Fachhochschule Nordwestschweiz

07 Graphen

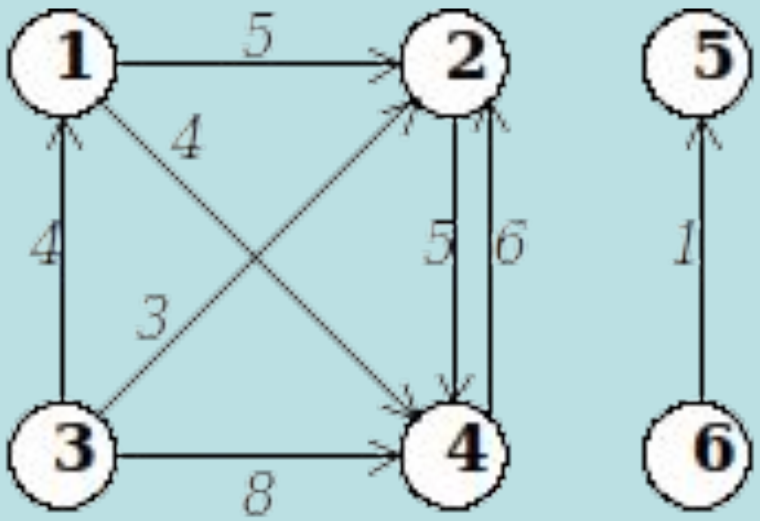
Algorithmen und Datenstrukturen 2

1. **Teil**
   - Graphrepräsentation
   - Topologisches Sortieren

2. **Teil**
   - Graphtraversierungen (DFS & BFS)
   - Anwendungen von DFS & BFS (Zusammenhangskomponenten & Spannbaum)

3. **Teil**
   - Kürzeste Wege

Speichern von Graphen



Adjazenzmatrix

n×n-Matrix (n = Anzahl Knoten)

- Gewichtet: Integer-Matrix
  `int[][] G = new int[n][n];`
- Ungewichtet: Boolean-Matrix
  `boolean[][] G = new boolean[n][n];`

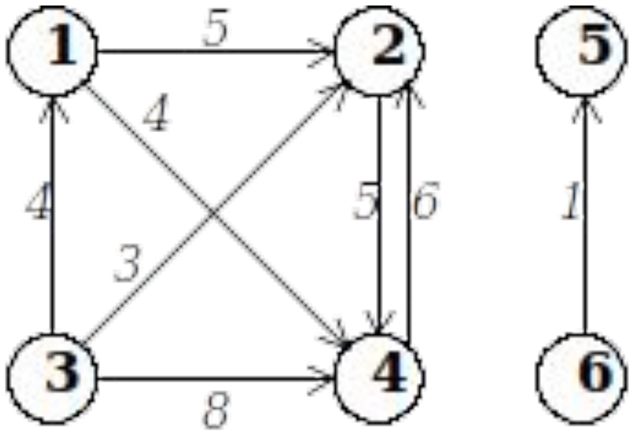
Ungerichtete Graphen: $G = G^T$

Speicherplatz: $O(n^2)$

Abbilidung “keine Kante” ist abhängig von Interpretation

- Distanz: Integer.MAX_VALUE
- Kapazität: 0

Bei richtiger Abbildung ist *keine* Fallunterscheidung im Algorithmus nötig



Zu

Von

	1	2	3	4	5	6
1	-	5	-	4	-	-
2	-	-				
3	4	3				
4						
5						
6						

Kantenliste

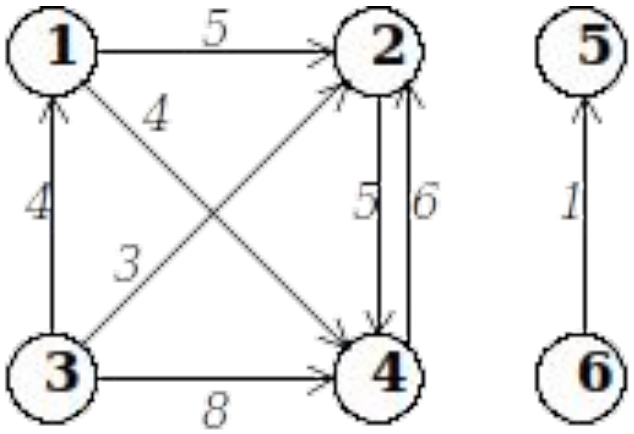
Tabelle / Liste mit Edge-Einträgen:

```
class Edge {
   int from, to, weight;
}
```

