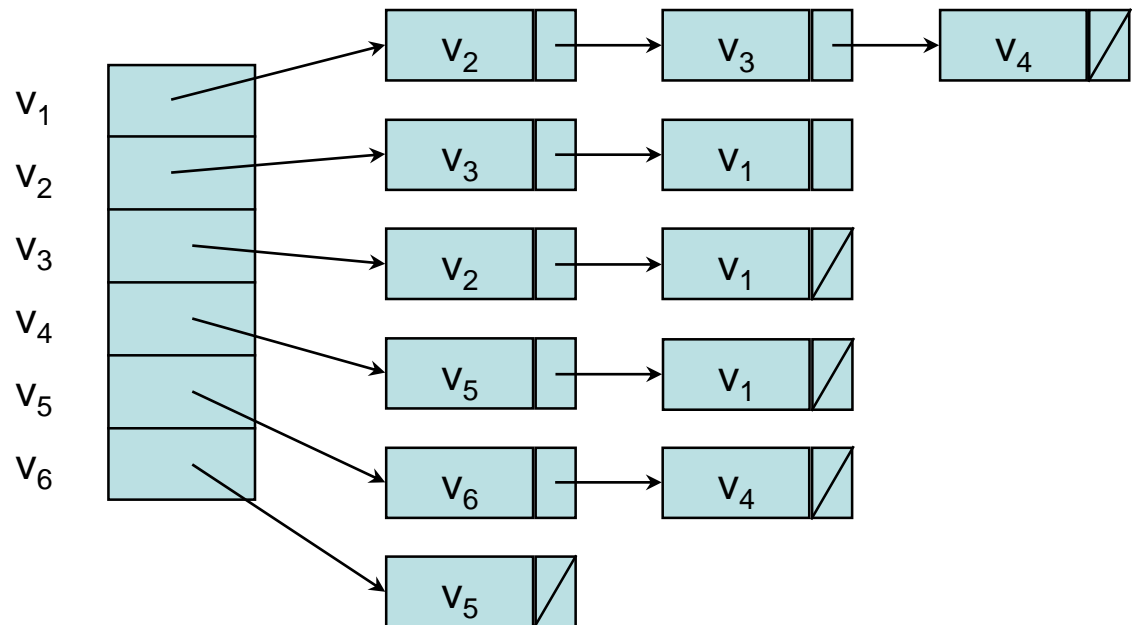
besuche-dfs (Knoten v_3)



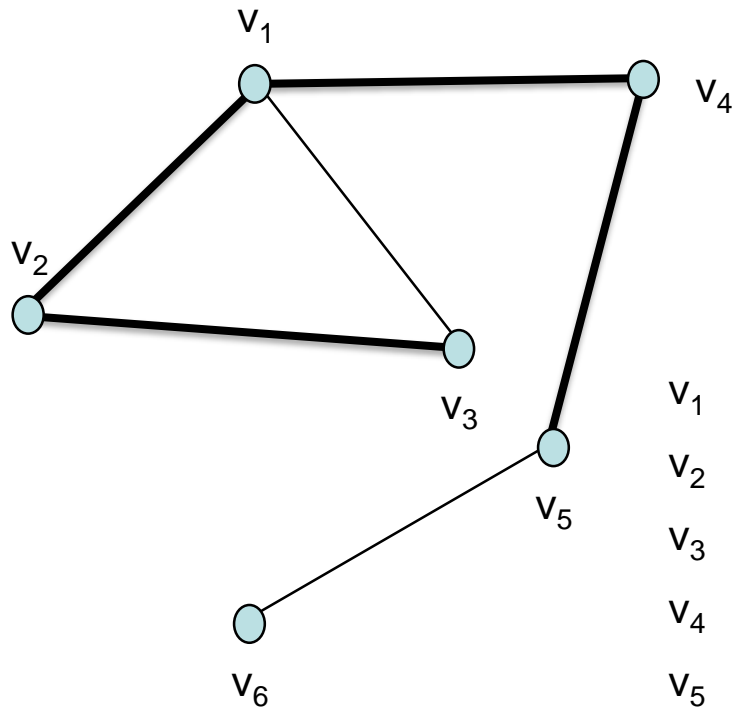
Markierung

v_1	v_2	v_3	v_4	v_5	v_6
b	b	b	b	u	u

Adjazenzliste



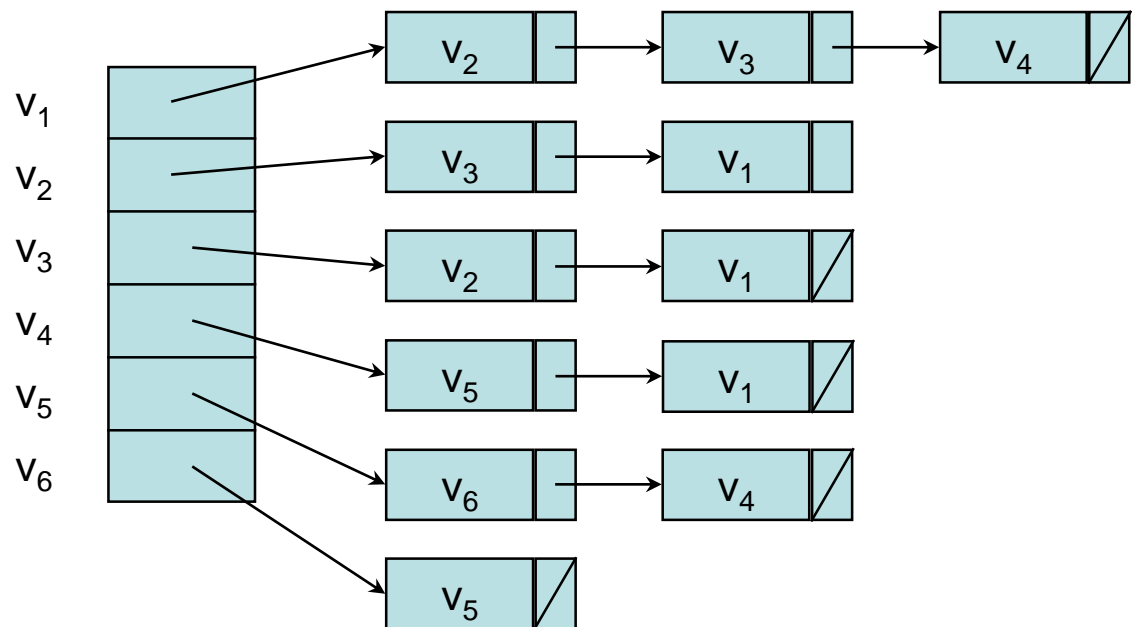
besuche-dfs (Knoten v_4)



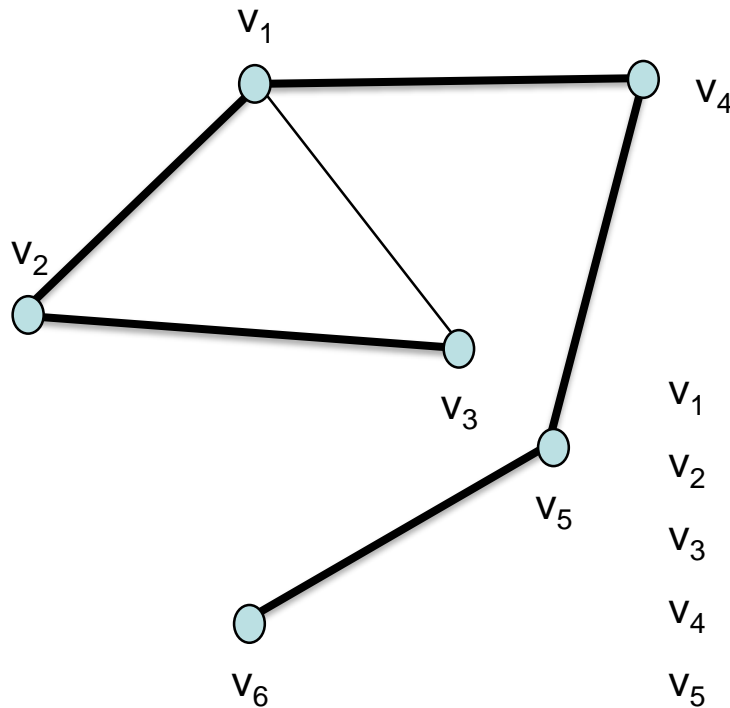
Markierung

v_1	v_2	v_3	v_4	v_5	v_6
b	b	b	b	b	u

Adjazenzliste



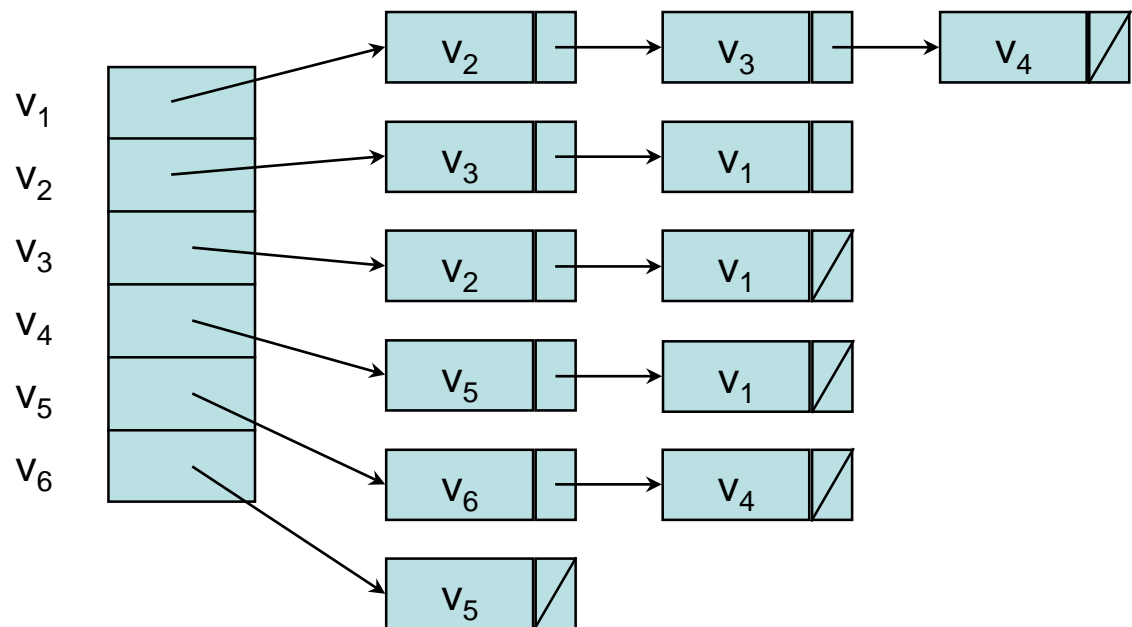
besuche-dfs (Knoten v_5)



Markierung

v_1	v_2	v_3	v_4	v_5	v_6
b	b	b	b	b	b

Adjazenzliste

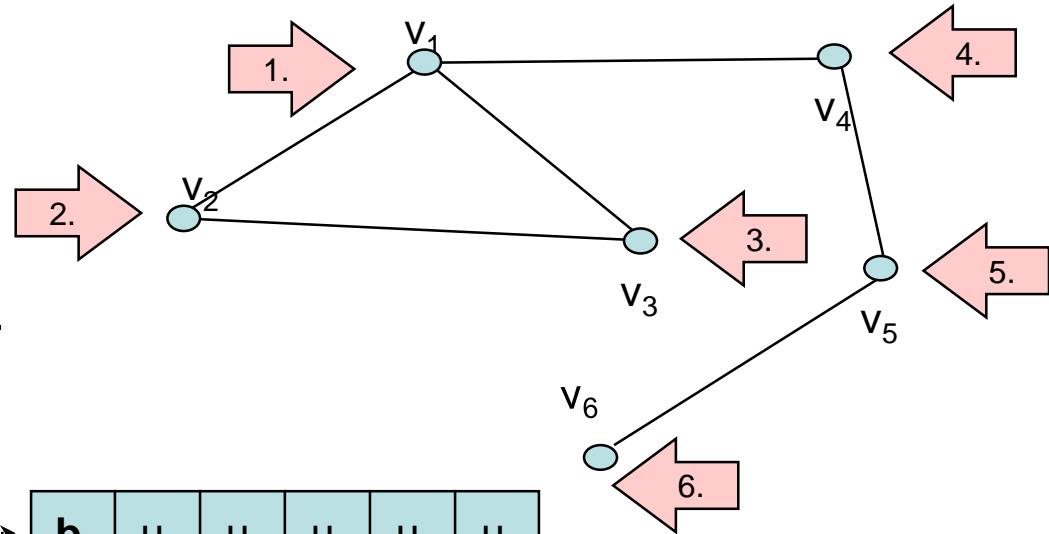


besuche-dfs (Knoten v_6)

dfs-Traversierung eines Graphen

Ausgehend vom Knoten v_1 wird der Graph mit dem dfs-Ansatz traversiert.

Besuchte Knoten werden auf dem Rekursionsstack vermerkt.



besuche-dfs(1)→	<table><tr><td>b</td><td>u</td><td>u</td><td>u</td><td>u</td><td>u</td></tr></table>	b	u	u	u	u	u
b	u	u	u	u	u			
besuche-dfs(2)→	<table><tr><td>b</td><td>b</td><td>u</td><td>u</td><td>u</td><td>u</td></tr></table>	b	b	u	u	u	u
b	b	u	u	u	u			
besuche-dfs(3)→	<table><tr><td>b</td><td>b</td><td>b</td><td>u</td><td>u</td><td>u</td></tr></table>	b	b	b	u	u	u
b	b	b	u	u	u			
besuche-dfs(4)→	<table><tr><td>b</td><td>b</td><td>b</td><td>b</td><td>u</td><td>u</td></tr></table>	b	b	b	b	u	u
b	b	b	b	u	u			
besuche-dfs(5)→	<table><tr><td>b</td><td>b</td><td>b</td><td>b</td><td>b</td><td>u</td></tr></table>	b	b	b	b	b	u
b	b	b	b	b	u			
besuche-dfs(6)→	<table><tr><td>b</td><td>b</td><td>b</td><td>b</td><td>b</td><td>b</td></tr></table>	b	b	b	b	b	b
b	b	b	b	b	b			

Woher weiss das Programm, welche Kanten von Knoten 1 noch nicht überprüft sind?

```
#define besucht 1
#define unbesucht 0
// maxV defined elsewhere

class node { public: int v; node *next;}
node *adjliste[maxV]; // Adjazenzliste
int mark[maxV]; // Knotenmarkierung

void traversiere() {
    int k;
    for (k = 0; k < maxV; ++k)
        mark[k] = unbesucht;
    for (k = 0; k < maxV; ++k)
        if (mark[k] == unbesucht)
            besuche-dfs(k);
}

void besuche-dfs(int k) {
    mark[k] = besucht;
    for (node *t = adjliste[k]; t != NULL; t = t->next)
        if (mark[t->v] == unbesucht)
            besuche-dfs(t->v);
}
```

Iterativer Ansatz

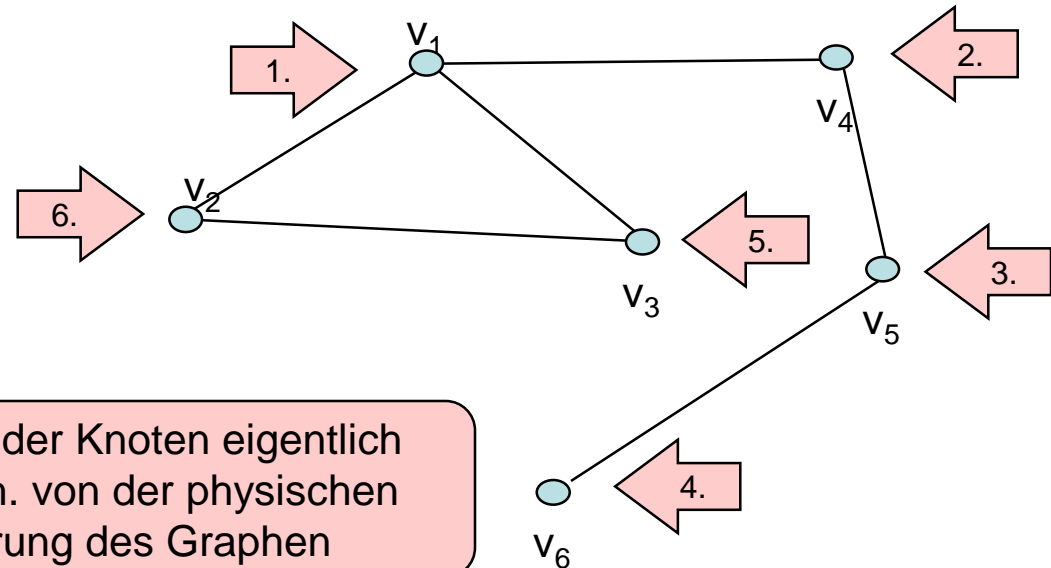
zu besuchende Knoten werden in einer Stack-Datenstruktur gespeichert,
nicht im (automatischen) Rekursionsstack

```
void besuche-dfs( Knoten x ) {  
    stelle (Push) Knoten x auf den Stack;  
    while( Stack nicht leer ) {  
        hole (Pop) letzten Knoten y vom Stack;  
        if ( y bereits 'besucht' ) continue;  
        markiere Knoten y 'besucht';  
        for (jeden zu y adjazenten Knoten v ) {  
            if ( v bereits 'besucht' ) continue; // optional  
            stelle (Push) v auf den Stack;  
        } // for  
    } // while  
}
```

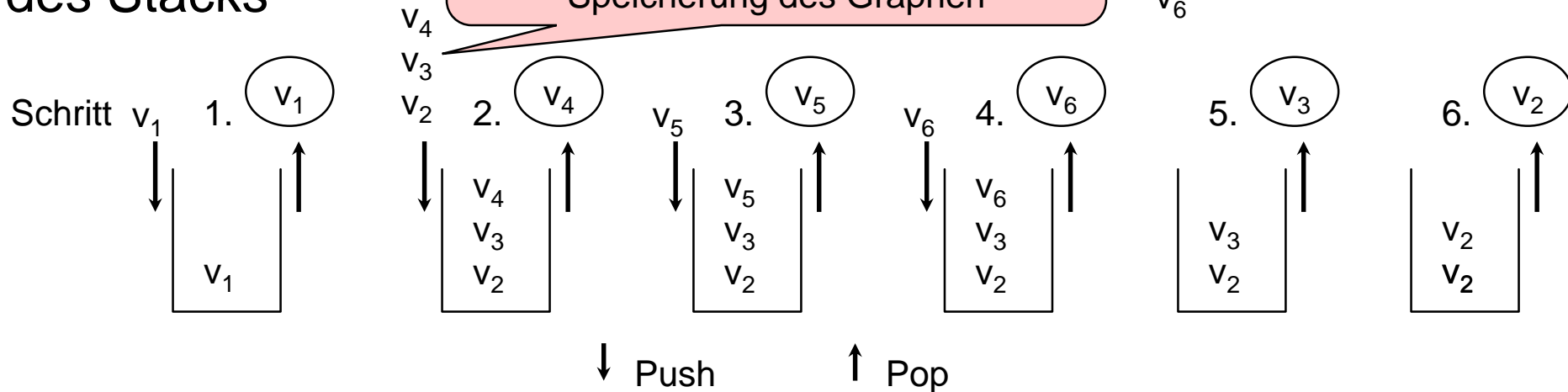
Iterativer Ansatz

dfs-Traversierung eines Graphen

Ausgehend vom Knoten v_1
wird der Graph mit dem
dfs-Ansatz traversiert.
Noch zu besuchende Knoten
werden auf einem Stack
vermerkt.



