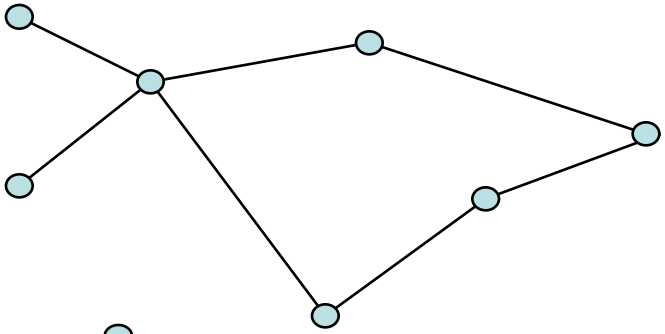
Graphen

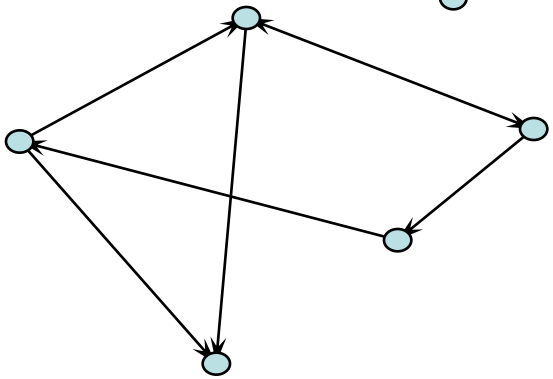
Graphen sind eine dominierende Datenstruktur in der Informatik

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

Ungerichteter Graph



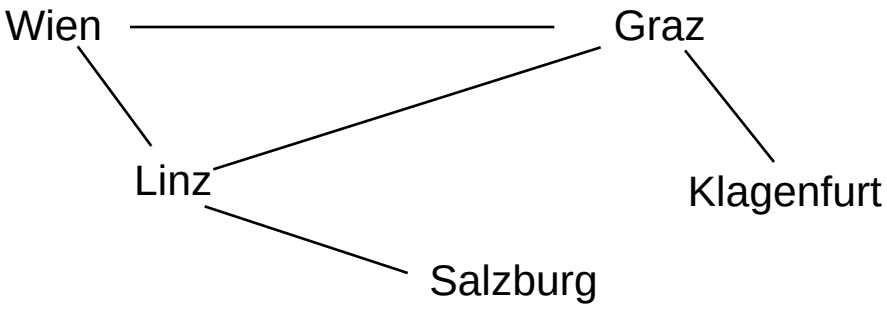
Gerichteter Graph



Graphen-Beispiele

Ortsverbindungen

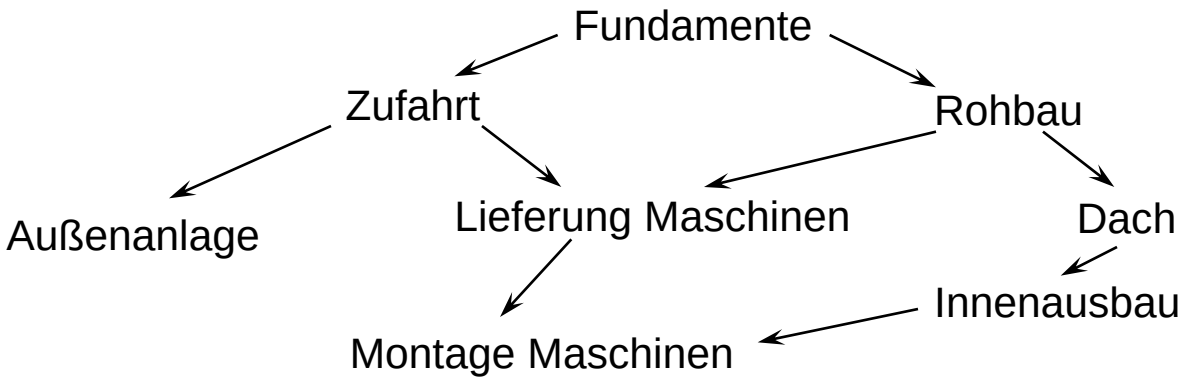
z.B. Züge



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.

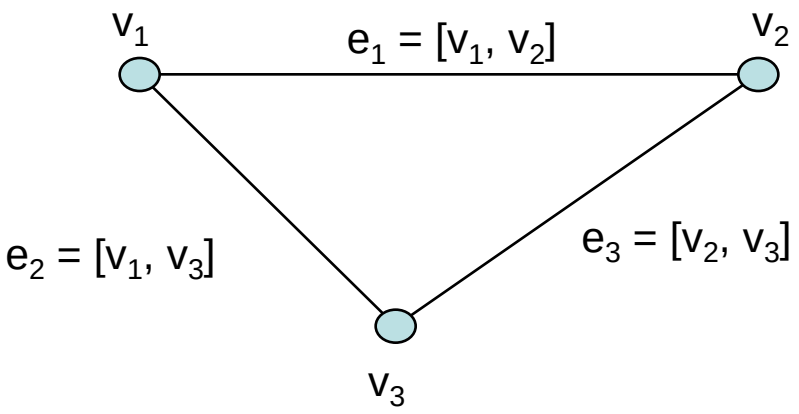
$V = \{v_1, v_2, \ldots, v_{|V|}\}$, $\boldsymbol{n} = |V|$

$E = \{e_1, e_2, \ldots e_{|E|}\}$, $\boldsymbol{m} = |E|$

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.

G(V, E)
$V = \{v_1, v_2, v_3\}$
$E = \{e_1, e_2, e_3\}$
$Grad(v_1) = 2$



Anzahl der Kanten

Die Anzahl der Kanten m erfüllt die folgende Bedingung:

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

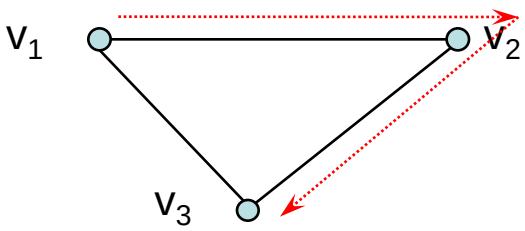
da jeder Knoten eine Kante zu jedem anderen Knoten haben kann.

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

Kantenfolge, Weg

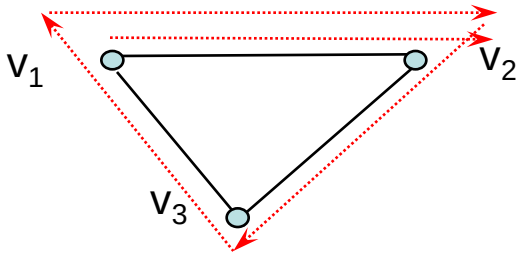
Eine ***Kantenfolge* (path)** von v_1 nach v_k in einem Graphen G ist eine endliche Folge von Kanten $[v_1,v_2]$, $[v_2,v_3]$, …, $[v_{k-1},v_k]$, wobei je 2 aufeinanderfolgende Kanten einen gemeinsamen Endpunkt haben

$KF_1 = [v_1,v_2], [v_2,v_3]$



oder

$KF_2 = [v_1,v_2], [v_2,v_3], [v_3,v_1], [v_1,v_2]$



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)

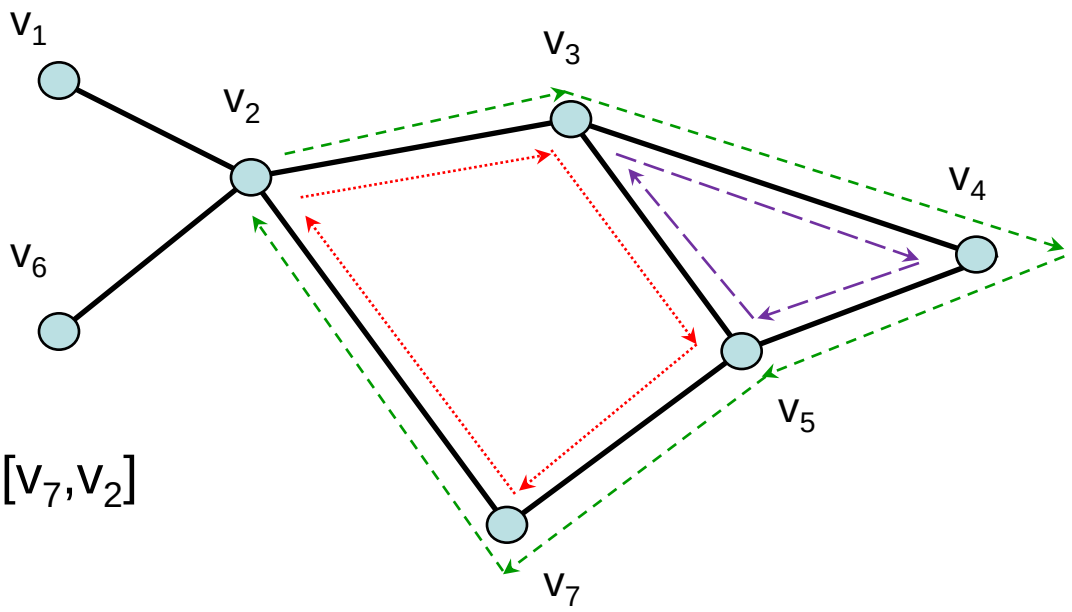
Die ***Länge*** einer Kantenfolge ist die Anzahl der Kanten auf der Kantenfolge.

Kreis

Ein ***Kreis (cycle)*** ist eine Kantenfolge, bei dem die Knoten v_1, v_2, …, 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 einen 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 höchstens $n - 1$ Kanten.

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

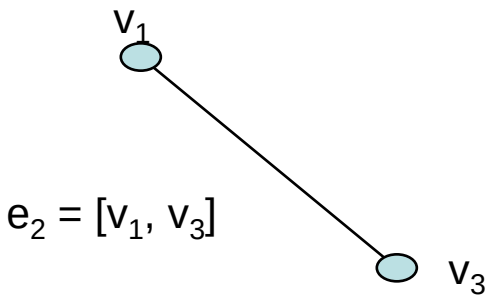
$n = 1$: Jeder Graph mit 1 Knoten ist kantenlos. Daher ist auch ein Baum mit 1 Knoten kantenlos. Daher stimmt die Induktionsbehauptung, dass ein Baum mit einem Knoten höchstens $n - 1 = 0$ Kanten hat.

$n > 1$: 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 wieder ein Baum, und zwar mit $n - 1$ Knoten. Wir nennen ihn T'. Nach der Induktionsannahme hat T' höchstens $n - 2$ Kanten. T kann von T' erzeugt werden, indem man v und die Kante von v hinzufügt. Daher hat T höchstens $n - 1$ Kanten.

Teilgraph

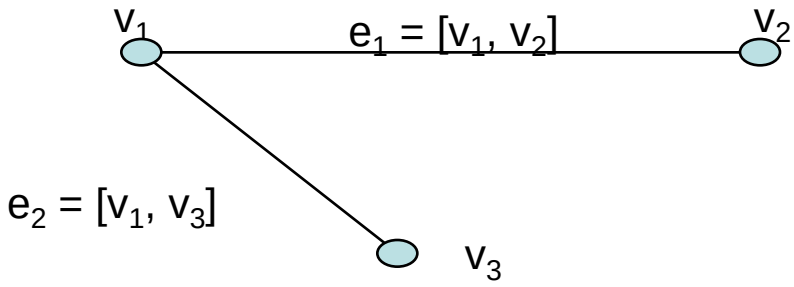
Ein Graph G'(V', E') heißt ***Teilgraph*** von G(V, E), wenn V' $\subseteq$ V und E' $\subseteq$ E

G'(V', E')
V' = {v_1, v_3}
E' = {e_2}

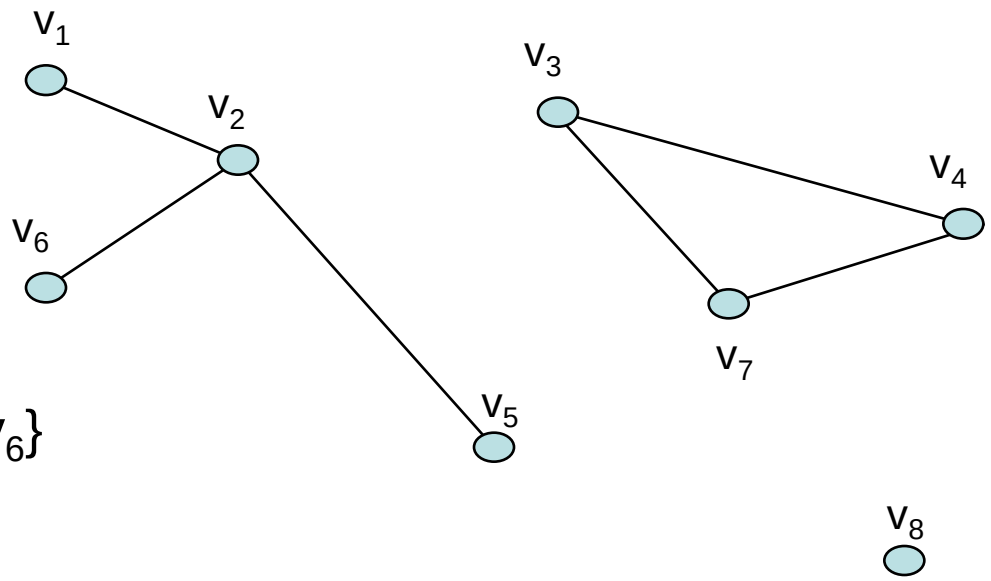


Ein Teilgraph G''(V'',E'') ist ***spannender* Baum** eines zusammenhängenden Graphens G(V,E), wenn V'' = V und G'' einen Baum bildet

G''(V'', E'')
V'' = {v_1, v_2, v_3}
E'' = {e_1, e_2}



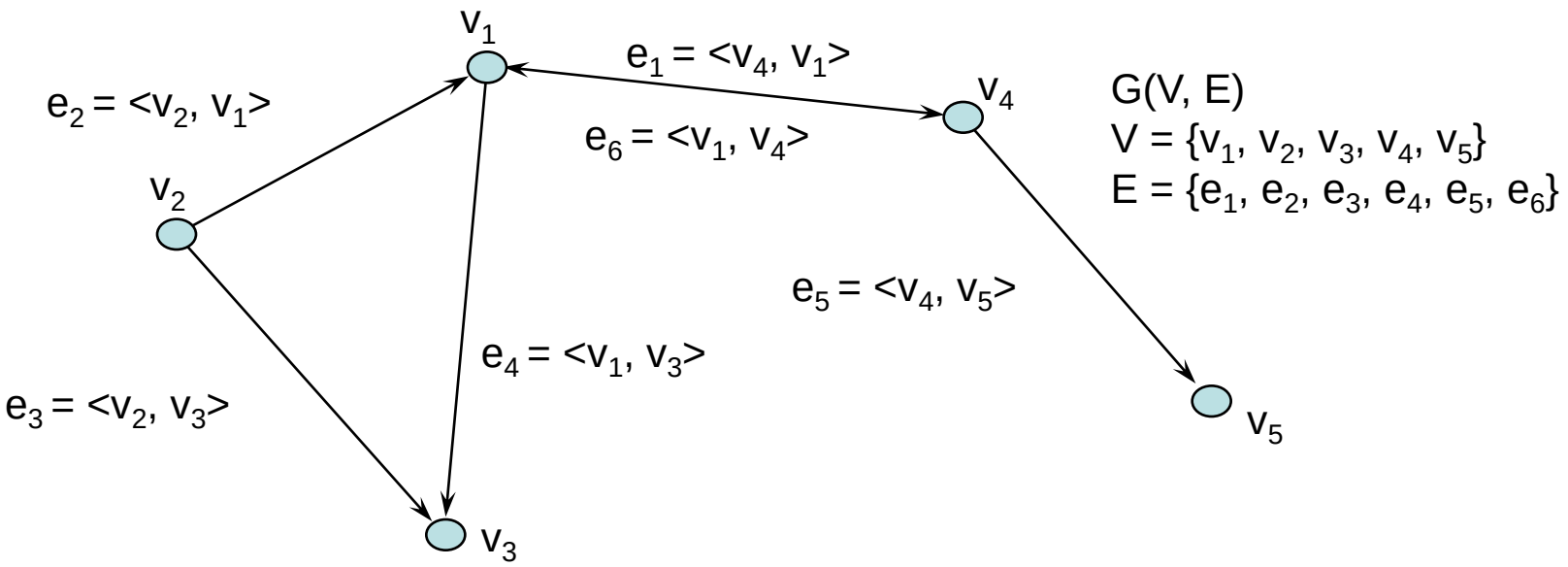
Jeder maximale verbundene Teilgraph heißt ***Komponente*** des Graphen



Komponenten:

$\{v_1, v_2, v_5, v_6\}$
$\{v_3, v_4, v_7\}$
$\{v_8\}$

Ein ***gerichteter Graph*** G(V, E) besteht aus einer Menge V von Knoten und einer Menge E von Kanten, wobei die Kanten *geordnete* Paare $<v_i,v_j>$ (oder (v_i,v_j)) von Knoten aus V sind
(***Ausgehende Kante von*** $\mathbf{v_i}$, ***eingehende Kante von*** $\mathbf{v_j}$)



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

Knotengrad

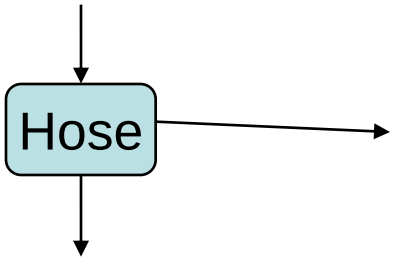
Eingangs- und Ausgangsgrad eines Knoten

indegree(v) mit v aus V: | { v' | <v', v> aus E} |

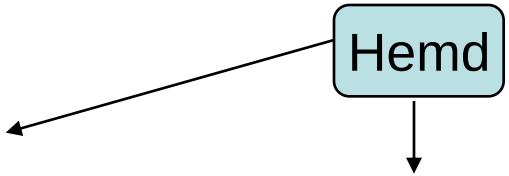
d.h. Anzahl der in v eingehenden Kanten

outdegree(v) mit v aus V: | { v' | <v, v'> aus E} |

d.h. Anzahl der von v ausgehenden Kanten



indegree: 1, outdegree: 2



indegree: 0, outdegree: 2

Anzahl der Kanten

Die Anzahl der Kanten m erfüllt die folgende Bedingung:

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

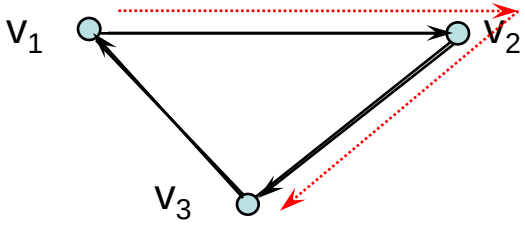
da jeder Knoten eine Kante zu jedem anderen Knoten haben kann.

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

Kantenfolge, Weg

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

$KF_1 = <v_1,v_2>, <v_2,v_3>$

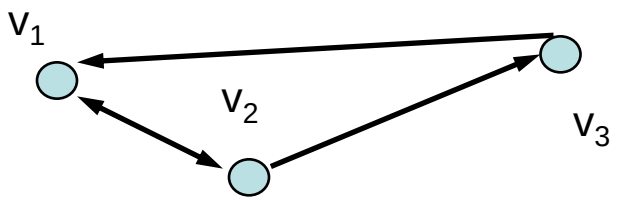


Ein ***gerichteter Weg (simple directed path)*** ist eine Kantenfolge in der alle Knoten verschieden sind (ein einzelner Knoten gilt auch als Weg)

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

Kreis

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



Kreise

$<v_1,v_2>$, $<v_2,v_1>$

$<v_1,v_2>$, $<v_2,v_3>$, $<v_3, v_1>$

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

Azyklische Graphen

Lemma 3: 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 $E' = \{< v_i, v_j > \mid < v_j, v_i > \ in\ E\}$. G' ist genauso wie G azyklisch.

Nimm einen beliebigen Knoten s und folge beginnend von s einen gerichteten 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 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.

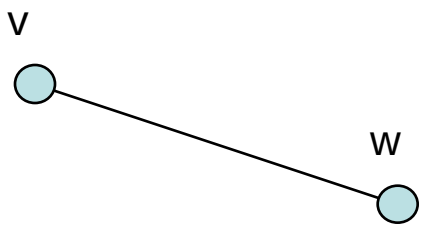
Daher hat v keine ausgehende Kante in G', dh keine eingehende Kante in G.

Wir unterscheiden 2 Methoden zur Speicherung von Graphen

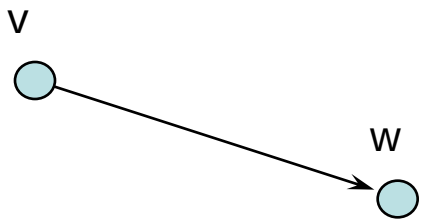
- Adjazenzmatrix-Darstellung
- Adjazenzlisten-Darstellung

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

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

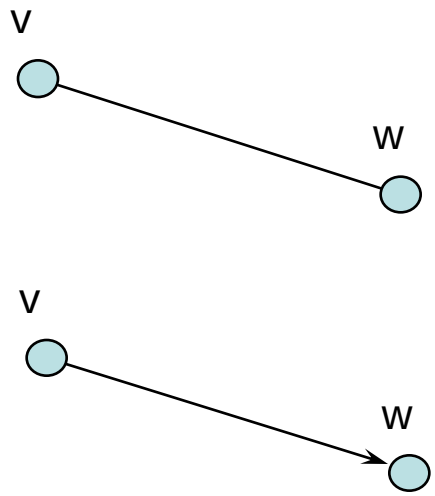


gerichtetete Kante <v,w>:
v adjazent zu w
w aber NICHT adjazent zu v



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

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



	…	v	…	w	…
…					
v				1	
…					
w		1			
…					

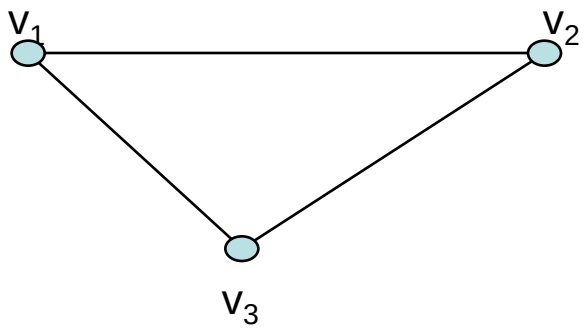
	…	v	…	w	…
…					
v				1	
…					
w					
…					

Bei ungerichteten Graphen ist die Matrix symmetrisch

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

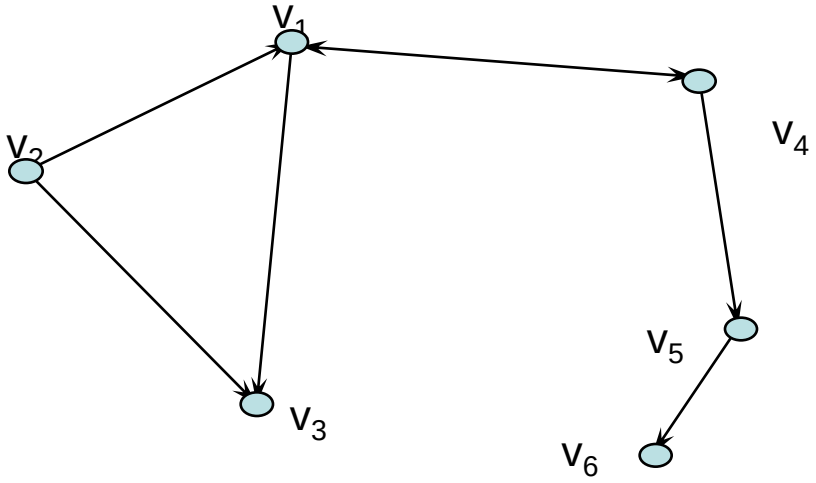
Adjazenzmatrix-Beispiele

Ungerichteter Graph



	v_1	v_2	v_3
v_1		1	1
v_2	1		1
v_3	1	1	

Gerichteter Graph

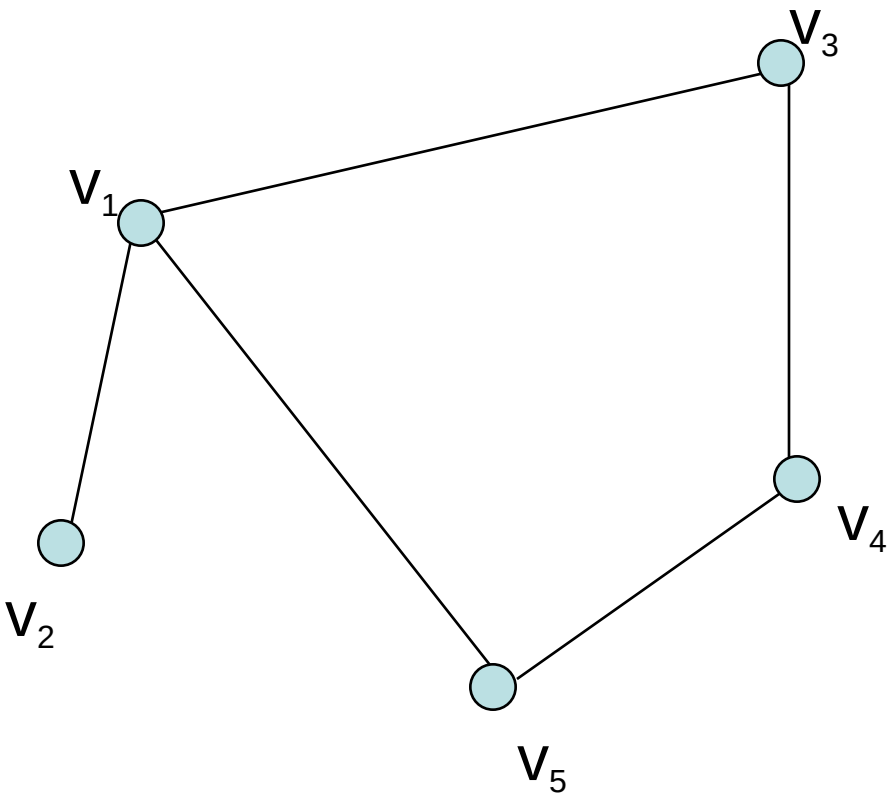


	v_1	v_2	v_3	v_4	v_5	v_6
v_1			1	1		
v_2	1		1			
v_3						
v_4	1				1	
v_5						1
v_6						

Beispiel: Adj. Matrix - ungerichteter Graph (D1)

universität wien

Ungerichteter Graph (Matrix)



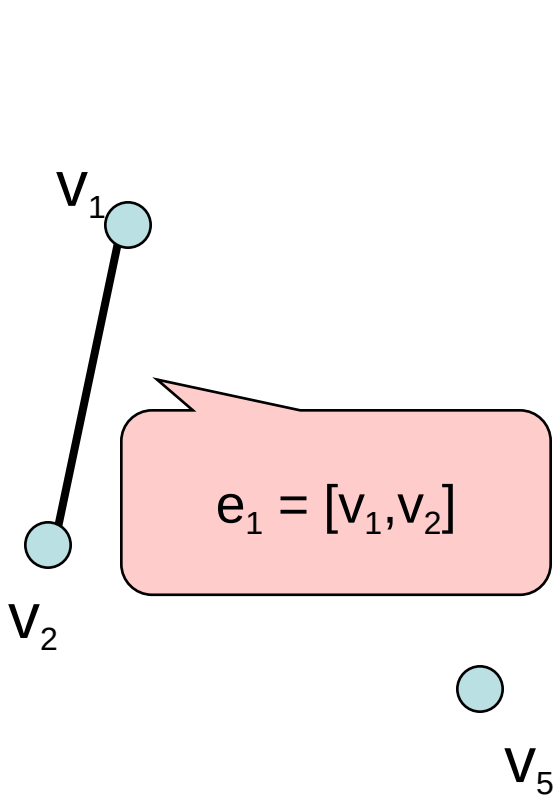
	v_1	v_2	v_3	v_4	v_5
v_1					
v_2					
v_3					
v_4					
v_5					