Ungewichtete Graphen: weight weglassen

Ungerichtete Graphen: 2x eintragen oder from / to gleich behandeln

Speicherplatz: O(m), m = Anzahl Kanten



from	1							
to	2							
weight	5							

Fachhochschule Nordwestschweiz

Adjazenzlisten

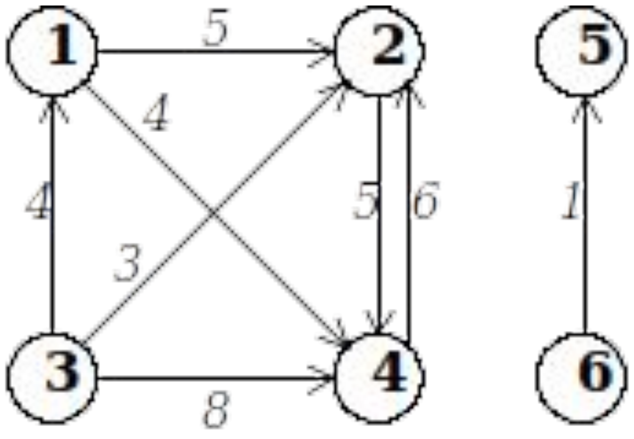
adjazent = benachbart

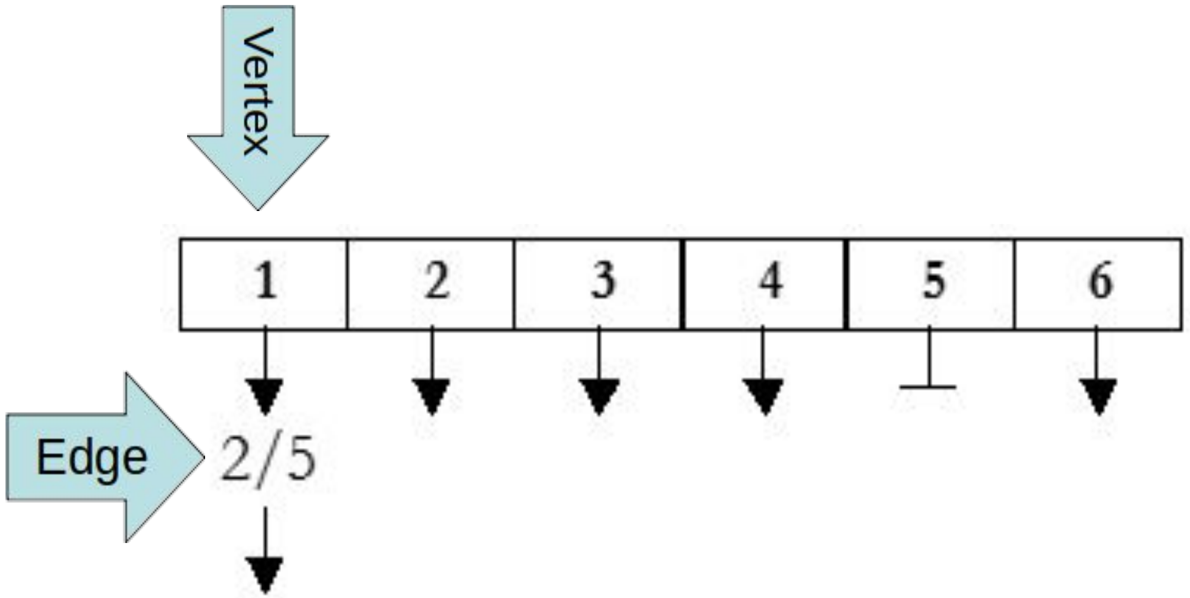
Pro Knoten eine Liste mit seinen ausgehenden Kanten

Adjazenzlisten in Array speichern (oder als Liste verbinden)

Ungerichtete Graphen: Kante in beiden Listen

Speicherplatz: O(n + m)



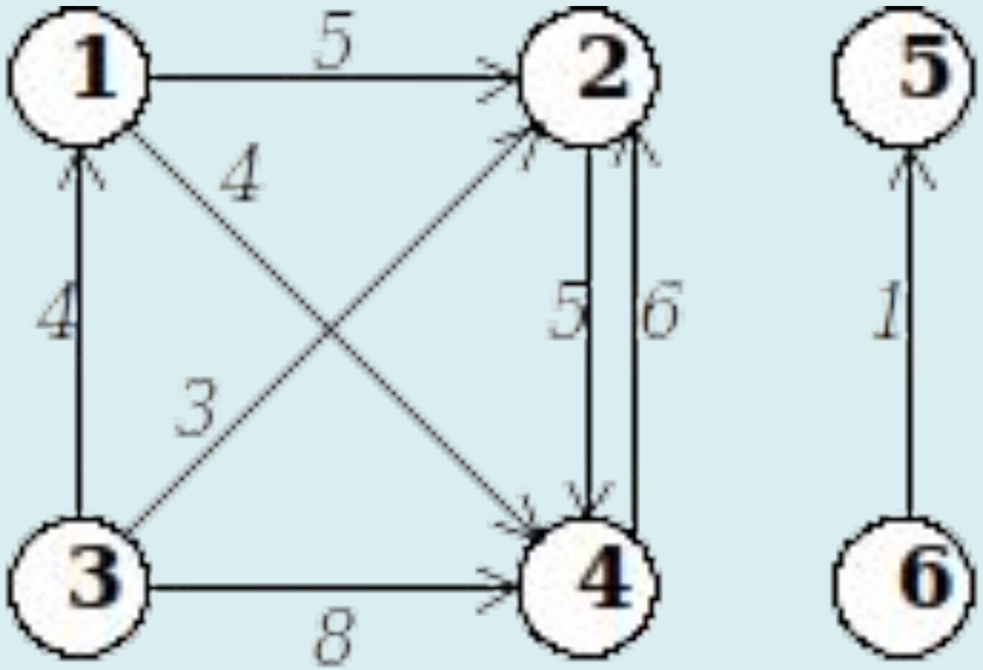


Aufgabe

Stellen Sie den folgenden Graphen als

- Adjazenzmatrix
- Kantentabelle
- Adjazenzliste

dar:



Aufgabe - Lösungen

1	2	3	4	5	6
↓	↓	↓	↓	↓	↓
2/5	4/5	1/4	2/6		5/1
↓		↓			
4/4		2/3			
		↓			
		4/8			

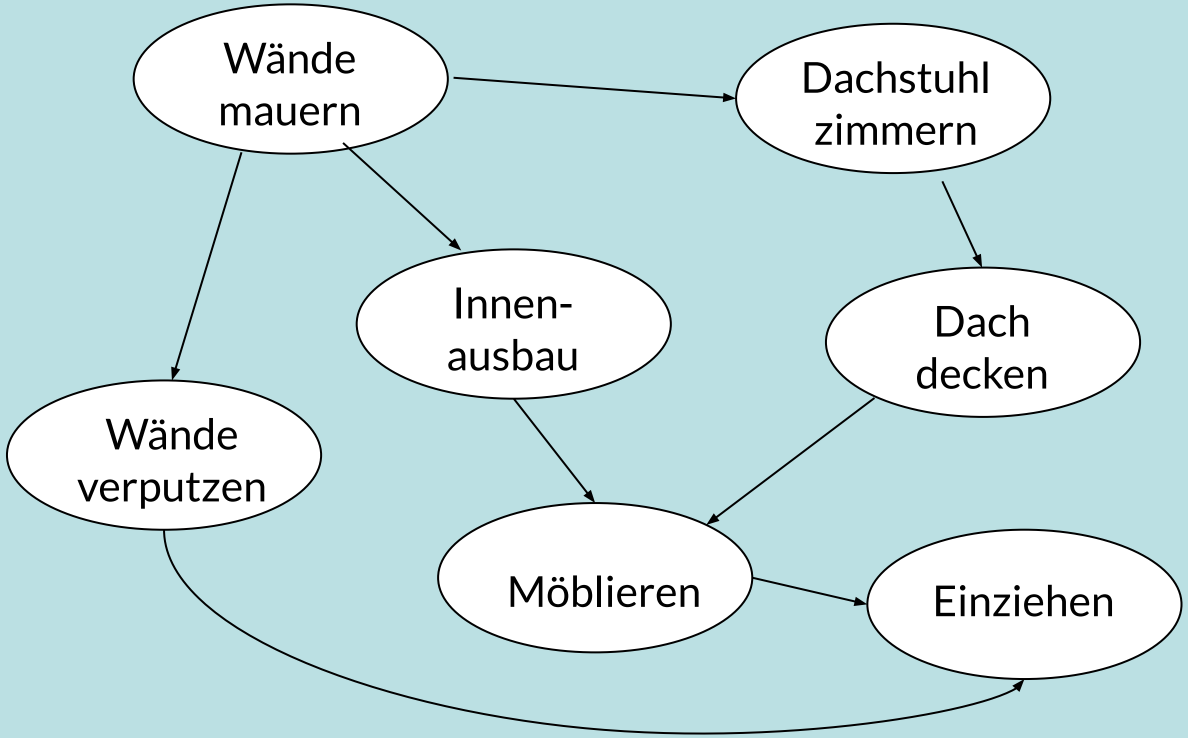
	1	2	3	4	5	6
1	-	5	-	4	-	-
2	-	-	-	5	-	-
3	4	3	-	8	-	-
4	-	6	-	-	-	-
5	-	-	-	-	-	-
6	-	-	-	-	1	-