Verhalten des Stacks



```
// besucht, unbesucht defined as before, maxV defined elsewhere
```

```
class node { public: int v; node *next;}
node *adjliste[maxV]; // Adjazenzliste
int mark[maxV]; // Knotenmarkierung
Stack stack(maxV*(maxV-1)/2);

void traversiere() {
    int k;
    for(k = 0; k < maxV; ++k) mark[k] = unbesucht;
    for(k = 0; k < maxV; ++k)
        if(mark[k] == unbesucht) besuche-dfs(k);
}

void besuche-dfs(int k) {
    stack.Push(k);
    while(!stack.IsStackEmpty()) {
        k = stack.Pop();
        if (mark[k] == besucht) continue;
        mark[k] = besucht;
        for(node *t = adjliste[k]; t != NULL; t = t->next){
            if(mark[t->v] == unbesucht)
                stack.Push(t->v);
        }
    }
}
```


Methode traversiere ohne besuche-dfs:

$O(n)$ Schritte

Methode besuche-dfs:

Mit Adjazenzliste:

Für jeden Knoten t : $O(1 + |adjliste[t]|)$ Schritte zum Finden der Nachfolgerknoten

Jeder Knoten wird einmal besucht \rightarrow Für alle Knoten:

$$O(\sum_{t \in V} (1 + |adjliste[t]|)) = O(\sum_{t \in V} 1 + \sum_{t \in V} |adjliste[t]|) = O(n + m)$$

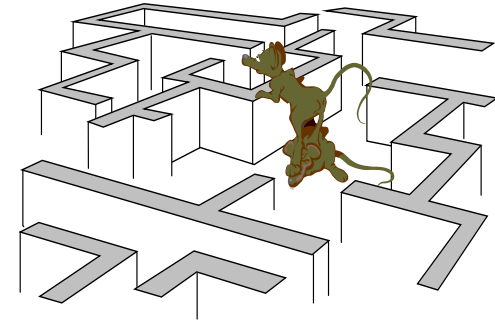
Total: $O(n + n + m) = O(n + m)$

Mit Adjazenzmatrix:

Für jeden Knoten t : $O(n)$ zum Finden der Nachfolgerknoten

Jeder Knoten wird einmal besucht \rightarrow Für alle Knoten: $O(n^2)$

Total: $O(n + n^2) = O(n^2)$



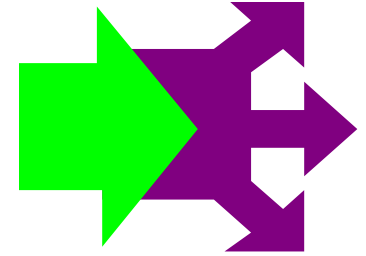
Da $m \in O(n^2)$ sind Adjazenzlisten im Allgemeinen schneller

Idee

Man leert einen Topf mit Tinte auf den Startknoten. Die Tinte ergießt sich in alle Richtungen (über alle Kanten) auf einmal

Beim breadth-first search Ansatz werden alle möglichen Alternativen auf einmal erforscht, über die gesamte Breite der Möglichkeiten

Dies bedeutet, dass zuerst alle möglichen, von einem Knoten weggehenden, Kanten untersucht werden, und danach erst zum nächsten Knoten weitergegangen wird



Analogie

Überblick über Buch verschaffen, Generelle Information suchen, Hierarchischer Lernansatz, Auswahlüberblick, Welle, etc.

Ein Ansatz

Iterativ mit Queue-Datenstruktur

Iterativer Ansatz

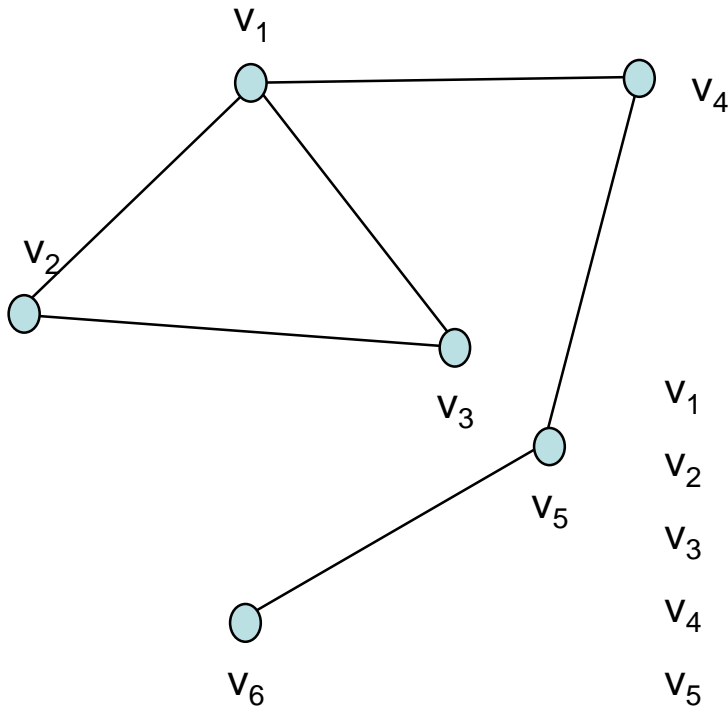
zu besuchende Knoten werden in einer Queue gemerkt

“Zuerst in die Breite, danach in die Tiefe gehen”

```
void besuche-bfs(Knoten x) {  
    stelle (Enqueue) Knoten x in Queue;  
    markiere Knoten x 'besucht';  
    while(Queue nicht leer) {  
        hole (Dequeue) ersten Knoten y von der Queue;  
        for(alle zu y adjazenten nicht besuchten Knoten v) {  
            stelle (Enqueue) v in die Queue;  
            markiere Knoten v 'besucht';  
        } // for  
    } // while  
}
```

Iterativer
Ansatz

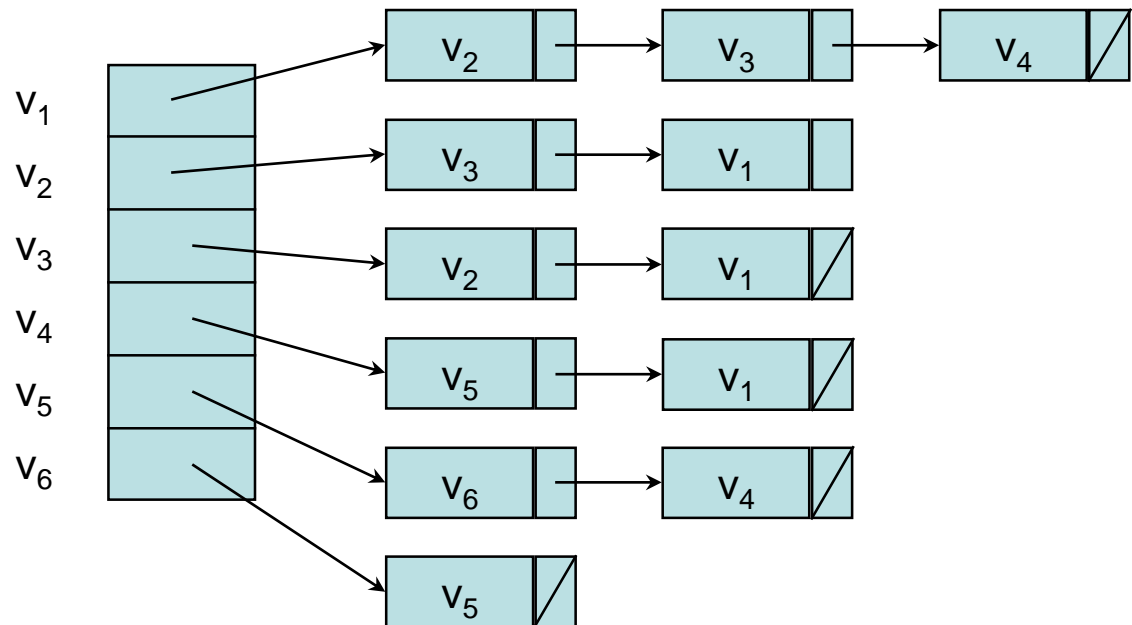
Beispiel 1, bfs



Markierung

v ₁	v ₂	v ₃	v ₄	v ₅	v ₆
u	u	u	u	u	u

Adjazenzliste

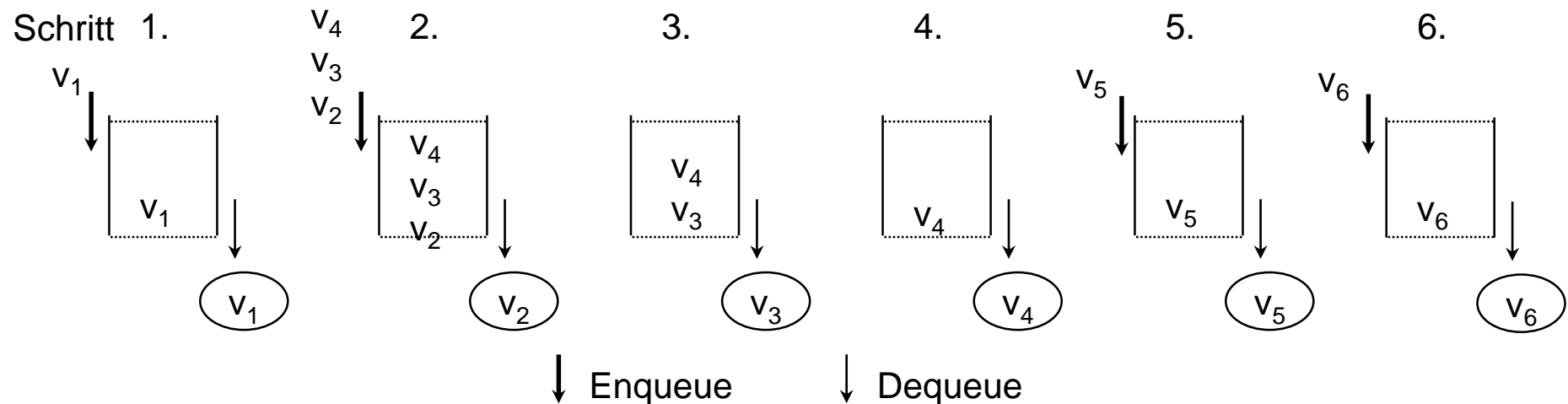
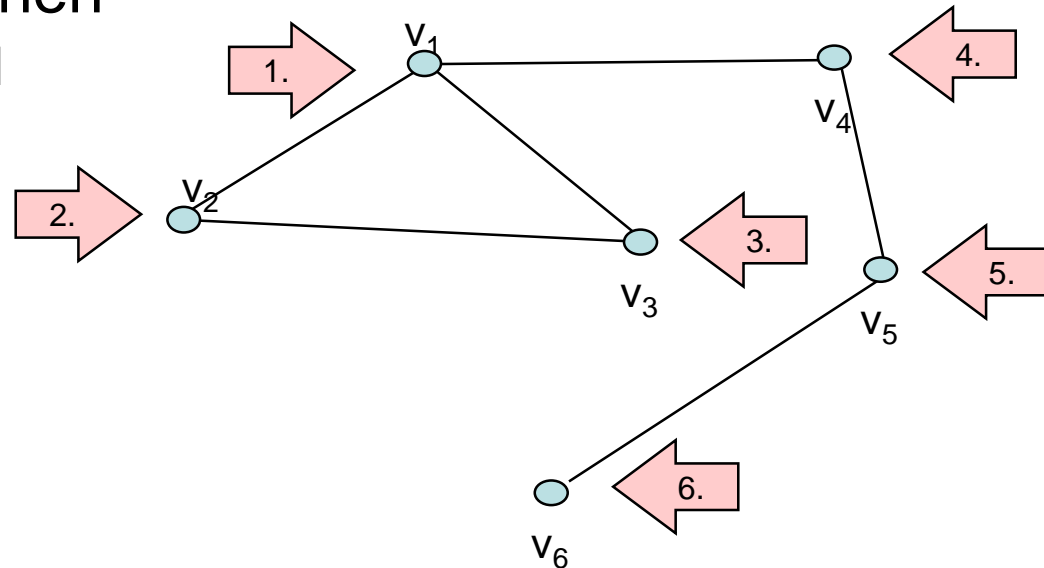


bfs-Traversierung eines Graphen

Ausgehend vom Knoten v_1 wird
der Graph mit dem
bfs-Ansatz traversiert

Besuchte Knoten werden in
einer Queue vermerkt

Verhalten der Queue



Beispiel 1, bfs (2)



besuche-bfs(1)

Enqueue(1), Dequeue(1)

Enqueue(2,3,4)

Dequeue(2,3,4)

Enqueue(5)

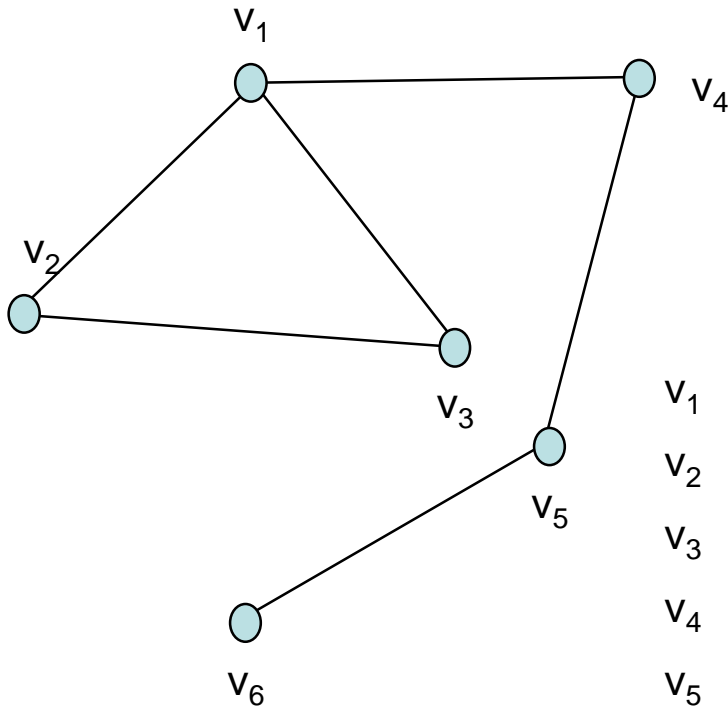
Dequeue(5)

Enqueue(6)

Dequeue(6)

v_1	v_2	v_3	v_4	v_5	v_6
b	u	u	u	u	u
b	b	u	u	u	u
b	b	b	u	u	u
b	b	b	b	u	u
b	b	b	b	b	u
b	b	b	b	b	b