Diagonale nicht definiert; zugleich Symmetrieachse

Beispiel: Adj. Matrix - ungerichteter Graph (D2)

universität wien

Ungerichteter Graph

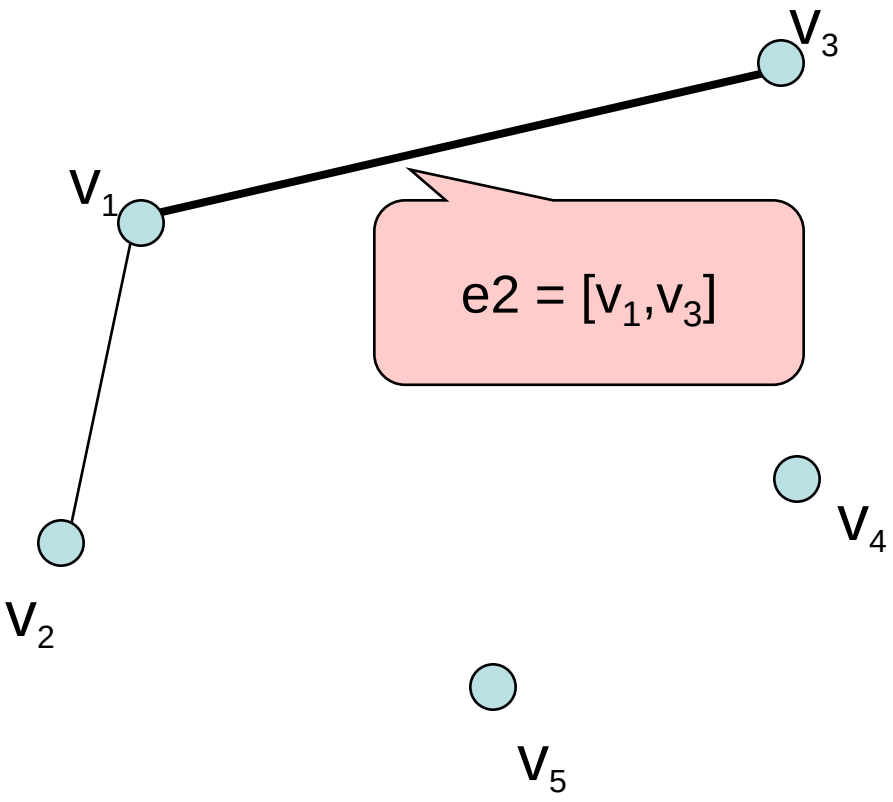


	v_1	v_2	v_3	v_4	v_5
v_1		1			
v_2	1				
v_3					
v_4					
v_5					

Beispiel: Adj. Matrix - ungerichteter Graph (D3)

universität wien

Ungerichteter Graph

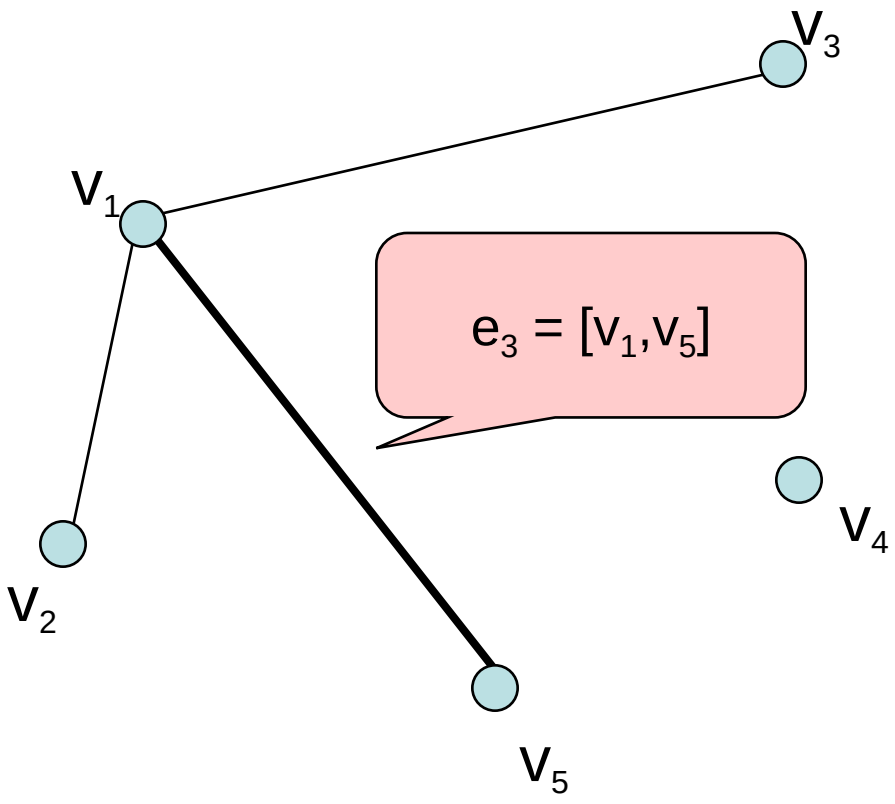


	v_1	v_2	v_3	v_4	v_5
v_1			**1**		
v_2	1				
v_3	**1**				
v_4					
v_5					

Beispiel: Adj. Matrix - ungerichteter Graph (D4)

universität wien

Ungerichteter Graph

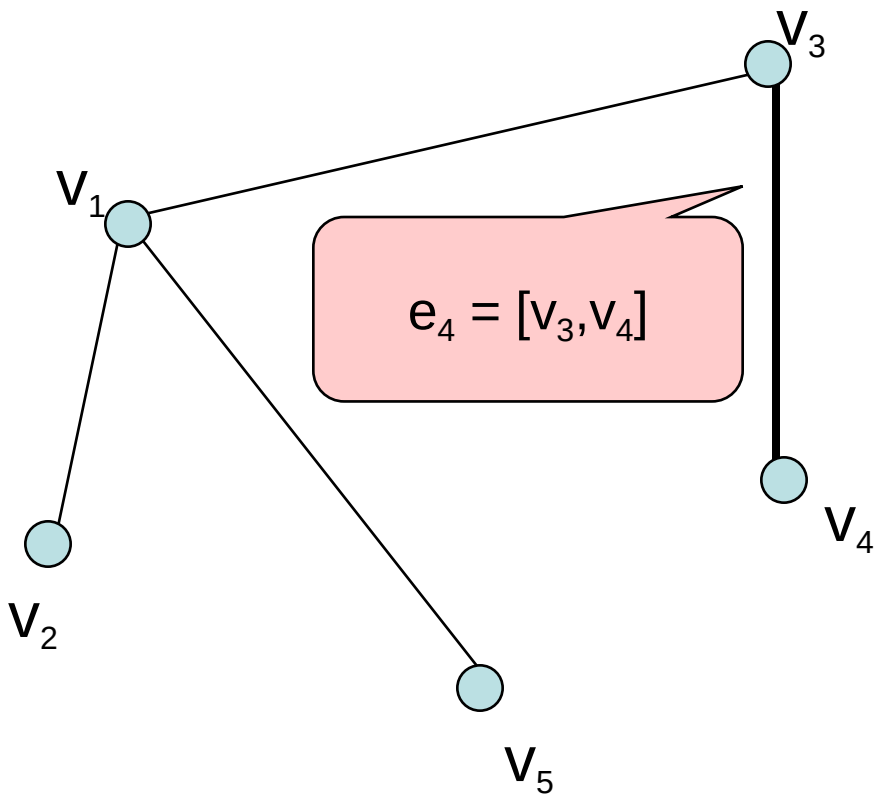


	v_1	v_2	v_3	v_4	v_5
v_1		1	1		**1**
v_2	1				
v_3	1				
v_4					
v_5	**1**				

Beispiel: Adj. Matrix - ungerichteter Graph (D5)

universität wien

Ungerichteter Graph

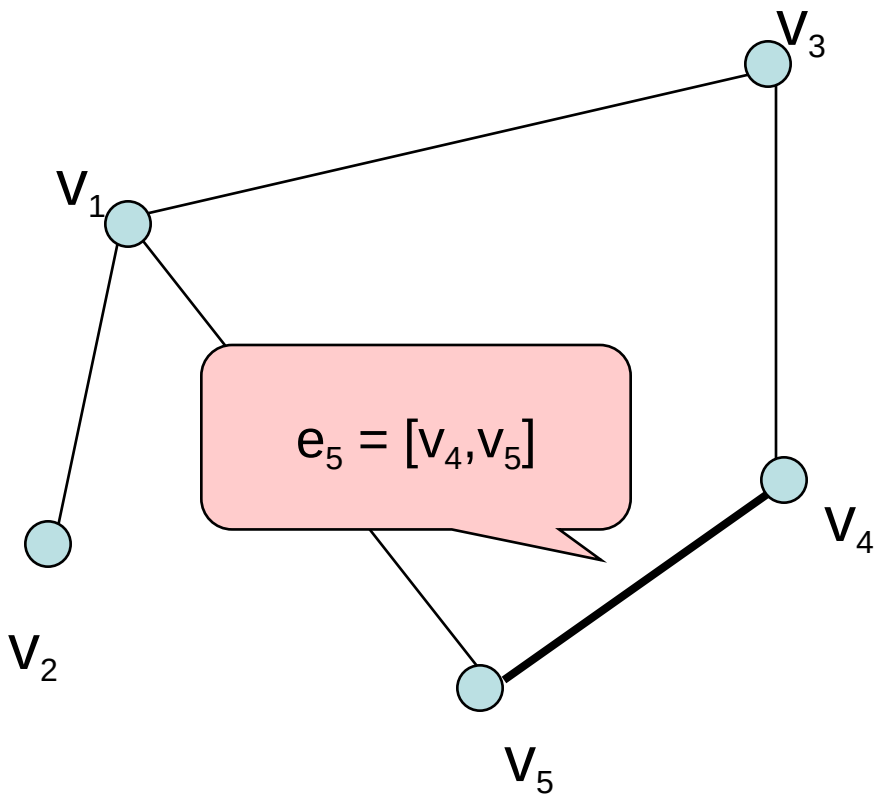


	v_1	v_2	v_3	v_4	v_5
v_1		1	1		1
v_2	1				
v_3	1			**1**	
v_4			**1**		
v_5	1				

Beispiel: Adj. Matrix - ungerichteter Graph (D6)

universität wien

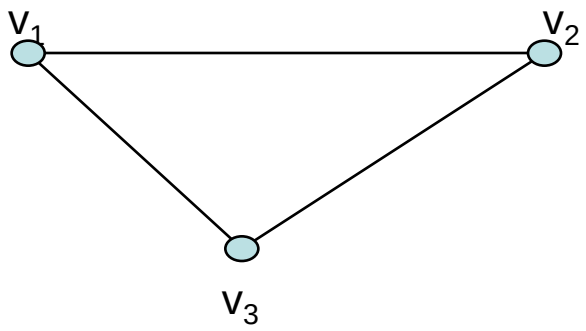
Ungerichteter Graph



	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**	

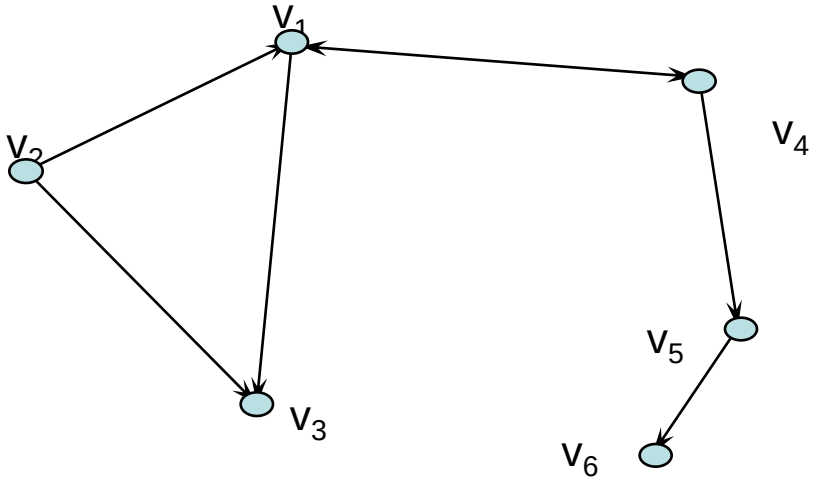
Adjazenzmatrix-Beispiele

Ungerichteter Graph



	v_1	v_2	v_3
v_1		1	1
v_2	1		1
v_3	1	1	

Gerichteter Graph

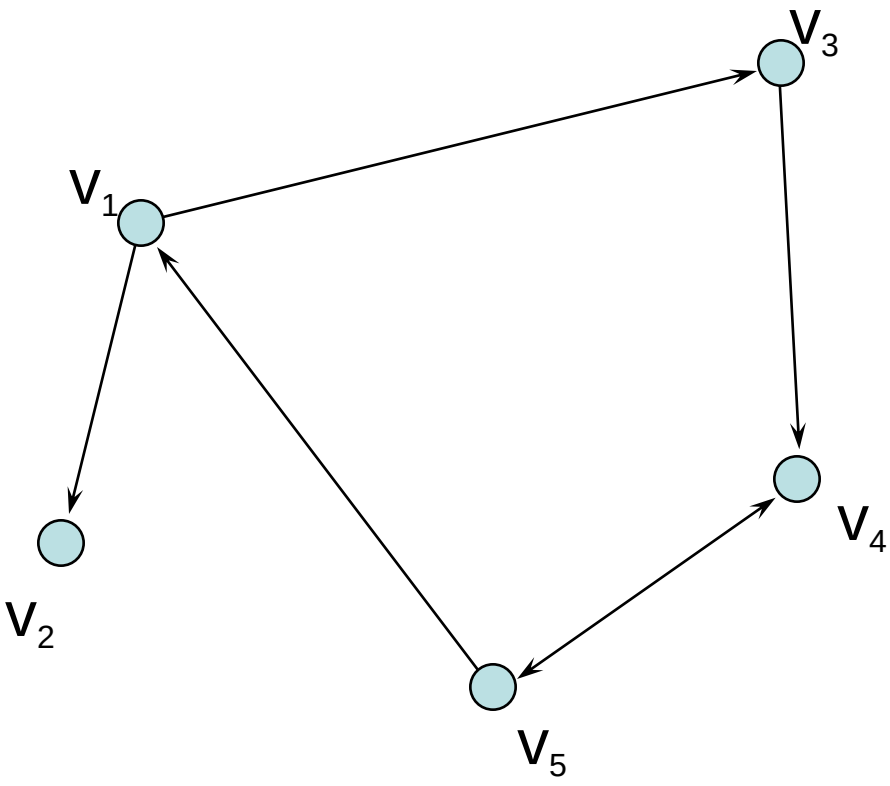


	v_1	v_2	v_3	v_4	v_5	v_6
v_1			1	1		
v_2	1		1			
v_3						
v_4	1				1	
v_5						1
v_6						

Beispiel: Adj. Matrix - gerichteter Graph (D1)

universität wien

Gerichteter Graph (Matrix)



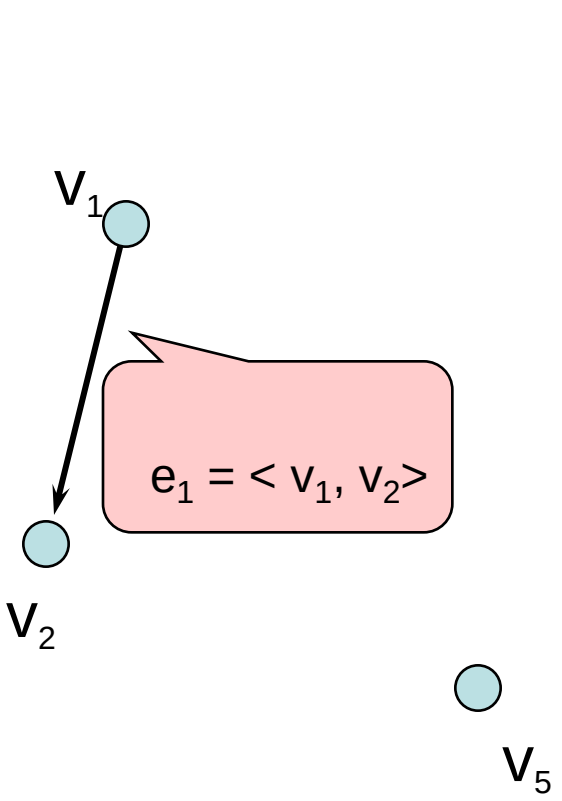
	V_1	V_2	V_3	V_4	V_5
V_1					
V_2					
V_3					

Verknüpfungen zu sich selbst - nicht definiert
Keine Symmetrieachse !

Beispiel: Adj. Matrix - gerichteter Graph (D2)

universität wien

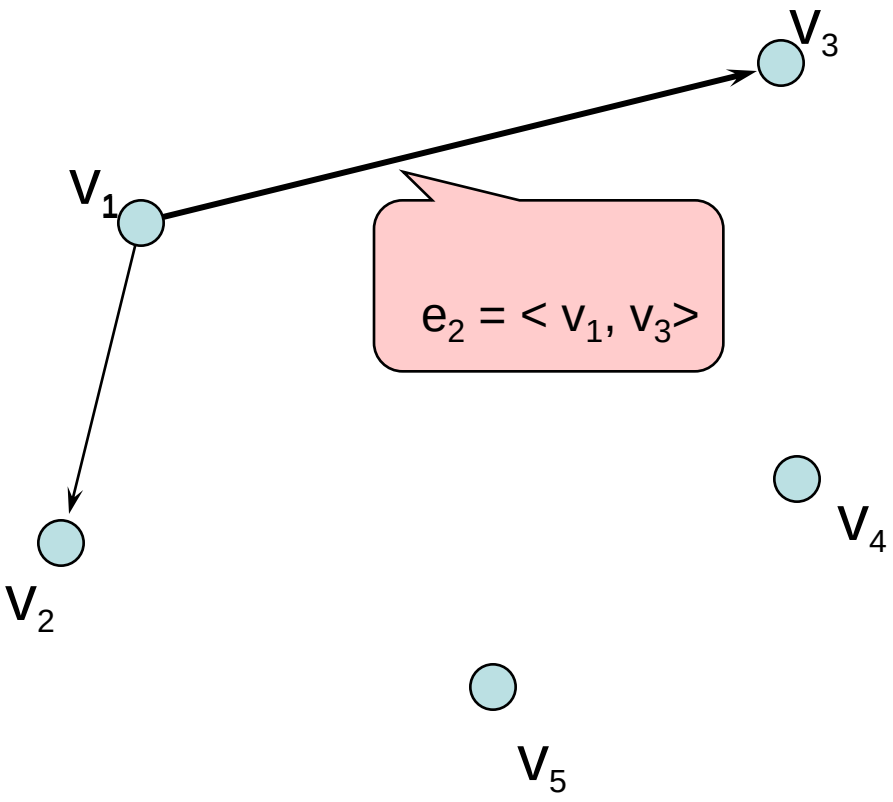
Gerichteter Graph



	v_1	v_2	v_3	v_4	v_5
v_1		1			
v_2					
v_3					
v_4					
v_5					

universität wien

Gerichteter Graph

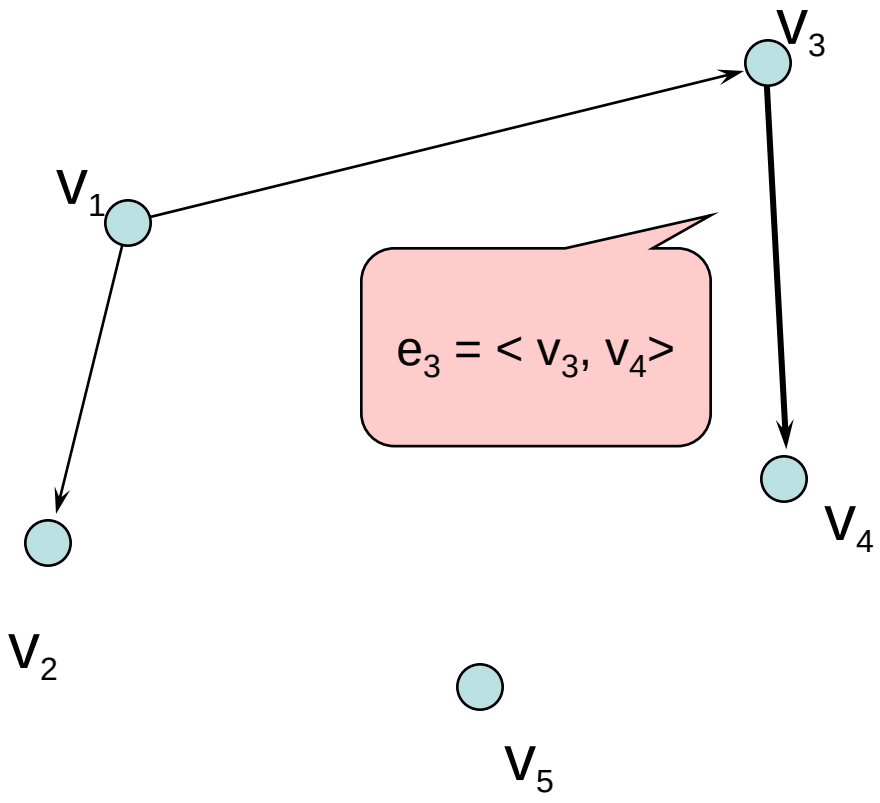


	v_1	v_2	v_3	v_4	v_5
v_1		1	**1**		
v_2					
v_3					
v_4					
v_5					

Beispiel: Adj. Matrix - gerichteter Graph (D4)

universität wien

Gerichteter Graph

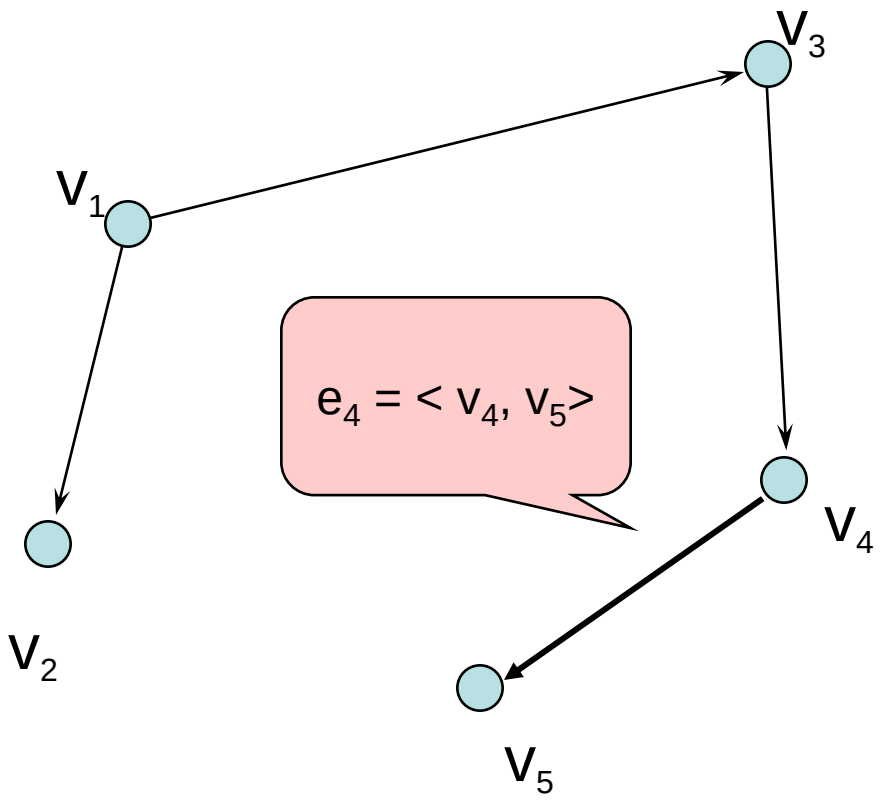


	v_1	v_2	v_3	v_4	v_5
v_1		1	1		
v_2					
v_3				1	
v_4					
v_5					

Beispiel: Adj. Matrix - gerichteter Graph (D5)

universität wien

Gerichteter Graph

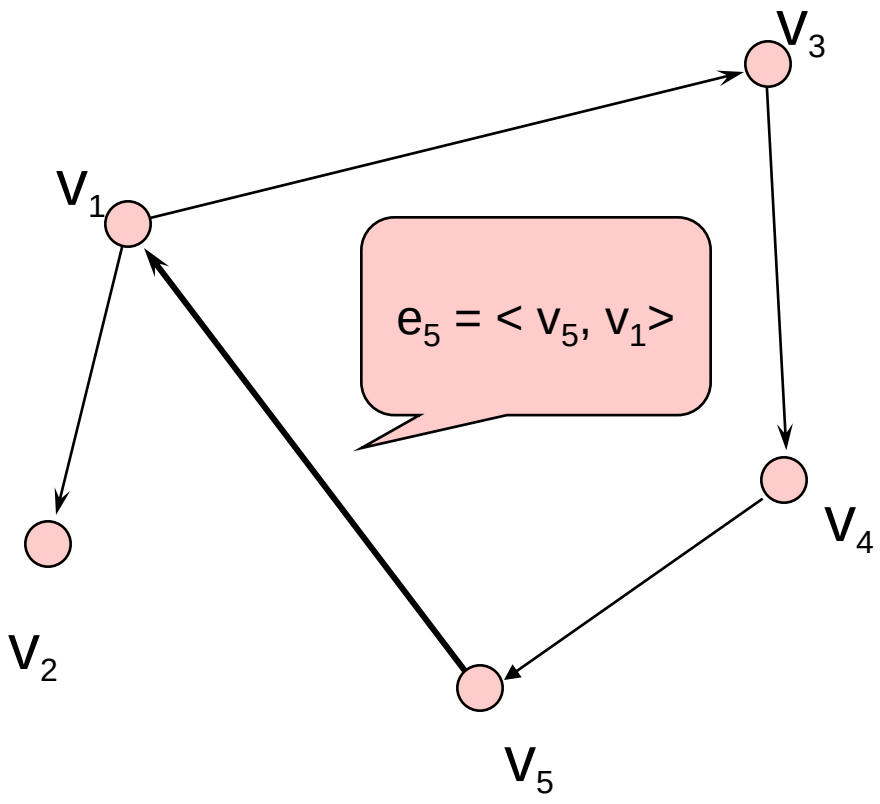


	v_1	v_2	v_3	v_4	v_5
v_1		1	1		
v_2					
v_3				1	
v_4					**1**
v_5					

Beispiel: Adj. Matrix - gerichteter Graph (D6)

universität wien

Gerichteter Graph

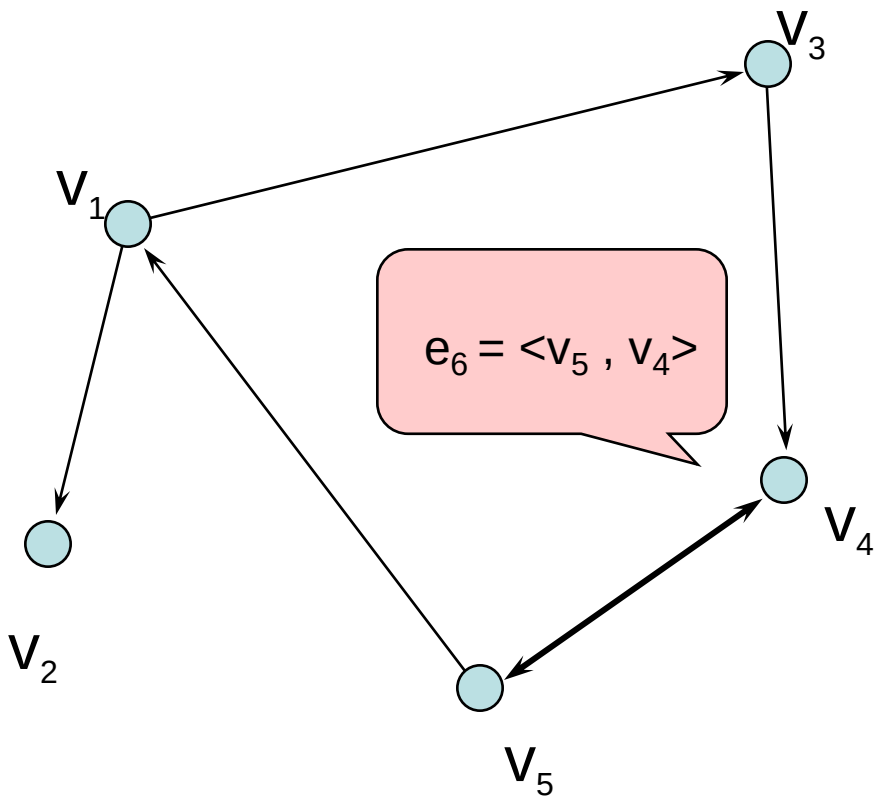


	v_1	v_2	v_3	v_4	v_5
v_1		1	1		
v_2					
v_3				1	
v_4					1
v_5	**1**				

Beispiel: Adj. Matrix - gerichteter Graph (D7)

universität wien

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**	

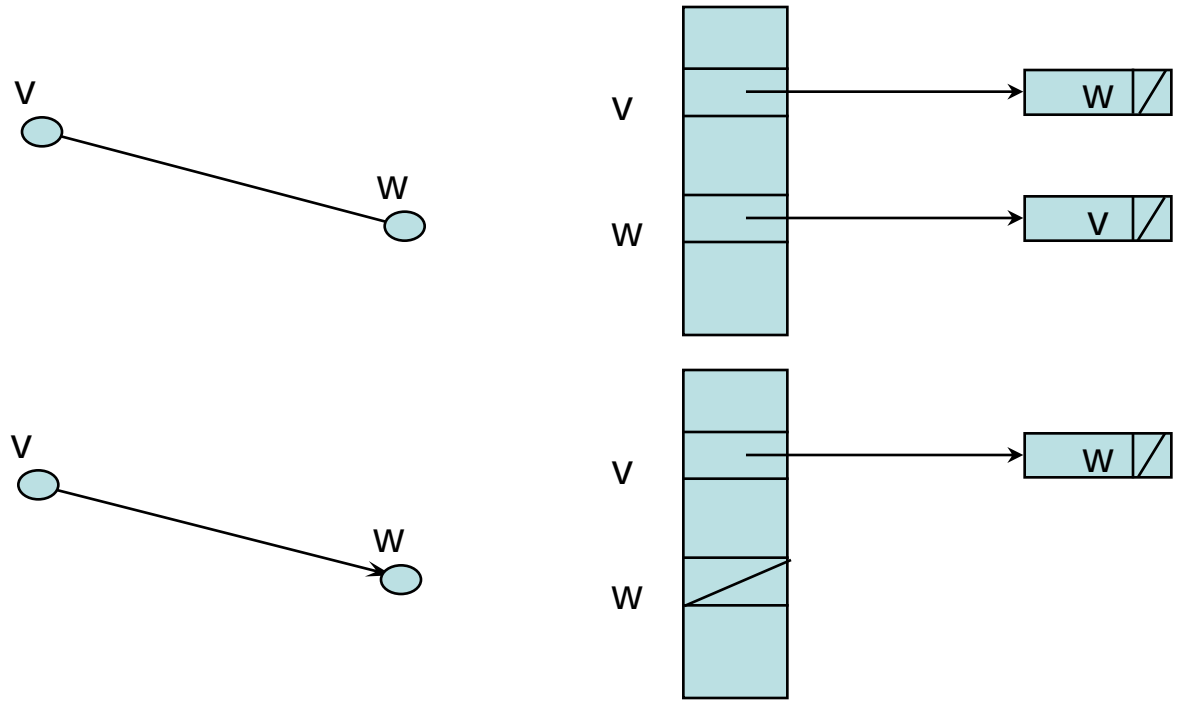
Eigenschaften

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

6.3.2 Adjazenzliste

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

Somit werden nur die auftretenden Kanten vermerkt

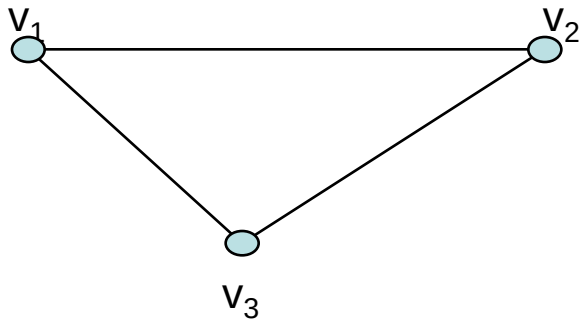


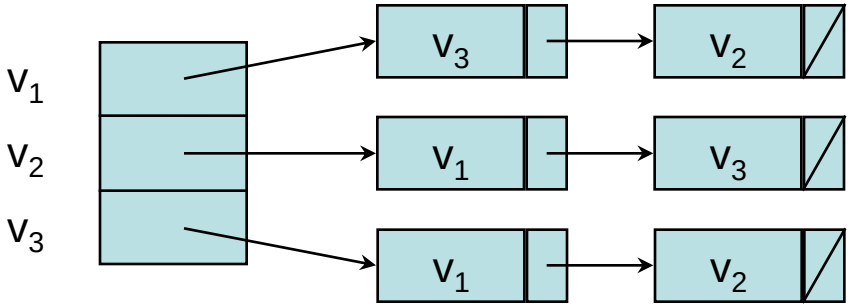
Mögliche C++ Datenstruktur:

```
class node { public: int v; node *next;}
node *adjliste[maxV]; // Adjazenzliste (maxV: maximale Anzahl von Knoten)
```