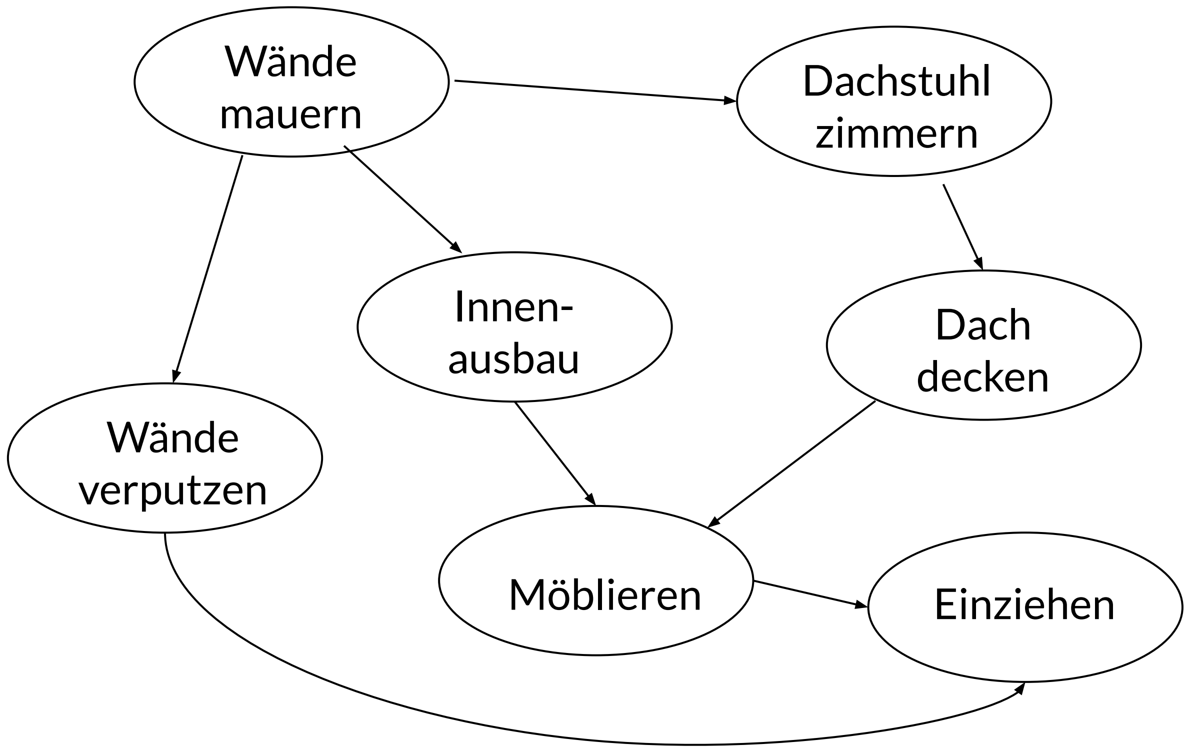
fr	1	1	2	3	3	3	4	6
to	2	4	4	1	2	4	2	5
weight	5	4	5	4	3	8	6	1

Komplexitätsanalyse

Lösen Sie sie Aufgabe 2 auf dem Arbeitsblatt 1.