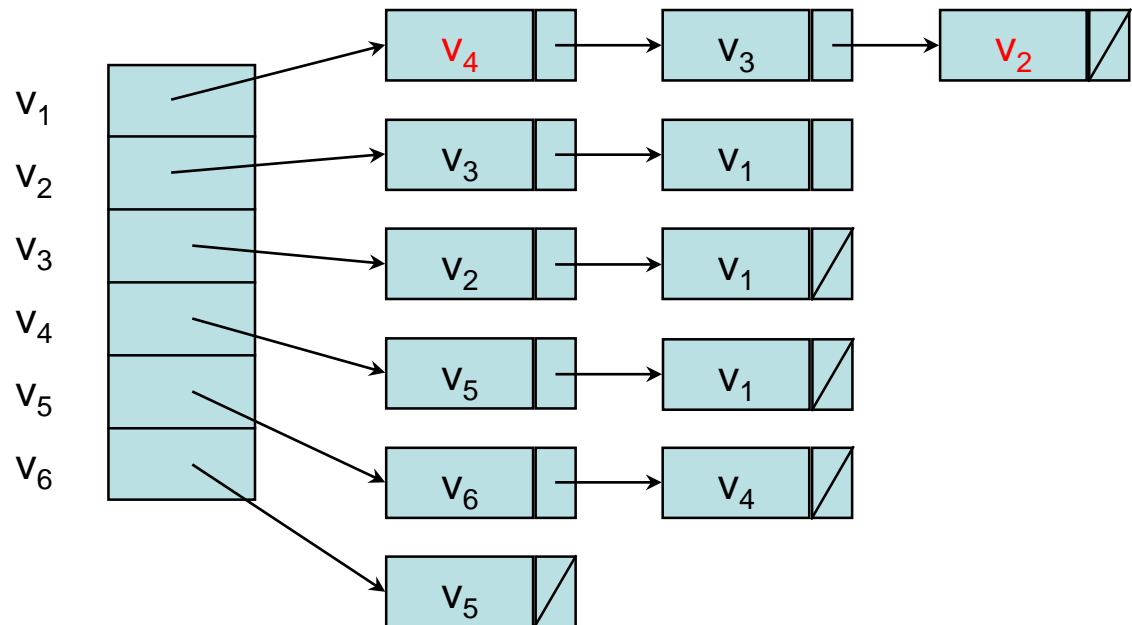
Beispiel 2, bfs



Markierung

v_1	v_2	v_3	v_4	v_5	v_6
u	u	u	u	u	u

Adjazenzliste

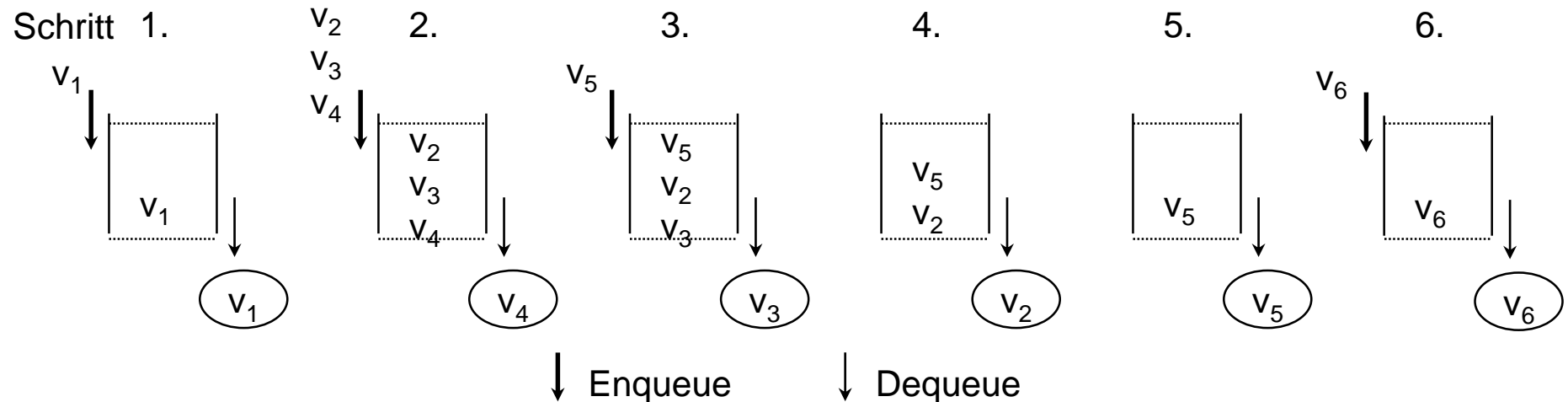
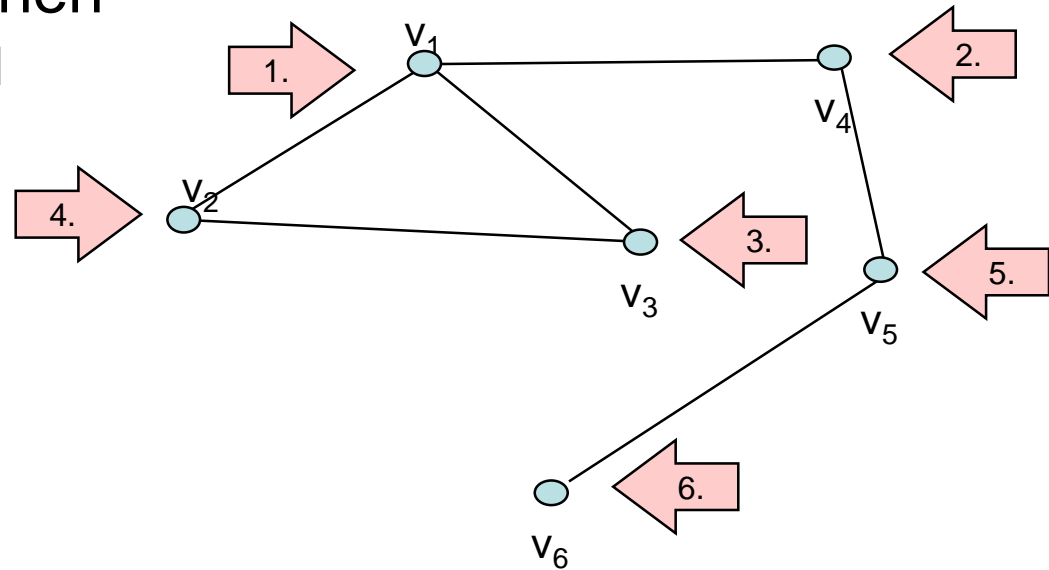


bfs-Traversierung eines Graphen

Ausgehend vom Knoten v_1 wird
der Graph mit dem
bfs-Ansatz traversiert

Besuchte Knoten werden in
einer Queue vermerkt

Verhalten der Queue



Beispiel 2, bfs (3)



besuche-bfs(1)

Enqueue(1), Dequeue(1)

Enqueue(4,3,2)

Dequeue(4)

Enqueue(5)

check 1 besucht

Dequeue(3)

check 2,1 besucht

Dequeue(2)

check 3,1 besucht

Dequeue(5)

Enqueue(6)

check 4 besucht

Dequeue(6)

check 5 besucht

v_1	v_2	v_3	v_4	v_5	v_6
b	u	u	u	u	u
b	u	u	b	u	u
b	u	b	b	u	u
b	b	b	b	u	u
b	b	b	b	b	u
b	b	b	b	b	b



Verschiedene Repräsentationen des Graphen führen zu verschiedener Reihenfolge der Knotenbesuche, aber unabhängig von der Repräsentation besucht BFS...

- ... zuerst alle Knoten, deren kürzester Weg zu v_1 Länge 1 hat,
- ... danach alle Knoten, deren kürzester Weg zu v_1 Länge 2 hat,
- ... danach alle Knoten, deren kürzester Weg zu v_1 Länge 3 hat
- ...

// besucht, unbesucht defined as before, maxV defined elsewhere

```
class node { public: int v; node *next;}
node *adjliste[maxV]; // Adjazenzliste
int mark[maxV]; // Knotenmarkierung
Queue queue(maxV);
void traversiere() {
    int k;
    for(k = 0; k < maxV; ++k) mark[k] = unbesucht;
    for(k = 0; k < maxV; ++k)
        if(mark[k] == unbesucht) besuche-bfs(k);
}
void besuche-bfs(int k) {
    queue.Enqueue(k);
    mark[k] = besucht;
    while(!queue.IsEmpty()) {
        k = queue.Dequeue();
        for(node *t = adjliste[k]; t != NULL; t = t->next) {
            if(mark[t->v] == unbesucht) {
                queue.Enqueue(t->v);
                mark[t->v] = besucht;
            }
        }
    }
}
```

Gleich!

Warum?

Gleiches Traversierungsprinzip, nur unterschiedliche
Datenstruktur (Stack oder Queue) mit gleicher Laufzeit für
Einfüge und Löschooperationen

Genereller iterativer Ansatz zur Traversierung eines Graphen mit Hilfe 2 (simpler) Listen

OpenList

Speichert bekannte aber noch nicht besuchte Knoten

CloseList

Speichert alle schon besuchten Knoten

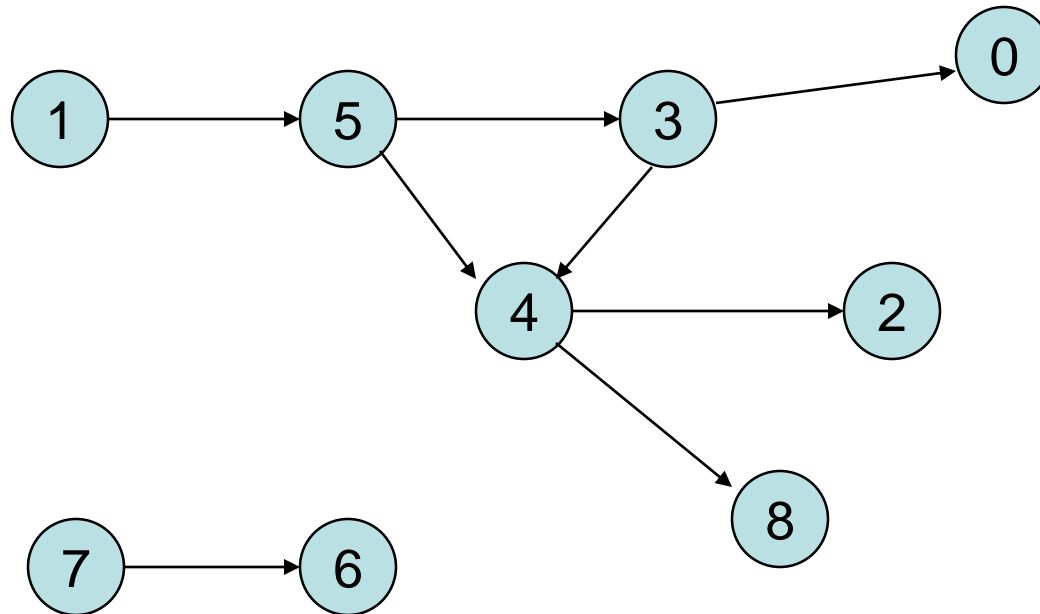
Expandieren eines Knoten

Generieren der Nachfolger eines Knoten

d.h. OpenList enthält alle generierten (bekannten), aber noch nicht expandierten Knoten, CloseList alle expandierten Knoten

Ohne weitere Steuerung Traversierungsansatz unsystematisch

```
Search(Knoten v) {  
    OpenList = [v];  
    CloseList = [];  
    while (OpenList != []) {  
        Akt = beliebigerKnotenAusOpenList; // ???  
        OpenList = OpenList \ [Akt];  
        CloseList = CloseList + [Akt];  
        process(Akt); // whatever to do  
        for(alle zu Akt adjazente Knoten v1){  
            if (v1  $\notin$  OpenList && v1  $\notin$  CloseList)  
                OpenList = OpenList + [v1];  
        }  
    }  
}
```



z.B.: **Search (5)**

OpenList: 5 | 3,4 | 4,0 | 0,2,8 | 2,8 | 8 |

CloseList: 5,3,4,0,2,8

Akt: 5 | 3 | 4 | 0 | 2 | 8

Es werden alle Knoten
besucht, die vom
Startknoten aus
erreichbar sind.
Unterschied gerichteter –
ungerichteter Graph?

Erweiterung von **Search**

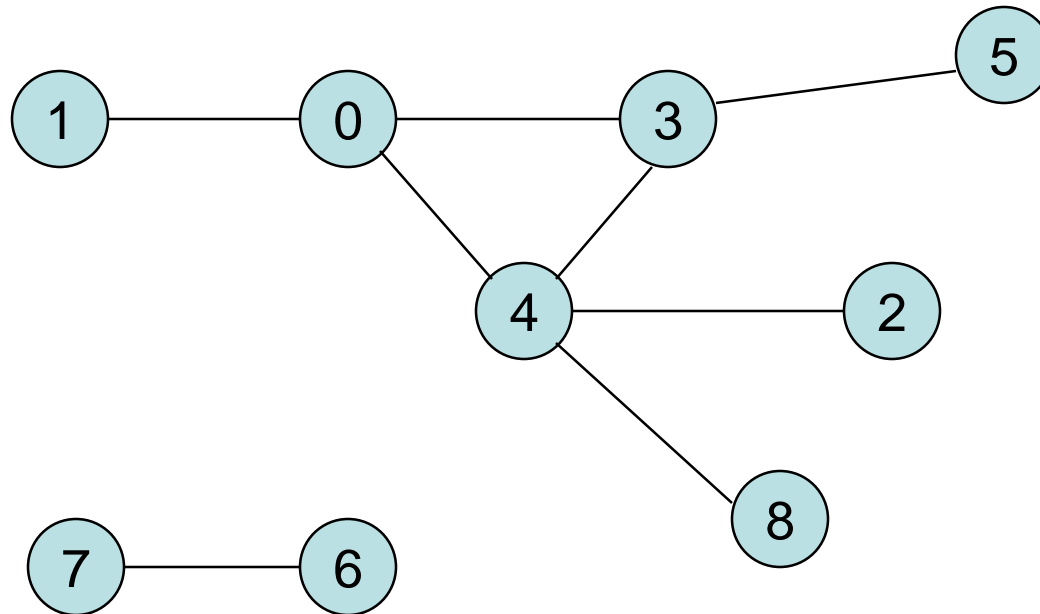
Feld zum Vermerken der Komponenten $K[n]$

```
int K_num = 0;
for (int i = 0; i < n; i++)
    if(K[i] == 0) { // i ist unbesucht
        K_num = K_num + 1;
        SearchAndMark(i, K_num);
    }
```

```
SearchAndMark(Knoten v, int K_num) {
    Version von Search mit
        process(Akt) { K[Akt] = K_num; }
}
```



```
SearchAndMark(Knoten v, int K_num) {
    OpenList = [v];
    CloseList = [];
    while (OpenList != []) {
        Akt = beliebigerKnotenAusOpenList;
        OpenList = OpenList \ [Akt];
        CloseList = CloseList + [Akt];
        K[Akt] = K_num;
        for(alle zu Akt adjazente Knoten v1) {
            if (v1  $\notin$  OpenList && v1  $\notin$  CloseList)
                OpenList = OpenList + [v1];
        }
    }
}
```



Frage:
Was wäre
bei einem
gerichteten
Graphen?

OpenList: 0 | 1,3,4 | 3,4 | 4,5 | 5,2,8 | 2,8 | 2 | 6 | 7

CloseList: 0,1,3,4,5,2,8 | 6,7

Akt: 0 | 1 | 3 | 4 | 5 | 2 | 8 | 6 | 7

K

0	1	2	3	4	5	6	7	8
1	1	1	1	1	1	2	2	1

Systematische Traversierung schon bekannt

Tiefensuche dfs

Breitensuche bfs

Unklarer Programmteil

`beliebigerKnotenAusOpenList`

Systematisierung durch Organisation der Auswahl

Aufbau der Liste

Position des einzufügenden Elements in der Liste

bfs: `OpenList = OpenList + [v1]`; am Ende

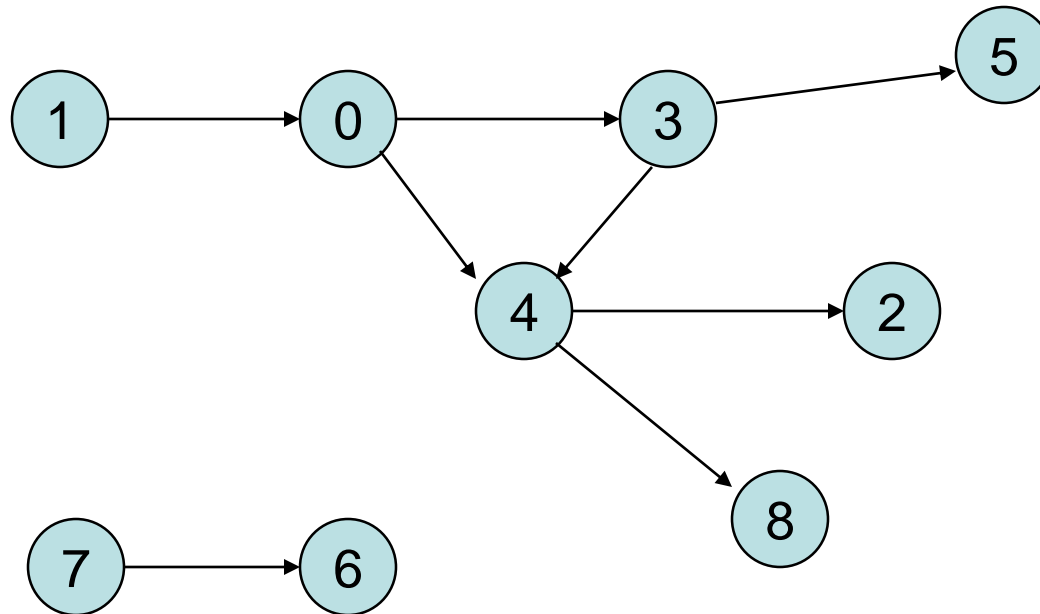
„Pseudo“-dfs/dfs-like: `OpenList = [v1] + OpenList`; am Anfang

Unterschied zu „echter“ dfs?

Zugriff auf die OpenList

Zugriff: `First(OpenList)`; Zugriff immer auf das erste Element

```
Search(Knoten v) {
  OpenList = [v];
  CloseList = [];
  while (OpenList != []) {
    Akt = First(OpenList);
    OpenList = OpenList \ [Akt];
    CloseList = CloseList + [Akt];
    process(Akt);
    for(alle zu Akt adjazente Knoten v1) {
      if (v1  $\notin$  OpenList && v1  $\notin$  CloseList)
        Pseudo-dfs: OpenList = [v1] + OpenList;
        bfs: OpenList = OpenList + [v1];
    }
  }
}
```



z.B.: **Search (0)** mit Pseudo-dfs

OpenList: 0|3,4|5,4|4|2,8|8|

CloseList: 0,3,5,4,2,8

Akt: 0|3|5|4|2|8

z.B.: **Search (0)** mit bfs

OpenList: 0|3,4|4,5|5,2,8|2,8|8|

CloseList: 0,3,4,5,2,8

Akt: 0|3|4|5|2|8

Systematisierung noch nicht ganz vollständig

Die Anweisung

```
for (alle zu Akt adjazente nicht besuchte Knoten v1)...
```

führt zu undefinierter Reihenfolge aller adjazenter Knoten bei der Weiterbearbeitung

→ Reihenfolge definiert durch die interne Graphenspeicherung
(Adjazenzliste, Adjazenzmatrix)

Annahme: adjazente Knoten werden aufsteigend sortiert eingetragen

Input: zusammenhängender Graph $G=(V,E)$, Knoten v in V

Search(Knoten v) {

 OpenList = [v]; CloseList = [];

$T = []$;

 while (OpenList != []) {

 Akt = *First*(OpenList);

 OpenList = OpenList \ [Akt];

 CloseList = CloseList + [Akt];

 process(Akt);

 for(alles zu Akt adjazente Knoten v_1) {

 if ($v_1 \notin$ OpenList && $v_1 \notin$ CloseList) {

 Pseudo-dfs: OpenList = [v_1] + OpenList;

 bfs: OpenList = OpenList + [v_1];

$T = T + [[\text{Akt}, v_1]]$

 }}}

}



Wenn G zusammenhängend ist, dann ist T ein spannender Baum von G .

- Traversierung mit BFS: T heißt *BFS-Baum*
- Traversierung mit **echtem** DFS: T heißt *DFS-Baum*

Anwendungsebene:

Beispiele

- günstigste Weg von a nach b
- gute Züge in einem Spiel

Werkzeugebene:

Beispiele

- Graphdurchquerung von v1 nach v2
- Topologische Anordnung

Implementationsebene:

Beispiele

- Rekursive Traversierung
- Iterative Traversierung

Beispiele

Weg von Knoten x nach y finden

Von x ausgehend Graph traversieren bis man zu y kommt (d.h. y markiert wird).

Beliebigen Kreis im ungerichteten Graph finden

Von jedem Knoten Traversierung des Graphen starten. Kreis ist gefunden, falls man zu einem markierten Knoten kommt (man war schon einmal da).

Beliebigen Kreis im gerichteten Graph finden

Von jedem Knoten Traversierung des Graphen starten. Kreis ist gefunden, falls man zu einem markierten Knoten kommt.

Wichtig bei gerichteten Graphen: Gesetzte Markierungen müssen beim Zurücksteigen wieder gelöscht werden.

...

6.5.4 Das “Bauer, Wolf, Ziege und Kohlkopf”-Problem



Klassisches Problem

Ein Bauer möchte mit einem Wolf, einer Ziege und einem Kohlkopf einen Fluss überqueren. Es steht ihm hierzu ein kleines Boot zur Verfügung, in dem aber nur zwei Platz haben.

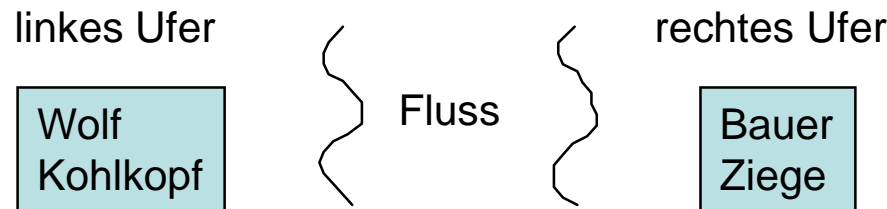
Weiters stellt sich das Problem, dass nur der Bauer rudern kann, und der Wolf mit der Ziege und die Ziege mit dem Kohlkopf nicht allein gelassen werden kann, da sonst der eine den anderen frisst.