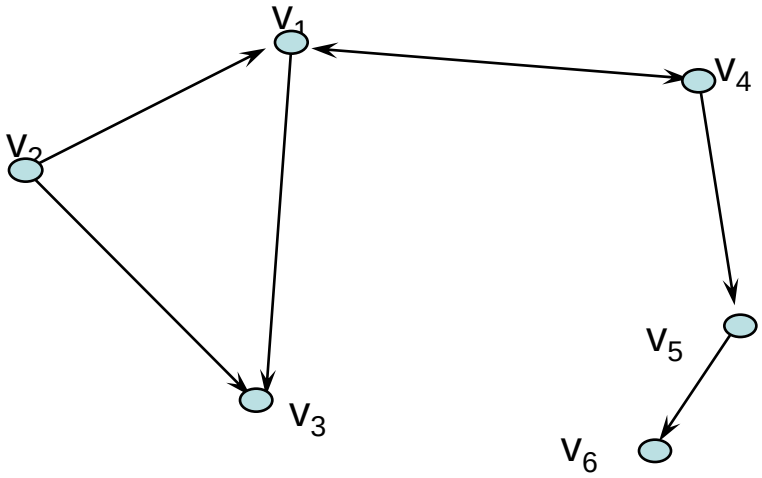
Adjazenzliste-Beispiele

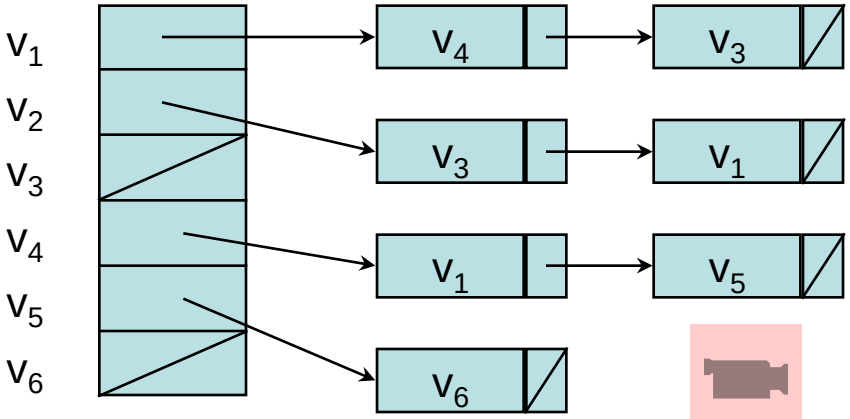
Ungerichteter Graph





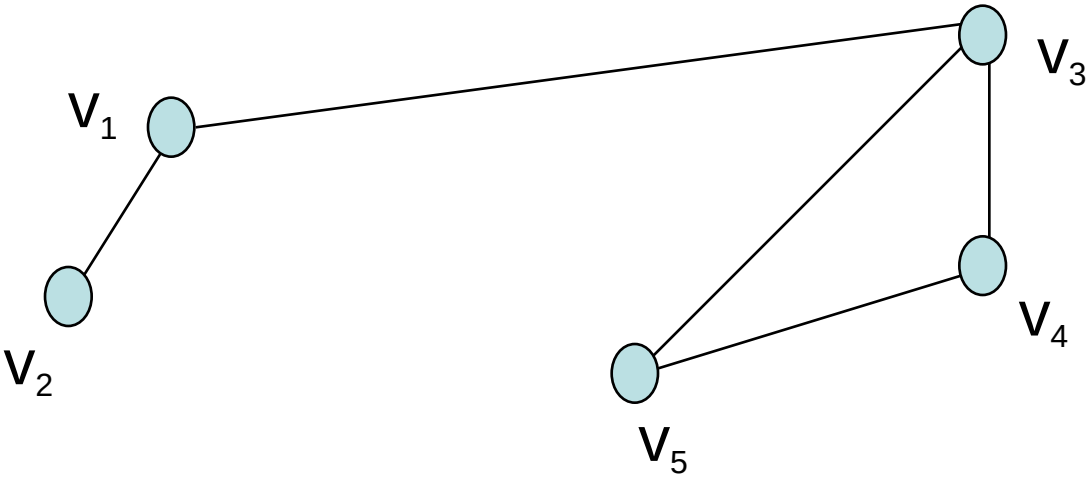
Gerichteter Graph

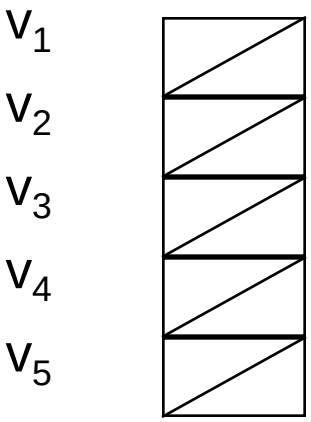




Beispiel: Adj. Liste - ungerichteter Graph (D1)

Ungerichteter Graph
(Liste)

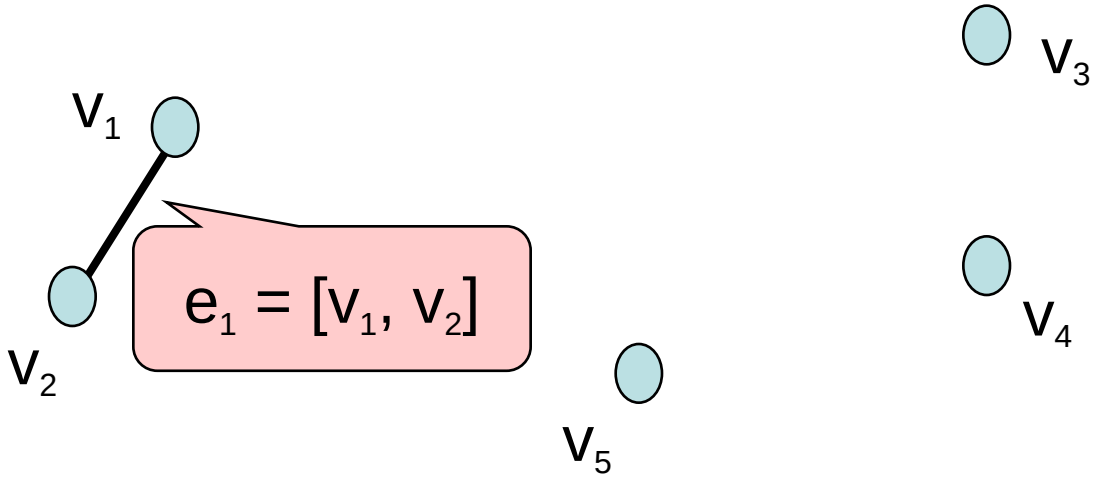


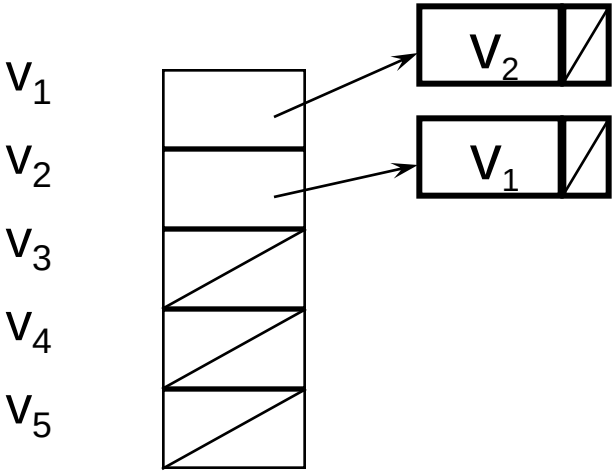


Beispiel: Adj. Liste - ungerichteter Graph (D2)

universität wien

Ungerichteter Graph

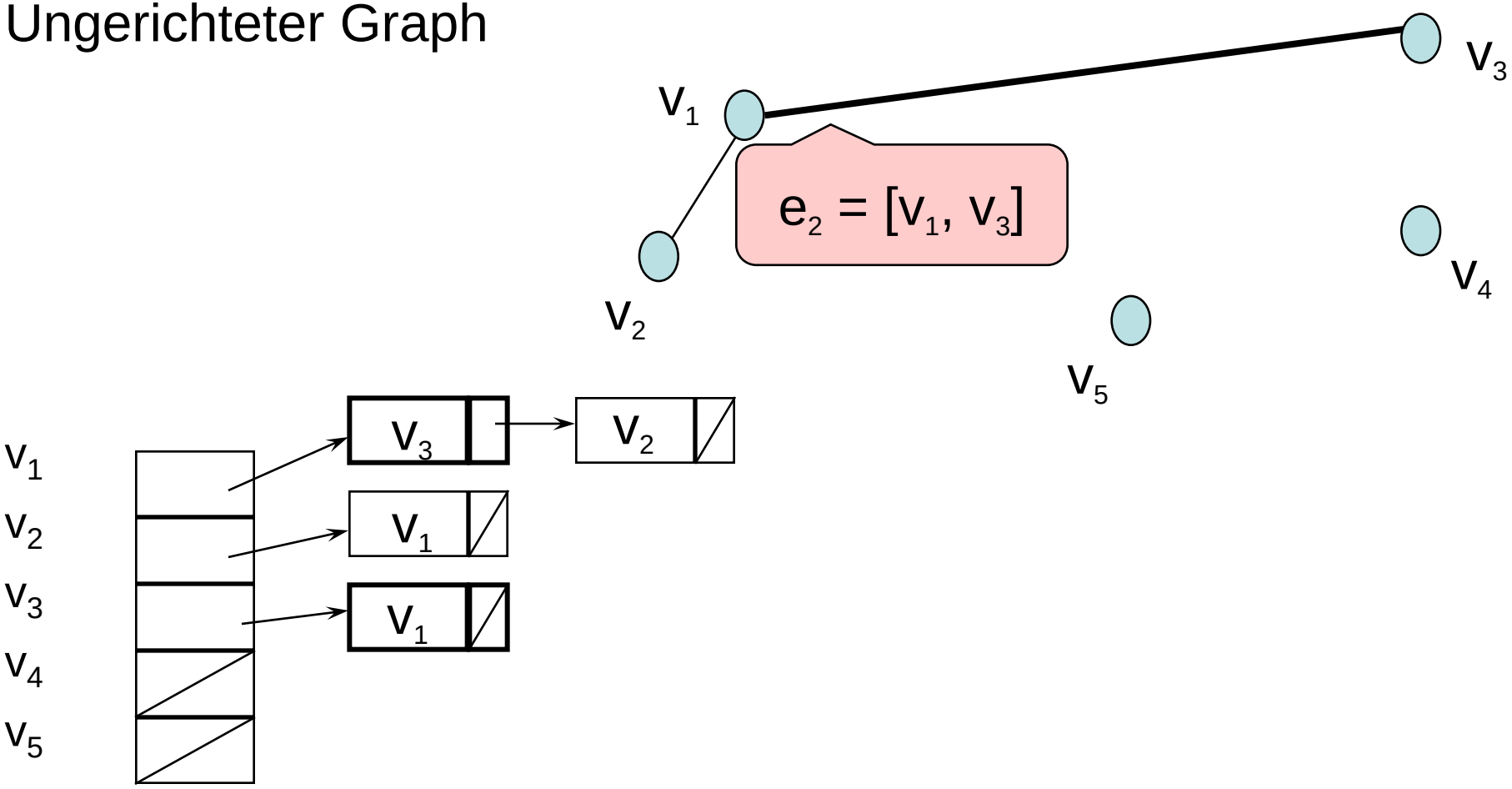




Beispiel: Adj. Liste - ungerichteter Graph (D3)

universität wien

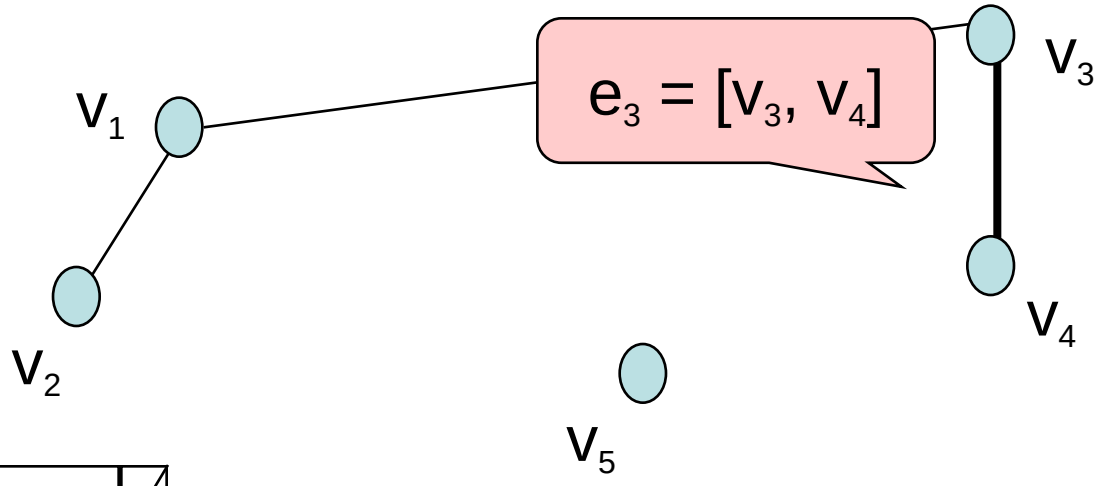
Ungerichteter Graph

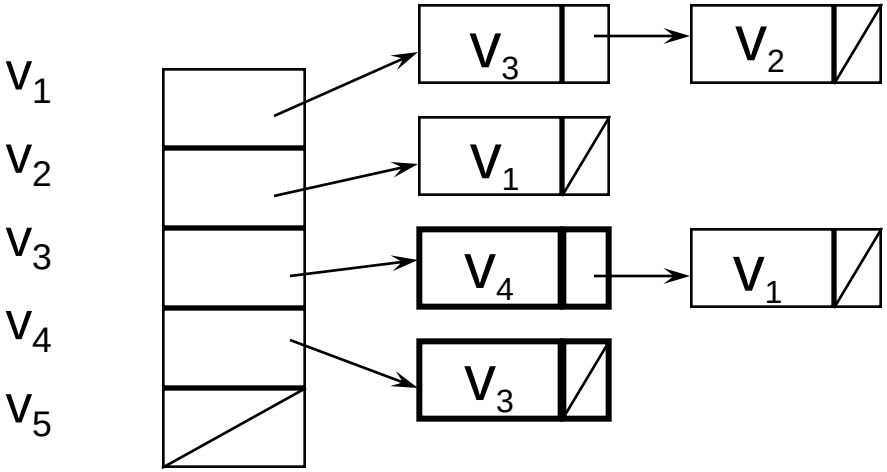


Beispiel: Adj. Liste - ungerichteter Graph (D4)

universität wien

Ungerichteter Graph

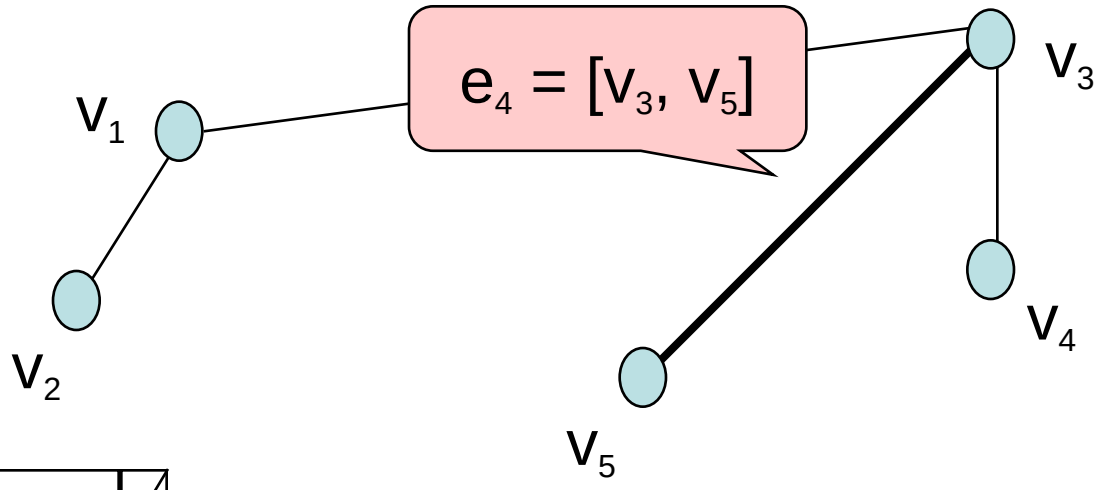


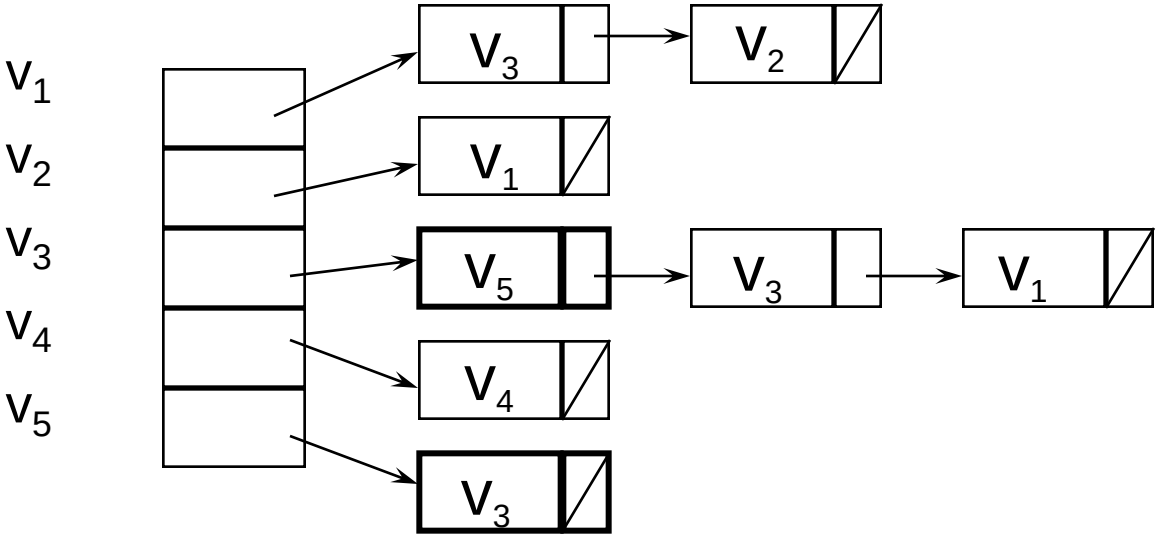


Beispiel: Adj. Liste - ungerichteter Graph (D5)

universität wien

Ungerichteter Graph

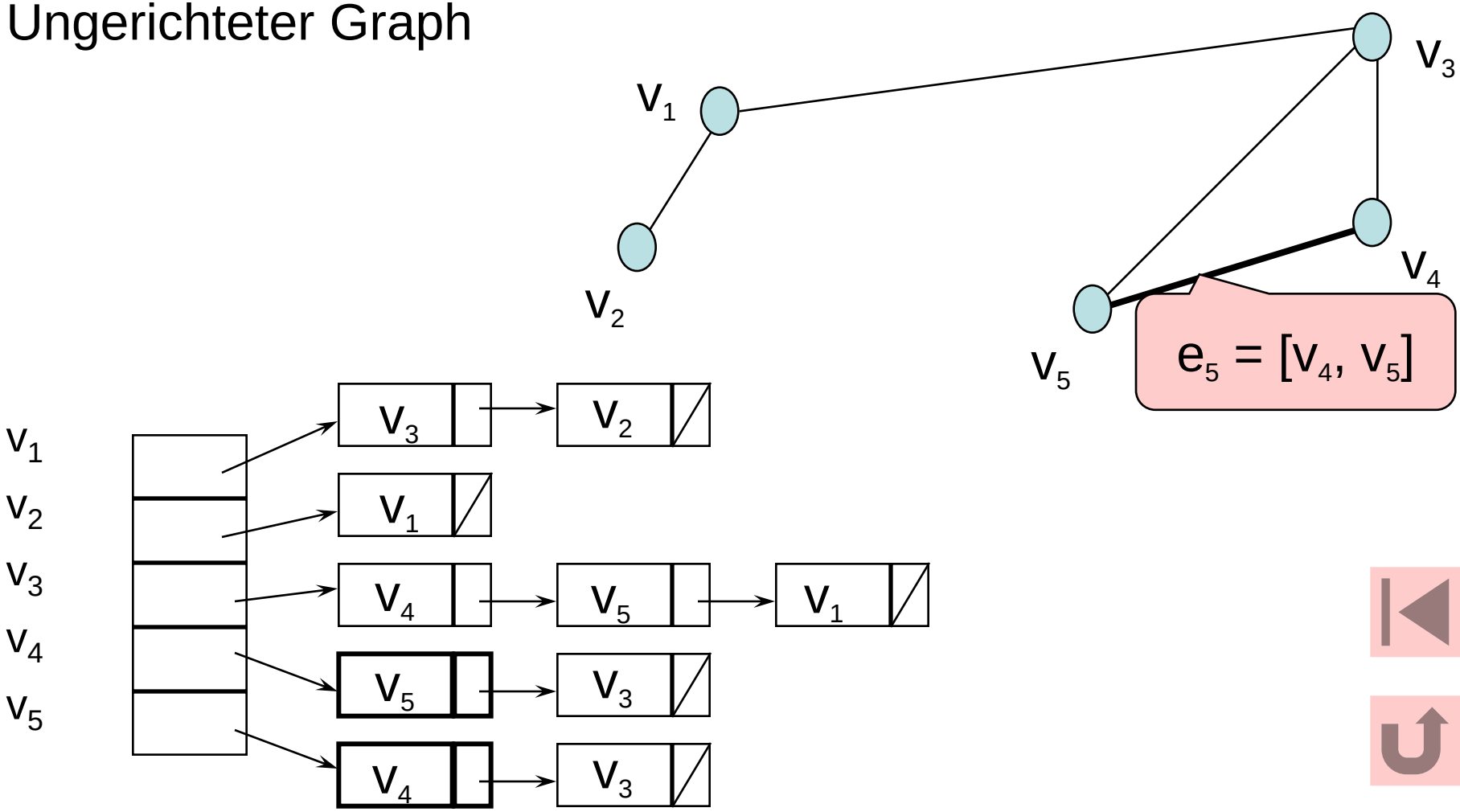




Beispiel: Adj. Liste - ungerichteter Graph (D6)

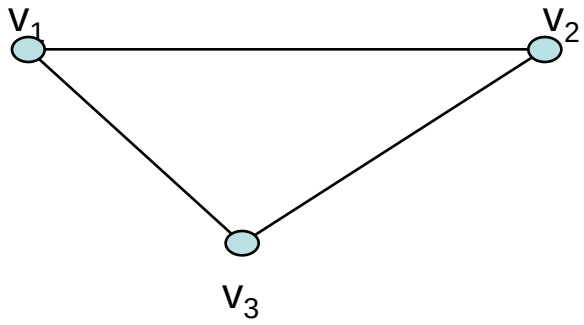
universität wien

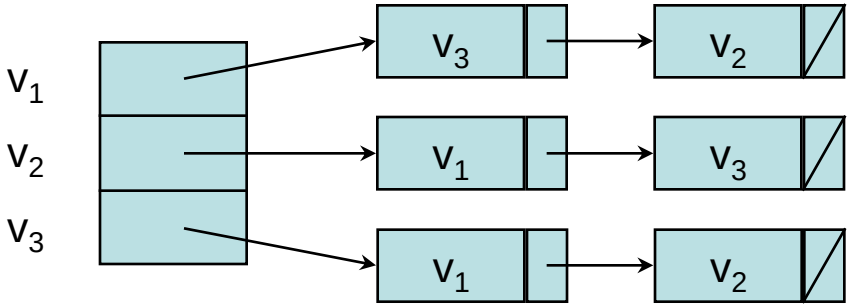
Ungerichteter Graph



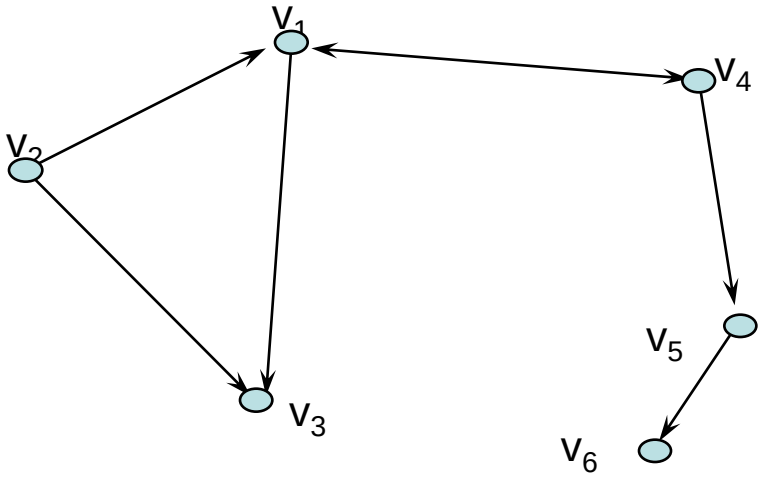
Adjazenzliste-Beispiele

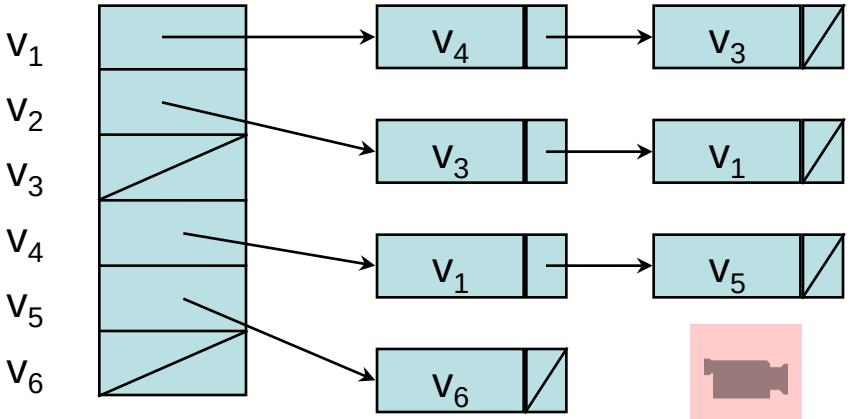
Ungerichteter Graph





Gerichteter Graph

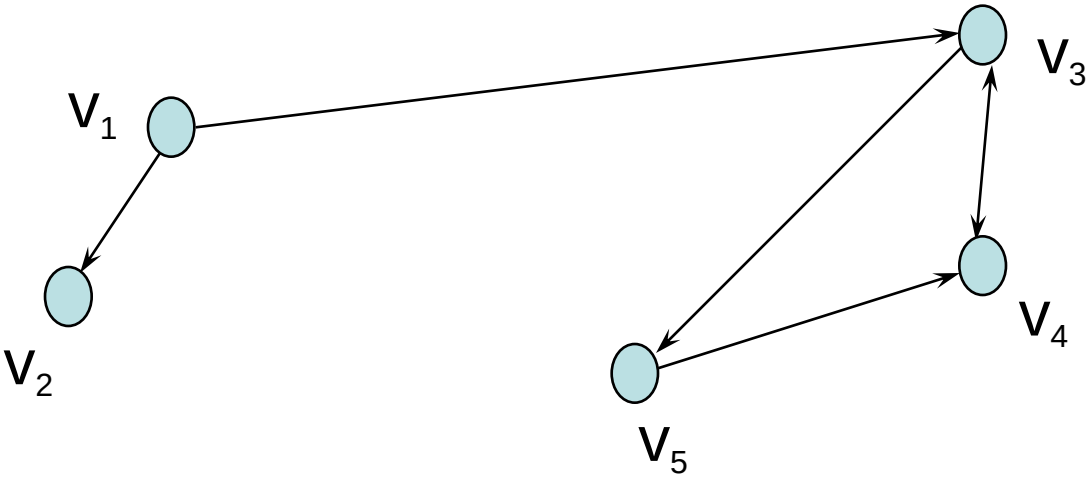


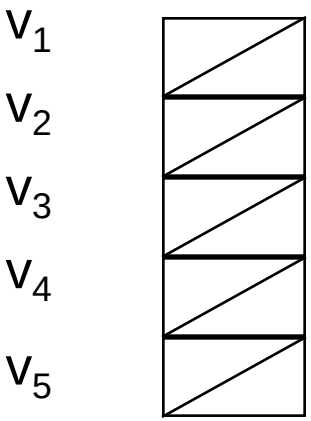


Beispiel: Adj. Liste - gerichteter Graph (D1)

universität wien

Gerichteter Graph
(Liste)