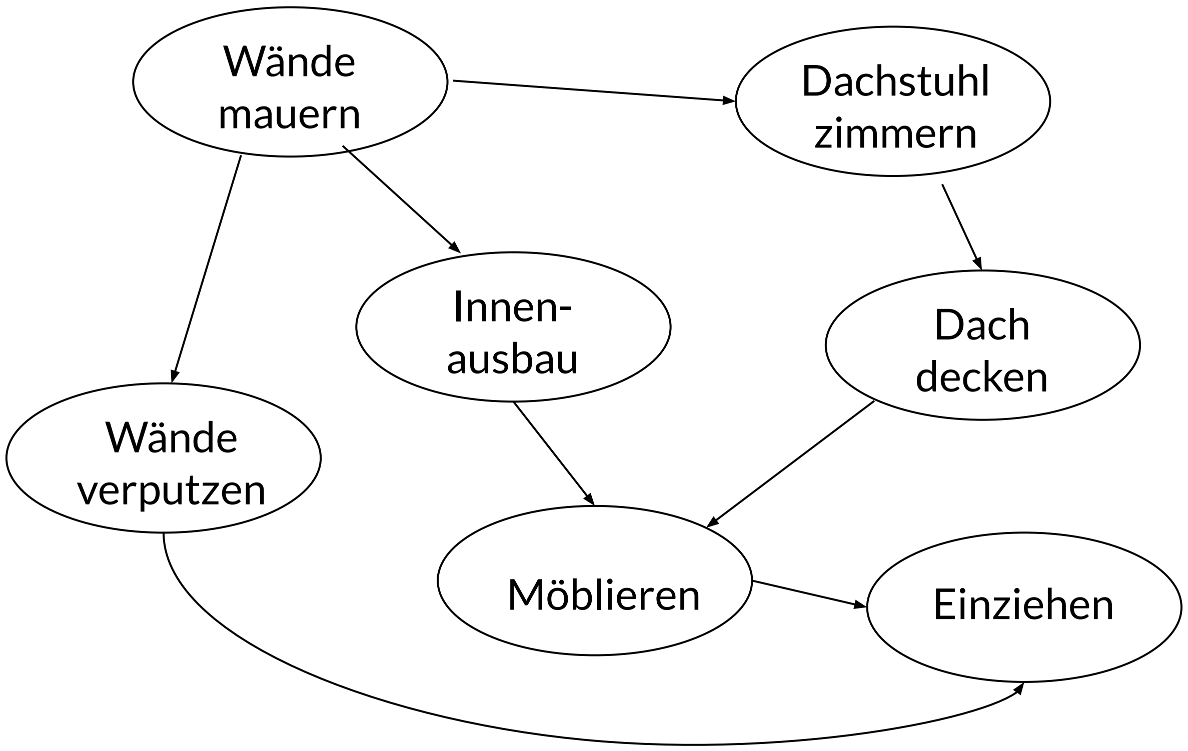
Topologisches Sortieren





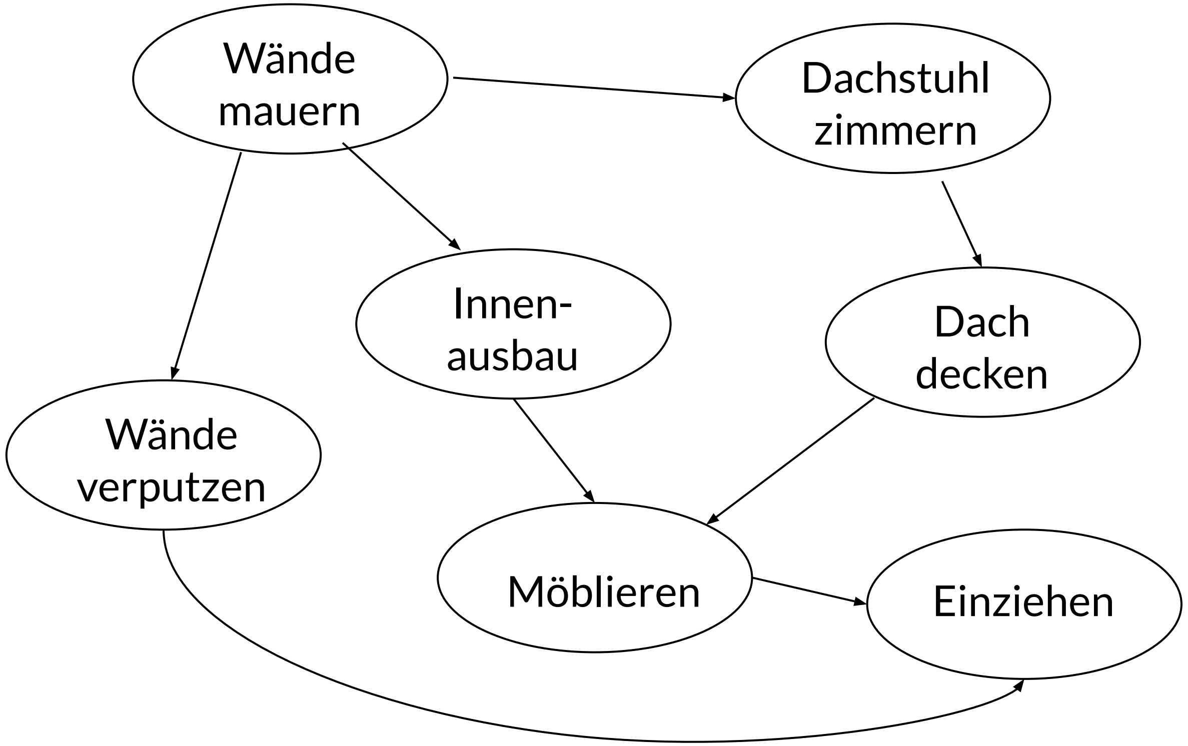
Definition

Eine *topologische Ordnung* eines gerichteten Graphen $G = (V, E)$ ist eine Reihenfolge $v_1, v_2, \ldots, v_n$ aller Knoten in V, so dass für jede Kante (v_i, v_j) in E gilt, dass $i < j$.



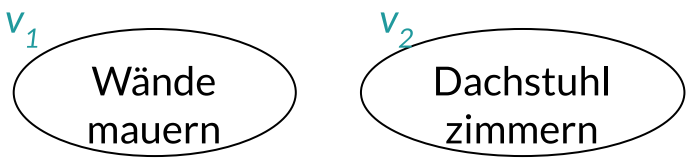
Definition

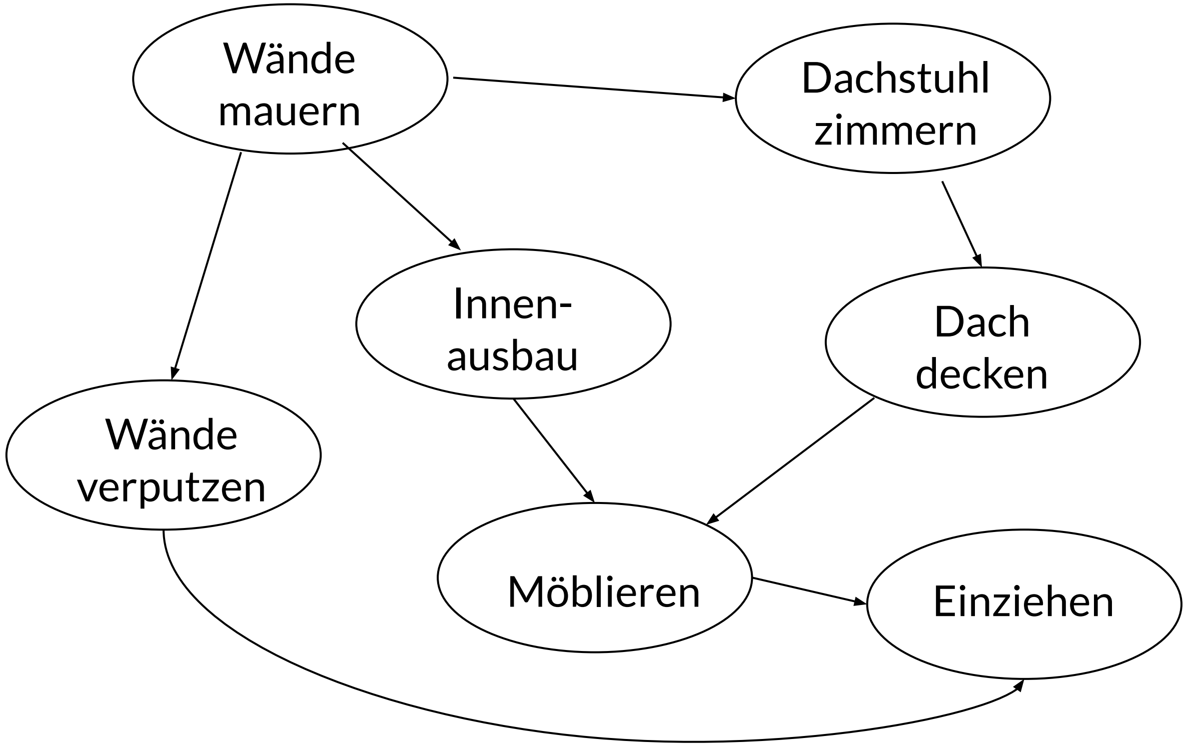
Eine *topologische Ordnung* eines gerichteten Graphen $G = (V, E)$ ist eine Reihenfolge $v_1, v_2, \dots, v_n$ aller Knoten in V, so dass für jede Kante (v_i, v_j) in E gilt, dass $i < j$.