Es soll eine Transportfolge gefunden werden, sodass alle ‘ungefressen’ das andere Ufer erreichen.



Das Problem wird durch einen Graphen dargestellt, wobei die Knoten die Positionen der zu Transportierenden und die Kanten die Bootsfahrten repräsentieren.

Eine **Position** (= **Ort** = Knoten) definiert, wer auf welcher Seite des Flusses ist,

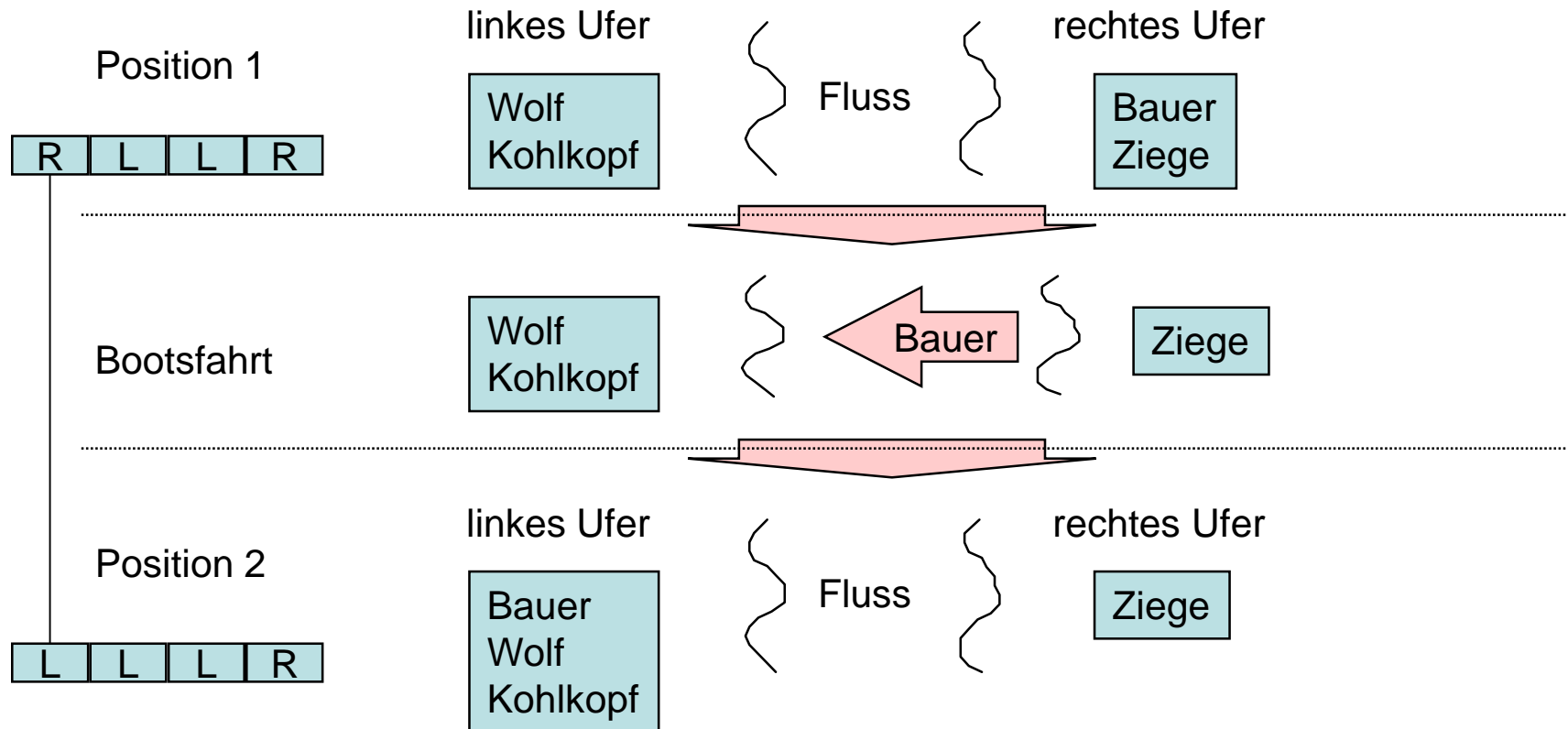


Dieses ist eine 'sichere' Position, da niemand gefressen wird.

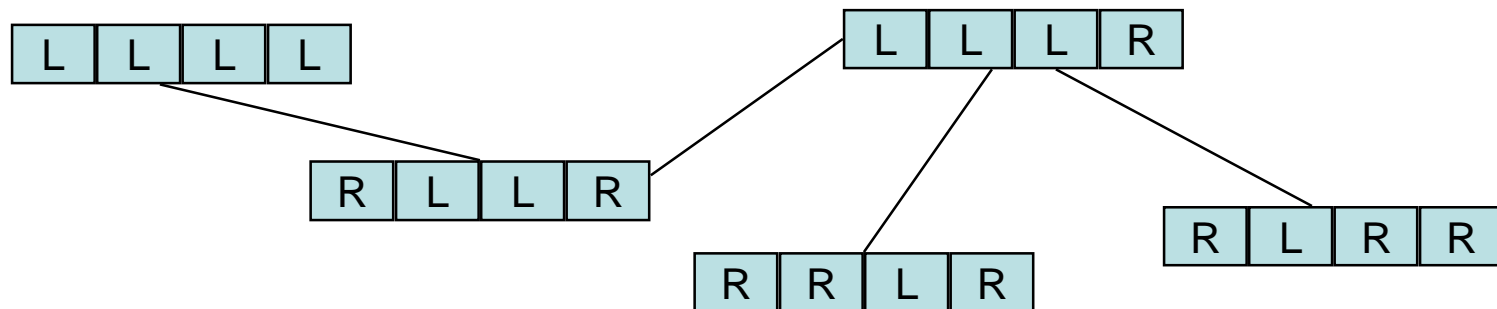
Eine Position kann in einem 4 elementigen Vektor kodiert werden, wobei jedem der 4 zu Transportierenden ein Vektorelement zugeordnet ist und L das linke Ufer bzw. R das rechte bezeichnet, d.h. der obigen Position entspricht der folgende Vektor

Bauer	Wolf	Kohlkopf	Ziege
R	L	L	R

Eine Bootsfahrt (Kante) gibt an, wer im Boot übersetzt, d.h. führt eine Position in die nächste über, d.h



Der gesamte Problembereich lässt sich somit durch einen Graphen darstellen, wobei die Positionen durch die Knoten und die Bootsfahrten durch die Kanten repräsentiert werden, z.B. (Ausschnitt)

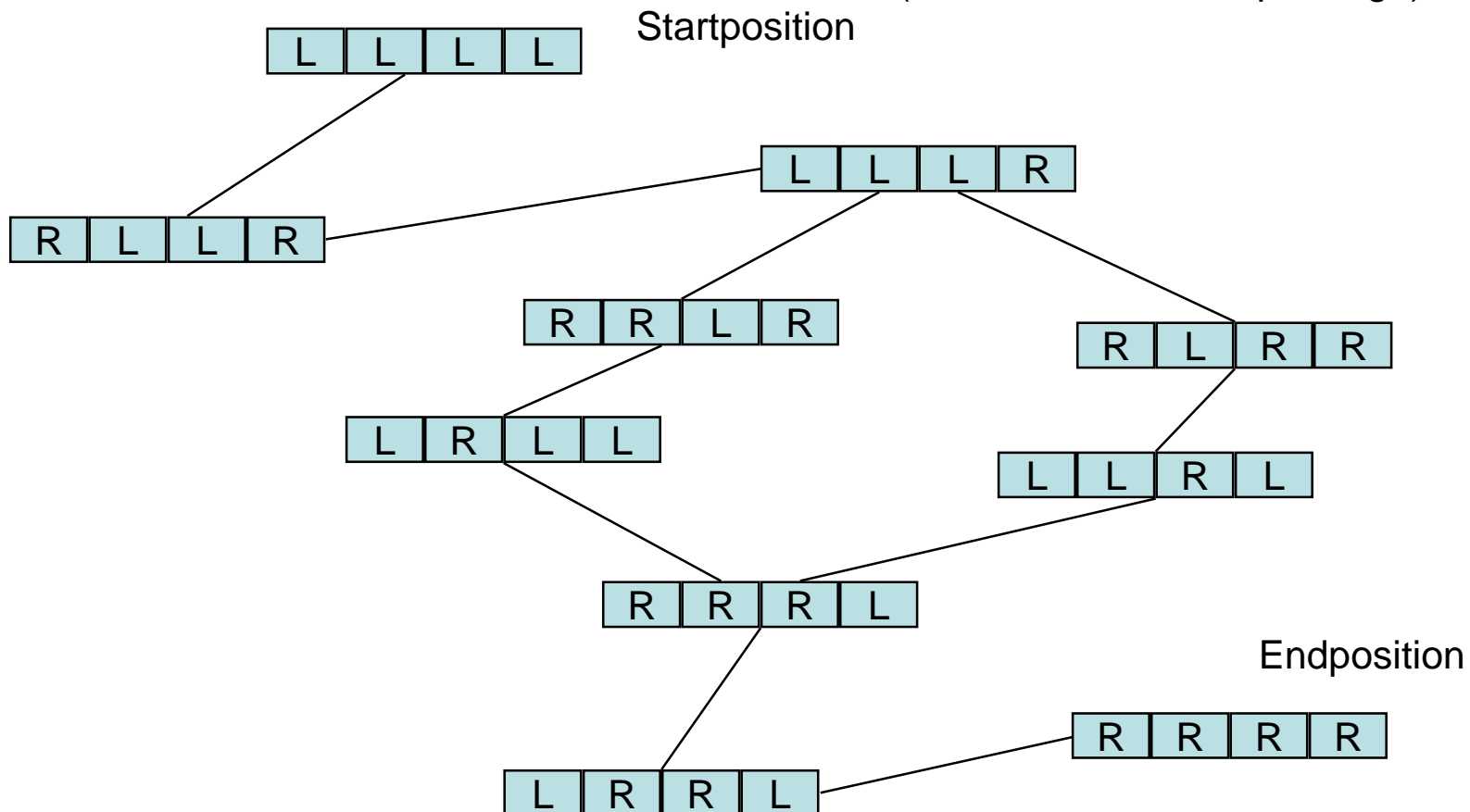


Die Lösung wird somit durch einen Weg bestimmt, der von der Anfangsposition (alle 4 auf dem linken Ufer = LLLL) zu der Endposition (alle 4 auf dem rechten Ufer = RRRR) führt.

Dies lässt sich durch eine simple Traversierung (z.B. DFS oder BFS) des Graphen lösen.

Lösungsweg

Kodierung:
(Bauer, Wolf, Kohlkopf, Ziege)



Wir wählen einen bfs-Ansatz (iterativ, mit Hilfe einer Queue)

Die Position wird in einer Integer-Variable binär (stellenwertig) kodiert

Bauer 3. Bit, Wolf 2. Bit, Kohlkopf 1. Bit, Ziege 0. Bit, wobei 0 linkes Ufer (L) und 1 rechtes Ufer (R) bedeutet. Beispiel: 1001 rechts: Bauer, Ziege

Die Funktionen **Bauer**, **Wolf**, **Kohl**, **Ziege** liefern die aktuelle Position zurück: 0 (links) oder 1 (rechts)

Die Funktion **sicher** bestimmt, ob eine Position sinnvoll ist, d.h. niemand wird gefressen.

Die Funktion **druckeort** dient zur verständlichen Ausgabe der Positionen.

In der Queue **zug** werden die zu besuchenden Knoten vermerkt.

Der beschrittene Weg (Traversierung) wird im Feld **Weg** gespeichert (max. 16 Positionen).

C-Spezialitäten	0x08	Hexadezimaldarstellung einer Zahl
	\wedge	bitweises exklusives Oder, XOR
	$\&$	bitweises Und
	$ $	bitweises Oder
	\ll	Linksshift Operator


```
int Ort = ...;
if (Ort & 0x08 == 0) {
    // teste Bauer-Bit, Bauer ist rechts
} else { ... }
    // nur Bauer wechselt Seite
int nOrt = Ort ^ 0x08;
    // 0001 : Ziegen-Bit
int Pers = 1;
    // Bauer und Ziege wechseln Seite
nOrt = Ort ^ (0x08 | Pers);
    // 0010 : Kohlkopf-Bit
nPers = Pers <<= 1;
```

```
int Bauer(int Ort) {return 0 != (Ort & 0x08);}  
// Ort & 0x08 == 1 wenn Bauer rechts ist, == 0 wenn Bauer links  
ist
```

```
int Wolf(int Ort) {return 0 != (Ort & 0x04);}  
int Kohl(int Ort) {return 0 != (Ort & 0x02);}  
int Ziege(int Ort) {return 0 != (Ort & 0x01);}
```

```
bool sicher(int Ort) {  
    if((Ziege(Ort) == Kohl(Ort)) &&  
        (Ziege(Ort) != Bauer(Ort))) return false;  
    if((Wolf(Ort) == Ziege(Ort)) &&  
        (Wolf(Ort) != Bauer(Ort))) return false;  
    return true;  
}
```

```
void DruckeOrt(int Ort) {  
    cout << ((Ort & 0x08) ? "R " : "L ");  
    cout << ((Ort & 0x04) ? "R " : "L ");  
    cout << ((Ort & 0x02) ? "R " : "L ");  
    cout << ((Ort & 0x01) ? "R " : "L ");  
    cout << endl;  
}
```

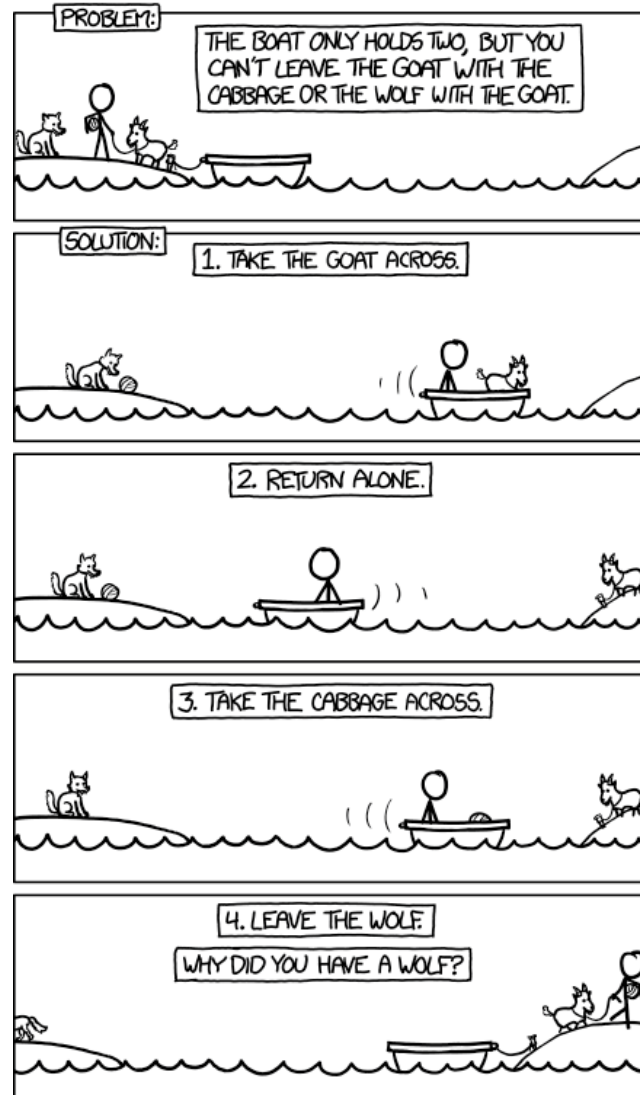
```
int Seite(int Ort, int Pers)
    if (Pers == 8) return Ort & 0x08;
// wenn Pers = Bauer und Bauer rechts ist, == 0 wenn Bauer links
ist
    if (Pers == 4) return Ort & 0x04;
    if (Pers == 2) return Ort & 0x02;
    if (Pers == 1) return Ort & 0x01;
}
```

```
void main() {
    Queue<int> Zug; // BFS queue
    int Weg[16];    // speichert Weg von LLLL zu RRRR
    for(int i = 0; i < 16; i++) Weg[i] = -1; // Initialisierung
    Zug.Enqueue(0x00); // starte bei LLLL
    while(!Zug.IsEmpty()) {
        int Ort = Zug.Dequeue(); // derzeitiger Knoten
        for (int Pers = 1; Pers <= 8; Pers <<= 1) { // adjazente
            // Kanten erzeugen: Pers nimmt nur Werte 1, 2, 4, 8 an
            if (Seite(Ort, Pers) != Bauer(Ort)) continue; // Pers
            // nicht auf gleicher Seite wie Bauer, also kann keine Kante,
            // die die Position von Pers verändert, existieren
            int nOrt = Ort ^ (0x08 | Pers); // benachbarter Knoten:
            // Bauer und Pers wechseln Seite
            if(sicher(nOrt) && (Weg[nOrt] == -1)) {
                // Kante existiert und nOrt noch nicht besucht
                Weg[nOrt] = Ort;
                Zug.Enqueue(nOrt);
            }
        }
    }
}
```

```
void main() {
    Queue<int> Zug;
    int Weg[16];
    for(int i = 0; i < 16; i++) Weg[i] = -1;
    Zug.Enqueue(0x00);
    while(!Zug.IsEmpty()) {
        int Ort = Zug.Dequeue();
        for (int Pers = 1; Pers <= 8; Pers <<= 1) {
            if (Seite(Ort, Pers) != Bauer(Ort)) continue;
            int nOrt = Ort ^ (0x08 | Pers);
            if(sicher(nOrt) && (Weg[nOrt] == -1)) {
                Weg[nOrt] = Ort;
                Zug.Enqueue(nOrt);
            }
        }
    }
}

// Ausgabe der Loesung
cout << "Weg:\n";
for(int Ort = 15; Ort > 0; Ort = Weg[Ort]) DruckeOrt(Ort);
cout << '\n';
}
```

Die "optimale Lösung"



<https://xkcd.com/1134/>

Ein kantengewichteter Graph $G = (V, E, w)$ ist ein ungerichteter oder gerichteter Graph, bei dem die Gewichtsfunktion w jeder Kante einen Wert zuordnet, d.h. für jede Kante $[u, v] \in E$ ($(u, v) \in E$) existiert ein Gewicht $w(u, v)$, welches die Kosten/Werte der Kante angibt.

Ein **Minimaler Spannender Baum MSB** (*minimum spanning tree*) ist ein spannender Baum eines **verbundenen**, ungerichteten Graphens, dessen Summe der Kantenwerte minimal ist.