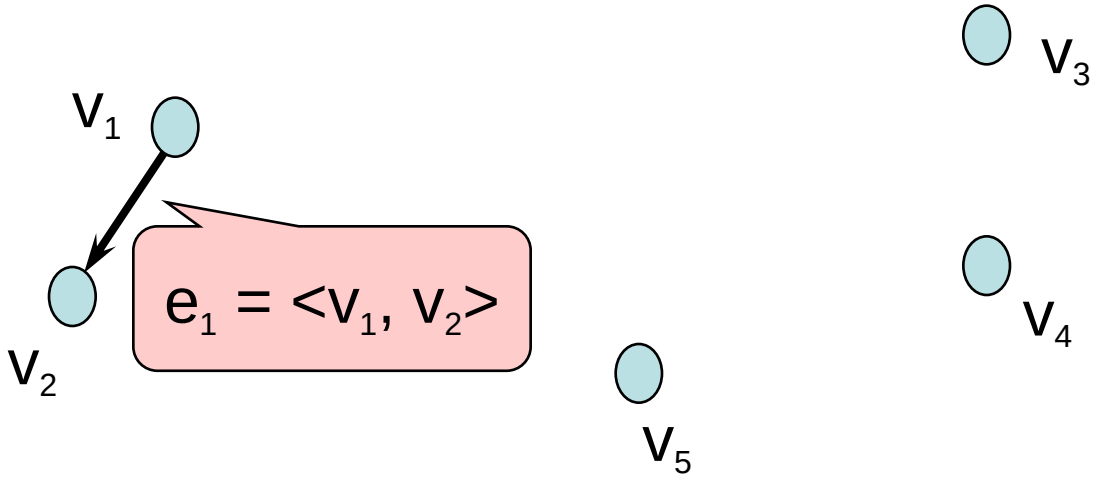


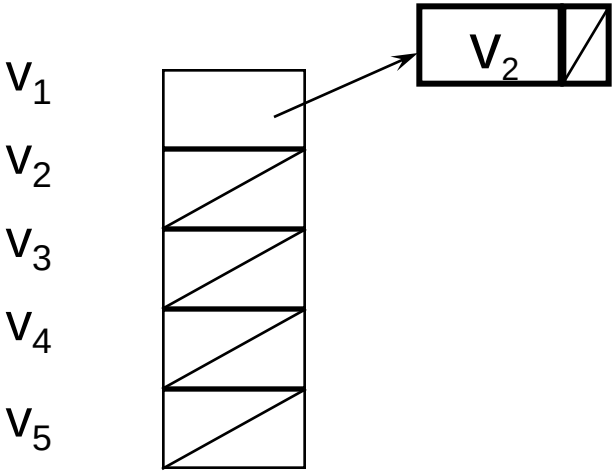


Beispiel: Adj. Liste - gerichteter Graph (D2)

universität wien

Gerichteter Graph

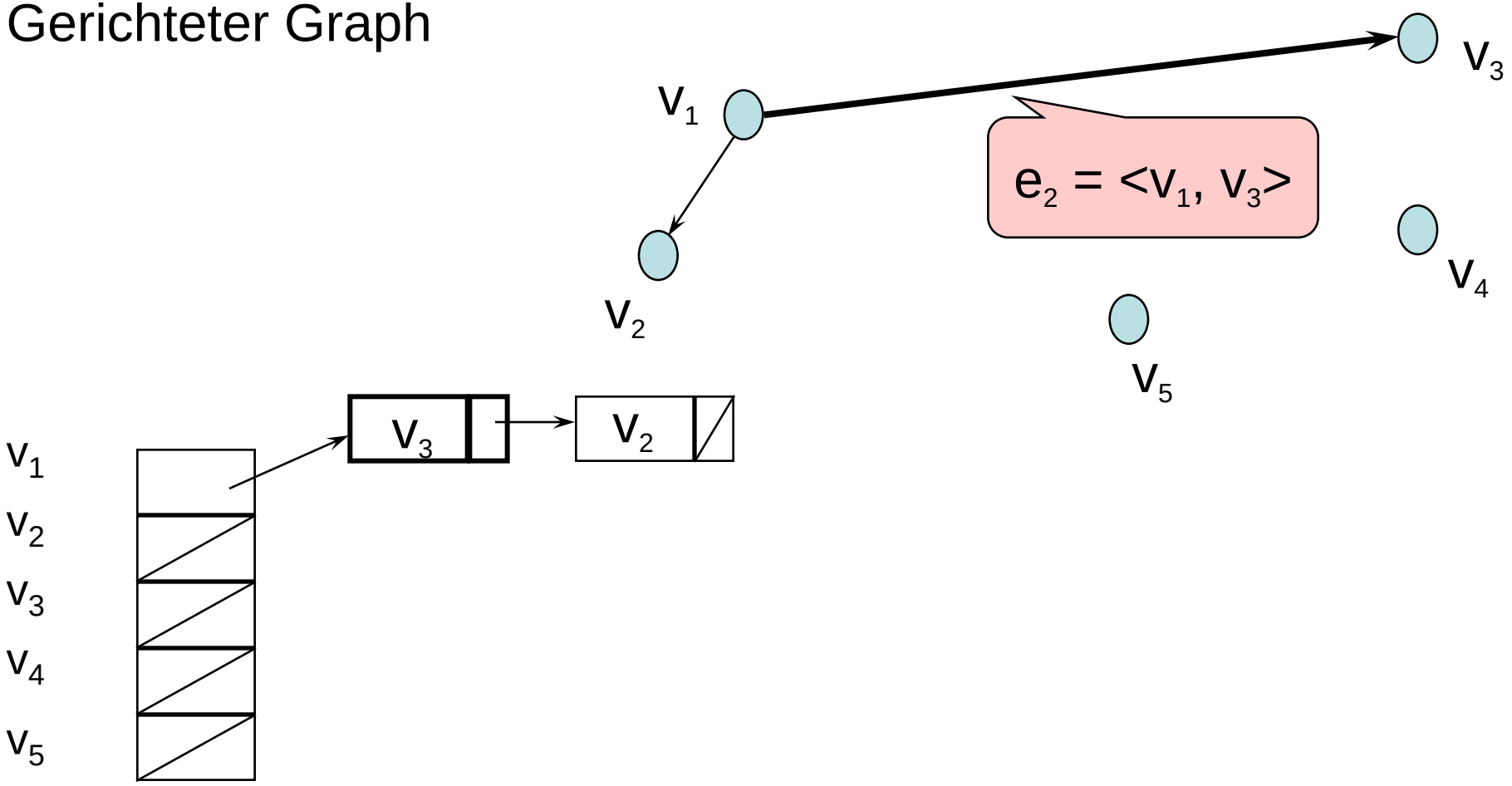




Beispiel: Adj. Liste - gerichteter Graph (D3)

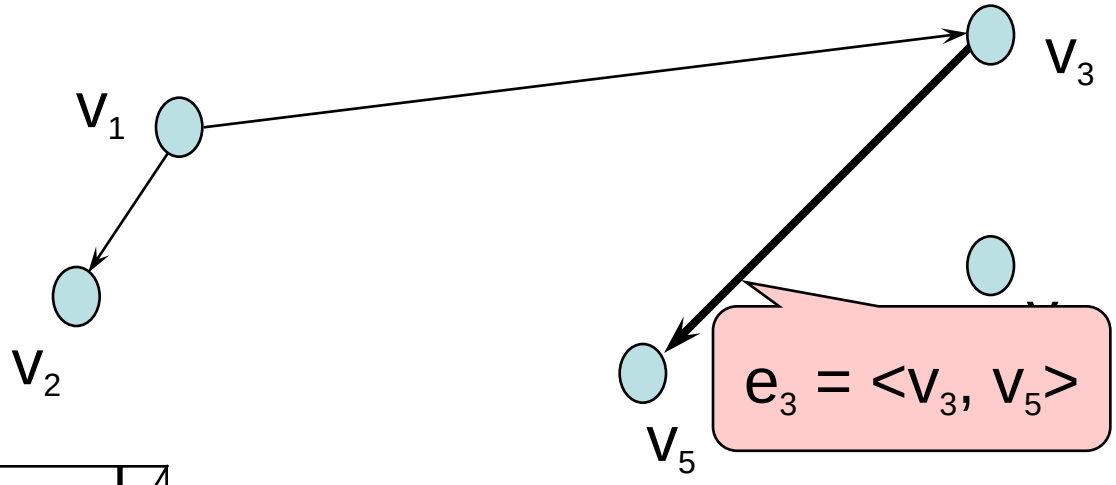
universität wien

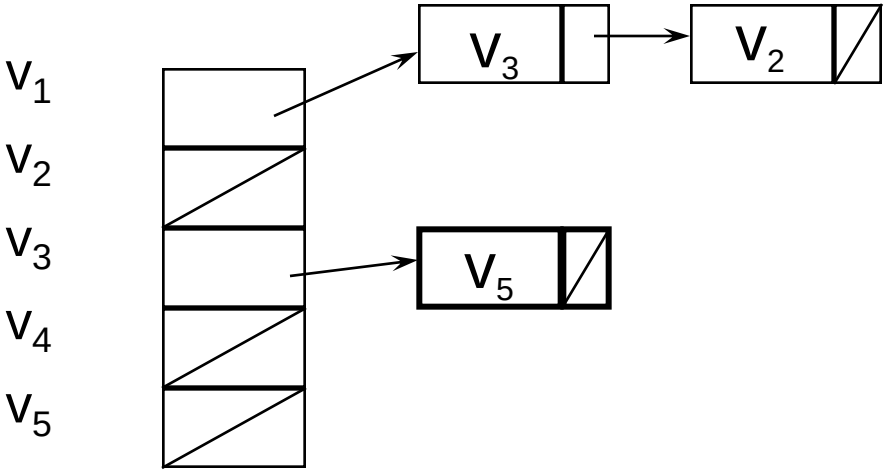
Gerichteter Graph



Beispiel: Adj. Liste - gerichteter Graph (D4)

Gerichteter Graph

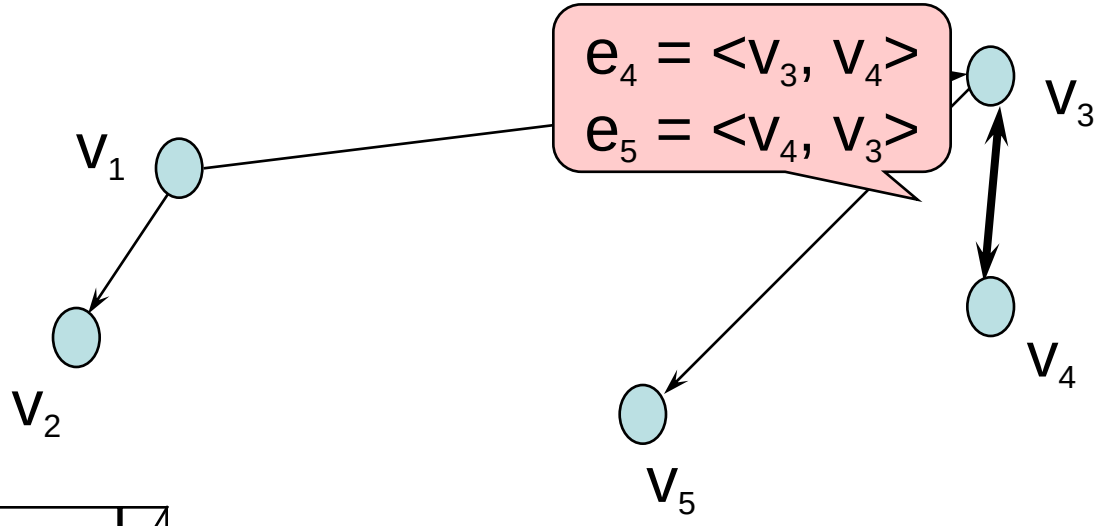


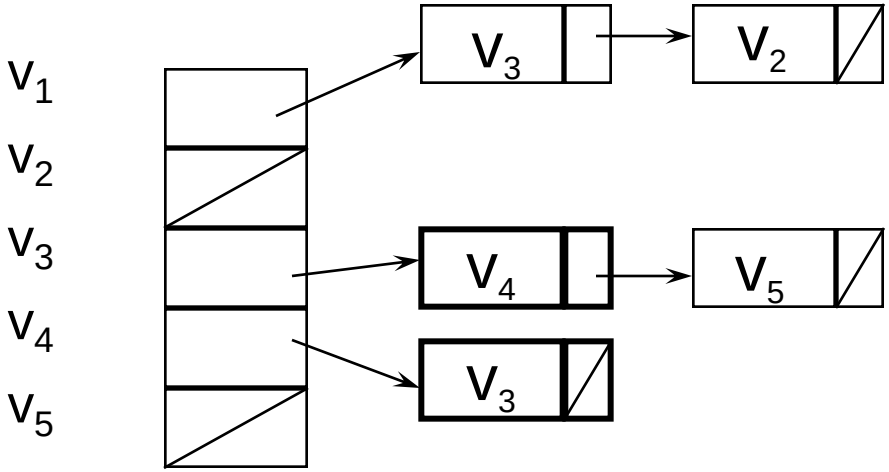


Beispiel: Adj. Liste - gerichteter Graph (D5)

universität wien

Gerichteter Graph

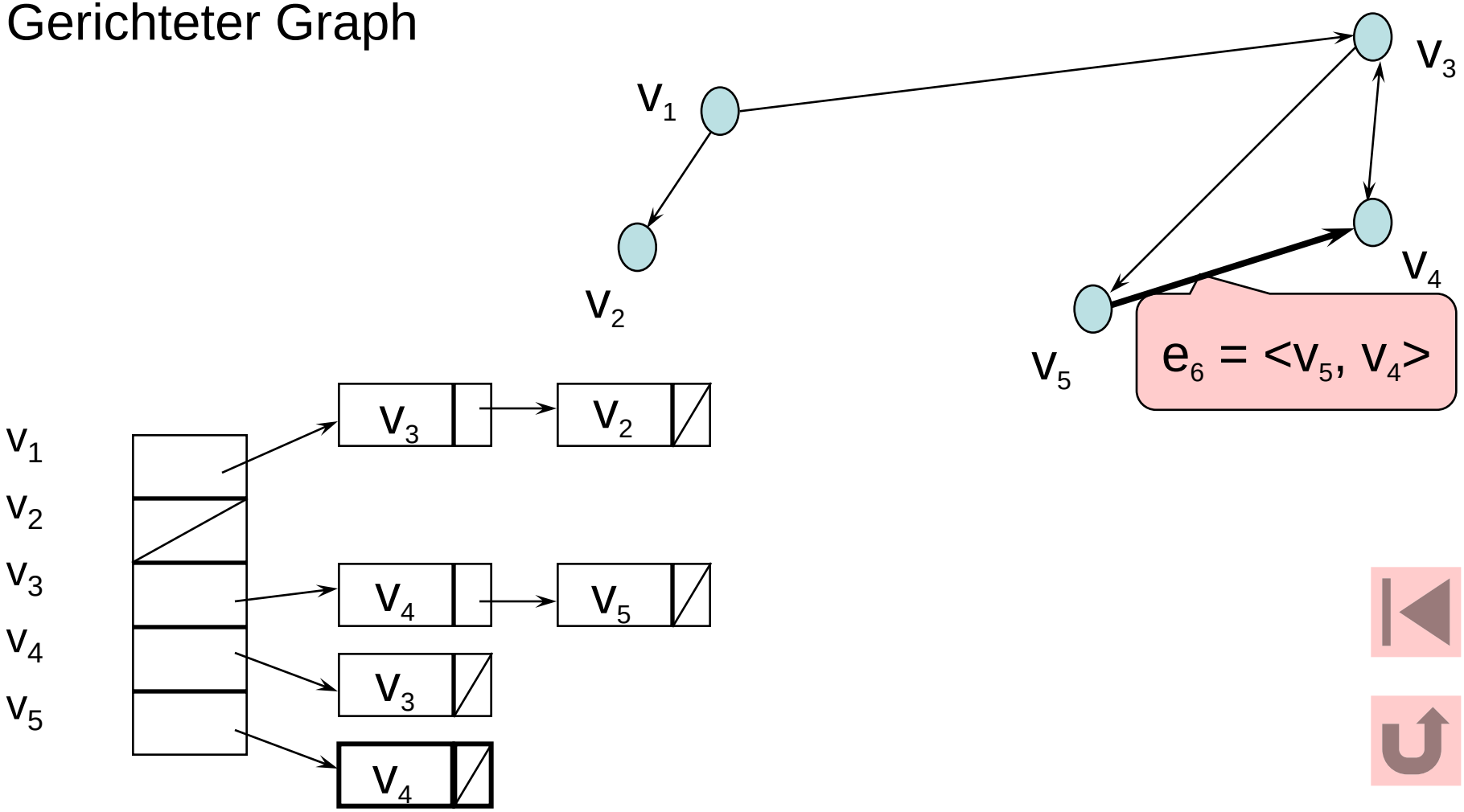




Beispiel: Adj. Liste - gerichteter Graph (D6)

universität wien

Gerichteter Graph



Eigenschaften

+ (linearer) Speicheraufwand O(|V|+|E|)
+ dfs gut, Rechenaufwand O(|V|+|E|)
- Überprüfung Adjazenzeigenschaft O(|V|)
- manche Algorithmen komplexer

6.4 Topologisches Sortieren

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

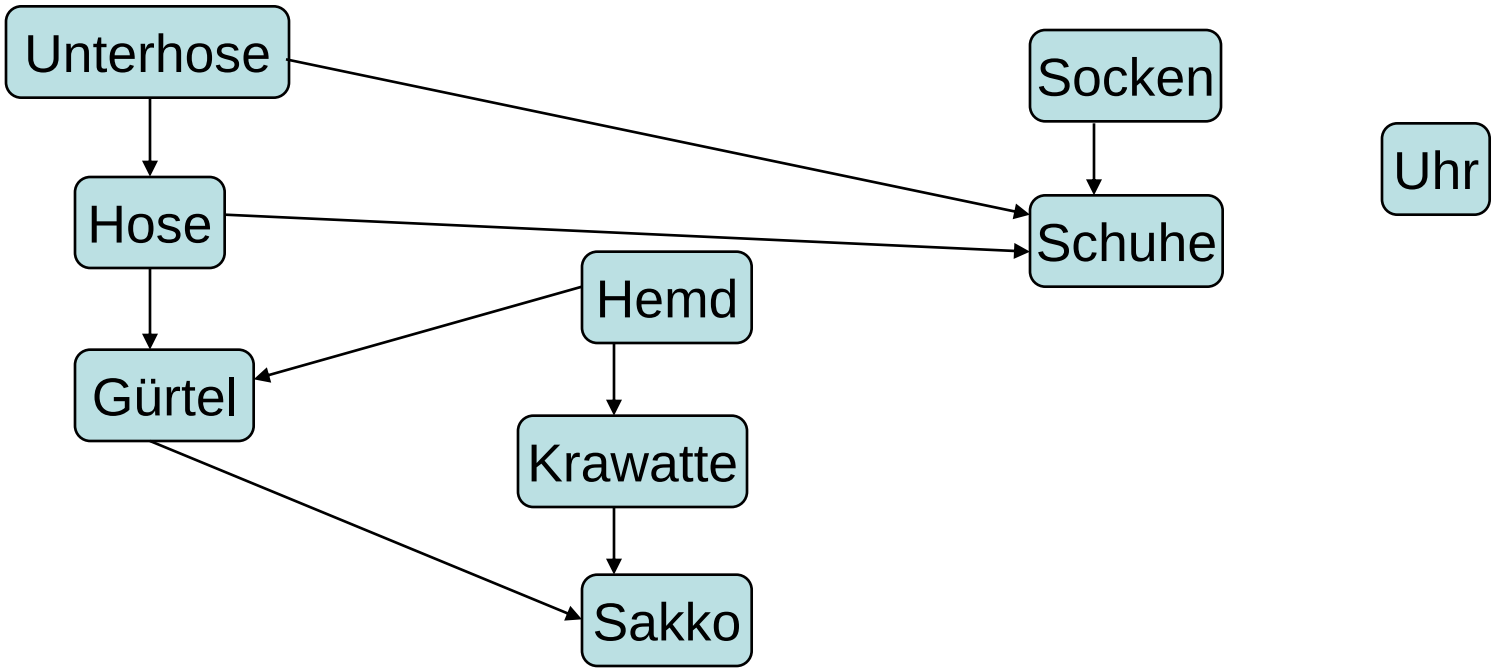
Beziehung: Kleidungsstück A muss vor B angezogen werden

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

Interpretation durch gerichteten Graphen

Binäre Beziehung zwischen Elementen ist gerichtete Kante zwischen entsprechenden Knoten

Prof. Bumstead



Topologische Ordnung eines DAG

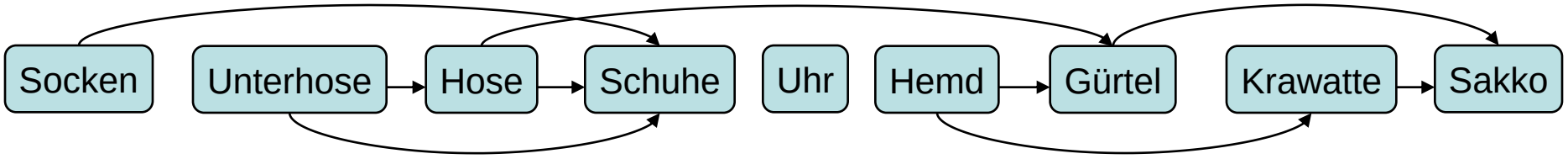
Eine ***Topologische Ordnung*** eines gerichteten azyklischen Graphs G (***directed acyclic graph, DAG***) ist eine lineare Anordnung aller Knoten, sodass u vor v in der Anordnung steht, wenn G eine Kante <u, v> enthält

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

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

Topologische Nummerierung impliziert topologische Ordnung

Topologische Ordnung: Professor Bumstead



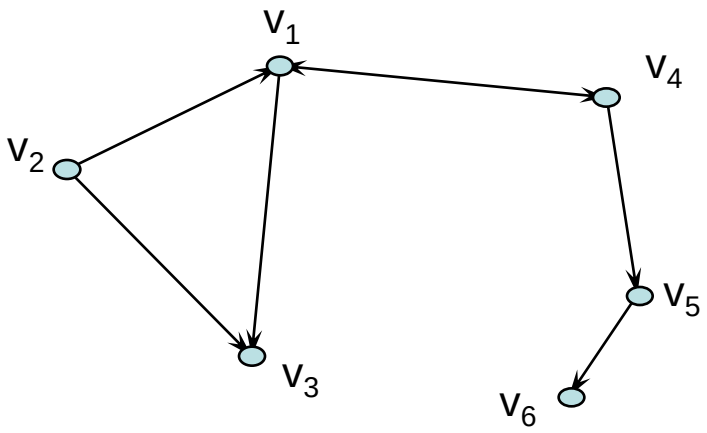
Eine mögliche Vorgangsweise

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 3 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 ldfNr um 1 und gib v Nummer lfdNr.
4. Lösche v aus dem Graphen
5. Weiter bei 2.
6. Falls es noch Knoten gibt, drucke “no topologic order exists”

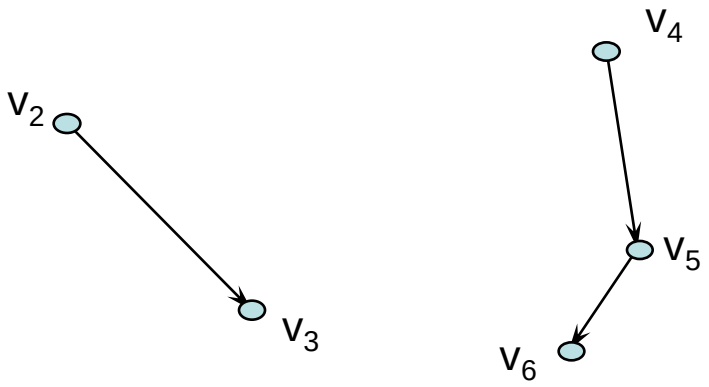
induzierter Teilgraph

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

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



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



Algorithmus topologisches Sortieren

```
lfdNr = 0;
while (∃ v ∈ G: indegree(v) = 0) {
  lfdNr = lfdNr + 1;
  N[v] = lfdNr;
  G = induzierter Teilgraph von V\{v}; // löscht v from G
}
if (V = {})
  G ist azyklisch; // N enthält die topologische Nummerierung für G
else
  G ist nicht azyklisch; // es existiert keine top. Ordnung
```

Implementierung mit Adjazenzmatrixdarstellung A

- Test am Anfang der While-Schleife dauert Zeit $O(n^2)$.
- Entfernen von einem Knoten dauert Zeit $O(n)$.
- Da die while-Schleife höchstens n Mal ausgeführt wird, ist die insgesamte Laufzeit $O(n^3)$.
- Kann mit zusätzlicher Datenstruktur auf $O(n^2)$ insgesamte Laufzeit verbessert werden:
  - Array IND, der für jeden Knoten seinen Eingangsgrad abspeichert
    - Initialisierung von IND vor der while-Schleife: Berechne IND mit der Adjazenzmatrix in Zeit $O(n^2)$, indem man die Einträge der Spalte von jedem Knoten w summiert und sie in IND[w] speichert.
    - Test am Anfang der While-Schleife läuft über IND und sucht nach dem ersten Knoten w mit IND[w] = 0. Daher dauert der Test Zeit $O(n)$.
    - 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 den Eingangsgrad von w um 1. Außerdem setze IND[v] auf -1 (um zu zeigen, dass v nicht mehr existiert). Das dauert Zeit $O(n)$

  $\Rightarrow$ Die insgesamte Laufzeit ist $O(n^2)$.

Algorithmus topologisches Sortieren

Implementierung mit Adjazenzlistendarstellung

Version 1: nur Adjazenzlistendarstellung von G

- Test am Anfang der While-Schleife dauert Zeit $O(n^2)$: Initialisiere einen Array IND, der für jeden Knoten seinen Eingangsgrad abspeichert, mit 0. Traversiere all Kanten, dh. alle Adjazenzlisten und für jede Kante <u,v> erhöhe IND[u] um 1. Das dauert Zeit $O(n^2)$,
- Entfernen von einem Knoten v dauert Zeit $O(n)$, da nur adjliste[v] auf NIL gesetzt wird.
- Da die while-Schleife höchstens n Mal ausgeführt wird, ist die insgesamte Laufzeit $O(n^3)$.

Algorithmus topologisches Sortieren

Implementierung mit Adjazenzlistendarstellung

Version 2: Adjazenzlistendarstellung von G und Adjazenzlistendarstellung vom umgekehrten Graphen G' (siehe Beweis von Lemma 3) mit Querverweisen zwischen den beiden Einträgen einer Kante in den zwei Adjazenzlistendarstellungen

- Test am Anfang der While-Schleife dauert Zeit $O(n)$: Finde einen Knoten, dessen Adjazenzliste in G' leer ist
- Entfernen von einem Knoten v dauert Zeit $O(n)$: Lösche jede Kante in , adjliste[v] aus der Adjazenzlistendarstellung von G'.Setze ein Eintrag von v im Array der Adjazenzlistendarstellung von G' auf -1 (um zu zeigen, dass v nicht mehr im Graphen ist.)
- Da die while-Schleife höchstens n Mal ausgeführt wird, ist die insgesamte Laufzeit $O(n^2)$.

6.5 Traversieren eines Graphen

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?

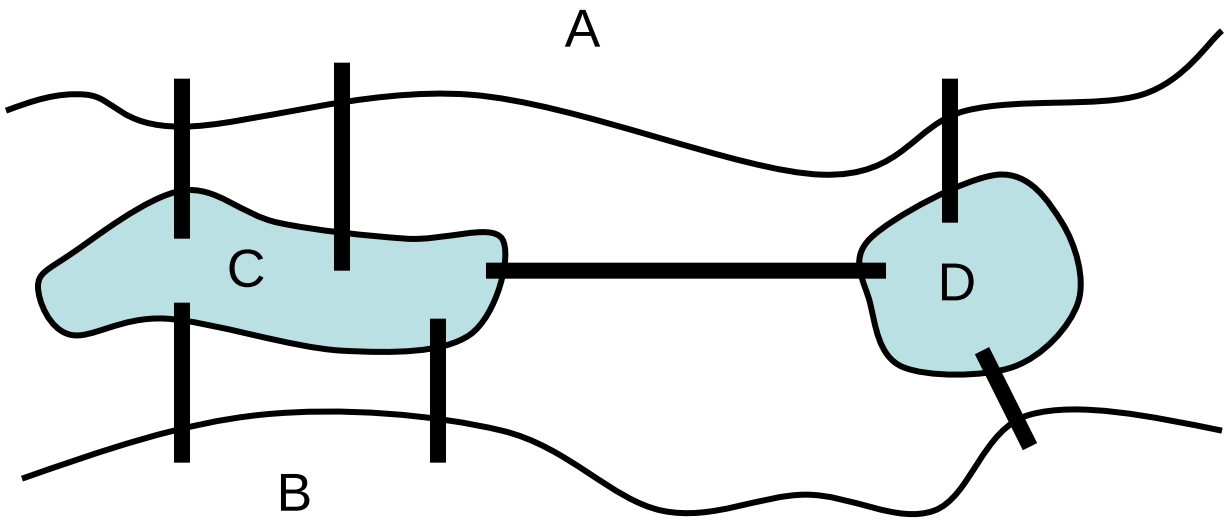
Finde alle Komponenten?

...

Das „Königsberger Brücken“ Problem

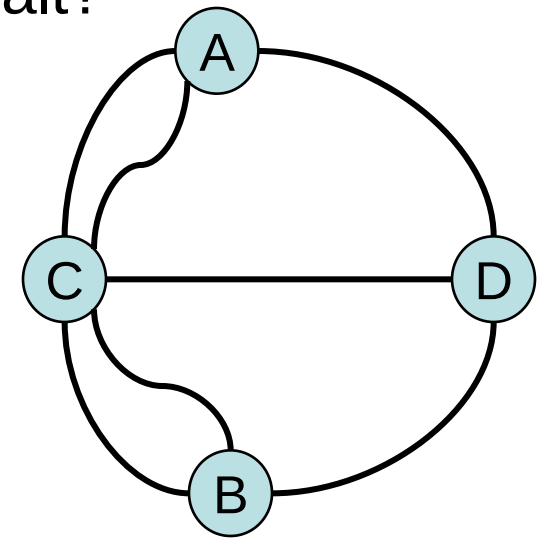
Leonhard Euler, 1736

Die Stadt Königsberg (Kaliningrad) liegt an den Ufern und auf 2 Inseln des Flusses Prigel 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 eine Kantenfolge im Graphen, die alle Kanten genau einmal enthält?



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

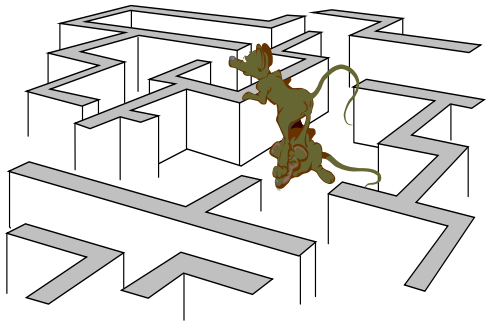
Solche Graphen werden Eulersche Graphen genannt.