Definition

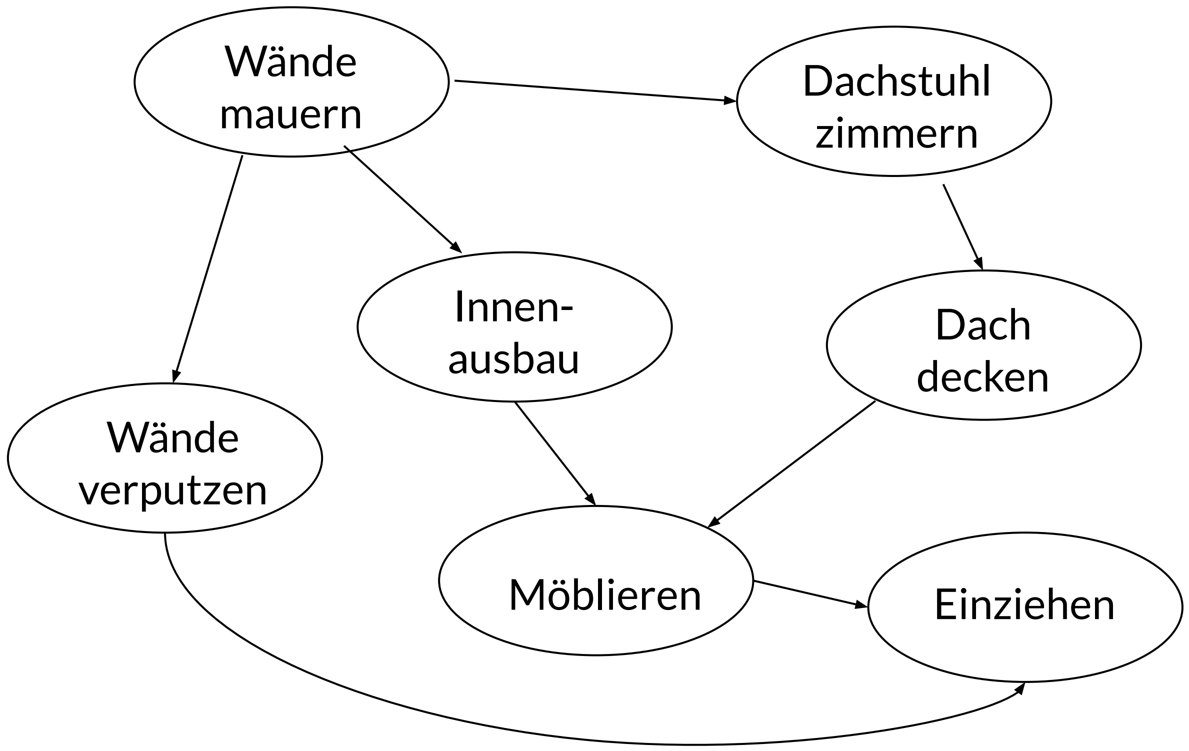
Eine *topologische Ordnung* eines gerichteten Graphen $G = (V, E)$ ist eine Reihenfolge $v_1, v_2, \ldots, v_n$ aller Knoten in V, so dass für jede Kante (v_i, v_j) in E gilt, dass $i < j$.





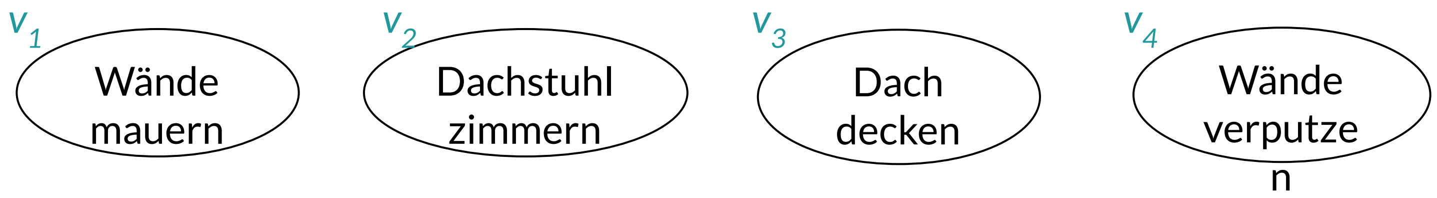
Definition

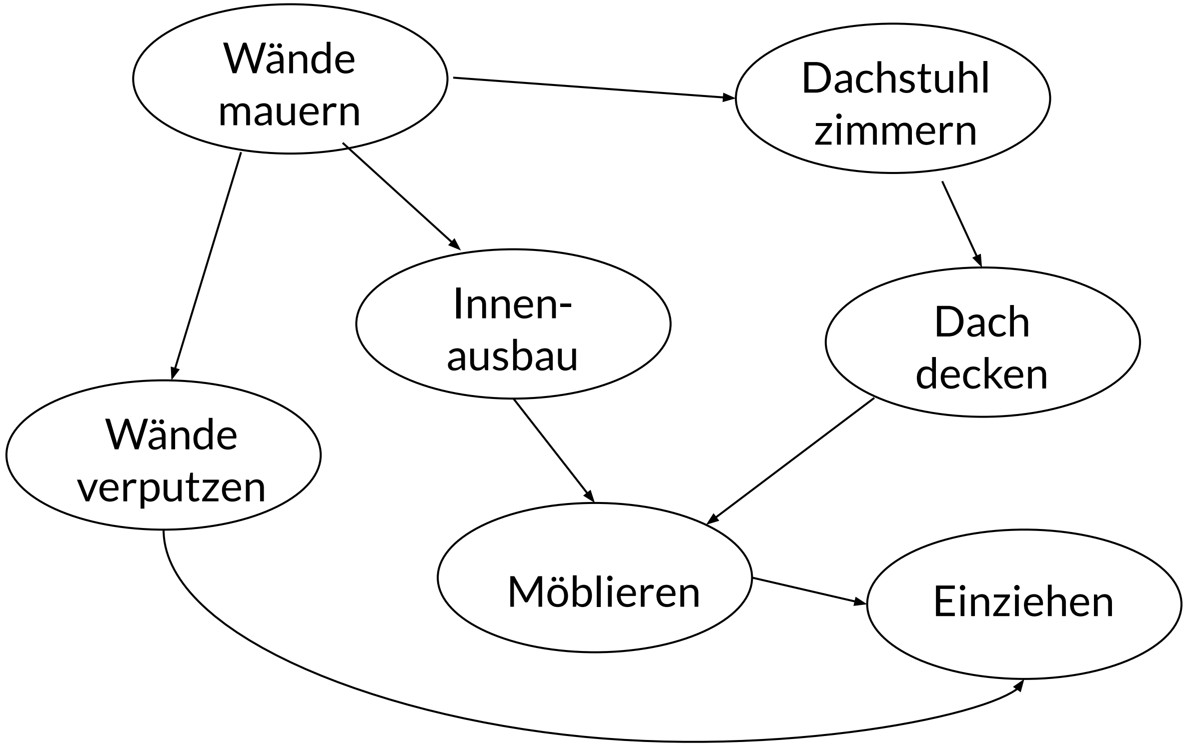
Eine *topologische Ordnung* eines gerichteten Graphen $G = (V, E)$ ist eine Reihenfolge $v_1, v_2, \ldots, v_n$ aller Knoten in V, so dass für jede Kante (v_i, v_j) in E gilt, dass $i < j$.