Ziel: Finde einen MSB, d.h. eine azyklische Teilmenge $T \subseteq E$, die alle Knoten verbindet und für die gilt:

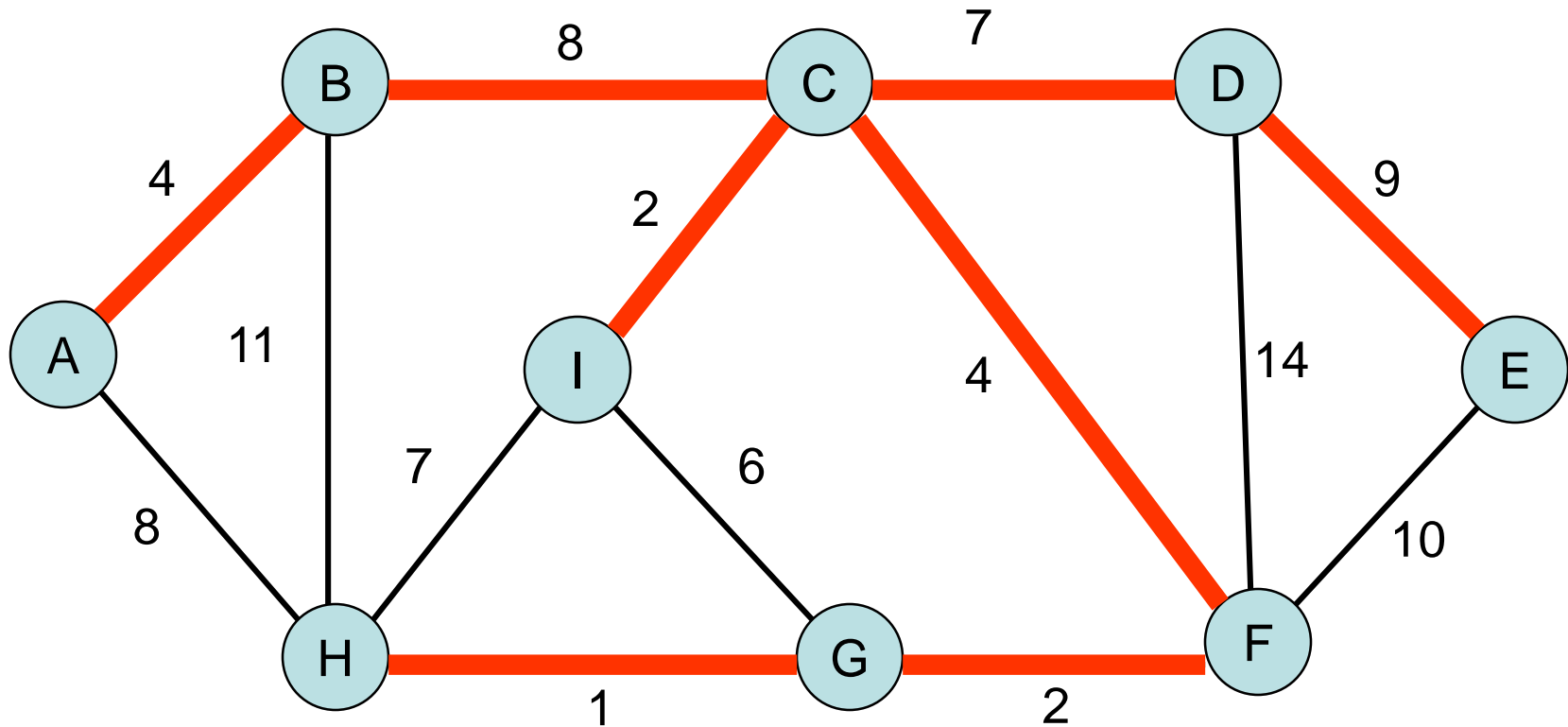
$$w(T) = \sum_{[u,v] \in T} w(u, v) \text{ ist ein Minimum}$$

Beispiel:

Verdrahtungsproblem in einem elektronischen Schaltplan

Alle Pins müssen untereinander verdrahtet werden, wobei die Drahtlänge minimal sein soll

d.h. V ist die Menge der Pins, E die Menge der möglichen Verbindungen zwischen Pin-Paaren, $w(u,v)$ Drahtlänge zwischen Pin u und v



Gewicht des MSB: 37

MSB ist nicht eindeutig, Kante [B, C] könnte durch [A, H] ersetzt werden

Ansatz durch Greedy Algorithmus

1. Algorithmus verwaltet eine Menge A von Kanten, die immer eine Teilmenge eines möglichen MSB sind
2. MSB wird schrittweise erzeugt, Kante für Kante, wobei für jede Kante überprüft wird, ob sie zu A hinzugefügt werden kann, ohne Bedingung 1 zu verletzen.

So eine Kante heißt „**sichere Kante für A** “ (**safe edge**): eine Kante e sodass $A \cup \{e\}$ eine Teilmenge eines möglichen MSB ist

```
Generic-MSB( $G=(V, E, w)$ ) {  
     $A = \{\}$ ;  
    while ( $A$  bildet keinen MSB) {  
        finde eine Kante  $[u,v]$ , die für  $A$  "sicher" ist  
         $A = A \cup \{ [u,v] \}$   
    }  
}
```

Wie finde ich so
eine sichere
Kante?

Ein **Wald** ist ein azyklischer Graph.

Unterschied zwischen Wald und Baum:

Ein Baum muss alle Knoten im Graphen verbinden, d.h. spannend sein.
Ein Wald muss nicht alle Knoten verbinden.

Grundidee:

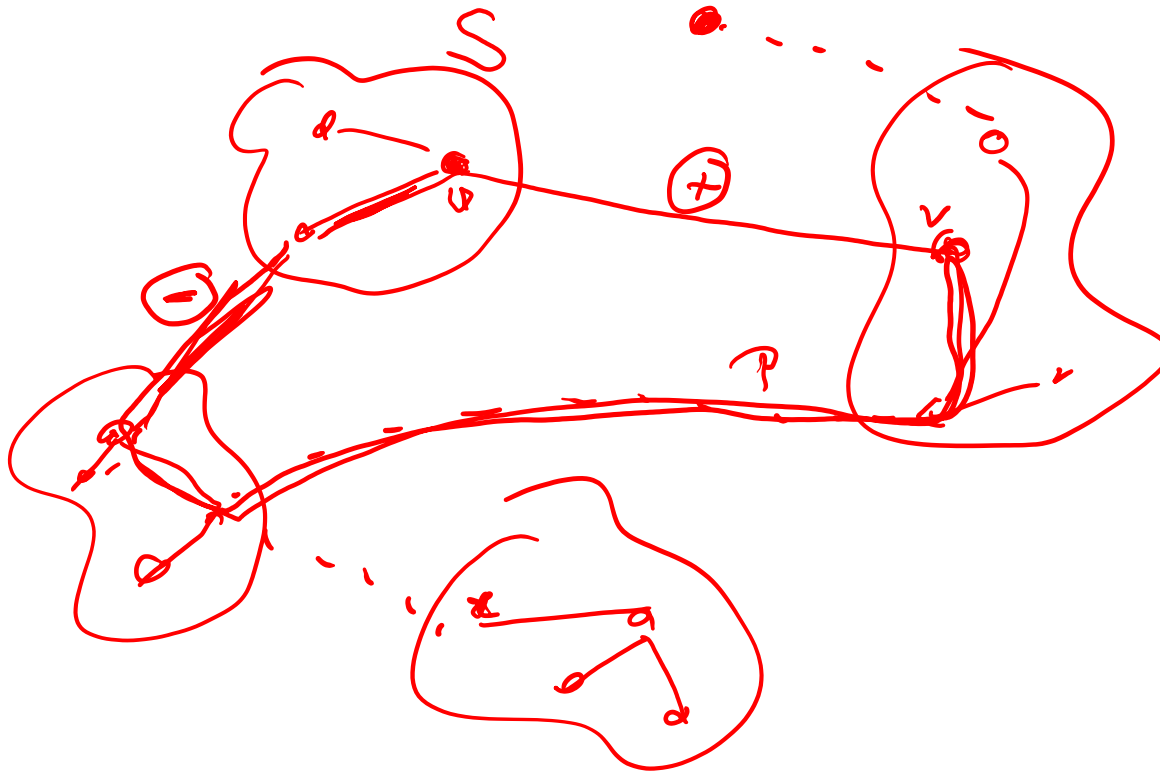
Die Menge der Knoten repräsentiert einen Wald bestehend aus $|V|$ Komponenten

→ an Beginn keine Kante

Lemma 4: *Sei A eine Menge von Kanten, sodass (V, A) ein Teilgraph eines MSB von $G = (V, E, w)$ ist. Die Kante mit dem niedrigsten Gewicht, die zwei unterschiedliche Komponenten von (V, A) verbindet ist eine sichere Kante für A .*

Greedy-Ansatz:

In jedem Schritt wird die Kante mit niedrigstem Gewicht zum Wald hinzugefügt.



Sei $G = (V, E, w)$ ein Graph und $X = (V, A)$ ein Untergraph eines MSB T von G , d.h. $A \subseteq T \subseteq E$. Sei $[u, v] \in E$ eine Kante mit minimalem Gewicht, die zwei Komponenten in X verbindet. Sei S eine Komponente von X , sodass $u \in S$ und $v \notin S$. Wir wollen zeigen, dass $[u, v]$ eine sichere Kante für A ist.

Wir nehmen an, dass $[u, v]$ nicht sicher für A ist und erzeugen einen Widerspruch. Das beweist, dass $[u, v]$ sicher für A ist.

Wenn $[u, v]$ nicht sicher für A ist, dann gehört $[u, v]$ zu keinem MSB, der alle Kanten von A enthält. Es folgt, dass T einen Weg P von u nach v enthält, der $[u, v]$ nicht enthält.

Sei $[x, y]$ die erste Kante auf P , sodass $x \in S$ und $y \notin S$. $[x, y]$ kann nicht zu A gehören, da $y \notin S$. Da $[x, y]$ die Komponente S mit einer anderen Komponente verbindet, ist nach der Definition von $[u, v]$ $w(u, v) \leq w(x, y)$.

Sei $T' = T \setminus \{[x, y]\} \cup \{[u, v]\}$. T' ist azyklisch und verbunden, d.h. T' ist ein spannender Baum. Aber $w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$, d.h. T' ist auch ein MSB. Da $A \cup \{[u, v]\}$ zu T' gehören, widerspricht das der Annahme, dass es keinen MSB gibt, der $A \cup \{[u, v]\}$ enthält, d.h. dass $[u, v]$ nicht sicher für A ist.

Das Entfernen von $[x, y]$ bricht T in 2 Komponenten, nämlich B und $V \setminus B$ sodass $u \in B$ und $v \notin B$. Daher verbindet das Hinzufügen von $[u, v]$ zu $T \setminus [x, y]$ die beiden Komponenten B und $V \setminus B$ wieder.

Daher ist $T' = T \setminus \{[x, y]\} \cup \{[u, v]\}$ verbunden.

Wenn wir $[u, v]$ zu T hinzufügen ohne $[x, y]$ zu entfernen, erzeugt das genau einen Kreis C in $T \cup \{[u, v]\}$, bestehend aus dem Pfad P und $[u, v]$. Da nach der Definition von $[x, y]$ die Kante $[x, y]$ auf dem Pfad P von u nach v liegt, zerstört das Entfernen von $[x, y]$ genau den Kreis C .

Daher ist $T' = T \setminus \{[x, y]\} \cup \{[u, v]\}$ azyklisch.

Es folgt, dass T' azyklisch und verbunden, und daher ein spannender Baum ist.

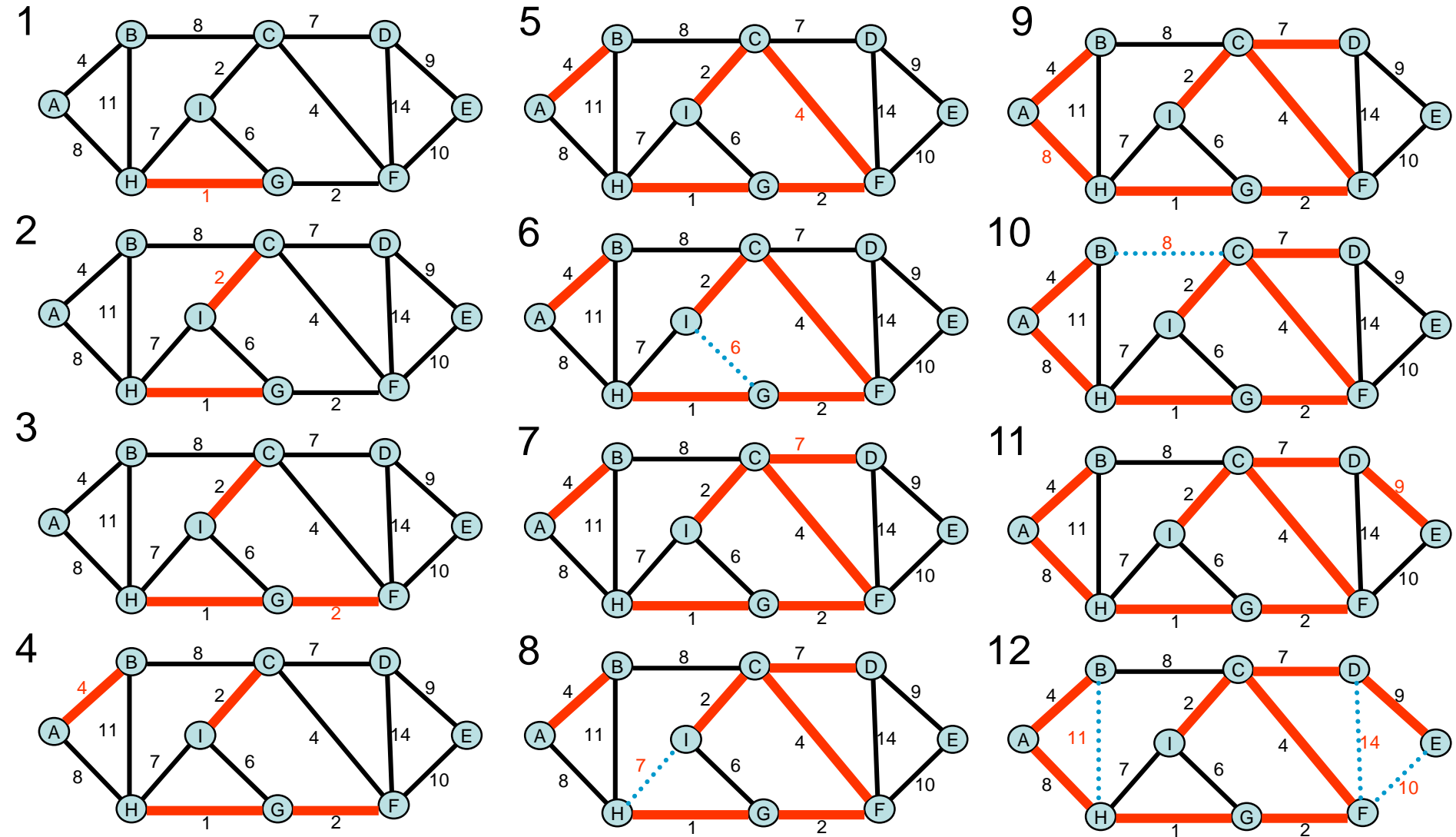
```
MSB-Kruskal (G=(V, E, w)) {  
    A = {};  
    for (jeden Knoten v ∈ V)  
        make-set(v);  
    sortiere die Kanten aus E nach aufsteigendem Gewicht w  
    for(jede Kante [u,v] ∈ E in der Reihenfolge der Gewichte)  
        if(find-set(u) != find-set(v)) {  
            A = A ∪ { [u,v] }  
            union(u, v)  
        }  
    return A;  
}
```

w enthält die Gewichte zwischen den Knoten, z.B. Gewicht zwischen u und v = **w(u,v)**

make-set(v) erzeugt eine Menge bestehend aus dem Element v
find-set(v) liefert ein repräsentatives Element für die Menge die v enthält

union(u,v) vereinigt Mengen von u und von v zu einer Menge
Hier: Jede Menge entspricht den Knoten in einer Komponente von (V,A)

Beispiel: Kruskal Algorithmus



- $O(m \log m) = O(m \log n)$ Kanten sortieren*, plus
- n make-set Operationen, plus
- $2m$ find-set Operationen, plus
- $\leq n - 1$ union Operationen
(ein Baum hat höchstens $n - 1$ Kanten)

Eine Datenstruktur, die make-set, find-set, und union Operationen implementiert, heißt **union-find Datenstruktur**

Der Aufwand hängt von der union-find Datenstruktur ab, die benützt wird.

* Anmerkung: Da $m < n^2$, folgt, dass $\log m < \log n^2 = 2 \log n$.

Idee

Set wird repräsentiert durch die Wurzel eines Baum

- `make-set(v)`: Generiere einen neuen Baum mit einem Knoten `v`
- `find-set(u)`: Gib die Wurzel des Baumes, der `u` enthält, aus
- `union(u, v)`: Mache aus den 2 Bäumen, die `u` and `v` enthalten, einen Baum indem die Wurzel des einen Baumes ein Kind der Wurzel des anderen Baumes wird

```
make-set(x) {  
    p[x] = x;  
}
```

```
link(x, y) {  
    p[y] = x;  
}
```

```
find-set(x) {  
    if(x != p[x]) return find-set(p[x]);  
    return p[x];  
}
```

```
union(x, y) {  
    link(find-set(x), find-set(y));  
}
```

$p[x]$ enthält den Elternknoten
von x (Vertreter der Teilmenge)
 $p[x] == x$ wenn x die Wurzel ist

Aufwand: Baum kann Höhe $n - 1$ haben

- $\text{make-set}(x): O(1)$
- $\text{link}(x, y): O(1)$
- $\text{find-set}(x): O(n)$
- $\text{union}(x, y): O(n)$

→ MSB-Kruskal: $O(m \log n + m \cdot n) = O(m \cdot n)$ mit dieser
Datenstruktur

Beispiel: Union-find

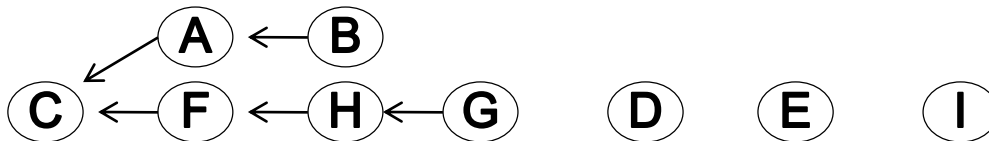


Elemente: A, B, C, D, E, F, G, H, I

p-Werte für
make-set(A), make-set(B), ..., make-set(I)
union(H,G)
union(F,G)
union(A,B)
union(C,F)
union(H,A)

A	B	C	D	E	F	G	H	I
A	B	C	D	E	F	G	H	I
A	B	C	D	E	F	H	H	I
A	B	C	D	E	F	H	F	I
A	A	C	D	E	F	H	F	I
A	A	C	D	E	C	H	F	I
C	A	C	D	E	C	H	F	I

Zu diesem Zeitpunkt haben die p-Werte folgende Struktur:



find-set(B) braucht 2 Look-ups, find-set(G) 3 Look-ups.