Für Kaliningrad gilt dies offensichtlich nicht.

Eulersche Graphen werden als das erste gelöste Problem der Graphentheorie angesehen

6.5.1 Depth-first Search (DFS)

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 markierten Kreuzung oder zu einer Kreuzung ohne unbesuchte Wegalternativen über einen noch nicht beschrittenen Weg, gehen wir diesen Weg wieder zurück bis zur ersten Kreuzung mit unbesuchter Wegalternative. Dies ist der nächste, noch nicht besuchte Weg.

DFS Ansätze

Analogie

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

2 Ansätze

- Rekursiv ohne explizite Datenstruktur
- Iterativ (nicht-rekursiv) mit Stack

Kann auf gerichteten und ungerichteten Graphen angewandt werden. Hier: ungerichtet

6.5.1.1 Rekursiver Ansatz

Rekursiver Algorithmus

zu besuchende Knoten werden (automatisch) am Rekursionsstack vermerkt.

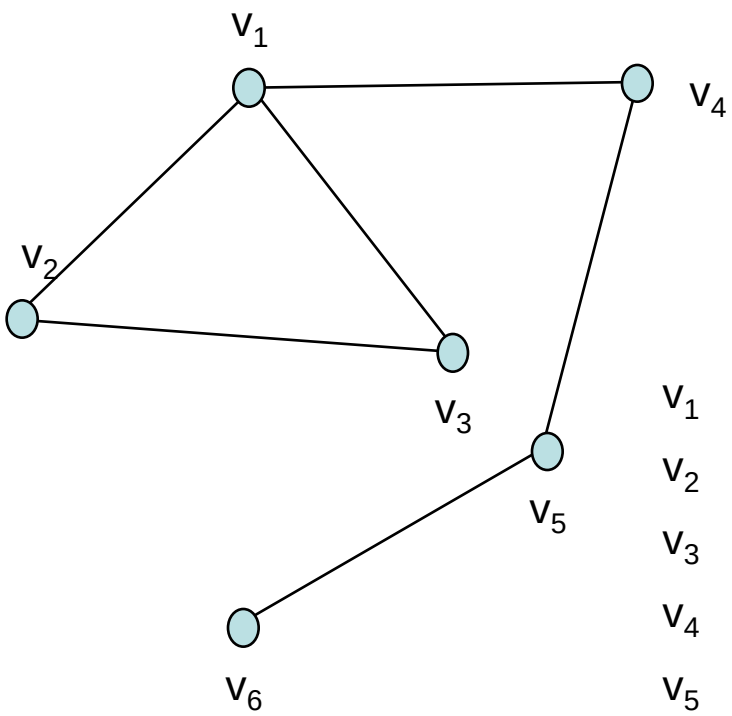
“Zuerst in die Tiefe , danach in die Breite gehen”

```
void besuche-dfs ( Knoten x ) {
  markiere Knoten x mit ‘besucht’;
  for ( alle zu x adjazente nicht besuchte Knoten v )
    besuche-dfs ( v );
```

Rekursiver Ansatz

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

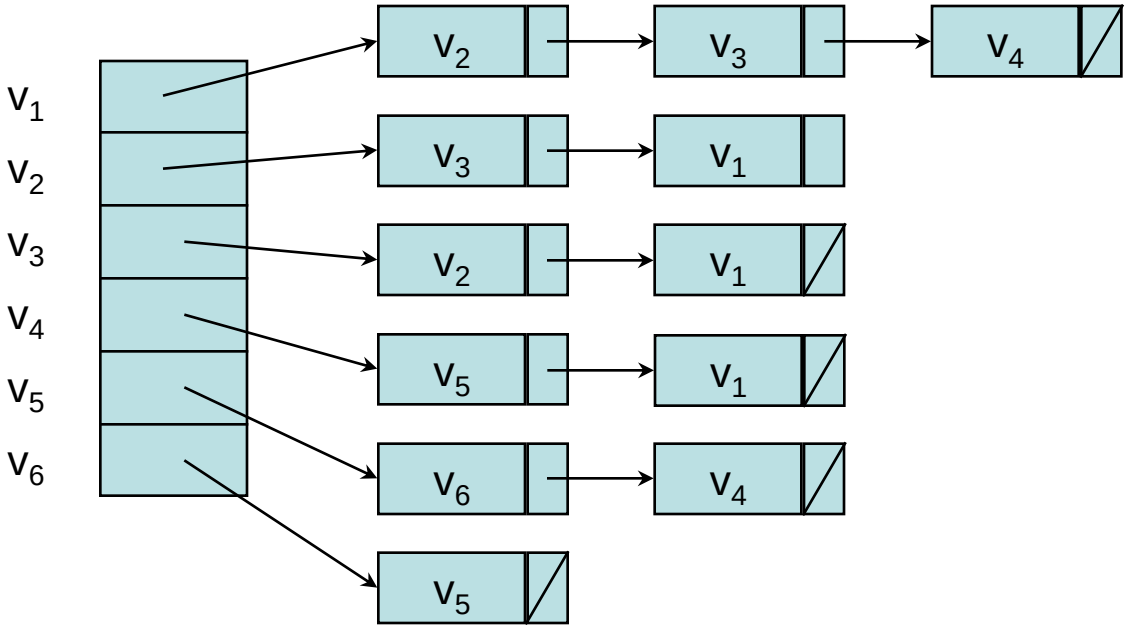
Beispiel, dfs



Markierung

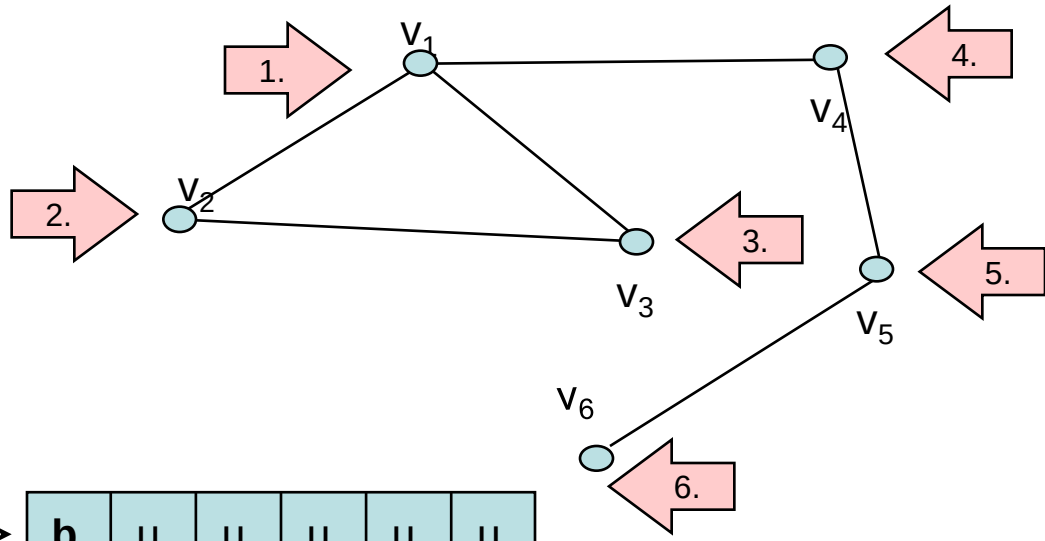
v_1	v_2	v_3	v_4	v_5	v_6
u	u	u	u	u	u

Adjazenzliste



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)	**b**	u	u	u	u	u
besuche-dfs(2)	b	**b**	u	u	u	u
besuche-dfs(3)	b	b	**b**	u	u	u
besuche-dfs(4)	b	b	b	**b**	u	u
besuche-dfs(5)	b	b	b	b	**b**	u
besuche-dfs(6)	b	b	b	b	b	**b**

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

C++ - Code für rekursiven Ansatz

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

6.5.1.2 Iterativer dfs Ansatz

Iterativer Ansatz

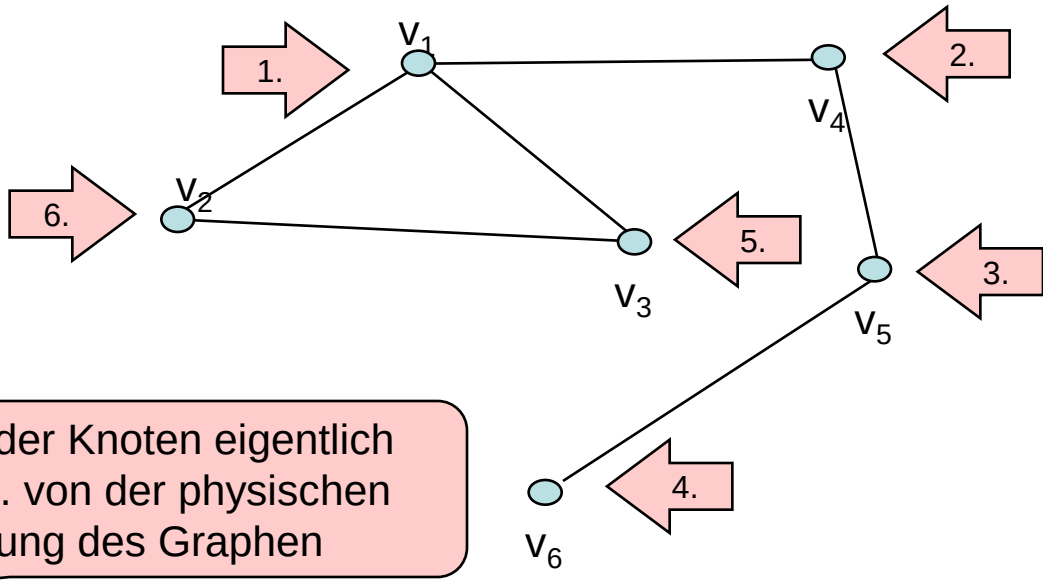
zu besuchende Knoten werden in einer Stack Datenstruktur gespeichert, nicht im (automatischen) Rekursionsstack

```
void besuche-dfs(Knoten x) {
  stelle (Push) Knoten x in Stack;
  markiere Knoten x 'besucht';
  while(Stack nicht leer) {
    hole (Pop) letzten Knoten y von der Stack;
    for(alle zu y adjazente nicht besuchte Knoten v) {
      stelle (Push) v in die Stack;
      markiere Knoten v 'besucht';
    }  // for
  }  // while
}
```

Iterativer Ansatz

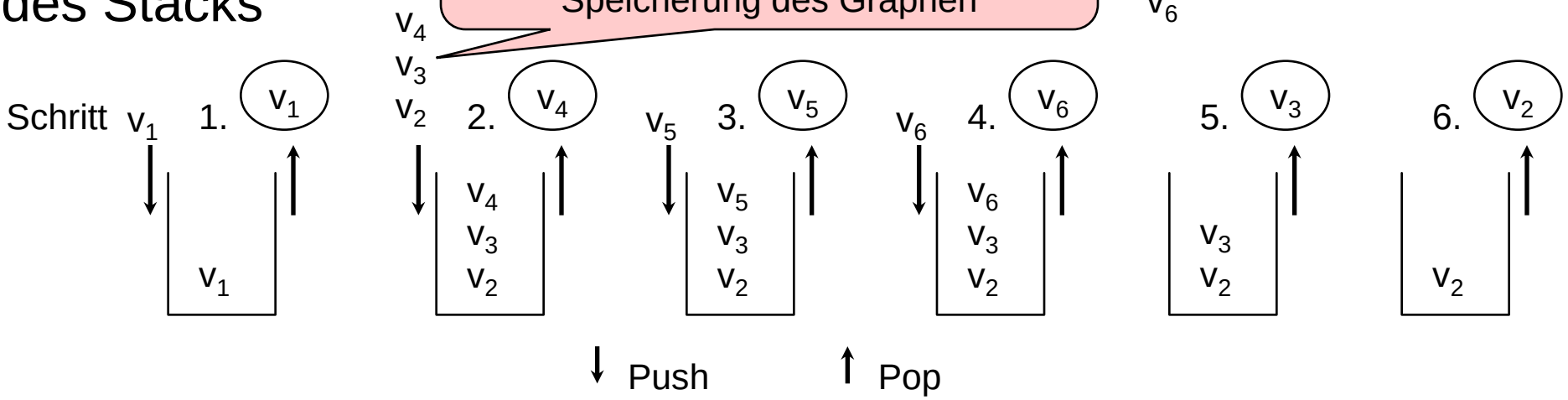
dfs-Traversierung eines Graphen

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



Verhalten des Stacks

Reihenfolge der Knoten eigentlich beliebig, abh. von der physischen Speicherung des Graphen



Beispiel, dfs (2)

besuche-dfs(1)

Push(1) Pop(1)	**b**	u	u	u	u	u
Push(2)	b	**b**	u	u	u	u
Push(3)	b	b	**b**	u	u	u
Push(4) Pop(4)	b	b	b	**b**	u	u
Push(5) Pop(5)	b	b	b	b	**b**	u
Push(6) Pop(6)	b	b	b	b	b	**b**
Pop(3) Pop(2)						

Die Ergebnisse der zwei Methoden von dfs sind identisch, doch die Knoten werden zu unterschiedlichen Zeitpunkten als besucht markiert:

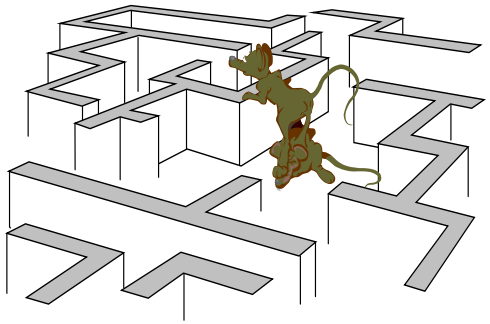
- Rekursiv: Als besucht markiert, wenn der Knoten besucht wird
- Iterative: Als besucht markiert, wenn der Knoten bekannt wird

Nicht-rekursiver C++ - Code

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

Aufwand (Laufzeitanalyse) beide Ansätze



Methode traversiere ohne besuche-dfs:

O(|V|) Schritte

Methode besuche-dfs:

Mit Adjazenzliste:

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

Jeder Knoten wird einmal besucht:

$$\text{Für alle Knoten}: O(\sum_t (1+|\text{adjliste}[t]|)) = O(\sum_t 1 + \sum_t |\text{adjliste}[t]|) = O(|V| + |E|)$$

Total: O(|V| + |E|)

Mit Adjazenzmatrix:

Für jeden Knoten t: O(|V|) zum Finden der Nachfolgerknoten

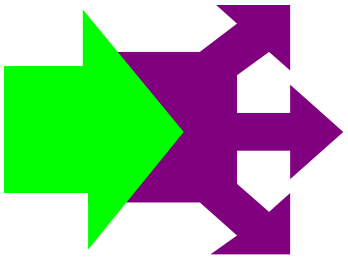
Jeder Knoten wird einmal besucht: Für alle Knoten: $O(|V|^2)$

Total: $O(|V| + |V|^2) = O(|V|^2)$

Da $|E| \leq O(|V|^2)$ sind Adjazenzlisten im Allgemeinen schneller

6.5.2 Breadth-first Search

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

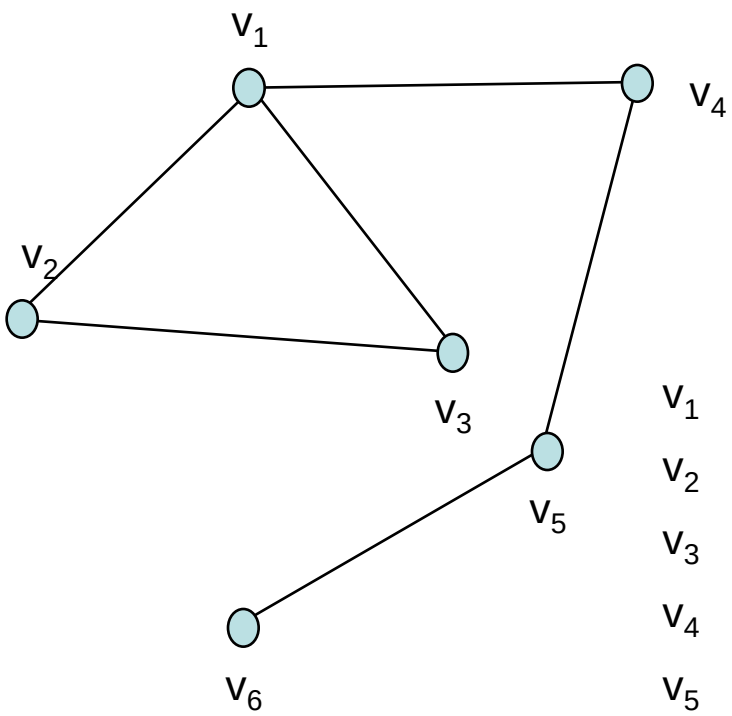
zu besuchende Knoten werden in einer Queue gemerkt

“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 adjazente nicht besuchte Knoten v) {
      stelle (Enqueue) v in die Queue;
      markiere Knoten v ‘besucht’;
    }  // for
  }  // while
}
```

Iterativer Ansatz

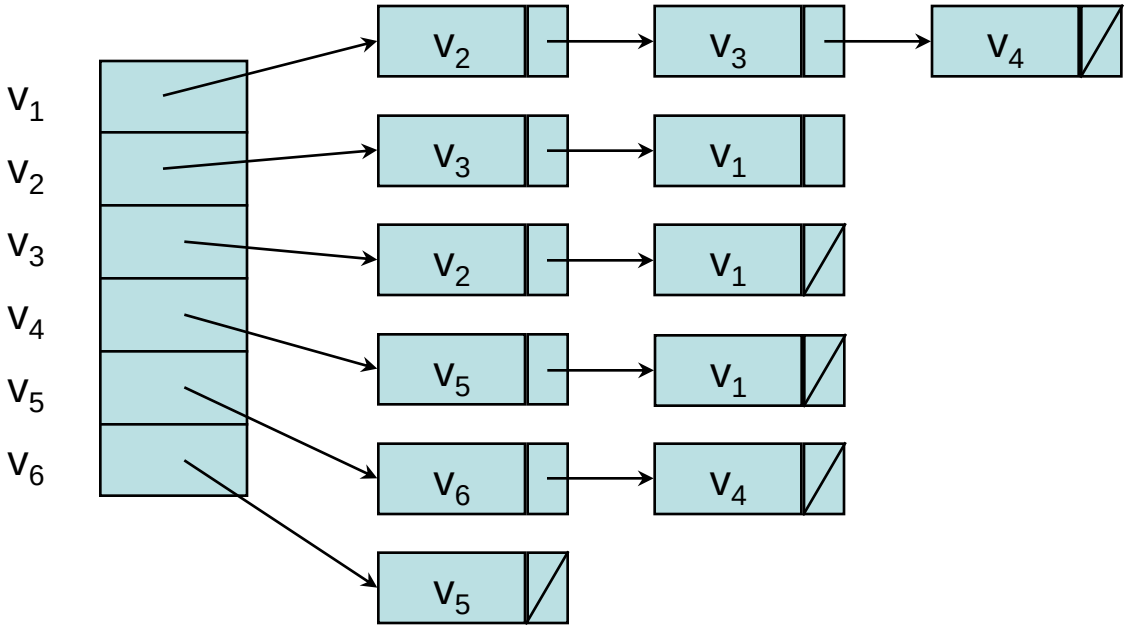
Beispiel 1, bfs



Markierung

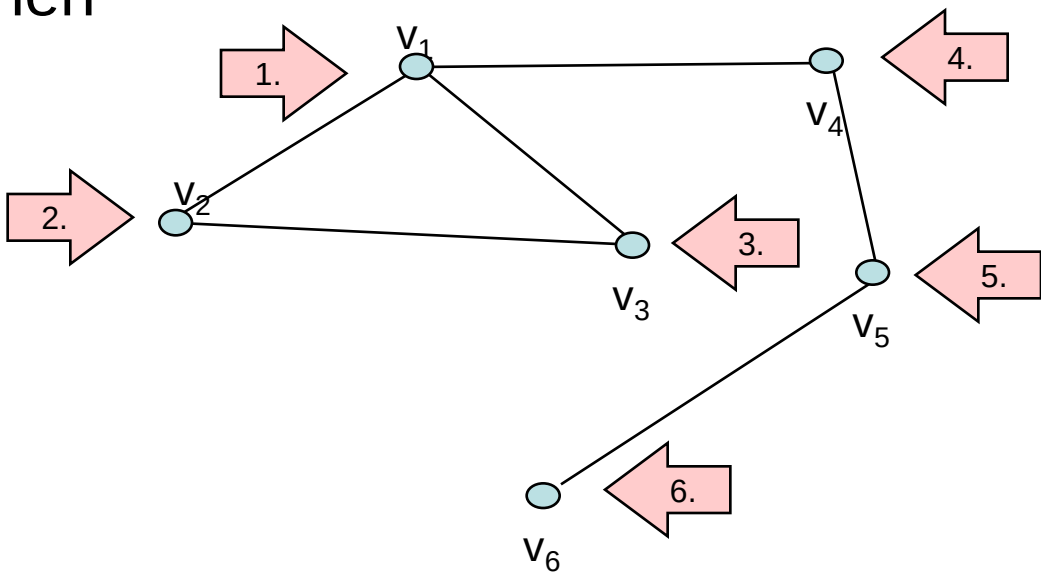
v_1	v_2	v_3	v_4	v_5	v_6
u	u	u	u	u	u

Adjazenzliste

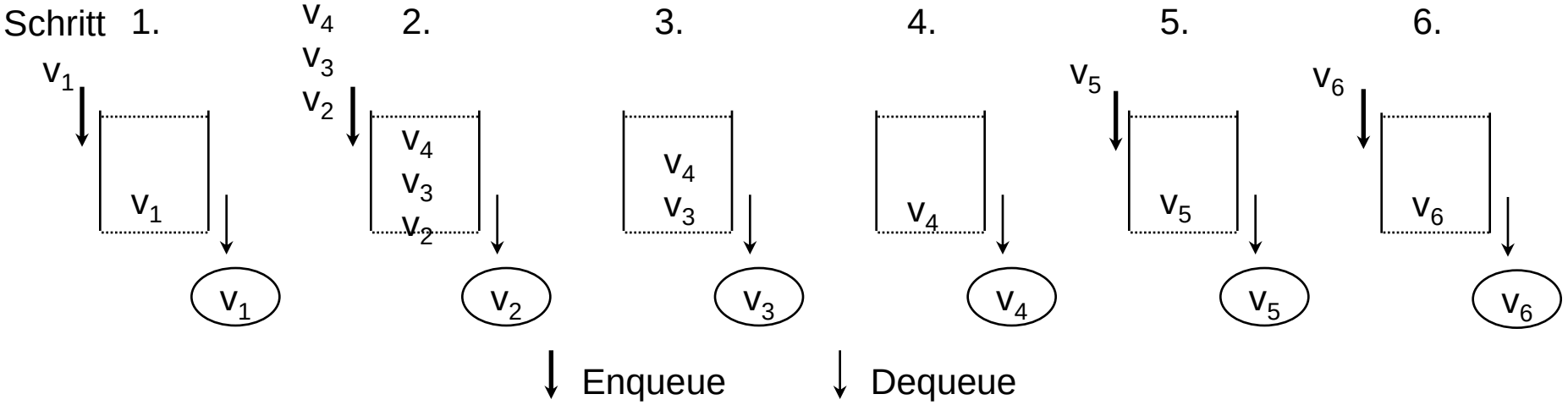


bfs-Traversierung eines Graphen

- Ausgehend vom Knoten v_1 wird der Graph mit dem bfs-Ansatz traversiert
- Besuchte Knoten werden in einer Queue vermerkt

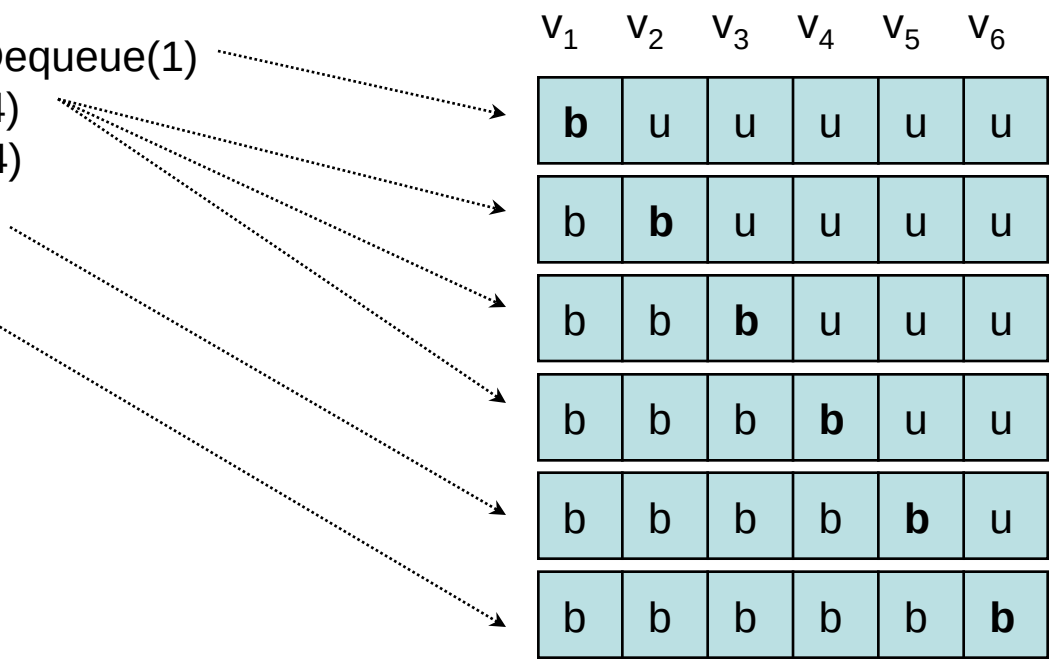


Verhalten der Queue

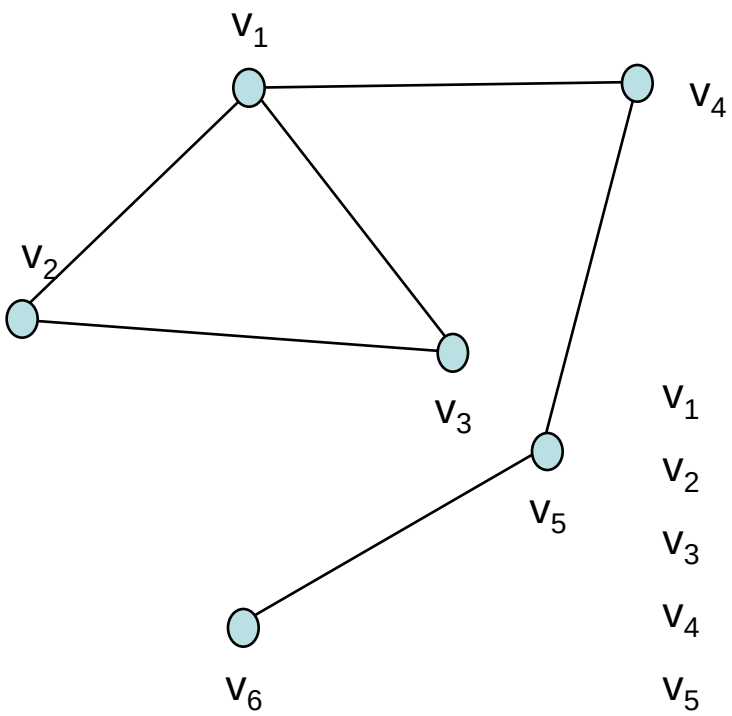


Beispiel 1, bfs (2)

besuche-bfs(1)
- Enqueue(1), Dequeue(1)
- Enqueue(2,3,4)
- Dequeue(2,3,4)
- Enqueue(5)
- Dequeue(5)
- Enqueue(6)
- Dequeue(6)



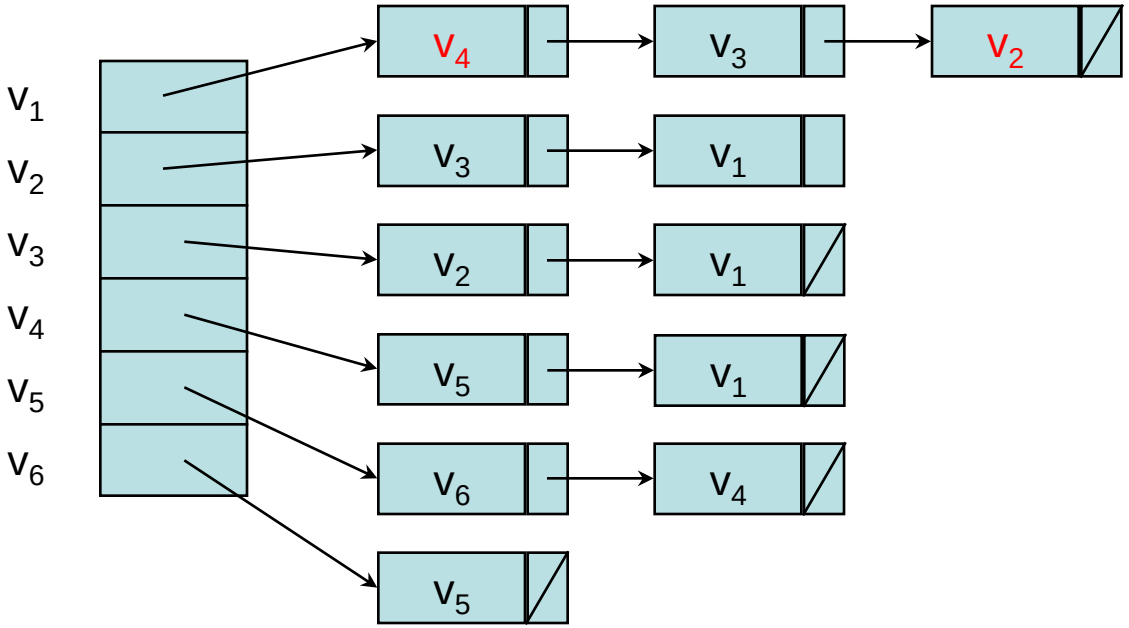
Beispiel 2, bfs



Markierung

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

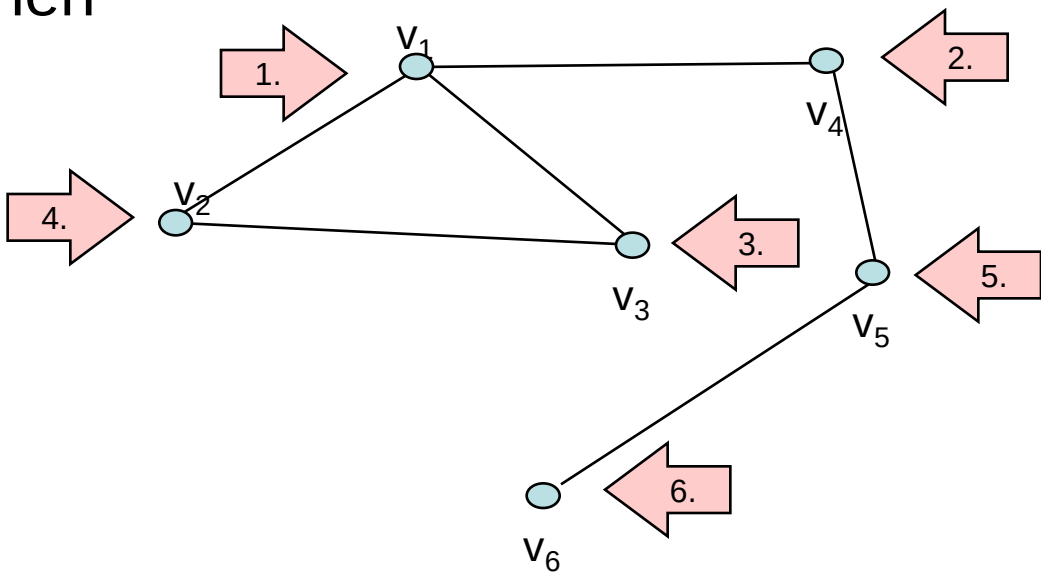
Adjazenzliste



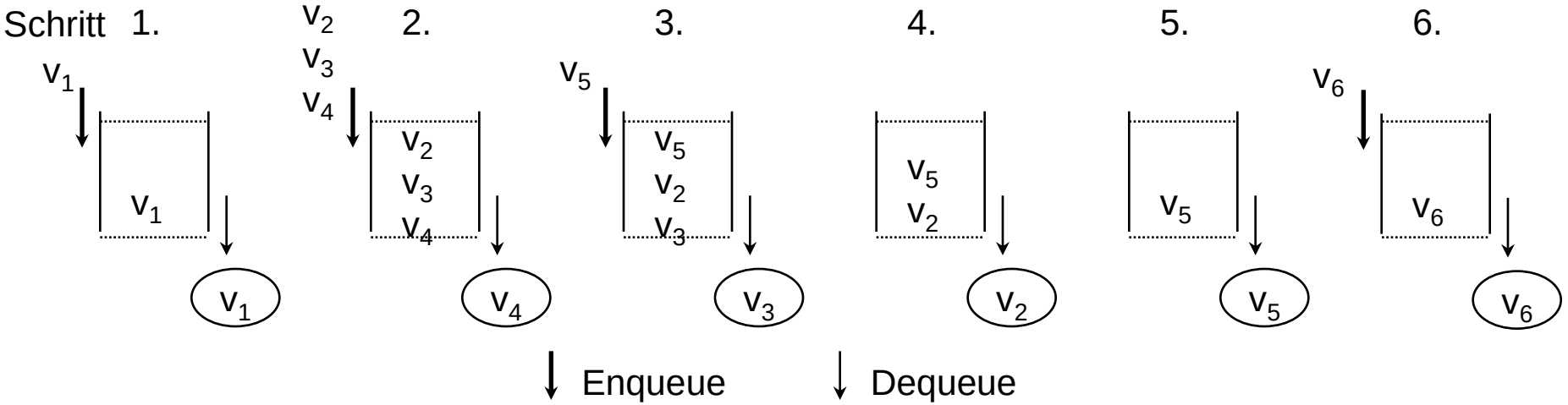
Methode bfs, Beispiel 2

bfs-Traversierung eines Graphen

- Ausgehend vom Knoten v_1 wird der Graph mit dem bfs-Ansatz traversiert
- Besuchte Knoten werden in einer Queue vermerkt

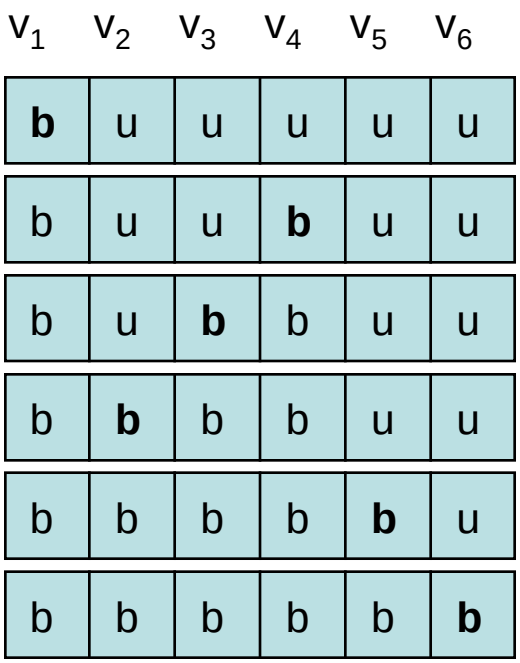


Verhalten der Queue



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



Reihenfolge der Knotenbesuche bei der bfs-Traversierung

Verschiedene Repräsentation des Graphs führt 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 …

C++ - Code