Definition

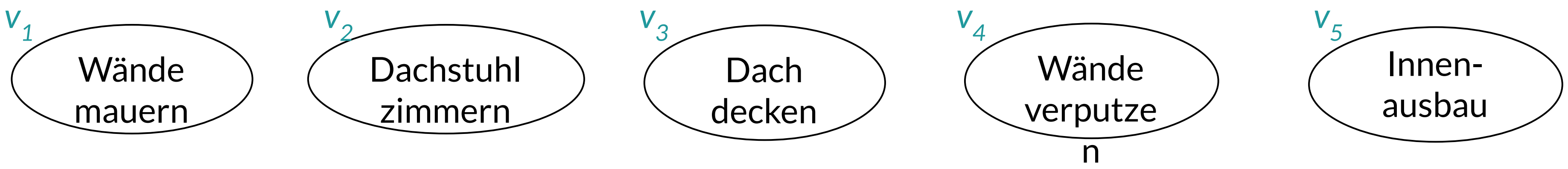
Eine *topologische Ordnung* eines gerichteten Graphen $G = (V, E)$ ist eine Reihenfolge $v_1, v_2, \ldots, v_n$ aller Knoten in V, so dass für jede Kante (v_i, v_j) in E gilt, dass $i < j$.

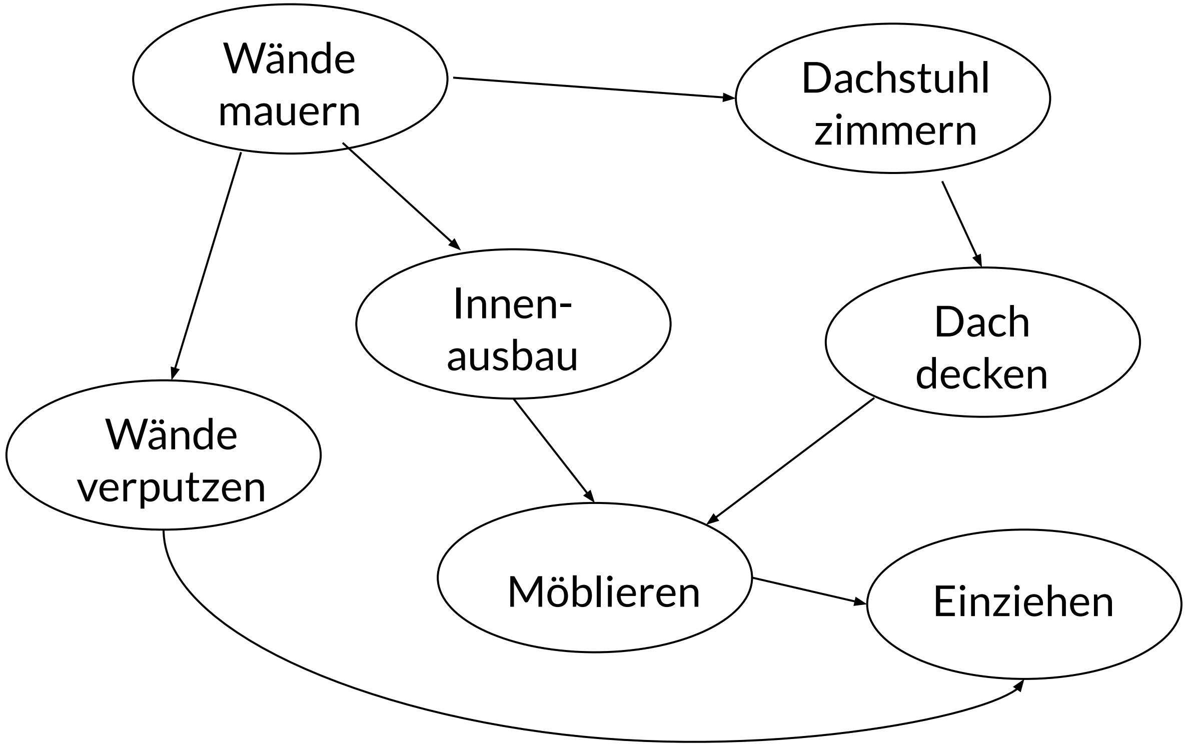




Definition

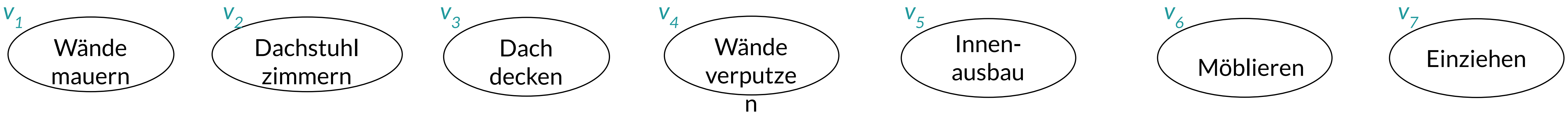
Eine *topologische Ordnung* eines gerichteten Graphen $G = (V, E)$ ist eine Reihenfolge $v_1, v_2, \ldots, v_n$ aller Knoten in V, so dass für jede Kante (v_i, v_j) in E gilt, dass $i < j$.