Bessere Union-find Datenstruktur (union-by-rank)



Idee: Reduziere den Aufwand von find-set indem link den Baum balanciert

```
make-set(x) {  
    p[x] = x;  
    rank[x] = 0;  
}  
  
link(x, y) {  
    if (rank[y] > rank[x]) p[x] = y;  
    else {  
        p[y] = x;  
        if (rank[x] == rank[y])  
            rank[x] = rank[x] + 1;  
    }  
}  
  
find-set(x) {  
    if(x != p[x]) return find-set(p[x]);  
    return p[x];  
}
```

p[x] enthält den Elternknoten von x (Vertreter der Teilmenge)
rank[x] enthält die Höhe von x (Länge des längsten Wegs zwischen x und einem Blatt im Unterbaum)

Beispiel: Union-find



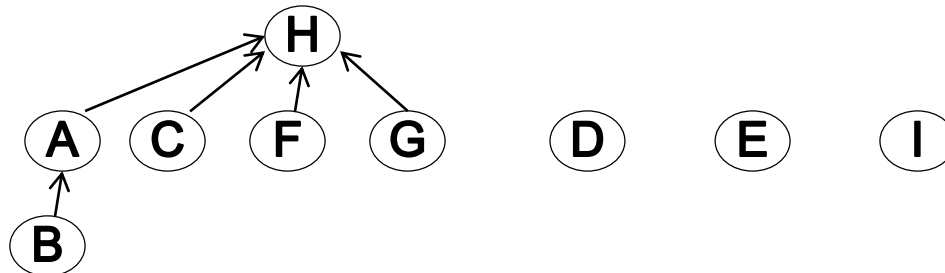
Elemente: A, B, C, D, E, F, G, H, I

p- und rank-Werte für
make-set(A), make-set(B), ..., make-set(I)
union(H,G)
union(F,G)
union(A,B)
union(C,F)
union(H,A)

A	B	C	D	E	F	G	H	I
A,0	B,0	C,0	D,0	E,0	F,0	G,0	H,0	I,0
A,0	B,0	C,0	D,0	E,0	F,0	H,0	H,1	I,0
A,0	B,0	C,0	D,0	E,0	H,0	H,0	H,1	I,0
A,1	A,0	C,0	D,0	E,0	H,0	H,0	H,1	I,0
A,1	A,0	H,0	D,0	E,0	H,0	H,0	H,1	I,0
H,1	A,0	H,0	D,0	E,0	H,0	H,0	H,2	I,0

Zu diesem Zeitpunkt haben die p-Werte folgende Struktur:

find-set(B) braucht
2 Look-ups,
find-set(G) nur mehr
1 Look-up



Lemma 5: Ein Knoten x mit $\text{rank}[x] = k$ hat mindestens 2^k Knoten in seinem Unterbaum.

Beweis: Induktion über k .

$k = 0$: Jeder Knoten liegt selbst in seinem Unterbaum, d.h. jeder Unterbaum hat mindestens $1 = 2^0$ Knoten, auch der Unterbaum eines Knotens mit rank 0.

$k - 1 \rightarrow k$: Der Rank eines Knotens y erhöht sich nur auf k , wenn es einen Knoten x mit gleichem Rank (vor der Erhöhung) gibt und x ein Kind von y wird. Durch die Induktionsannahme wissen wir, dass zu diesem Zeitpunkt im Unterbaum von x sowie im Unterbaum von y mindestens 2^{k-1} Knoten sind. Daher hat der vereinte Baum mindestens $2^{k-1} + 2^{k-1} = 2^k$ Knoten.

→ Wenn es n Elemente gibt, dann gibt es $\leq n$ Knoten in jedem Baum, d.h. $2^k \leq n$. Es folgt, dass $k \leq \log n$, d.h. jeder Knoten x hat $\text{rank}[x] \leq \log n$, d.h., jeder Baum höchstens Höhe $\log n$.

Aufwand: Jeder Baum hat Höhe $\leq \log n$

- $\text{make-set}(x) : O(1)$
- $\text{link}(x, y) : O(1)$
- $\text{find-set}(x) : O(\log n)$
- $\text{union}(x, y) : O(\log n)$

$$m < n^2$$
$$\log m < \log n^2 = 2 \log n$$

→ MSB-Kruskal: $O(m \log n + m \log n) = O(m \log n)$ mit dieser
Datenstruktur

Beste Union-find Datenstruktur (union-by-rank with path compression)



```
make-set(x) {  
    p[x] = x;  
    rank[x] = 0;  
}  
  
link(x, y) {  
    if (rank[y] > rank[x]) p[x] = y;  
    else {  
        p[y] = x;  
        if (rank[x] == rank[y])  
            rank[x] = rank[x] + 1;  
    }  
}  
  
find-set(x) {  
    if (x != p[x]) p[x] = find-set(p[x]);  
    return p[x];  
}
```

p[x] enthält den Elternknoten von x (Vertreter der Teilmenge)
rank[x] ist eine **obere Grenze** der Höhe von x (Anzahl der Kanten zwischen x und einem Nachfolger-Blatt)
find-set(v) sucht die Wurzel und **trägt sie danach jedem Knoten als Elternknoten ein**

Path compression: Alle Knoten auf dem Weg zur Wurzel werden Kinder der Wurzel

Beispiel: Union-find



Elemente: A, B, C, D, E, F, G, H, I

p- und r-Werte für

make-set(A), make-set(B), ..., make-set(I)

union(H,G)

union(F,G)

union(A,B)

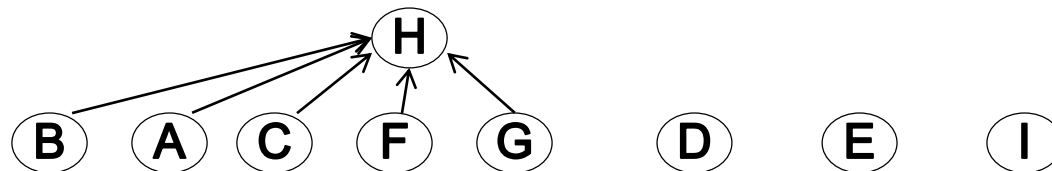
union(C,F)

union(H,A)

find-set(B)

A	B	C	D	E	F	G	H	I
A,0	B,0	C,0	D,0	E,0	F,0	G,0	H,0	I,0
A,0	B,0	C,0	D,0	E,0	F,0	H,0	H,1	I,0
A,0	B,0	C,0	D,0	E,0	H,0	H,0	H,1	I,0
A,1	A,0	C,0	D,0	E,0	H,0	H,0	H,1	I,0
A,1	A,0	H,0	D,0	E,0	H,0	H,0	H,1	I,0
H,1	A,0	H,0	D,0	E,0	H,0	H,0	H,2	I,0
H,1	H,0	H,0	D,0	E,0	H,0	H,0	H,2	I,0

Zu diesem Zeitpunkt haben die p-Werte folgende Struktur:



Komplizierte Analyse

- `make-set(x)`: $O(1)$
- `link(x, y)`: $O(1)$
- `find-set(x)`: $O(\alpha(n))$ (fast $O(1)$)
- `union(x, y)`: $O(\alpha(n))$ (fast $O(1)$)

$A(m, n)$: Ackermannfunktion,
extrem schnell wachsende
Funktion

$f(n) := A(n, n)$

$\alpha(n) := f^{-1}(n)$, extrem langsam
wachsende Funktion

$\alpha(n) < 5$ für alle praktisch
relevanten n

→ MSB-Kruskal: $O(m \log n + m \cdot \alpha(n)) = O(m \log n)$ mit dieser

Datenstruktur

Lemma 4: Sei A eine Menge von Kanten, sodass (V, A) ein Teilgraph eines MSB von $G = (V, E, w)$ ist. Die Kante mit dem niedrigsten Gewicht, die zwei unterschiedliche Komponenten von (V, A) verbindet ist eine sichere Kante für A .

Lemma 4a: Sei A eine Menge von Kanten, sodass (V, A) ein Teilgraph eines MSB von $G = (V, E, w)$ ist, **und sei S eine Komponente von A** . Die Kante mit dem niedrigsten Gewicht, **die S mit einer anderen Komponente** von (V, A) verbindet ist eine sichere Kante für A .

Beweis: Im Beweis von Lemma 4, ersetze

“Sei $[u, v] \in E$ eine Kante mit minimalem Gewicht, die zwei Komponenten in X verbindet. Sei S eine Komponente von X , sodass $u \in S$ und $v \notin S$.”

durch:

“Sei $[u, v] \in E$ eine Kante mit minimalem Gewicht, die S mit einer anderen Komponente von X verbindet.”

Lemma 4a: *Die Kante mit dem niedrigsten Gewicht, die S mit einer anderen Komponente von (V, A) verbindet ist eine sichere Kante für A .*

Grundidee:

1. Die Menge A bildet einen einzelnen Baum T
2. Die sichere Kante, die hinzugefügt wird, ist immer die Kante $[u, x]$ mit dem niedrigsten Gewicht, die T mit einem Knoten u verbindet, der noch nicht in T ist
 1. Speichere für jeden Knoten v , der noch nicht in T ist, in der Variable $key[v]$ das Gewicht der “leichtesten” Kante, die v mit T verbindet
 2. Der Knoten u mit minimalen $key[u]$ -Wert ist der Knoten, der noch nicht in T ist und (von allen solchen Knoten) eine Kante mit niedrigsten Gewicht zu T hat

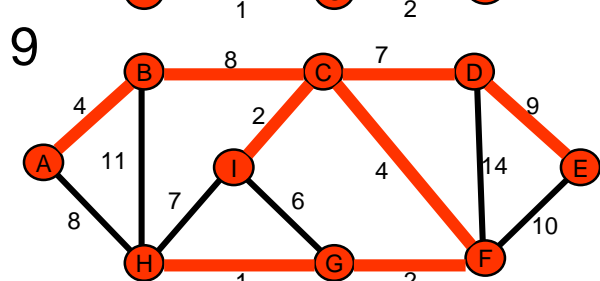
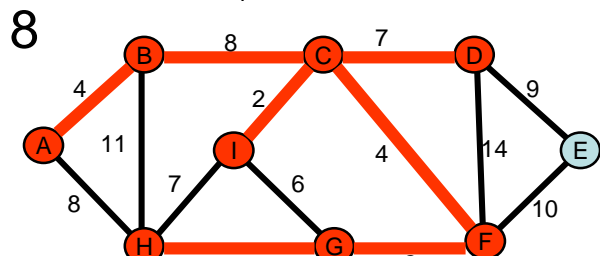
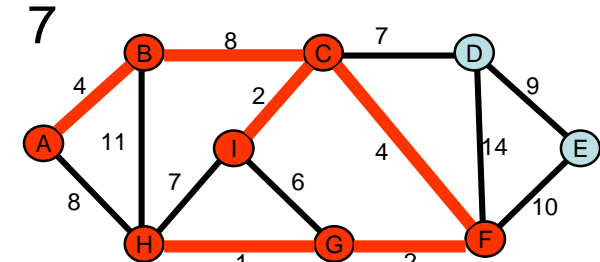
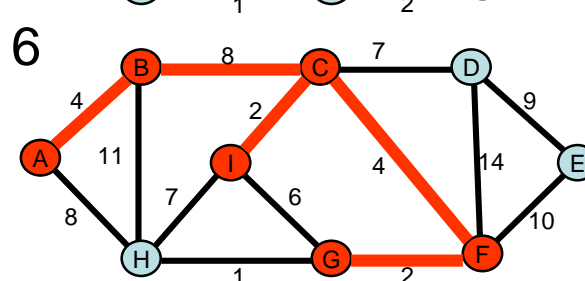
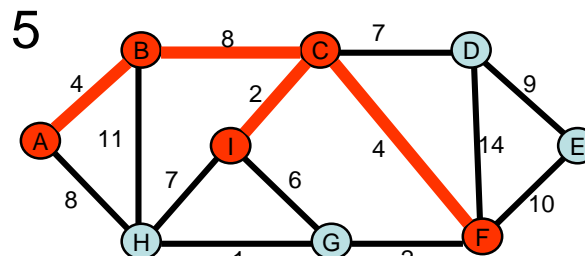
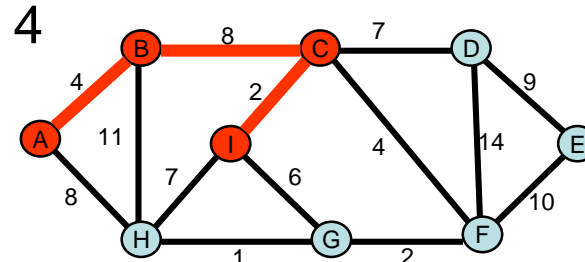
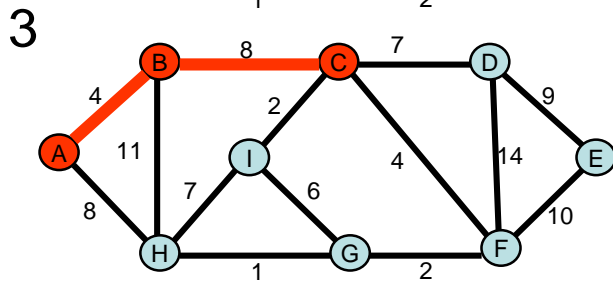
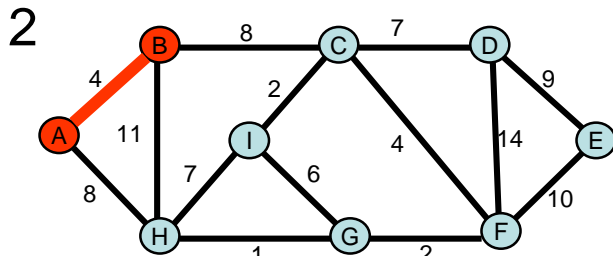
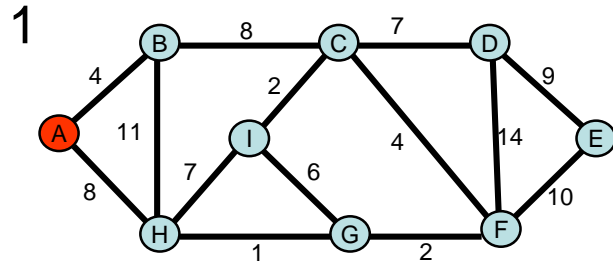
→ Greedy-Ansatz

Vorteil: Sortieren der Kanten nach Gewicht ist nicht nötig

```
MSB-Prim( $G=(V,E,w)$ ,  $r$ ) {  
     $Q = V$ ;  
    for(jeden Knoten  $u \in Q$ )  
         $key[u] = \infty$ ;  
     $key[r] = 0$ ;  
     $pred[r] = NIL$ ;  
    while( $Q \neq \{\}$ ) {  
         $u = \text{extract-min}(Q)$   
        if ( $pred(u) \neq NIL$ ) {  
             $A = A \cup \{ [u, pred[u]] \}$   
        }  
        for(jeden Knoten  $v$  adjazent zu  $u$ )  
            if ( $v \in Q \ \&\& \ w(u,v) < key[v]$ ) {  
                 $pred[v] = u$ ;  
                 $\text{decrease-key}(v, w(u,v))$ ;  
            }  
    }  
}
```

r ist die Wurzel (Startknoten) des MSB T
 $key[v]$ speichert das leichteste Gewicht von v zu einem Knoten in T
 $pred[v]$ speichert einen Knoten u sodass $[u,v]$ Gewicht $key[v]$ hat
 Q ist eine Priority Queue, die alle Knoten nicht in T entsprechend ihres key Wertes speichert;

Beispiel: Prim Algorithmus



- $O(n + m)$, plus
- n Einfügeoperationen in Q , plus
- n Extract-Min Operationen, plus
- $\leq m$ Decrease-Key Operationen

→ Aufwand der Algorithmen stark abhängig von der Implementation der Mengenoperationen bzw. der Datenstrukturen für die Mengenverwaltung (Priority Queue, ...)

→ Heaps von 5.9.1: Insert Operationen können zu Decrease-Key Operationen modifiziert werden

Aufwand pro Insert, Extract-Min, Decrease-Key Operation: $O(\log n)$

Totaler Aufwand: $O(n + m + n \log n + m \log n) = O(m \log n)$

Lässt sich durch Fibonacci-Heap verbessern auf $O(n \log n + m)$

In einem gewichteten Graph können Gewichte zwischen Knoten als **Weglängen** oder Kosten angesehen werden

Es ergibt sich oft die Fragestellung nach dem (den) kürzesten Weg(en) zwischen

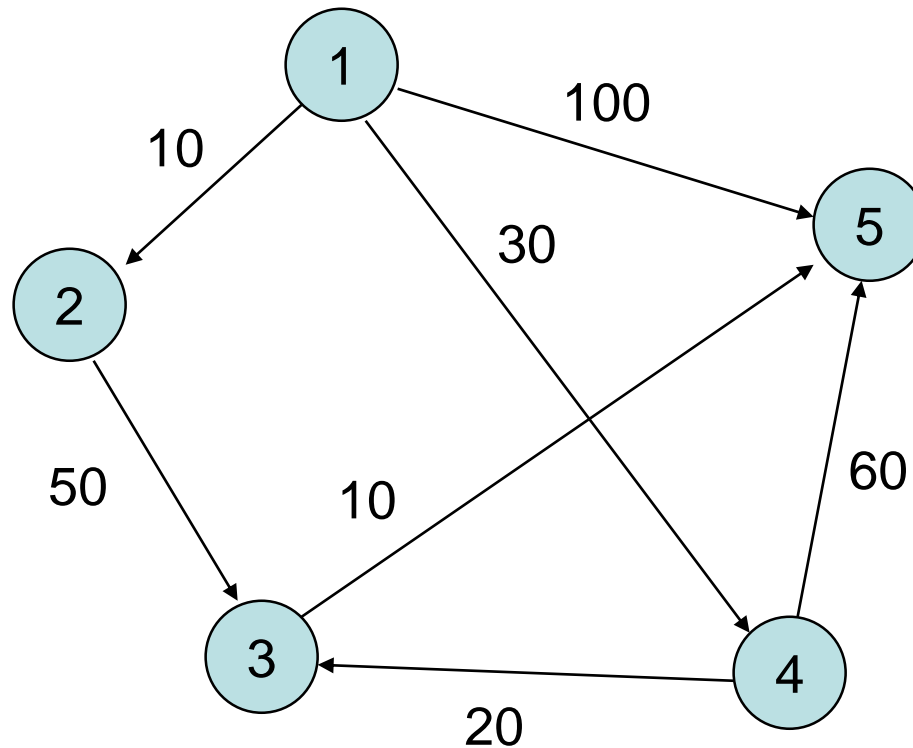
- einem Knoten und allen anderen Knoten
- zwei Knoten
- allen Knoten

Annahme: alle Kosten sind gespeichert in der Kostenmatrix C ($C[u,v]$ enthält Kosten von der Kante (u,v) wenn sie existiert und ∞ , wenn (u,v) nicht existiert.)