```
// 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) {
  node *t;
  queue.Enqueue(k);
  mark[k]= besucht;
  while(!queue.IsQueueEmpty()) {
    k = queue.Dequeue();
    for(t = adjliste[k]; t != NULL; t = t->next)
      if(mark[t->v] == unbesucht) {
        queue.Enqueue(t->v); mark[t->v] = besucht;
      }
    }
  }
}
```

Aufwand von dfs und bfs

Gleich!

Warum?

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

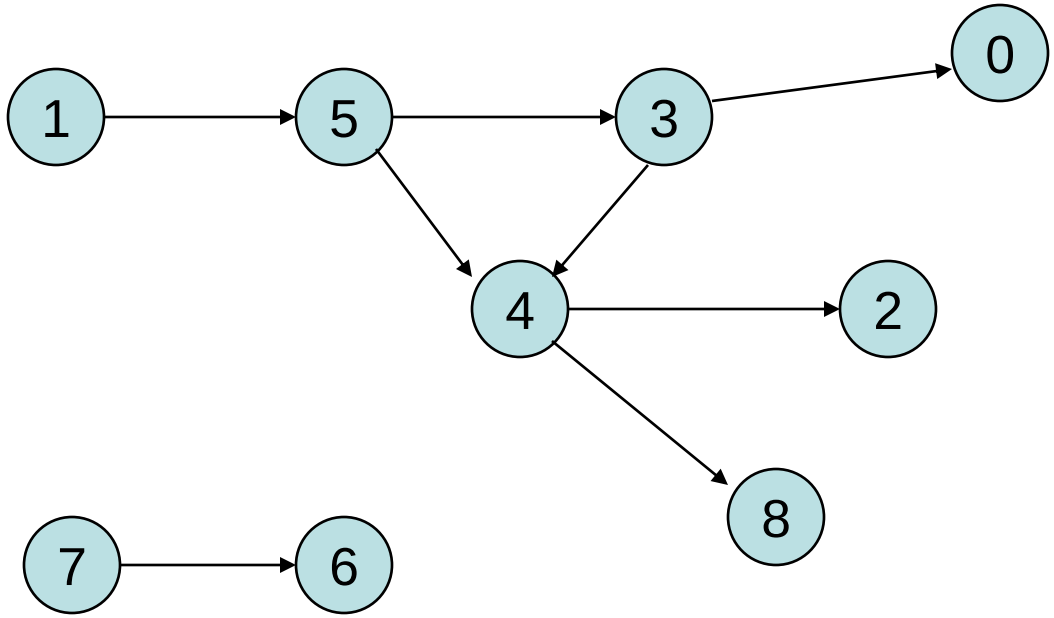
6.5.3 Genereller iterativer Ansatz

- 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

Unsystematische Traversierung

```
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 ∉ OpenList && v1 ∉ CloseList)
        OpenList = OpenList + [v1];
    }
  }
}
```

Beispiel (F)



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?

Zusammenhangskomponente im ungerichteten Graphen

Erweiterung von **`Search`**

Feld zum Vermerken der Komponenten K[|V|]

```
int Vanz = |V|;
int Knum = 0;
for (int i = 0; i < Vanz; i++)
  if(K[i] == 0) { // i ist unbesucht
    Knum = Knum + 1;
    SearchAndMark(i, Knum);
  }

SearchAndMark(Knoten v; int Knum) {
```

neue Version von **`Search`** wobei

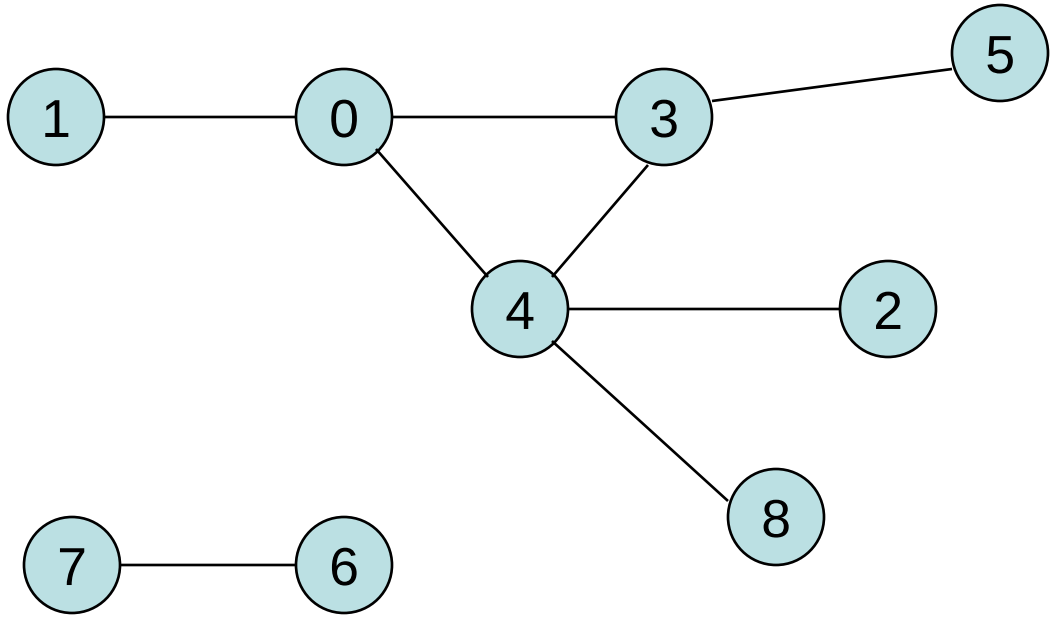
`process(Akt);` entspricht **`K[Akt] = Knum;`**

```
}
```

Zusammenhangskomponente

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

Beispiel (F)



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

Systematisierung der Traversierung

universität wien

Systematische Traversierung schon bekannt

- Tiefensuche dfs
- Breitensuche bfs

Unklarer Programmteil

- **`beliebigerKnotenAusOpenList`**

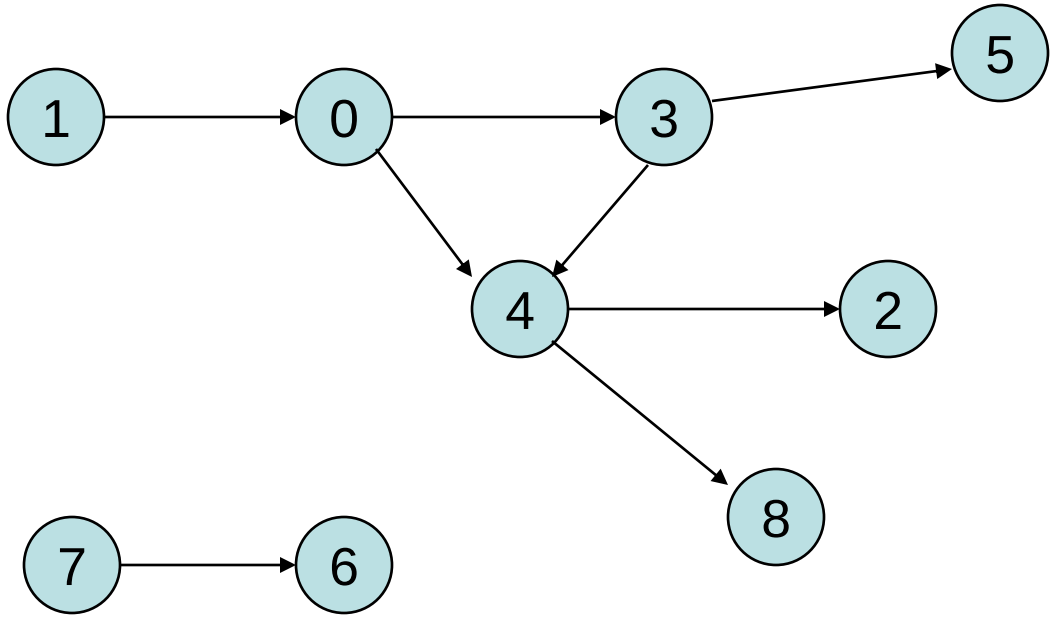
Systematisierung durch Organisation der Auswahl

- Aufbau der Liste
  - Position des einzufügenden Elements in der Liste
  - bfs: **`OpenList = OpenList + [v1];`** am Ende
  - dfs: **`OpenList = [v1] + OpenList;`** am Anfang
- Zugriff auf die OpenList
  - Zugriff: **`First(OpenList);`** Zugriff immer auf das erste Element

Systematische Traversierung

```
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 ∉ OpenList && v1 ∉ CloseList)
        dfs: OpenList = [v1] + OpenList;
        bfs: OpenList = OpenList + [v1];
    }
  }
}
```

Beispiel (F)



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

Noch ein kleines Problem

Systematisierung noch nicht ganz vollständig

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

Berechnung eines spannenden Baumes

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(alle zu Akt adjazente Knoten v1){
      if (v1 ∉ OpenList && v1 ∉ CloseList)
        dfs: OpenList = [v1] + OpenList;
        bfs: OpenList = OpenList + [v1];
        T = T + [[Akt, v1]]
    }
  }
}
```

Berechnung eines spannenden Baumes

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

- BFS: T heißt BFS-Baum
- DFS: T heißt DFS-Baum

Schichtenkonzept zur Problemlösung

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 2 Platz haben.

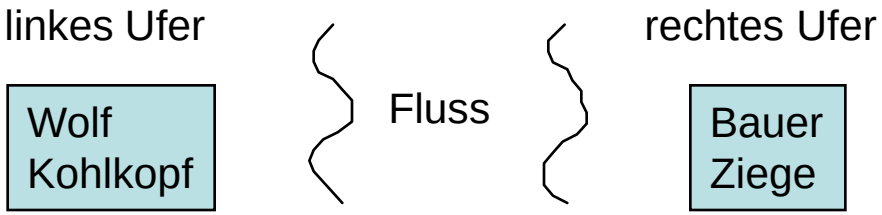
Weiters stellt sich das Problem, dass nur der Bauer rudern kann, und der Wolf mit der Ziege und die Ziege mit dem Kohlkopf nicht allein gelassen werden kann, da sonst der eine den anderen frisst.

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

Kodierung des Problems (1)

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,

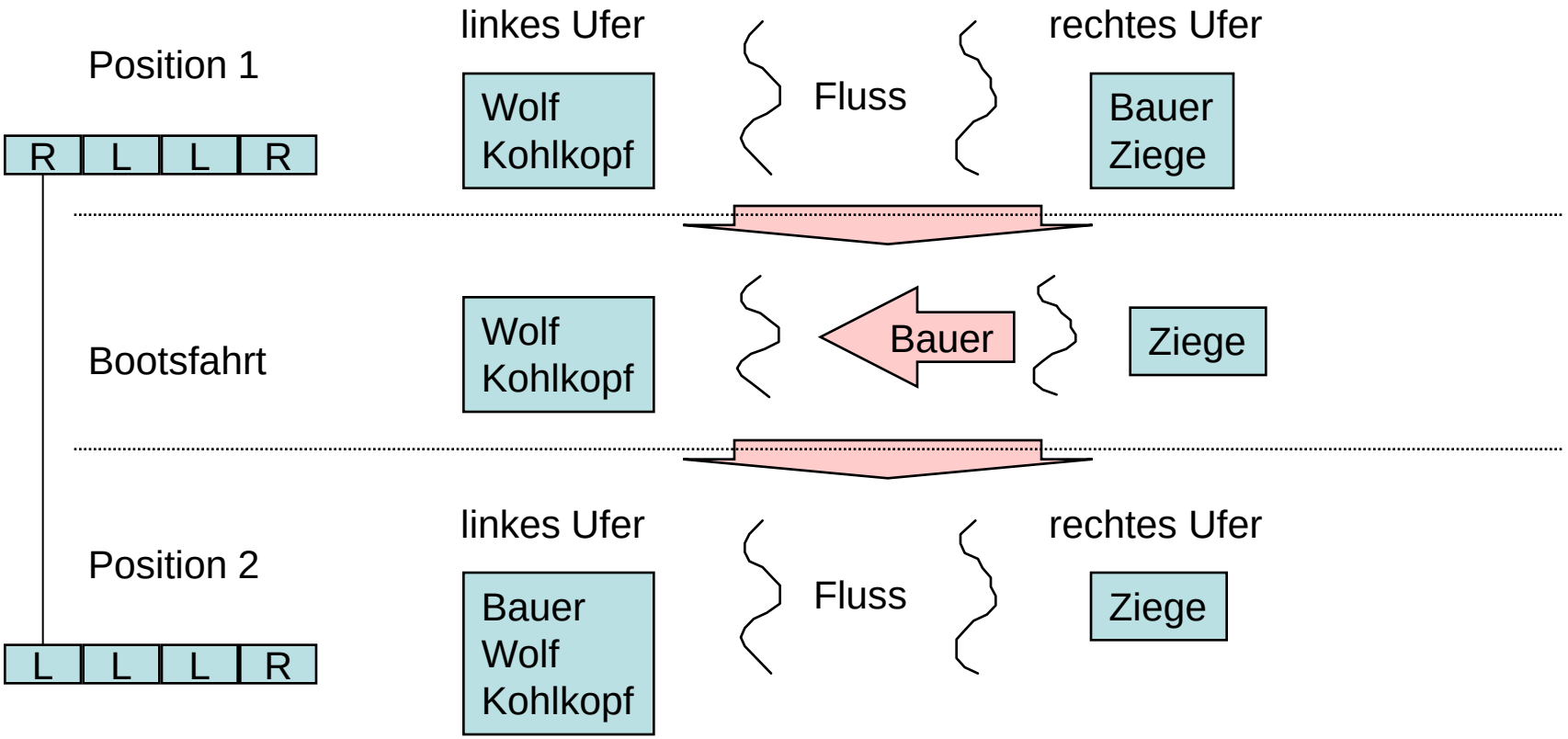


Dieses ist eine ‘sichere’ Position, da niemand gefressen wird.

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

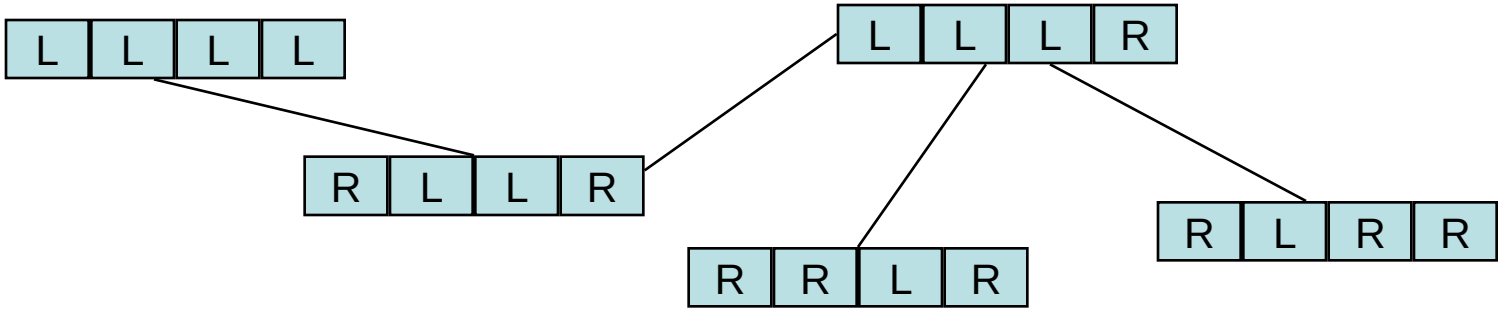
Kodierung des Problems (2)

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



Problemrepräsentation

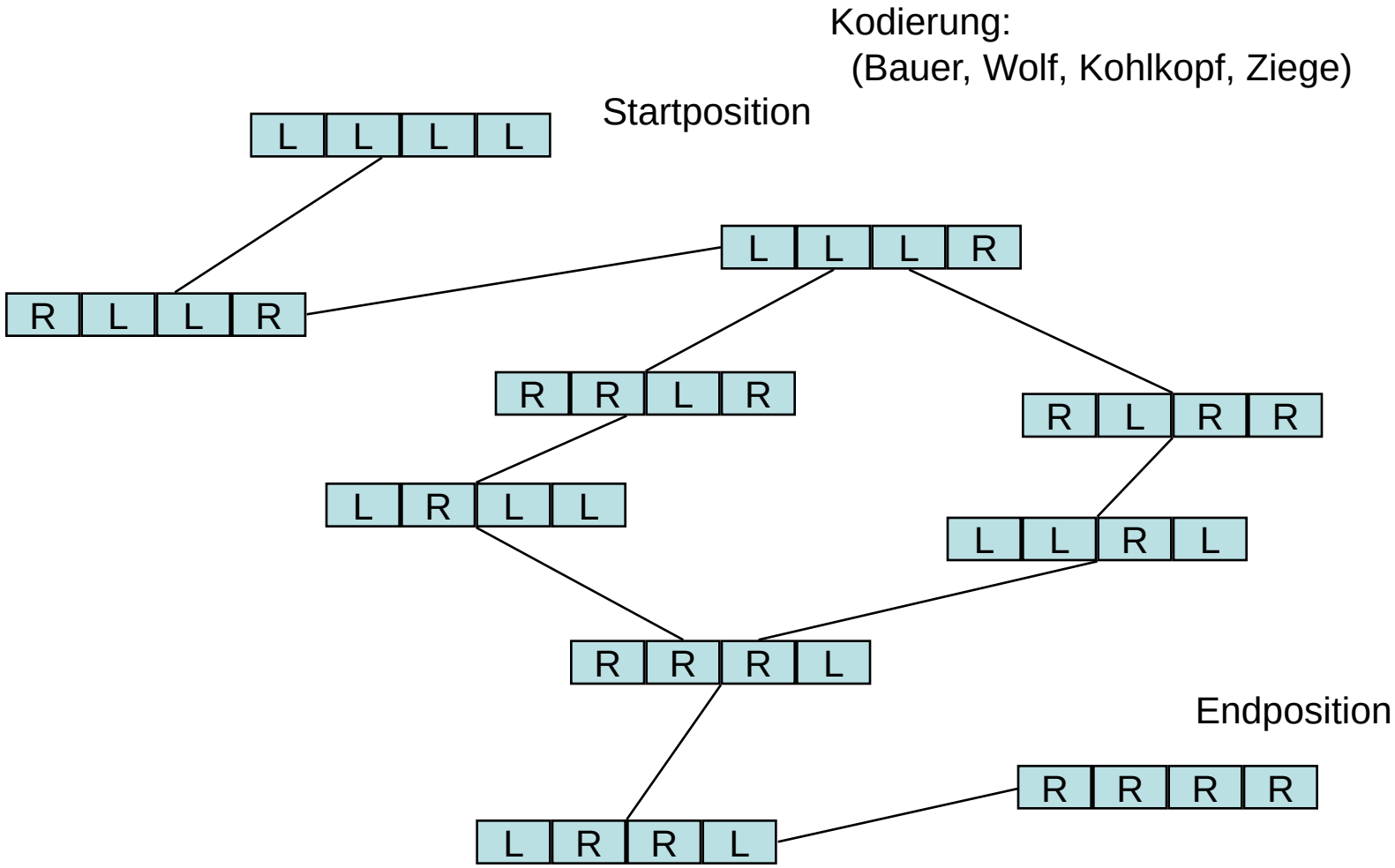
Der gesamte Problembereich lässt sich somit durch einen Graphen darstellen, wobei die Positionen durch die Knoten und die Bootsfahrten durch die Kanten repräsentiert werden, z.B. (Ausschnitt)



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)
Startposition
L L L L
R L L R
L L L R
R R L R
R L R R
L R L L
L L R L
R R R L
Endposition
R R R R
L R R L

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

- Die Position wird in einer integer Variable Zug 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
- Die Funktionen ‘Bauer’, ‘Wolf’, ‘Kohl’ und ‘Ziege’ liefern die aktuelle Position zurück.
- Die Funktion ‘sicher’ bestimmt, ob eine Position sinnvoll ist, d.h. niemand wird gefressen.
- ‘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
	^	bitweises exklusives Oder, XOR
	&	bitweises Und
	|	bitweises Oder
	<<	Linksshift Operator

```
int Ort = …;
if (Ort & 0x08 == 0) {

} else {

}
int nOrt = Ort ^ 0x08;

int Pers = 1;
nOrt = Ort ^ (0x08 | Pers);

nPers = Pers <<= 1;
```

Hilfsfunktionen

```
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;
}
```

Hilfsfunktionen

```
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;
}
```

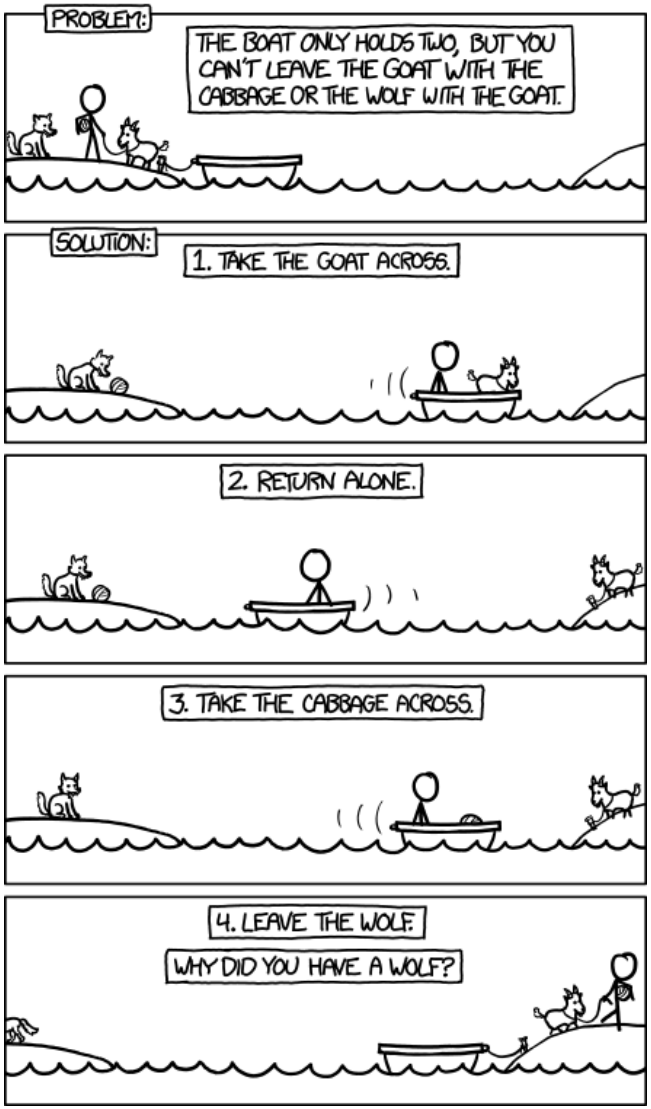
Traversierung

```
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 veraendert, 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';
}
```

Wie wird aus diesem bfs Ansatz ein dfs Ansatz?

Die "optimale Lösung"



https://xkcd.com/1134/

6.6 Minimaler Spannender Baum

Den Kanten des Graphen G(V, E) ist ein Wert zugeordnet (Netzwerk), d.h. für jede Kante $[u, v] \in E$ 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** 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, $w(T) = \Sigma_{(u,v)\in T} w(u, v)$ ist ein Minimum

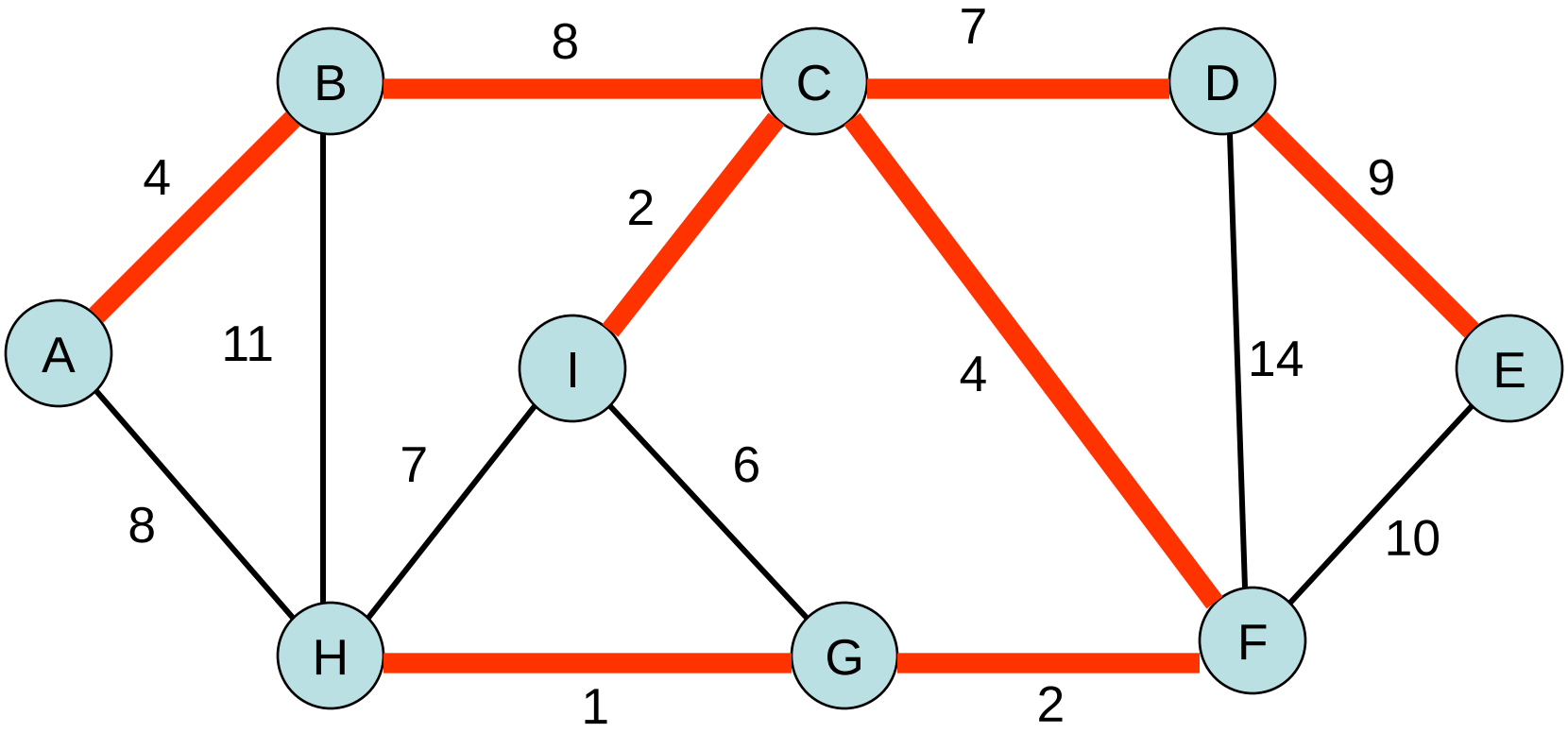
Beispiel:

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

Beispiel: Minimaler Spannender Baum



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 ∪ {e} eine Teilmenge eines möglichen MSB ist

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

Wie finde ich so eine sichere Kante?