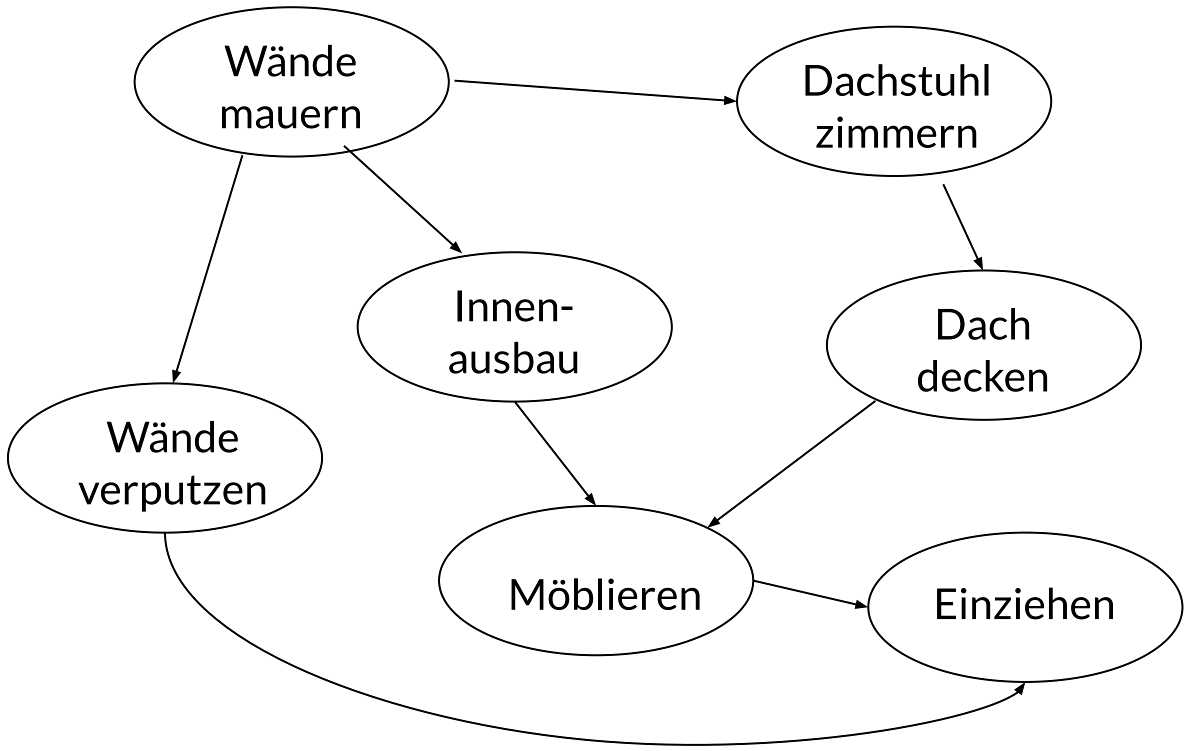




Definition

Eine *topologische Ordnung* eines gerichteten Graphen $G = (V, E)$ ist eine Reihenfolge $v_1, v_2, \ldots, v_n$ aller Knoten in V, so dass für jede Kante (v_i, v_j) in E gilt, dass $i < j$.

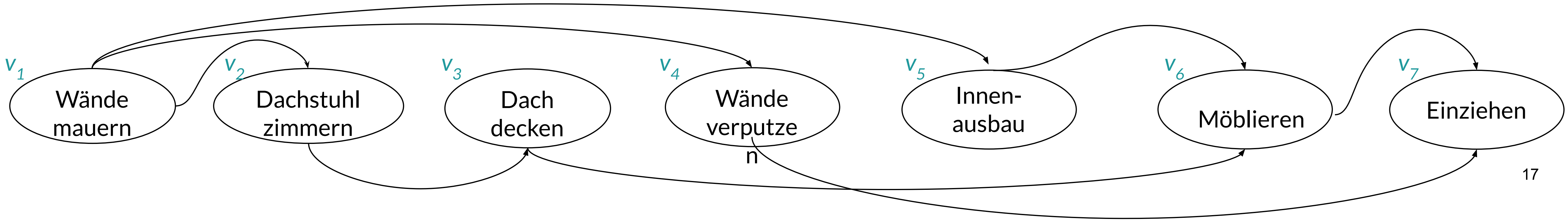




Definition

Eine *topologische Ordnung* eines gerichteten Graphen $G = (V, E)$ ist eine Reihenfolge $v_1, v_2, \ldots, v_n$ aller Knoten in V, so dass für jede Kante (v_i, v_j) in E gilt, dass $i < j$.

In einer topologischen Ordnung zeigen alle Kanten von links nach rechts:



Eigenschaften von topologischen Ordnungen

Beantworten Sie folgende Fragen (4 Minuten):

1. *Kann ein Graph mehrere topologische Ordnungen haben?*
2. *Kann ein nicht-zusammenhängender Graph eine topologische Ordnung haben?*
3. *Hat jeder gerichtete Graph eine topologische Ordnung?*
4. *Welche Eigenschaft muss ein Knoten haben, um in einer topologischen Ordnung zu an erster Position zu stehen?*

Eigenschaften von topologischen Ordnungen

1. *Kann ein Graph mehrere topologische Ordnungen haben?*
   Ja.
2. *Kann ein nicht-zusammenhängender Graph eine topologische Ordnung haben?*
   Ja.
3. *Hat jeder gerichtete Graph eine topologische Ordnung?*
   Nein. Nicht, wenn er einen Kreis enthält.
   Der Graph muss ein DAG sein (directed acyclic graph).
4. *Welche Eigenschaft muss ein Knoten haben, um in einer topologischen Ordnung zu an erster Position zu stehen?*
   Der Knoten muss Eingangsgrad (indegree) 0 haben.

Algorithmus

intuitiv

Solange der Graph Knoten hat:

1. Suche v in V mit Eingangsgrad 0
2. Füge v der topologischen Ordnung hinzu
3. Lösche v und alle ausgehenden Kanten

Algorithmus

intuitiv

Solange der Graph Knoten hat:

1. Suche v in V mit Eingangsgrad 0
2. Füge v der topologischen Ordnung hinzu
3. Lösche v und alle ausgehenden Kanten

- Klappt nur bei einem DAG, läuft sonst unendlich.
- Kanten wirklich zu löschen ist umständlich.

Algorithmus

0. Berechne indeg(v) für jeden Knoten v,

   füge v zu S, wenn indeg(v)=0

Solange S nicht leer ist:

1. Entferne einen Knoten v aus S
2. Füge v der topologischen Ordnung hinzu
3. Reduziere indeg(u) für alle Kanten <v,u> um 1 und aktualisiere S

Falls nicht alle Knoten in der topologischen Ordnung “landen”, enthält der Graph einen Kreis!

Algorithmus

0. Berechne indeg(v) für jeden Knoten v,
   füge v zu S, wenn indeg(v)=0

Solange S nicht leer ist:

1. Entferne einen Knoten v aus S
2. Füge v der topologischen Ordnung hinzu
3. Reduziere indeg(u) für alle Kanten <v,u> um 1 und aktualisiere S

Falls nicht alle Knoten in der topologischen Ordnung “landen”, enthält der Graph einen Kreis!