Fragestellung existiert in gerichteten und ungerichteten Graphen, hier: gerichtete Graphen

Annahme: nur positive Kantenwerte

Manche Algorithmen funktionieren nur mit positiven Werten



6.7.1 Dijkstra Algorithmus

Single Source Shortest Paths



Algorithmus zur Suche des kürzesten Weges
von einem Startknoten s zu allen anderen
Knoten

Annahme: *nur positive Kantenwerte*

Idee

Beginne beim Startknoten s .

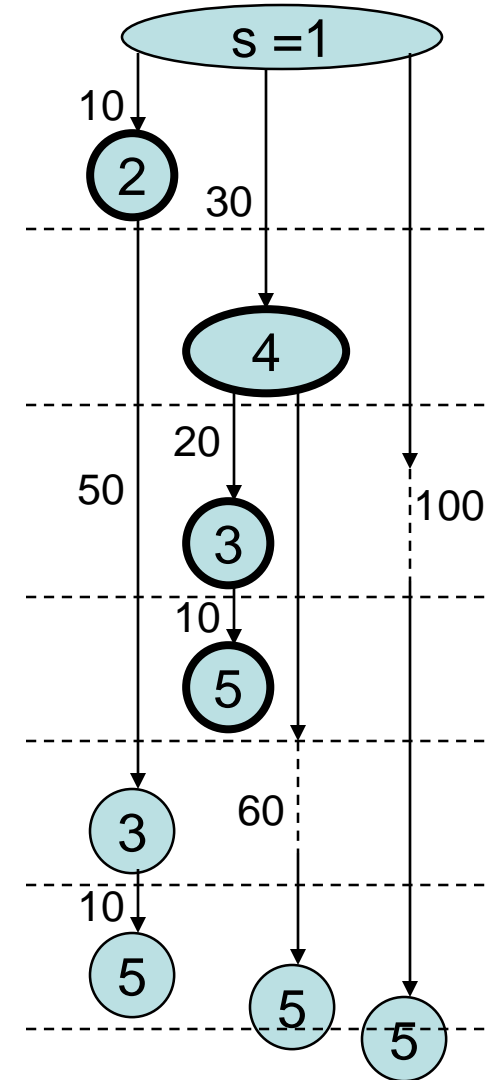
Suche den Knoten mit der kürzesten Distanz zu s ,

Danach den Knoten mit der zweitkürzesten Distanz,
usw. Merke die jeweiligen kürzesten Wege.

→ Greedy Ansatz, ähnlich Prim's Algorithmus

Wenn alle Kantenlängen = 1, dann Reihenfolge der
Knotenbesuche bei Dijkstra's Algorithmus gleich
der Reihenfolge von BFS

⇒ Algorithmus konstruiert einen shortest-path tree
mit Wurzel s . (Was ist der Unterschied zum MST?)



Grundidee:

1. S ist die Menge der schon bearbeiteten Knoten
2. Der nächste Knoten, der hinzugefügt wird, ist immer ein Knoten von $V \setminus S$, der den kürzesten Weg mit inneren Knoten nur von S zu s hat
 1. Speichere für jeden Knoten v , der noch nicht in S ist, in der Variable $\text{minC}[v]$ die Länge des kürzesten Weges von s nach v , der nur Knoten von S als innere Knoten benutzt
 2. Der Knoten u mit minimalem $\text{minC}[u]$ Wert ist der Knoten, der noch nicht in S ist und (von allen solchen Knoten) den kürzesten Weg von s mit inneren Knoten nur aus S hat
 - (innere Knoten = nicht Start- oder Endknoten des Weges)

Greedy Ansatz

```
MSB-Prim( $G=(V,E,w)$ ,  $r$ ) {  
     $Q = V$ ;  
    for(jeden Knoten  $u \in Q$ )  
         $key[u] = \infty$ ;  
     $key[r] = 0$ ;  
     $pred[r] = NIL$ ;  
    while( $Q \neq \{\}$ ) {  
         $u = \text{Extract-Min}(Q)$   
        if  $pred(u) \neq NIL$  {  
             $A = A \cup \{ [u, pred[u]] \}$   
        }  
        for(jeden Knoten  $v$  adjazent zu  $u$ )  
            if ( $v \in Q \ \&\& \ w(u,v) < key[v]$ ) {  
                 $pred[v] = u$ ;  
                 $\text{Decrease-Key}(v, w(u,v))$ ;  
            }  
    }  
}
```

r ist die Wurzel (Startknoten) des MSB T
 $key[v]$ speichert das leichteste Gewicht von v zu einem Knoten in T
 $pred[v]$ speichert einen Knoten u sodass $[u,v]$ Gewicht $key[v]$ hat
 Q ist eine Priority Queue, die alle Knoten nicht in T entsprechend ihres key Wertes speichert;

```
Dijkstra( $G=(V,E,w)$ ,  $s$ ) {
```

```
1.  $Q = V$ ;  $S = \{\}$ ;
```

```
2. for(jeden Knoten  $v \in Q$ )
```

```
3.      $MinC[v] = \infty$ ;
```

```
4.  $MinC[s] = 0$ ;
```

```
5. while( $Q \neq \{\}$ ) {
```

```
6.      $v = \text{Extract-Min}(Q)$ ;
```

```
7.     if  $MinC[v] < \infty$  {
```

```
8.          $S = S \cup \{v\}$ ;
```

```
9.         for (jeden Knoten  $w$  adjazent zu  $v$ )
```

```
10.             if ( $(w \in Q) \ \&\& \ (MinC[v] + C[v,w] < MinC[w])$ ) {
```

```
11.                 Decrease-Key( $w, MinC[v] + C[v,w]$ );
```

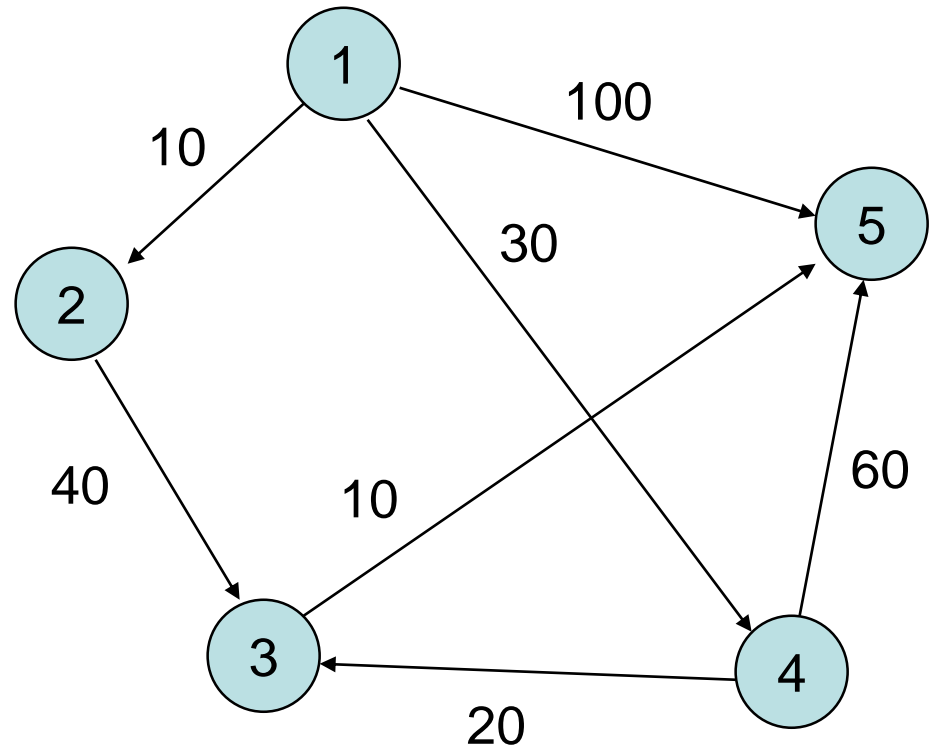
```
12.             }
```

```
13.     }
```

```
14. }
```

s Menge der bereits bearbeiteten Knoten
 v Knoten, der gerade bearbeitet wird
 C Kostenmatrix
 $MinC$ bisher bekannter kürzester Weg
 Q ist eine Priority Queue (=Heap), die alle Knoten nicht in S entsprechend ihres $MinC$ Wertes speichert

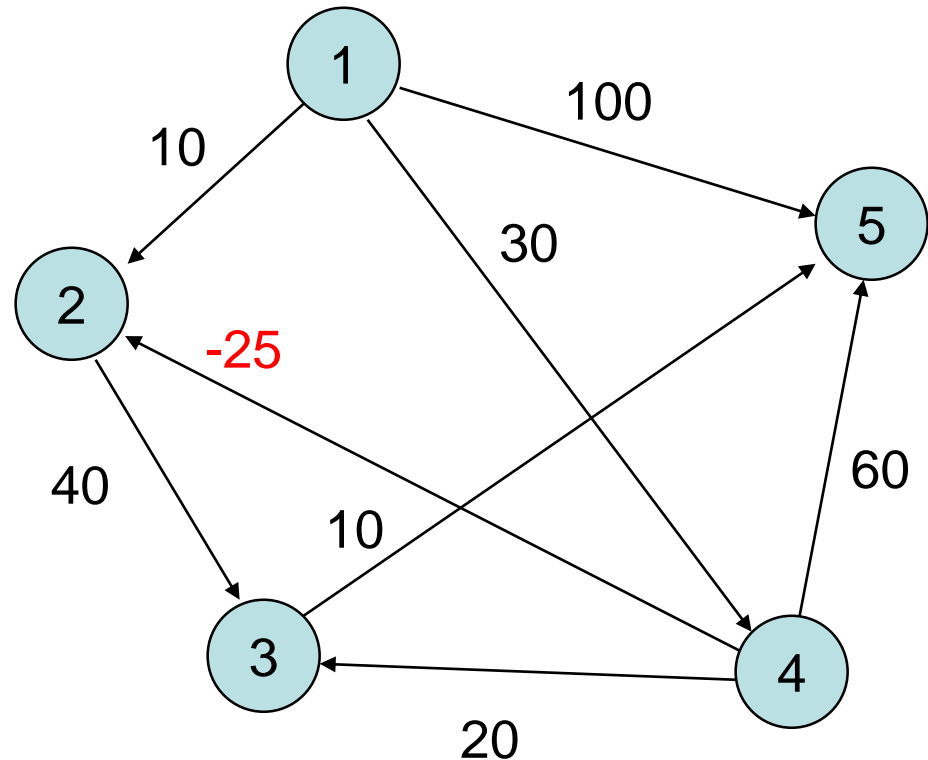
S	Akt	MinC
{}		$[0, \infty, \infty, \infty, \infty]$
{1}	1	$[0, 10, \infty, 30, 100]$
{1,2}	2	$[0, 10, 50, 30, 100]$
{1,2,4}	4	$[0, 10, 50, 30, 90]$
{1,2,3,4}	3	$[0, 10, 50, 30, 60]$



Was passiert bei negativen Kantenkosten?



S	Akt	MinC
{}		$[0, \infty, \infty, \infty, \infty]$
{1}	0	$[0, 10, \infty, 30, 100]$
{1,2}	2	$[0, 10, 50, 30, 100]$
{1,2,4}	4	$[0, 10, 50, 30, 90]$



2 ist nicht mehr in Q. Daher können die kürzesten Wege, die durch 2 gehen nicht mehr korrigiert werden.

{1,2,3,4} 3 $[0, 10, 50, 30, 60]$ falsche Werte

- $O(n + m)$ (wie bei BFS), plus
- n Insert und Extract-Min Operationen
- $\leq m$ Decrease-Key Operationen

→ Heaps von 5.9.1: Insert Operationen können zu Decrease-Key Operationen modifiziert werden

Aufwand pro Insert, Extract-Min, Decrease-Key Operation: $O(\log n)$

Totaler Aufwand: $O(m \log n + n \log n) = O(m \log n)$

Lässt sich durch Fibonacci-Heap verbessern auf
 $O(m + n \log n)$

Bevor wir die Fragestellung der kürzesten Wege zwischen allen Knoten klären, wollen wir zuerst die (einfachere) Frage behandeln, zwischen welchen Knoten existieren überhaupt Wege

Führt zu 2 Algorithmen

Transitive Hülle: welche Knoten sind durch Wege verbunden

APSP: alle kürzesten Wege zwischen den Knoten

Fragestellung welche Knoten durch Wege verbunden (erreichbar) sind, führt zur Frage nach der **Transitiven Hülle** (**transitive closure**)

Ein gerichteter Graph $G' = (V, E')$ wird **transitive** (und **reflexive**) Hülle eines Graphen $G = (V, E)$ genannt, genau dann wenn:
 $(v, w) \in E' \Leftrightarrow$ es existiert ein Weg von v nach w in G

Ausgangspunkt: Adjazenzmatrix von G

Idee 1

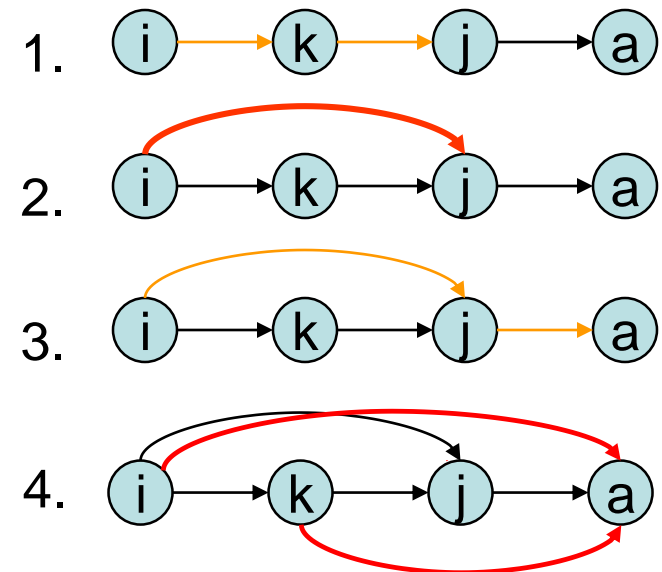
Iteratives Hinzufügen von Kanten im Graphen für Wege der Länge 2

d.h. Weg (i,k) und (k,j) wird durch neue Kante (i,j) beschrieben

in weiterer Folge wird dann natürlich auch Weg (i,j) und (j,a) als Weg der Länge 2 gefunden (in Wirklichkeit klarerweise Weg der Länge 3), usw.

Führt dazu, dass die neuen Kanten immer längere Wege beschreiben, bis alle möglichen Wege gefunden sind

Ansatz durch 3 verschachtelte Schleifen, ähnlich der Matrizenmultiplikation



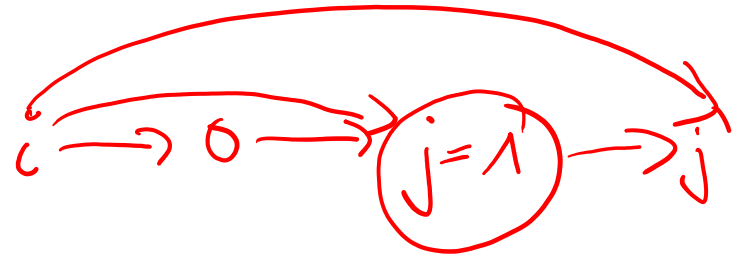
Ansatz: **Dynamisches Programmieren:**

Löse “kleinere Subprobleme” um Gesamtproblem zu lösen

In welcher Reihenfolge sollen die Knoten/Wege geprüft werden?

Idee 2: Für alle Knotenpaare (i,j) :

- Erster Schleifendurchlauf: Teste ob es einen Weg durch Knoten 0 gibt.
 - Falls es Kanten $(i,0)$ und $(0,j)$ gibt, füge die Kante (i,j) hinzu
- Zweiter Schleifendurchlauf: Teste ob es einen Weg durch Knoten 1 gibt.
 - Falls es Kanten $(i,1)$ und $(1,j)$ gibt, füge die Kante (i,j) hinzu
 - Nachdem schon alle Wege durch 0 nach 1 gefunden wurden, findet dieser Schritt auch alle Wege $(i,0)$ $(0,1)$ und $(1,j)$. Dasselbe gilt für alle Wege von 1 durch 0, d.h. alle Wege $(i,1)$ und $(1,0)$ $(0,j)$.
 - Daher findet dieser Schritt alle Wege, die als innere Knoten nur Knoten aus $\{0,1\}$ benutzen
- **Allgemeine Invariante: Die Ausführung der äußersten Schleife findet für den jeweiligen Wert k alle Wege von i nach j , die als innere Knoten nur Knoten aus $\{0, \dots, k\}$ benutzen**
(innere Knoten = nicht Start- oder Endknoten des Weges)



```
Transitive-Hülle (Matrix a) {  
    n = a.numberOfRows();  
    for(int k = 0; k < n; ++k)  
        /* Test all pairs (i,j) */  
        for(int i = 0; i < n; ++i)  
            for(int j = 0; j < n; ++j)  
                a[i,j] = a[i,j] | a[i,k] & a[k,j];  
}
```

a Adjazenzmatrix
bei **a** als Bit-Matrix
| binärer OR Operator
& binärer AND Operator

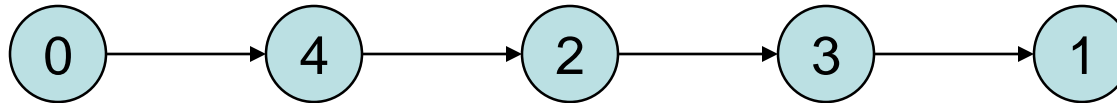
Fügt Weg als neue
Kante (falls sie noch
nicht existiert) in
den Graphen ein

True (1), falls Weg
zwischen i und j
über k existiert

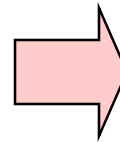
Mit Adjazenzmatrixrepräsentation:

- $O(n)$ für die innerste Schleife,
- $O(n^2)$ für die zwei inneren Schleifen,
- $O(n^3)$ für alle drei Schleifen

Beispiel: Floyd-Warshall für Transitive Hülle (1)



	0	1	2	3	4
0	0	0	0	0	1
1	0	0	0	0	0
2	0	0	0	1	0
3	0	1	0	0	0
4	0	0	1	0	0



	0	1	2	3	4
0	0	1	1	1	1
1	0	0	0	0	0
2	0	1	0	1	0
3	0	1	0	0	0
4	0	1	1	1	0

Beispiel: Floyd-Warshall für Transitive Hülle (2)



Werte für k (Durchläufe äußere Schleife):