Wald

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.

6.6.1 Kruskal Algorithmus

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) 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

Beweis von Lemma 4

Sei (V,A) ein Untergraph eines MSB. Sei [u,v] eine Kante mit minimalen Gewicht die zwei Komponenten vom Graphen (V,A) verbindet. Sei S eine Komponente von (V, A) 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 gehoert [u,v] zu keinem MSB, der alle Kanten von A enthaelt. Sei T ein MSB der alle Kanten von A enthaelt. Es folgt, dass T einen Weg P von u nach v enthaelt, der [u,v] nicht enthaelt.

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) ≤ w(x,y).

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

Warum ist T' ein spannender Baum?

Das Entfernen von [x,y] bricht T in 2 Komponenten, naemlich B und V – B sodass u $\in$ B und v $\notin$ B.

Daher verbindet das Hinzufügen von [u,v] zu T – {[x,y]} B und V – B wieder.

Daher ist T' = T – {[x,y]} + {[u,v]} verbunden.

Das Hinzufügen von [u,v] zu T erzeugt genau einen Kreis C in T + {[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 – {[x,y]} + {[u,v]} azyklisch.

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

Kruskal Algorithmus (2)

```
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 ein Menge bestehend aus dem Element v

find-set(v) liefert ein repräsentatives Element für die Menge die v enthält

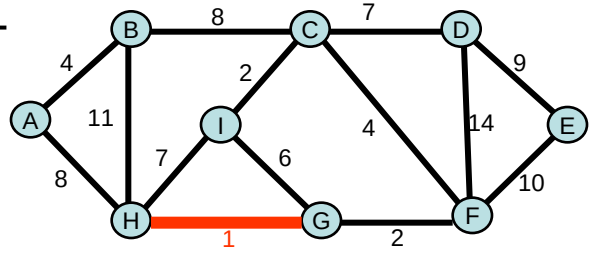
union(u,v) vereinigt Mengen von u und von v zu einer Menge

Hier: Jede Menge entspricht den Knoten in einer Komponente von (V,A)

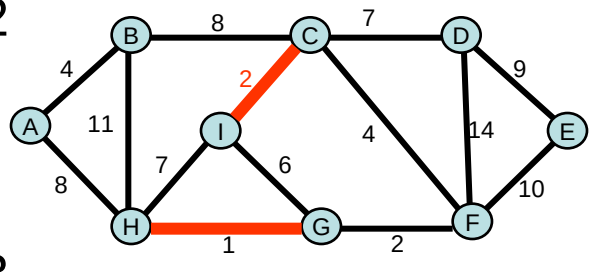
Beispiel: Kruskal Algorithmus

universität wien

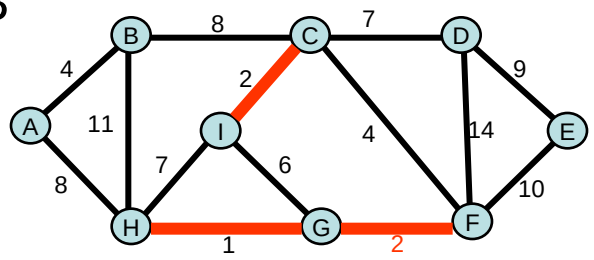
1



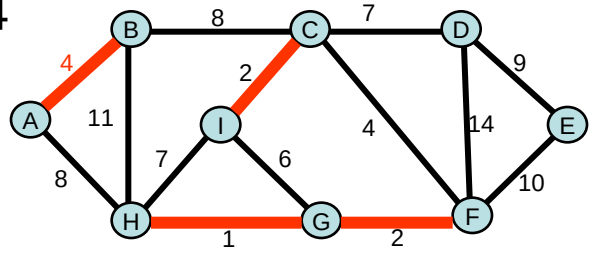
2



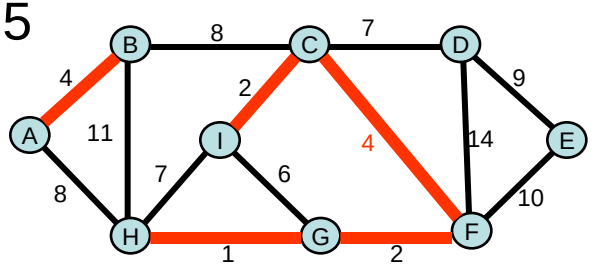
3



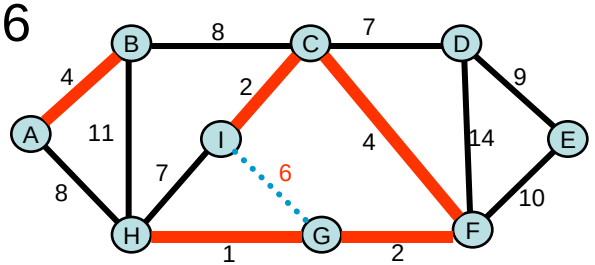
4



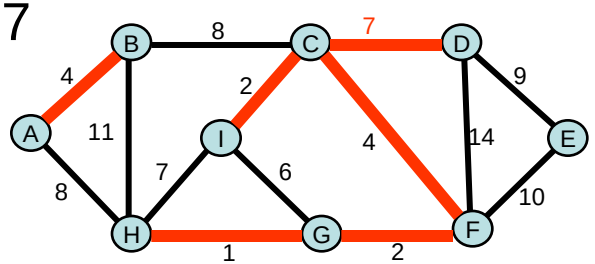
5



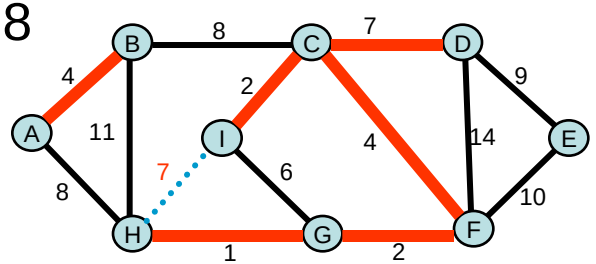
6



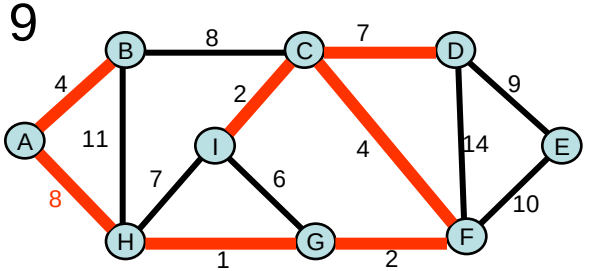
7



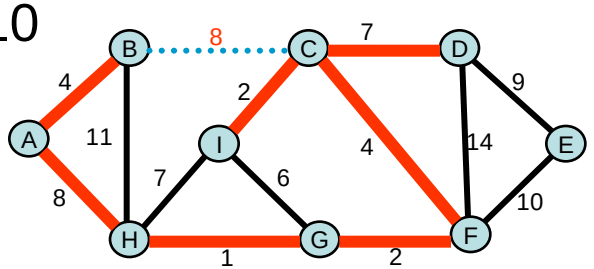
8



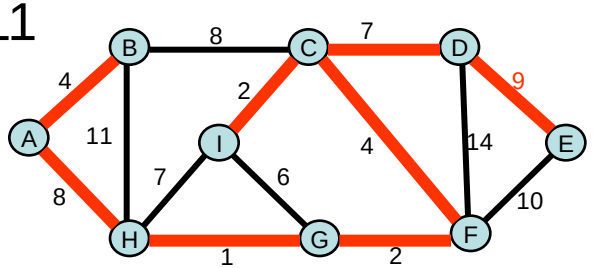
9



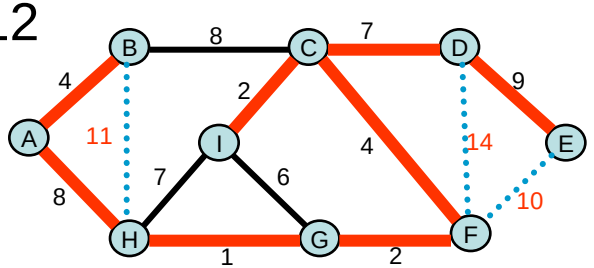
10



11



12



Aufwand von MSB-Kruskal

- O(|E| log |E|) Kanten sortieren plus
- |V| Make-Set Operationen
- 2|E| Find-Set Operationen
- ≤ |V|-1 Union Operationen
  (ein Baum hat höchstens |V|-1 Kanten)

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

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

Union-Find Datenstruktur

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

Einfache Union-find Datenstruktur

```
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 |V| haben

- Make-set(x): O(1)
- Link(x,y): O(1)
- Find-set(x): O(|V|)
- Union(x,y): O(|V|)

→ MSB-Kruskal:O(|E| |V|) 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)

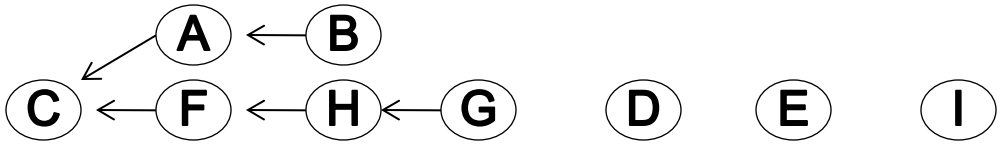
Union(A,B)

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];
}
```

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

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)

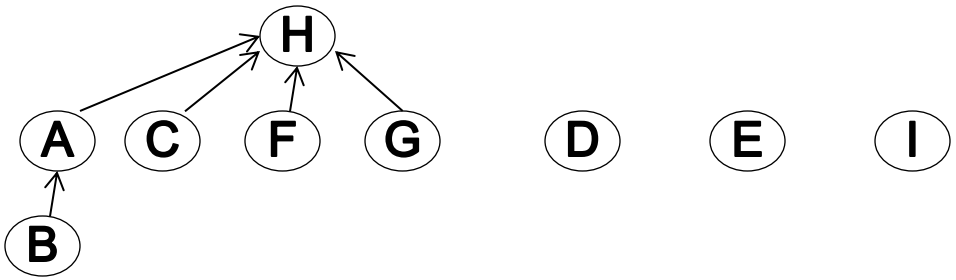
Union(A,B)

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

Eigenschaft von union-by-rank

Lemma 5: Ein Knoten x mit 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 > 0: 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 und auch 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

$\rightarrow$ Wenn es höchstens 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 $rank[x] \leq \log n$, d.h., jeder Baum höchstens Höhe log n

Aufwand von union-by-rank

Aufwand: Jeder Baum hat Höhe ≤ log |V|

- Make-set(x): O(1)
- Link(x,y): O(1)
- Find-set(x): O(log |V|)
- Union(x,y): O(log |V|)

→ MSB-Kruskal: O(|E| log |V| + |E| log |E|) = O(|E| log |E|) 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];
}
```

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

p[x] enthält den Elternknoten von x (Vertreter der Teilmenge)
rank[] 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)

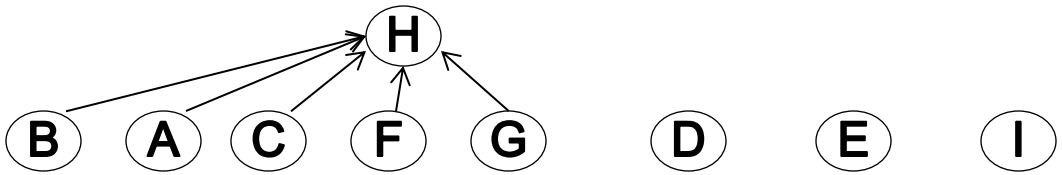
Union(C,F)

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:



Aufwand von union-by-rank mit path compression

Komplizierte Analyse

- Make-set(x): O(1)
- Link(x,y): O(1)
- Find-set(x): O(α(|V|)) (fast O(1))
- Union(x,y): O(α(|V|)) (fast O(1))

A(m,n): Ackermannfunktion, extrem schnell wachsende Funktion
f(n) = A(n,n)
α(n): $f^{-1}(n)$, extrem langsam wachsende Function
α(n) < 5 für alle praktisch relevanten n

→ MSB-Kruskal: O(|E| α(|V|) + |E| log |E|) = O(|E| log |E|) mit dieser Datenstruktur

6.6.2 Prim Algorithmus

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

- **Lemma 4':** *Sei A eine Menge von Kanten, sodass (V,A) ein Teilgraph eines MSB von G=(V,E) 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] eine Kante mit minimalen Gewicht die zwei Komponenten vom Graphen (V,A) verbindet. Sei S eine Komponente von (V, A) sodass u $\in$ S und v $\notin$ S."
  - durch:
    "Sei [u,v] eine Kante mit minimialen Gewicht, die S mit einer anderen Komponente von (V,A) verbindet."

6.6.2 Prim Algorithmus

Lemma 4': *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 fuer 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

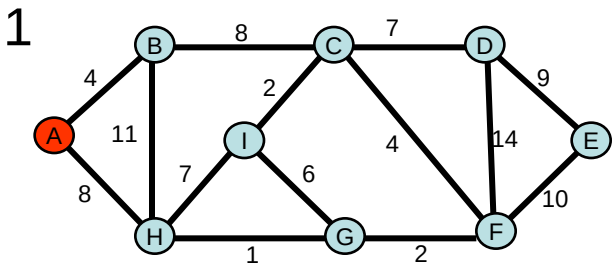
Vorteil: Sortieren der Kanten nach Gewicht ist nicht nötig

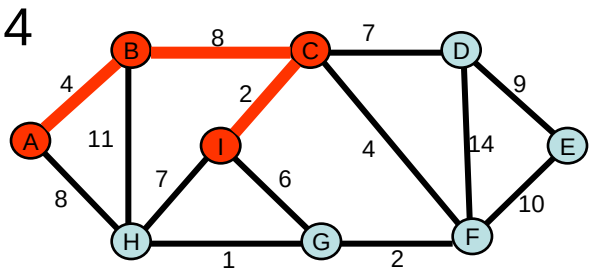
Prim Algorithmus

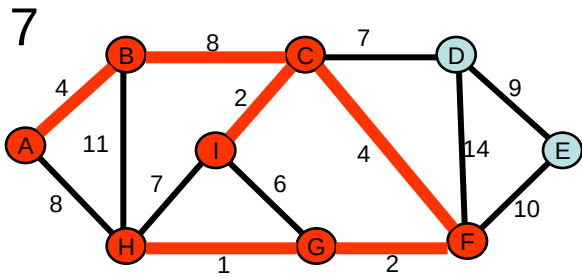
```
MSB-Prim(G(V,E), w, r) {
  Q = V;
  for(jeden Knoten u ∈ Q)
    key[u] = ∞;
  key[r] = 0;
  pred[r] = NIL;
  while(Q != {}) {
    u = Extract-Min(Q)
    if pred(u) != NIL) {
         A = A ∪ {  [u, pred[u]] }
    }
    for(jeden Knoten v adjazent zu u)
      if(v ∈ Q && w(u,v) < key[v]) {
        pred[v] = u;
        Decrease-Key(v, w(u,v));
      }
  }
}
```

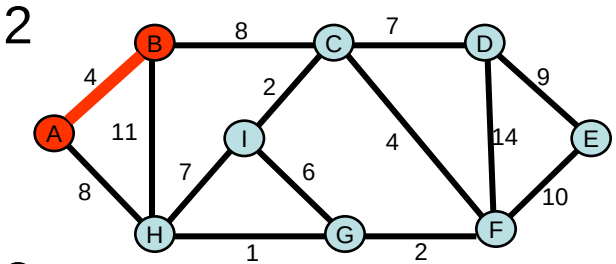
r ist die Wurzel (Startknoten) des MSB T
key[v] speichert das leichteste Gewicht von v zu einem Knoten in T
pred[v] speichert einen Knoten **u** sodass **[u,v]** Gewicht **key[v]** hat
Q ist eine Priority Queue, die alle Knoten nicht in T entsprechend ihres **key** Wertes speichert;

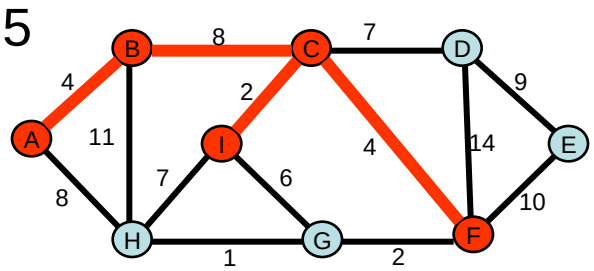
Beispiel: Prim Algorithmus

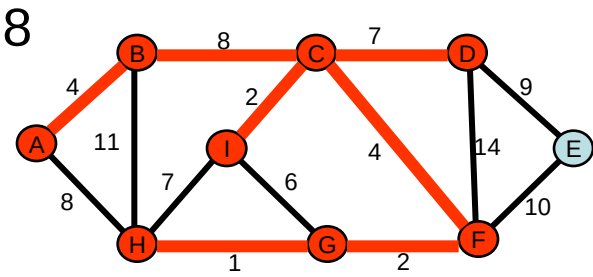


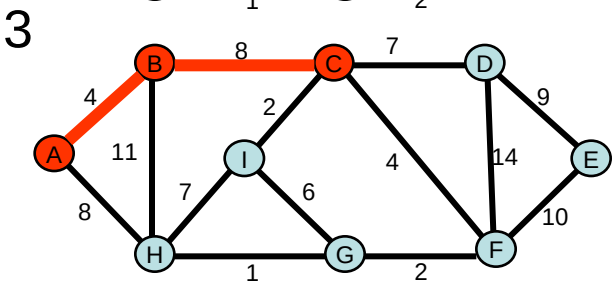


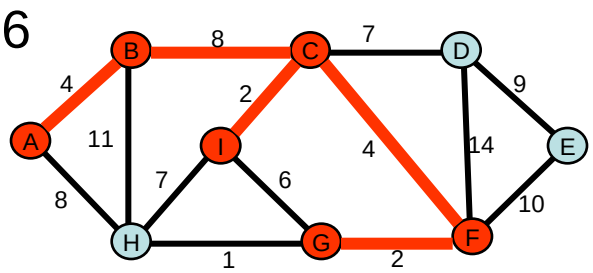


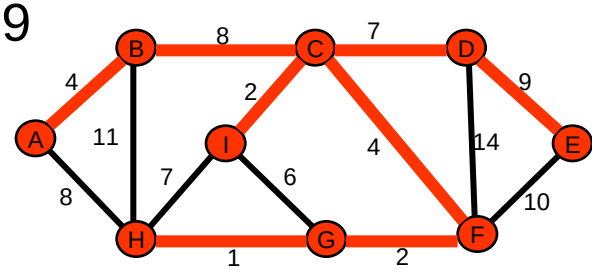












Aufwand von Prim Algorithmus

- O(|V| + |E|) plus
- |V| Einfüge Operationen in Q
- |V| Extract-Min Operationen
- ≤ |E| 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 |V|)

Totaler Aufwand: O(|E| log |V| + |V| log |V|) =
O(|E| log |V|)

Lässt sich durch Fibonacci-Heap verbessern auf O(|E| + |V| log |V|)

6.7 Kürzeste Wege

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

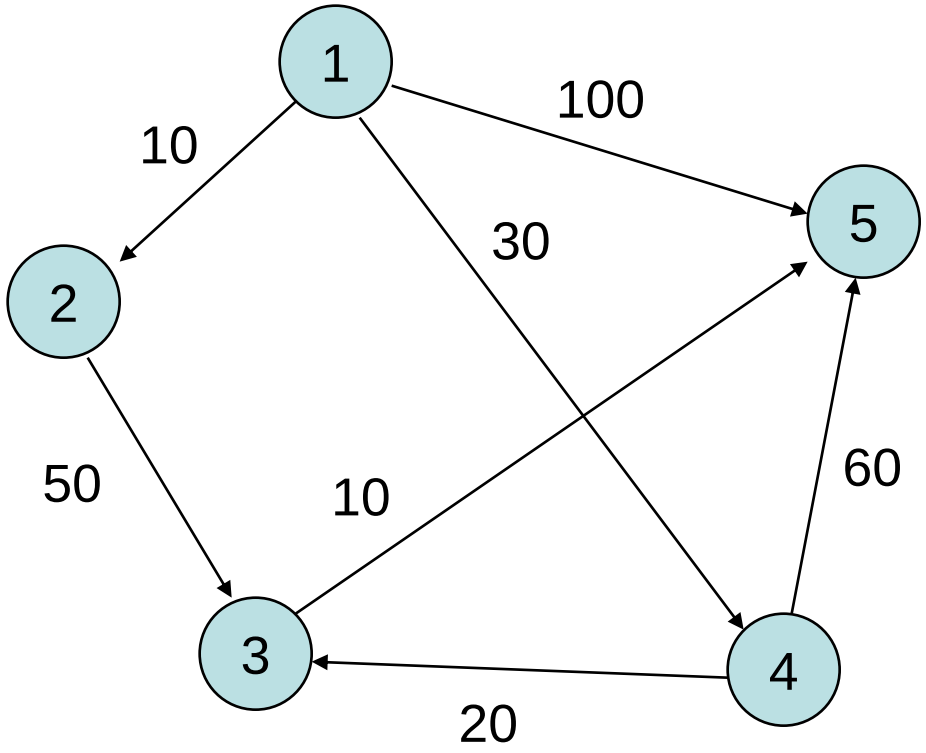
Annahme: alle Kosten sind gespeichert in der Kostenmatrix C (C[u,v] enthält Kosten von der Kante (u,v) wenn sie existiert und ∞, wenn (u,v) nicht existiert.)

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

Beispiel: Streckennetz einer Fluglinie

Annahme: nur positive Kantenwerte

Manche Algorithmen funktionieren nur mit positiven Werten



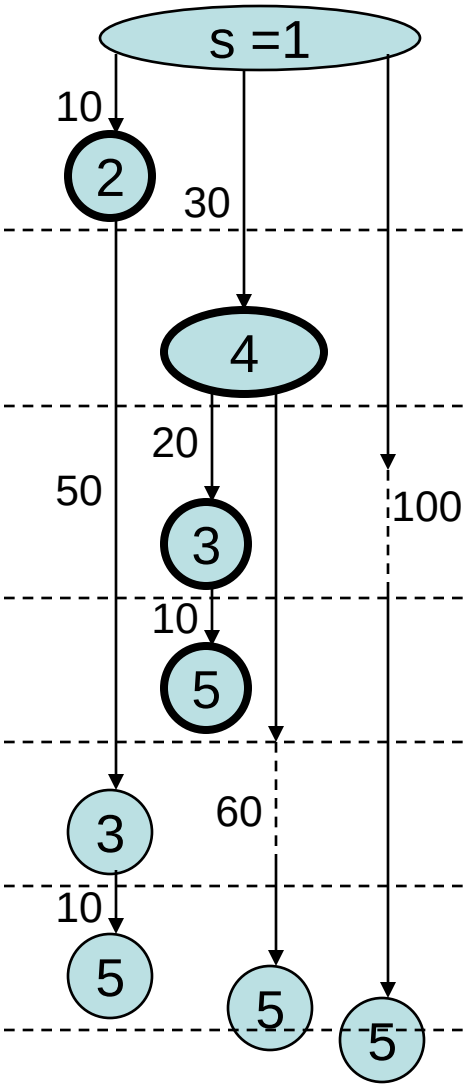
Algorithmus zur Suche des kürzesten Weges von einem Startknoten s zu allen anderen Knoten

Annahme: *nur positive Kantenwerte*

Idee

- Ausgehend vom Startknoten werden, beginnend mit dem kürzesten Weg zu einem Knoten, die nächst-längeren Wege zu allen anderen Knoten gesucht und die kürzesten Wege für die entsprechenden Knoten vermerkt.
  - 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?)



Dijkstra's Algorithmus

Grundidee:

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

Greedy Ansatz

Prim Algorithmus

```
MSB-Prim(G(V,E), w, r) {
  Q = V;
  for(jeden Knoten u ∈ Q)
    key[u] = ∞;
  key[r] = 0;
  pred[r] = NIL;
  while(Q != {}) {
    u = Extract-Min(Q)
    if pred(u) != NIL) {
         A = A ∪ {  [u, pred[u]] }
    }
    for(jeden Knoten v adjazent zu u)
      if(v ∈ Q && w(u,v) < key[v]) {
        pred[v] = u;
        Decrease-Key(v, w(u,v));
      }
  }
}
```

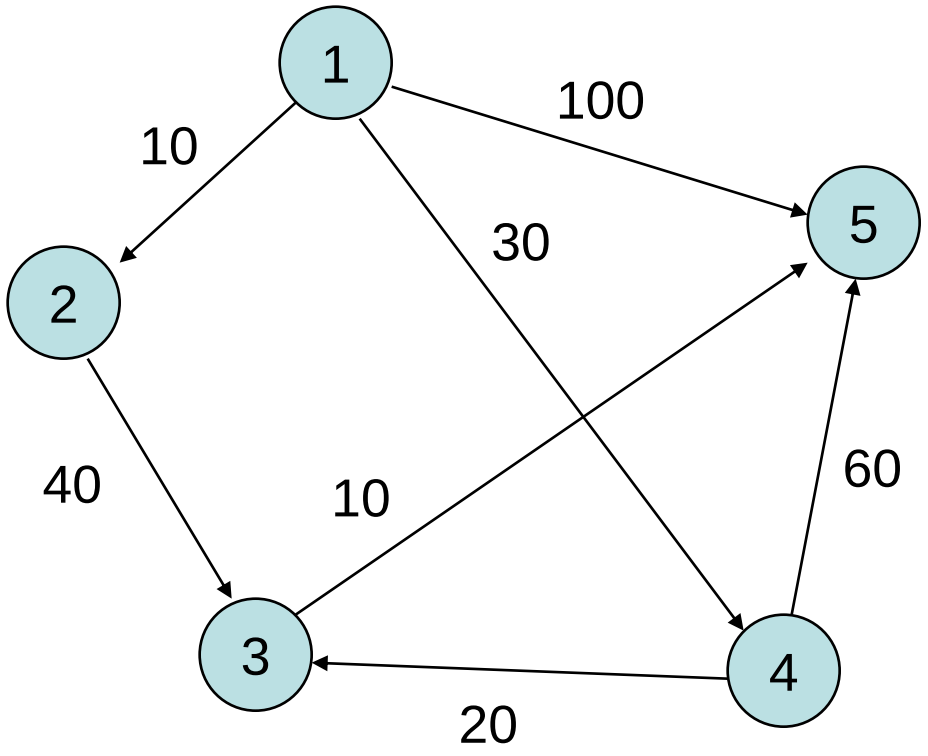
r ist die Wurzel (Startknoten) des MSB T
key[v] speichert das leichtestes Gewicht von v zu einem Knoten in T
pred[v] speichert einen Knoten **u** sodass **[u,v]** Gewicht **key[v]** hat
Q ist eine Priority Queue, die alle Knoten nicht in T entsprechend ihres **key** Wertes speichert;

Dijkstra Algorithmus

```
1.Q = V;
2. for(jeden Knoten v ∈ Q)
3.     MinC[v] = ∞;
4. MinC[s] = 0;
5. while(Q != {}){
6.   v = Extract-Min(Q);
7.   if MinC[v] < ∞{
8.         S = S ∪ { v };
9.       for (jeden Knoten w sodass v adjazent zu w ist)
10.        if w∈Q && MinC[v] + C[v,w] < MinC[w] {
11.          Decrease-Key(w, MinC[v] + C[v,w]);
12.        }
13.  }
14.}
```

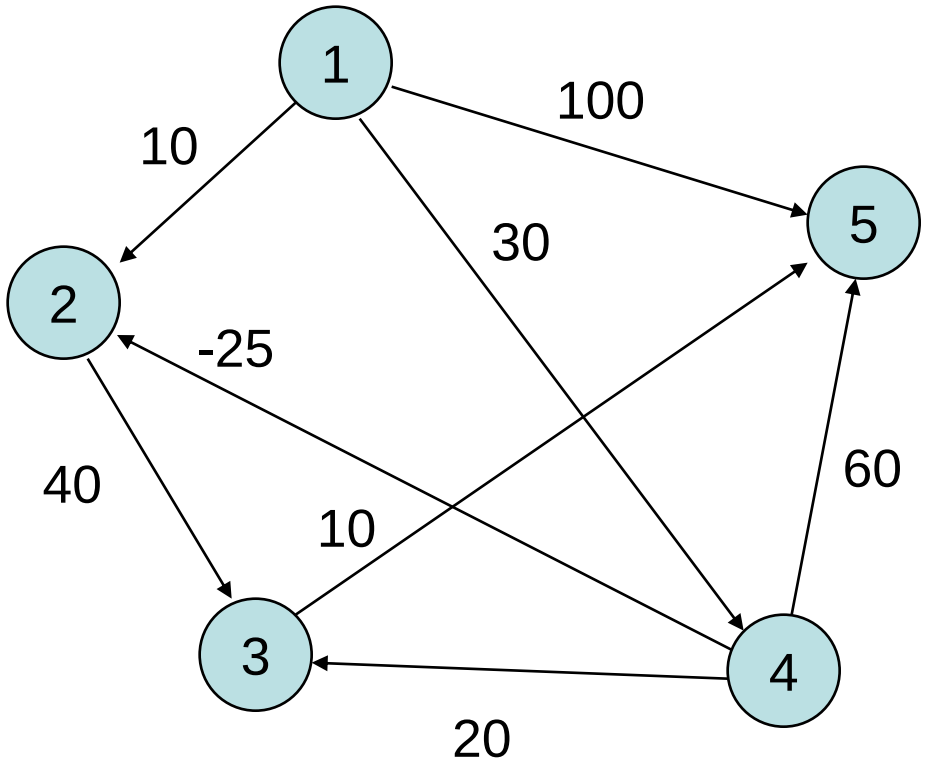
S Menge der bereits bearbeiteten Knoten
u Knoten, der gerade bearbeitet wird
C Kostenmatrix
MinC bisher bekannter kürzester Weg
Q ist eine Priority Queue (=Heap), die alle Knoten nicht in S entsprechend ihres **MinC** Wertes speichert

Berechnung für s = 1



S	Akt	MinC
{1}	-	[/, 10, ∞, 30, 100]
{1,2}	2	[/, 10, 50, 30, 100]
{1,2,4}	4	[/, 10, 50, 30, 90]
{1,2,3,4}	3	[/, 10, 50, 30, 60]

Was passiert bei negativen Kantenkosten?