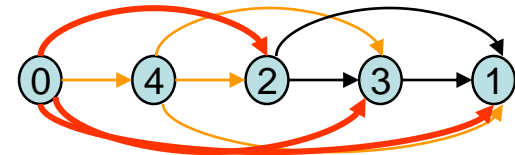
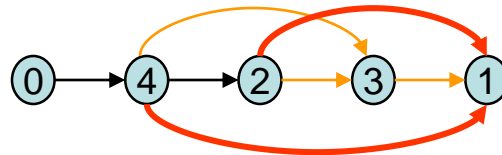
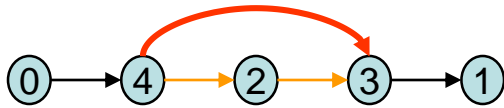
0: keine Kante, da $a(?,0)$ nicht möglich

1: keine Kante, da $a(1,?)$ nicht möglich

2: Kante $a(4,2)$, $a(2,3) \Rightarrow a(4,3)$

3: $a(2,3)$, $a(3,1) \Rightarrow a(2,1)$
 $a(4,3)$, $a(3,1) \Rightarrow a(4,1)$

4: $a(0,4)$, $a(4,1) \Rightarrow a(0,1)$
 $a(0,4)$, $a(4,2) \Rightarrow a(0,2)$
 $a(0,4)$, $a(4,3) \Rightarrow a(0,3)$



	0	1	2	3	4
0	0	0	0	0	1
1	0	0	0	0	0
2	0	0	0	1	0
3	0	1	0	0	0
4	0	0	1	1	0

	0	1	2	3	4
0	0	0	0	0	1
1	0	0	0	0	0
2	0	1	0	1	0
3	0	1	0	0	0
4	0	1	1	1	0

	0	1	2	3	4
0	0	1	1	1	1
1	0	0	0	0	0
2	0	1	0	1	0
3	0	1	0	0	0
4	0	1	1	1	0



Berechnung der kürzesten Wege zwischen allen Knoten ist simpel aus dem Algorithmus für Transitive Hüllen ableitbar

Unterschied

Adjazenzmatrix speichert die Weglängen zwischen den Knoten (entspricht der Kostenmatrix)

Statt 0/1 Werte die Wegkosten

Beim Feststellen eines Weges über 2 Kanten einfache Berechnung der Weglänge dieser neuen Kante und Vergleich mit aktueller Weglänge (falls schon vorhanden)

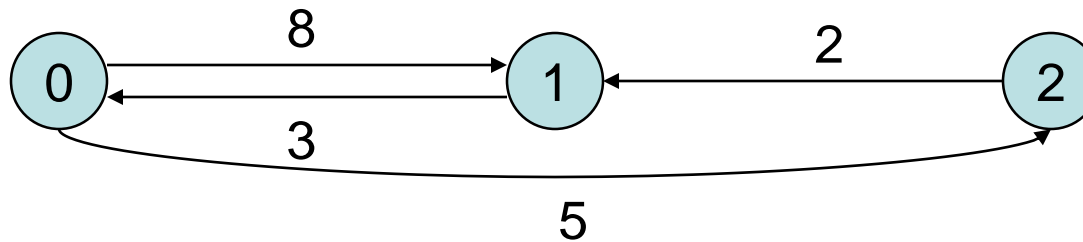
Statt logisches AND die Berechnung der Kostensumme

```
Floyd-Warshall (Matrix a) {  
    n = a.numberOfRows();  
    for(int k = 0; k < n; k++)  
        for(int i = 0; i < n; i++)  
            for(int j = 0; j < n; j++) {  
                int newPathLength = a[i,k] + a[k,j];  
                if(newPathLength < a[i,j])  
                    a[i,j] = newPathLength;  
            }  
}
```

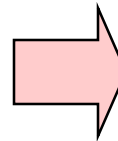
Eintrag in die Kostenmatrix, falls
neuer Weg kürzer als möglicherweise
vorhandener alter Weg

Berechnung der
Weglänge über
neuen Weg

Invariante: Die k -te Ausführung (Zählung beginnend bei 0) der äußersten Schleife findet die Länge des kürzesten Weges von i nach j , der als innere Knoten nur Knoten aus $\{0, \dots, k\}$ benützt



	0	1	2
0	0	8	5
1	3	0	∞
2	∞	2	0

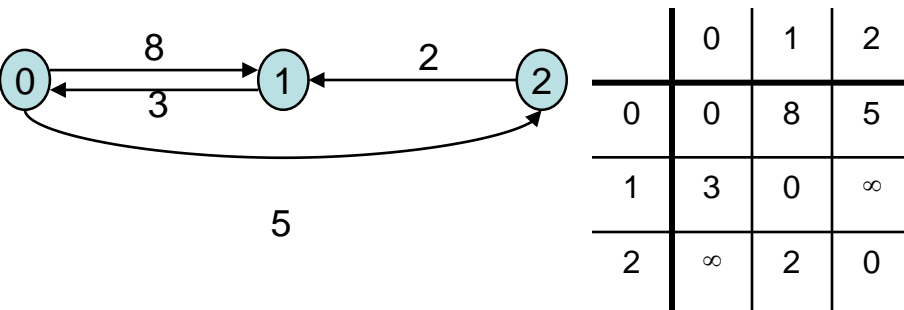


	0	1	2
0	0	7	5
1	3	0	8
2	5	2	0

Beispiel: Floyd-Warshall für kürzeste Wege (2)



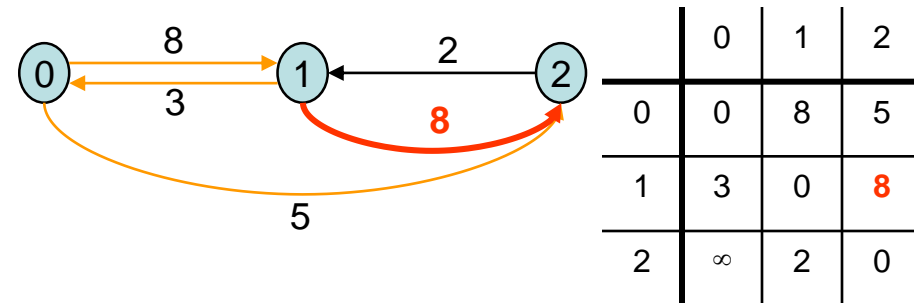
Ausgangsposition



k: 0

$(1,0) (0,2) \Rightarrow (1,2)$, Kosten: 8

$(1,0) (0,1) \Rightarrow (1,1)$, Kosten: 11



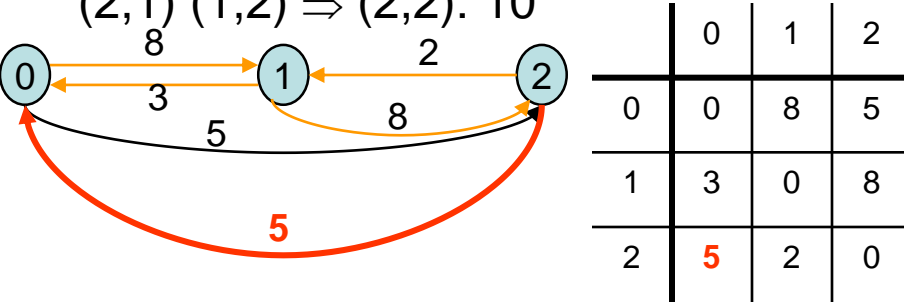
k: 1

$(0,1) (1,2) \Rightarrow (0,2)$: 16

$(0,1) (1,0) \Rightarrow (0,0)$: 11

$(2,1) (1,0) \Rightarrow (2,0)$: 5

$(2,1) (1,2) \Rightarrow (2,2)$: 10



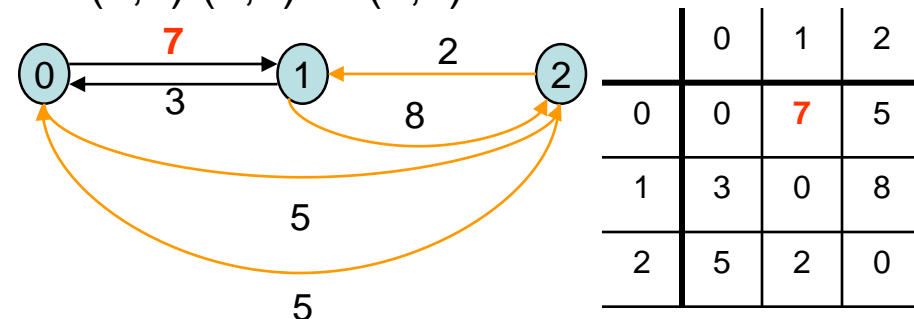
k: 2

$(0,2) (2,0) \Rightarrow (0,0)$: 10

$(0,2) (2,1) \Rightarrow (0,1)$: 7

$(1,2) (2,0) \Rightarrow (1,0)$: 13

$(1,2) (2,1) \Rightarrow (1,1)$: 10

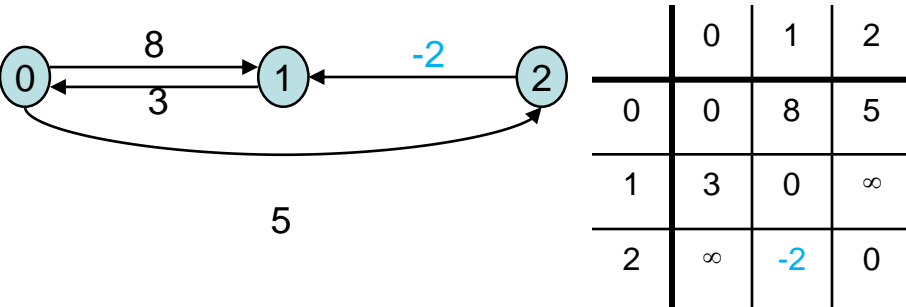


Floyd-Warshall für kürzeste Wege

Was geschieht bei negativen Kosten?



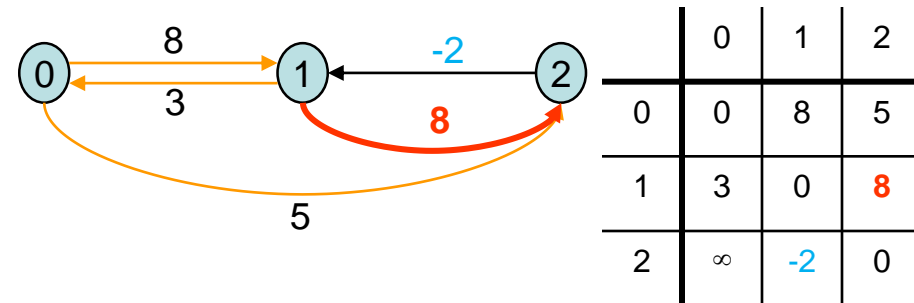
Ausgangsposition



k: 0

$(1,0) (0,2) \Rightarrow (1,2)$, Kosten: 8

$(1,0) (0,1) \Rightarrow (1,1)$, Kosten: 11



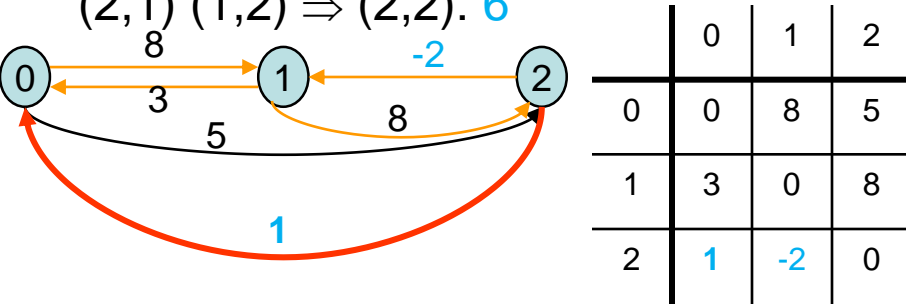
k: 1

$(0,1) (1,2) \Rightarrow (0,2)$: 16

$(0,1) (1,0) \Rightarrow (0,0)$: 11

$(2,1) (1,0) \Rightarrow (2,0)$: 1

$(2,1) (1,2) \Rightarrow (2,2)$: 6



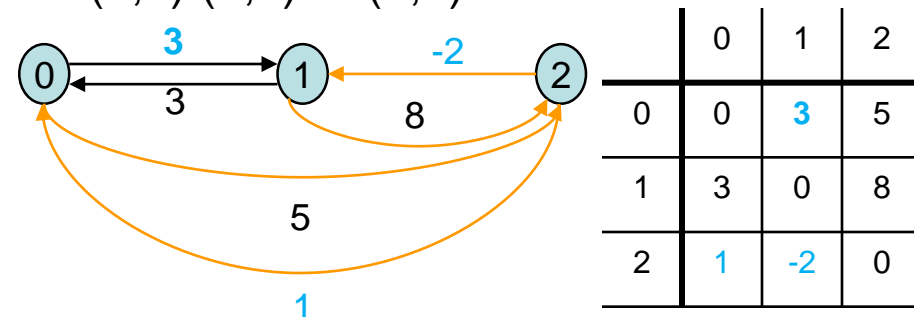
k: 2

$(0,2) (2,0) \Rightarrow (0,0)$: 6

$(0,2) (2,1) \Rightarrow (0,1)$: 3

$(1,2) (2,0) \Rightarrow (1,0)$: 9

$(1,2) (2,1) \Rightarrow (1,1)$: 6



Was geschieht bei negativen Kantenkosten?



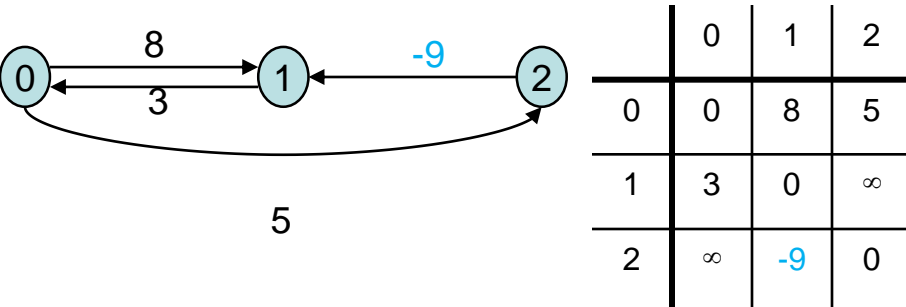
Negative Kantenkosten verursachen kein Problem solange sie keine Zyklen erzeugen, bei denen die Summe der Kantenkosten negativ ist (***negativer Zyklus***)

Floyd-Warshall für kürzeste Wege

Was geschieht bei negativen Zyklen?



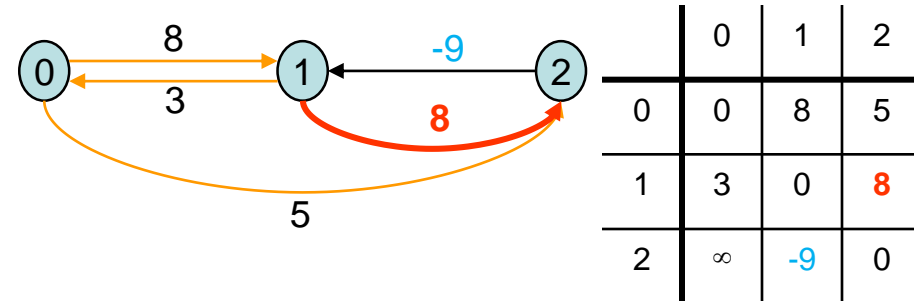
Ausgangsposition



k: 0

$(1,0) (0,2) \Rightarrow (1,2)$, Kosten: 8

$(1,0) (0,1) \Rightarrow (1,1)$, Kosten: 11



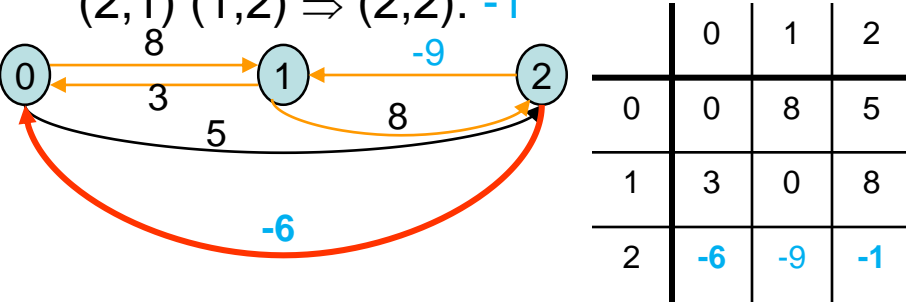
k: 1

$(0,1) (1,2) \Rightarrow (0,2)$: 16

$(0,1) (1,0) \Rightarrow (0,0)$: 11

$(2,1) (1,0) \Rightarrow (2,0)$: -6

$(2,1) (1,2) \Rightarrow (2,2)$: -1



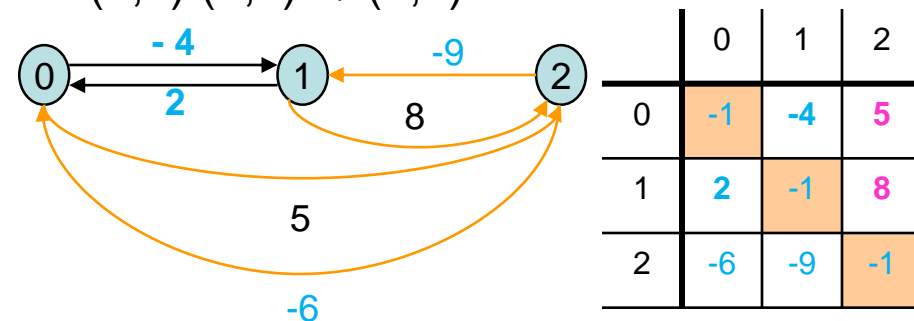
k: 2

$(0,2) (2,0) \Rightarrow (0,0)$: -1

$(0,2) (2,1) \Rightarrow (0,1)$: -4

$(1,2) (2,0) \Rightarrow (1,0)$: 2

$(1,2) (2,1) \Rightarrow (1,1)$: -1



Bei negativen Zyklen gibt es am Ende des Algorithmus einen Knoten i mit $a[i,i] < 0$

Wenn es keine negativen Zyklen gibt, gibt es am Ende keinen Knoten i mit $a[i,i] < 0$

→ Floyd-Warshall kann dazu benützt werden, die Existenz von negativen Zyklen zu **testen**

Aufwand: $O(n^3)$

Achtung: Wenn es ein Weg zwischen zwei Knoten einen negativen Zyklus enthält, ist ihre Distanz $-\infty$ / undefiniert.

Graphen

- Definitionen

- Gerichtete und ungerichtete Graphen

Topologisches Sortieren

Traversierung

- BFS und DFS

- „Bauer, Wolf, Ziege und Kohlkopf“-Problem

- Genereller iterativer Ansatz

Spannende Bäume

Kürzeste Wege