S	Akt	MinC
{1}	-	[/, 10, ∞, 30, 100]
{1,2}	2	[/, 10, 50, 30, 100]
{1,2,4}	4	[/, 5, 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 [/,5, 50, 30, 60] falsche Werte

Aufwand

- O(|V| + |E|) plus
- |V| Insert und Extract-Min Operationen
- ≤ |E| 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 |V|)

Totaler Aufwand: O(|E| log |V| + |V| log |V|) = O(|E| log |V|)

Lässt sich durch Fibonacci-Heap verbessern auf O(|E| + |V| log |V|)

6.7.2 All Pairs Shortest Paths (APSP)

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

Transitive Hülle

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:

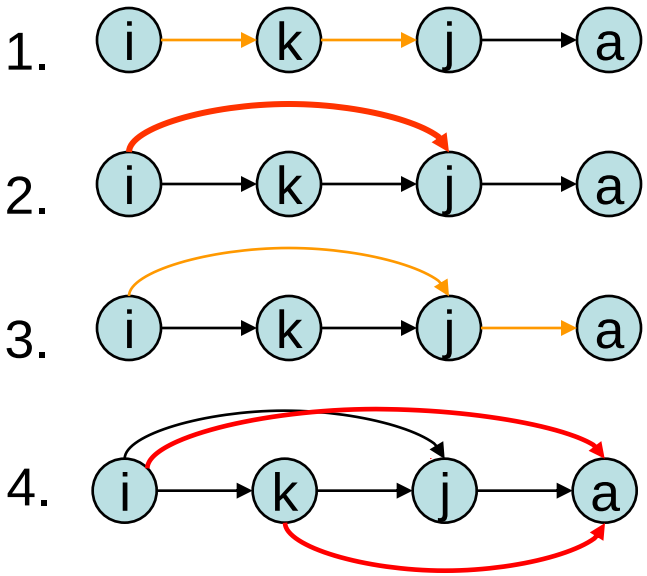
$(v, w) \in E' \Leftrightarrow$ es existiert ein Weg von v nach w in G

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**:
Loese “kleinere Subprobleme” um Gesamtproblem zu loesen



Floyd-Warshall Algorithmus (2)

In welcher Reihenfolge sollen die KnotenWege geprueft 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, fuege 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, fuege 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} benützen
- 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, …, k} benützen
  (innere Knoten = nicht Start- oder Endknoten des Weges)

Floyd-Warshall Algorithmus (2)

```
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

Aufwand

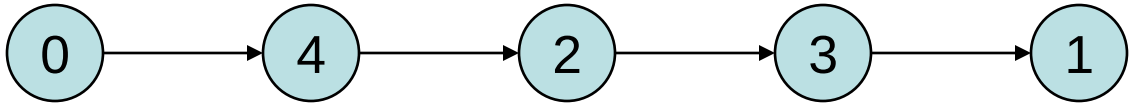
Mit Adjazenzmatrixrepresäntation:

$O(|V|)$ für die innerste Schleife,

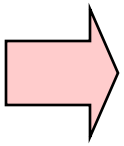
$O(|V|^2)$ für die zwei inneren Schleifen,

$O(|V|^3)$ 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):

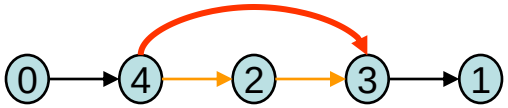
0: keine Kante, da a(?,0) nicht möglich

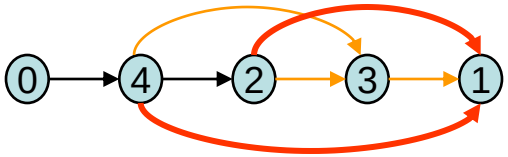
1: keine Kante, da a(1,?) nicht möglich

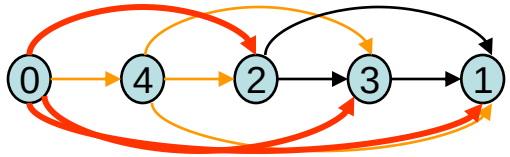
2: Kante a(4,2), a(2,3) ⇒ a(4,3)

3: a(2,3), a(3,1) ⇒ a(2,1)
a(4,3), a(3,1) ⇒ a(4,1)

4: a(0,4), a(4,1) ⇒ a(0,1)
a(0,4), a(4,2) ⇒ a(0,2)
a(0,4), a(4,3) ⇒ 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

Floyd-Warshall Algorithmus für kürzeste Wege

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 Algorithmus (2)

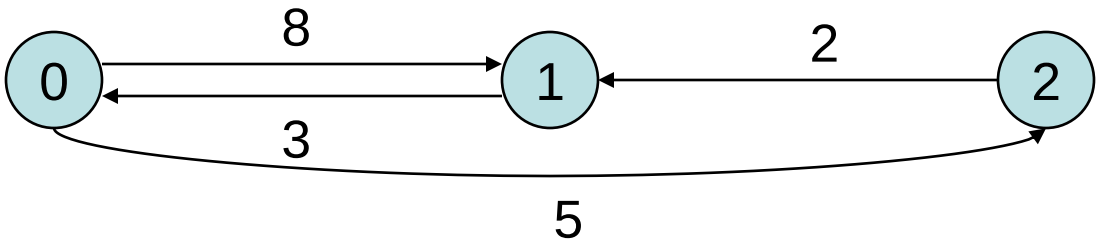
```
Floyd-Warshall (Matrix a) {
  n = a.numberofRows();
  for(int k = 0; k < n; k++)
    for(int i = 0; i < n; i++)
      for(int j = 0; j < n; j++) {
        int newPathLength = a[i,k] + a[k,j];
        if(newPathLength < a[i,j])
          a[i,j] = newPathLength;
      }
}
```

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

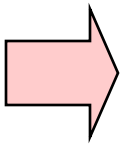
Berechnung der Weglänge über neuen Weg

Invariante: Die k-te Ausführung der äußersten Schleife findet die Länge des kürzesten Weges von i nach j, der als innere Knoten nur Knoten aus {0, …, k-1} benützt

Beispiel: Floyd-Warshall für kürzeste Wege



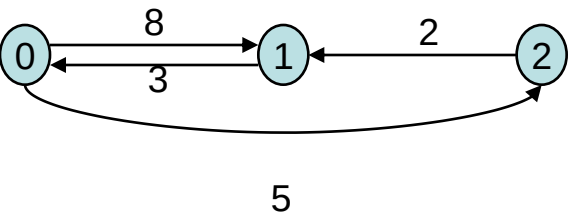
	0	1	2
0	0	8	5
1	3	0	∞
2	∞	2	0



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

Beispiel: Floyd-Warshall für kürzeste Wege (2)

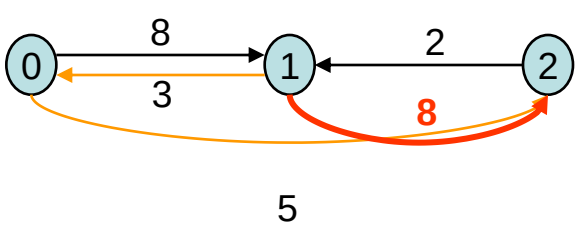
Ausgangsposition



	0	1	2
0	0	8	5
1	3	0	∞
2	∞	2	0

k: 0

Kanten: (1,0) (0,2) ⇒ (1,2), Kosten: 8



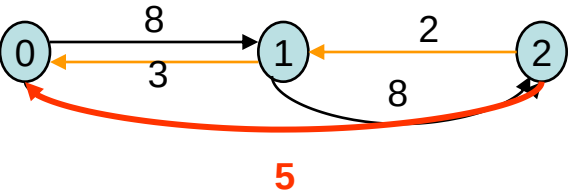
	0	1	2
0	0	8	5
1	3	0	**8**
2	∞	2	0

k: 1

(0,1) (1,2) ⇒ (0,2): 16

(2,1) (1,0) ⇒ (2,0): 5

(2,1) (1,2) ⇒ (2,2):10



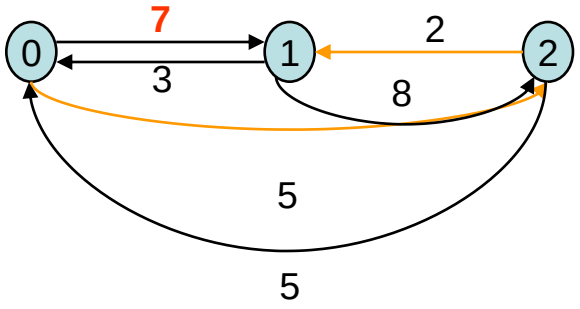
	0	1	2
0	0	8	5
1	3	0	8
2	**5**	2	0

k: 2

(0,2) (2,0) ⇒ (0,0): 10

(0,2) (2,1) ⇒ (0,1): 7

(1,2) (2,0) ⇒ (1,0): 13

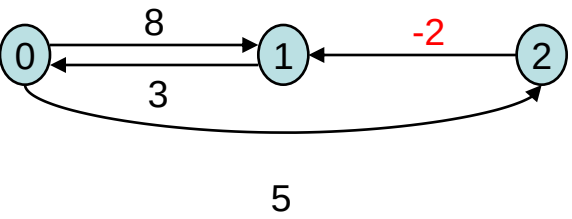


	0	1	2
0	0	**7**	5
1	3	0	8
2	5	2	0

Floyd-Warshall für kürzeste Wege: Was geschieht bei negativen Kantenkosten?

universität wien

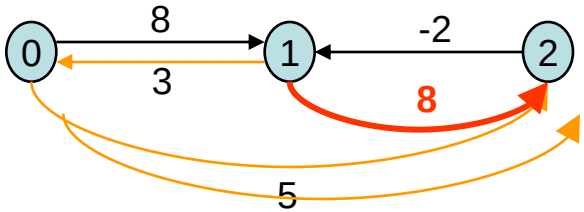
Ausgangsposition



	0	1	2
0	0	8	5
1	3	0	∞
2	∞	-2	0

k: 0

Kanten: (1,0) (0,2) ⇒ (1,2), Kosten: 8



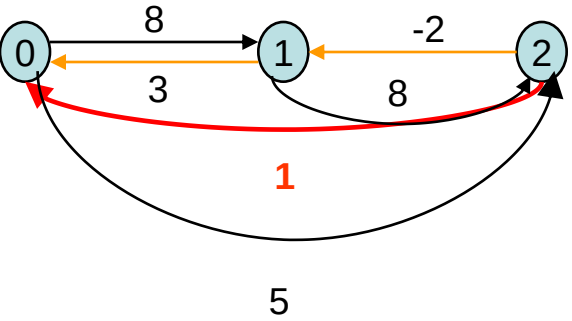
	0	1	2
0	0	8	5
1	3	0	**8**
2	∞	-2	0

k: 1

(0,1) (1,2) ⇒ (0,2): 16

(2,1) (1,0) ⇒ (2,0): 1

(2,1) (1,2) ⇒ (2,2): 6



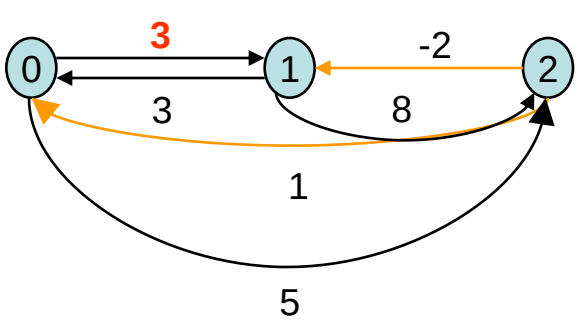
	0	1	2
0	0	8	5
1	3	0	8
2	**1**	-2	0

k: 2

(0,2) (2,0) ⇒ (0,0): 6

(0,2) (2,1) ⇒ (0,1): 3

(1,2) (2,0) ⇒ (1,0): 9



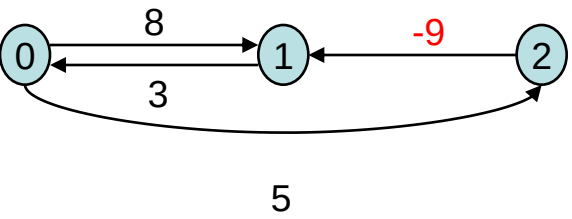
	0	1	2
0	0	**3**	5
1	3	0	8
2	1	-2	0

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?

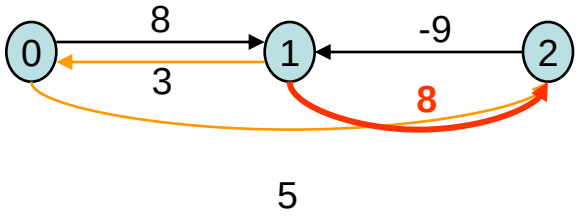
Ausgangsposition



	0	1	2
0	0	8	5
1	3	0	∞
2	∞	-9	0

k: 0

Kanten: (1,0) (0,2) ⇒ (1,2), Kosten: 8



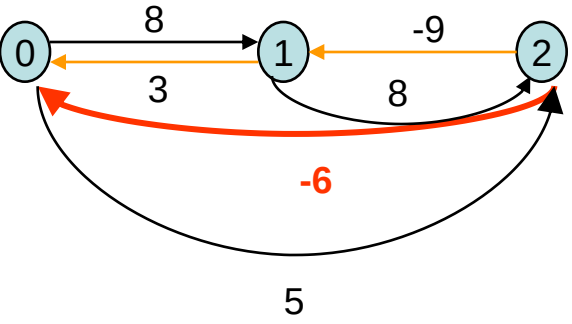
	0	1	2
0	0	8	5
1	3	0	**8**
2	∞	-9	0

k: 1

(0,1) (1,2) ⇒ (0,2): 16

(2,1) (1,0) ⇒ (2,0): -6

(2,1) (1,2) ⇒ (2,2): -1



	0	1	2
0	0	8	5
1	3	0	8
2	**-6**	-9	-1

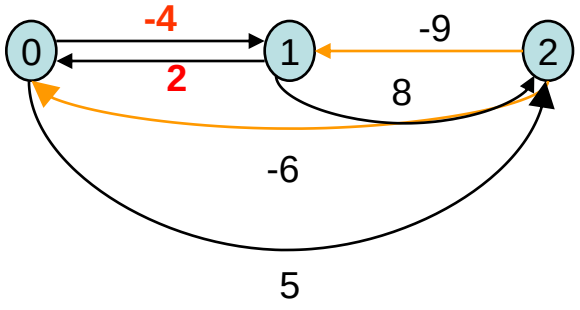
k: 2

(0,2) (2,0) ⇒ (0,0): -1

(0,2) (2,1) ⇒ (0,1): -4

(1,2) (2,0) ⇒ (1,0): 2

(1,2) (2,1) ⇒ (1,1): -1



	0	1	2
0	**-1**	**-4**	5
1	**2**	**-1**	8
2	-6	-9	-1

Was geschieht bei negativen Zyklen?

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

$\rightarrow$ Floyd-Warshall kann dazu benützt werden, die Existenz von negativen Zyklen zu testen

Aufwand: $O(|V|^3)$

Was nehmen wir mit?

- Graphen
  - Definitionen
  - Gerichtete und ungerichtete Graphen
- Topologisches Sortieren
- Traversierung
  - Bfs und dfs
  - „Bauer, Wolf, Ziege und Kohlkopf“ Problem
  - Generell iterativer Ansatz
- Spannende Bäume
- Kürzeste Wege

Dynamisches Programmieren kann man anwenden wenn:

- ein Problem aus voneinander abhaengigen Unterproblemen besteht

(wenn die Unterprobleme unabhaengig sind, benuetze Divide&Conquer)

- die Unterprobleme Unter-Unterprobleme teilen, die nur einmal geloest werden muessen und deren Loesung (z.B. in einem Array) gespeichert werden kann

- Beispiele:
  - Transitive Hülle, Floyd-Warshall Algorithmus
  - Fibonacci Zahlen
  - Knapsack Problem

Beispiel: Fibonacci Zahlen

Problem: Berechnung von Fibonacci Zahlen

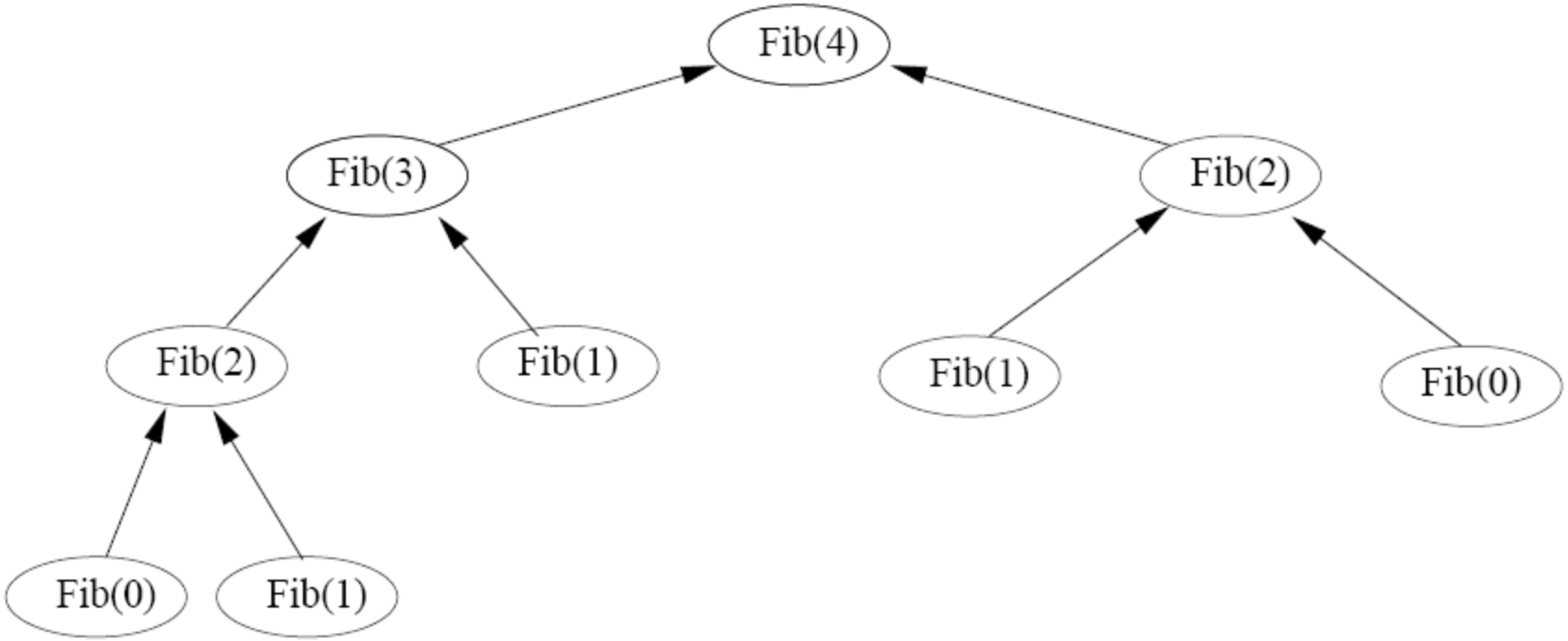
$$\begin{aligned} F_0 &= 0, \\ F_1 &= 1, \\ F_n &= F_{n-1} + F_{n-2}, \quad n \geq 2 \end{aligned}$$

Input: Integer n ≥ 0

Output: F_n

Rekursive Loesung:

```
Fib (int n) {
  if (n ≤ 1) {
    return n;
  } else {
    return Fib(n-1) + Fib(n-2)
  }
}
```


Fib(4)
Fib(3)
Fib(2)
Fib(2)
Fib(1)
Fib(1)
Fib(0)
Fib(0)
Fib(1)

$T(n) = T(n-1) + T(n-2) + 1$

Lemma: Fuer $n \geq 1$, $T(n) \geq \varphi^{n-1}$, wobei $\varphi = (1+\sqrt{5})/2 = 1.618\ldots$

Note: $\varphi^2 = \varphi+1$

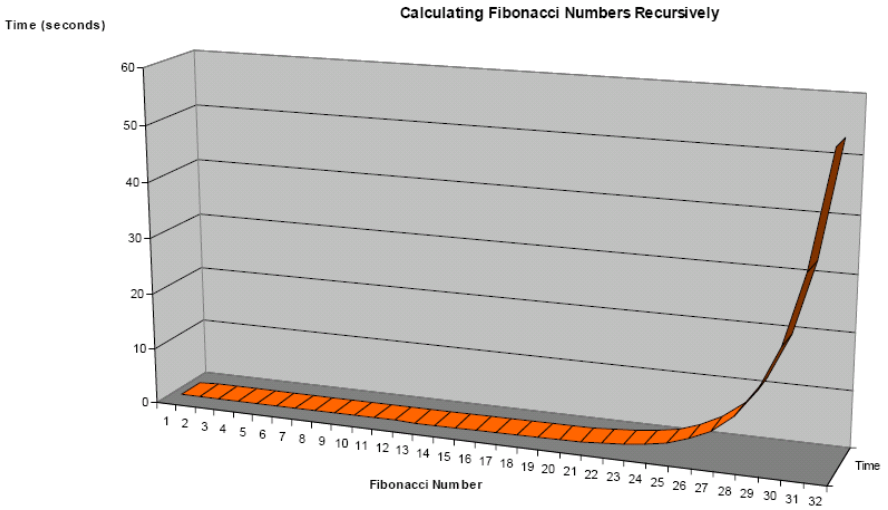
Proof: Induktionsbeweis:

Induktionsbasis: $T(0) = 1 \geq \varphi^0$, $T(1) = 1 \geq \varphi^0$

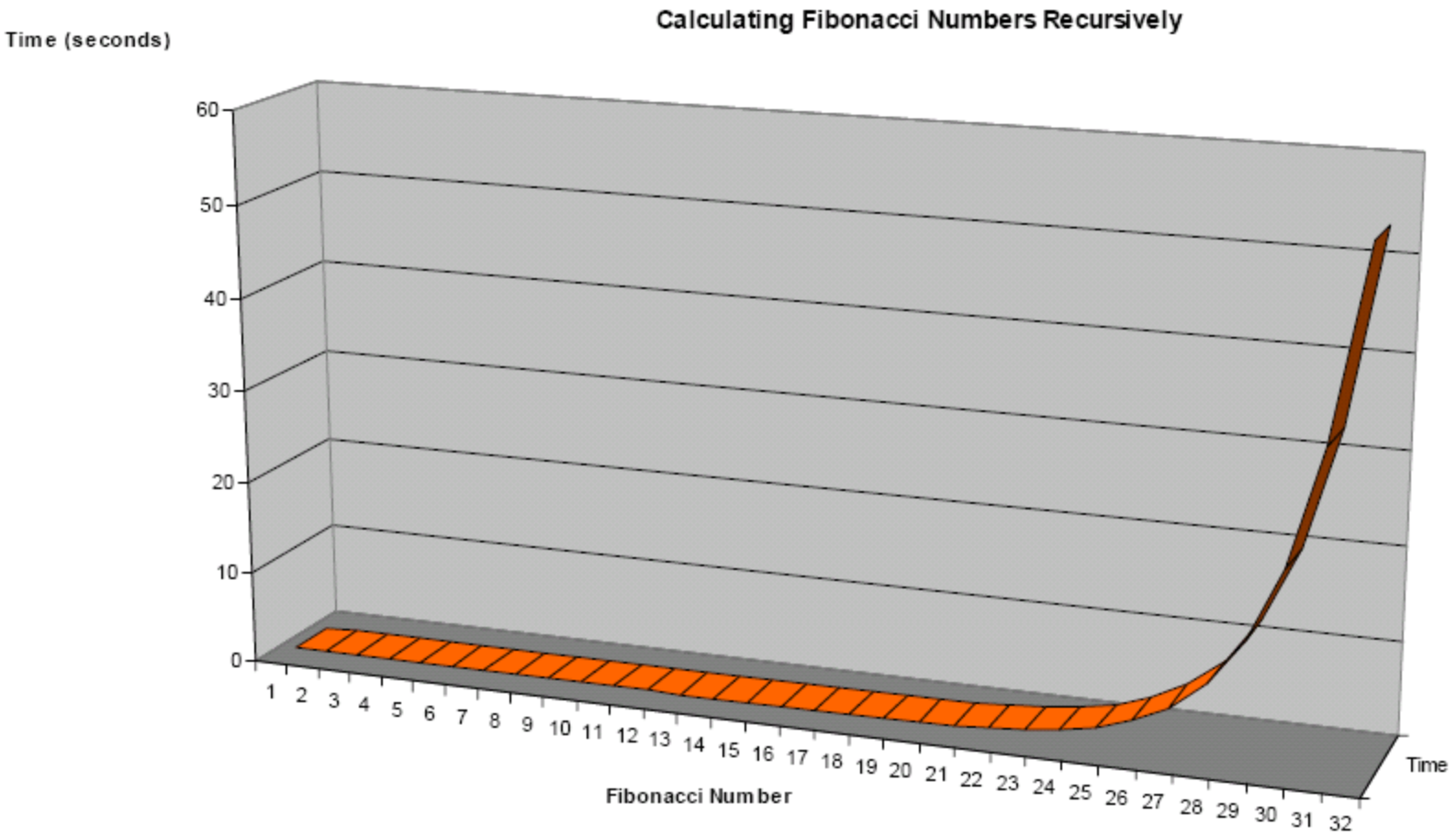
Induktionsschritt: $n > 1$:

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + 1 \\ &\geq \varphi^{n-2} + \varphi^{n-3} \\ &= \varphi^{n-3}(\varphi+1) \\ &= \varphi^{n-3}\varphi^2 \\ &= \varphi^{n-1} \end{aligned}$$

⇨ $T(n) = \Omega(\varphi^n)$, exponentiell weil es die Werte fuer Fib(n-2), Fib(n-3) etc. mehrmals berechnet.



Running Time



Idee: Speichere Ergebnisse von Unterproblemen in einem Vektor F anstatt sie wiederzuberechnen.

```
FibonacciDP (int n) {
  F[0] = 0;
  F[1] = 1;
  for(int k = 2; k ≤ n; ++k)
      F[k] = F[k-1] + F[k-2];
  }
  return F[n];
}
```

Laufzeit: O(n)

0/1 Knapsack Problem

Input: a knapsack of capacity W and a set of *n* objects numbered *1,2,…,n*. Each object *i* has weight w_i and profit v_i.

Output: A vector x = $[x_1, x_2, \ldots, x_n]$ in which

$x_i = 0$ if object *i* is not in the knapsack, and
$x_i = 1$ if it is in the knapsack.

such that $\sum_{i=1}^{n} w_i x_i \leq W$

(that is, the objects fit into the knapsack) and

$$\sum_{i=1}^{n} v_i x_i$$

is maximized (that is, the profit is maximized).

0/1 Knapsack Problem

Naive algorithm: consider all 2^n possible subsets of the n objects and choose the one that fits into the knapsack and maximizes the profit

$\Rightarrow$ running time exponential in n

0/1 Knapsack Problem: DP solution

Let $c_{i,w}$ be the maximum profit for a knapsack of capacity w using only objects $\{1,2,\ldots,i\}$. The DP formulation is:

$$c_{i,w} = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ c_{i-1,w} & \text{if } w_i > w \\ \max\{v_i + c_{i-1,w-w_i}, c_{i-1,w}\} & \text{if } i > 0 \text{ and } w_i \leq w \end{cases}$$

Construct a table F of size $n \times W$ in row-major order.

Filling an entry in a row requires two entries from the previous row: one from the same column and one from the column offset by the weight of the object corresponding to the row.

0/1 Knapsack Problem: DP Algorithm

```
1:  for w = 0...W do
2:      c_{0,w} ← 0
3:  end for
4:  for i = 1...n do
5:      c_{i,0} ← 0
6:      for w = 1...W do
7:          if w_i ≤ w then
8:              if v_i + c_{i-1,w-w_i} > c_{i-1,w} then
9:                  c_{iw} = v_i + c_{i-1,w-w_i}
10:             else
11:                 c_{iw} = c_{i-1,w}
12:             end if
13:         else
14:             c_{iw} = c_{i-1,w}
15:         end if
16:     end for
17: end for
18: return c_{nW}.
```

0/1 Knapsack Problem

Computing each entry takes constant time ⇨ running time is $\Theta(nW)$.

No algorithm is known that takes time polynomial in n alone

Was nehmen wir mit?

- Graphen
  - Definitionen
  - Gerichtete und ungerichtete Graphen
- Topologisches Sortieren
- Traversierung
  - Bfs und dfs
  - „Bauer, Wolf, Ziege und Kohlkopf“ Problem
  - Generell iterativer Ansatz
- Spannende Bäume
- Kürzeste Wege
- Dynamisches Programmieren