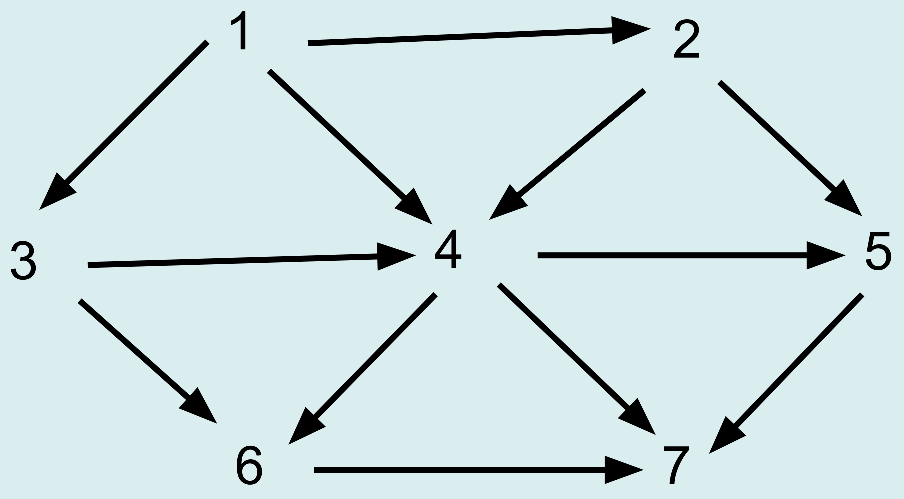
Aufwand (n Knoten, m Kanten, Adjazenzlisten)

- indeg berechnen für alle Knoten: **O(n + m)**
  alle Knoten mit indeg 0 in S einfügen: **O(n)**
- jede Kante wird 1x verfolgt, um den indeg des Zielknotens zu reduzieren: **O(m)**
  jeder Knoten wird einmal in S eingefügt und einmal gelöscht: **O(n)**

Total: O(m + n)

Aufgabe

Finden Sie alle Toplogischen Ordnungen des folgenden Graphen:



Aufgabe - Lösung

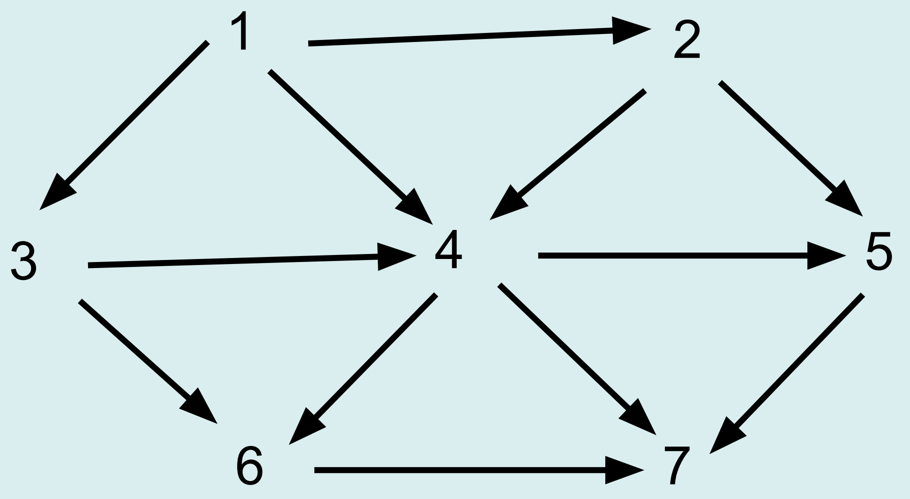
Finden Sie alle Toplogischen Ordnungen des folgenden Graphen:

1 - 2 - 3 - 4 - 5 - 6 - 7

1 - 3 - 2 - 4 - 5 - 6 - 7

1 - 2 - 3 - 4 - 6 - 5 - 7

1 - 3 - 2 - 4 - 6 - 5 - 7



Beispiel: einfacher Adjazenzmatrix-Graph

```
public class GraphI {
    private final boolean[][] adjMatrix;
    public final int n;

    public GraphI(int numNodes) {
        if (numNodes < 1) throw new IllegalArgumentException();
        this.adjMatrix = new boolean[numNodes][numNodes];
        this.n = numNodes;
    }

    public boolean addEdge(int u, int v){
        if(0<= u && u<n && 0 <= v && v<n){
            if(adjMatrix[u][v]) return false;
            adjMatrix[u][v] = adjMatrix[v][u] = true;
            return true;
        }
        throw new IndexOutOfBoundsException();
}   }
```

Aufgabe: Topsort in einfachem Adjazenzmatrix-Graph

```
public class GraphI {
    private final boolean[][] adjMatrix;
    public final int n;
     ...
    public int[] topsort(){
        // indegree berechnen



        // topsort init queue



        // topsort step









}   }
```

Lösung: Topsort in einfachem Adjazenzmatrix-Graph

```
public class GraphI {
    private final boolean[][] adjMatrix;
    public final int n;
     ...
    public int[] topsort(){
        int[] indeg = new int[n];                               // indegree berechnen
        for (int i=0; i<n;i++){
            for (int j=0; j<n; j++){
                indeg[i] += adjMatrix[j][i] ? 1:0;
        }   }
        Queue S = new LinkedList();                              // topsort init queue
        for (int i=0; i<n; i++){
            if(indeg[i]==0) S.add(i);
        }
        int[] result = new int[n];                               // topsort step
        for(int i=0; i<n; i++){
            if(!S.isEmpty()){
                result[i] = (int) S.remove();
                for(int j=0; j<n; j++){
                    if(adjMatrix[result[i]][j]) {
                        indeg[j]--;
                        if(indeg[j] == 0) S.add(j);
            }   }   }
            return null;
        }
        return result;
}   }
```

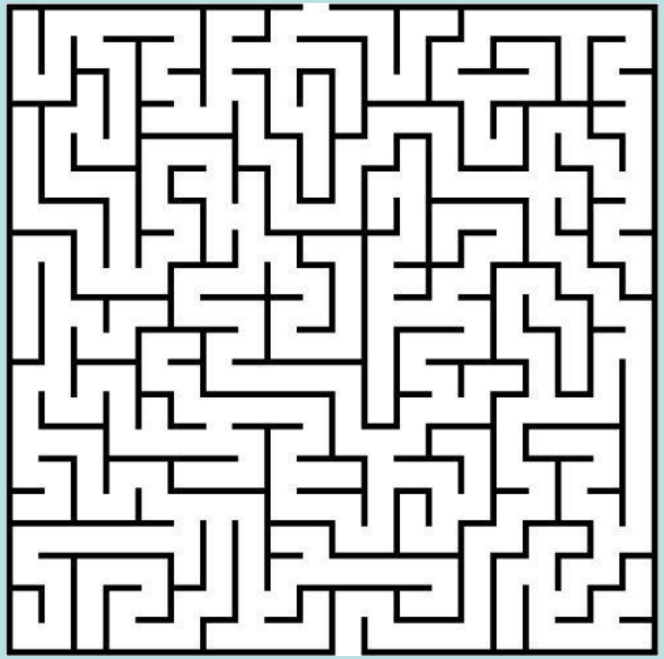
Fachhochschule Nordwestschweiz

Hausaufgaben

- Programmieren 1 (Adjazenzlisten und TopSort)

Fachhochschule Nordwestschweiz

Graphtraversierungen



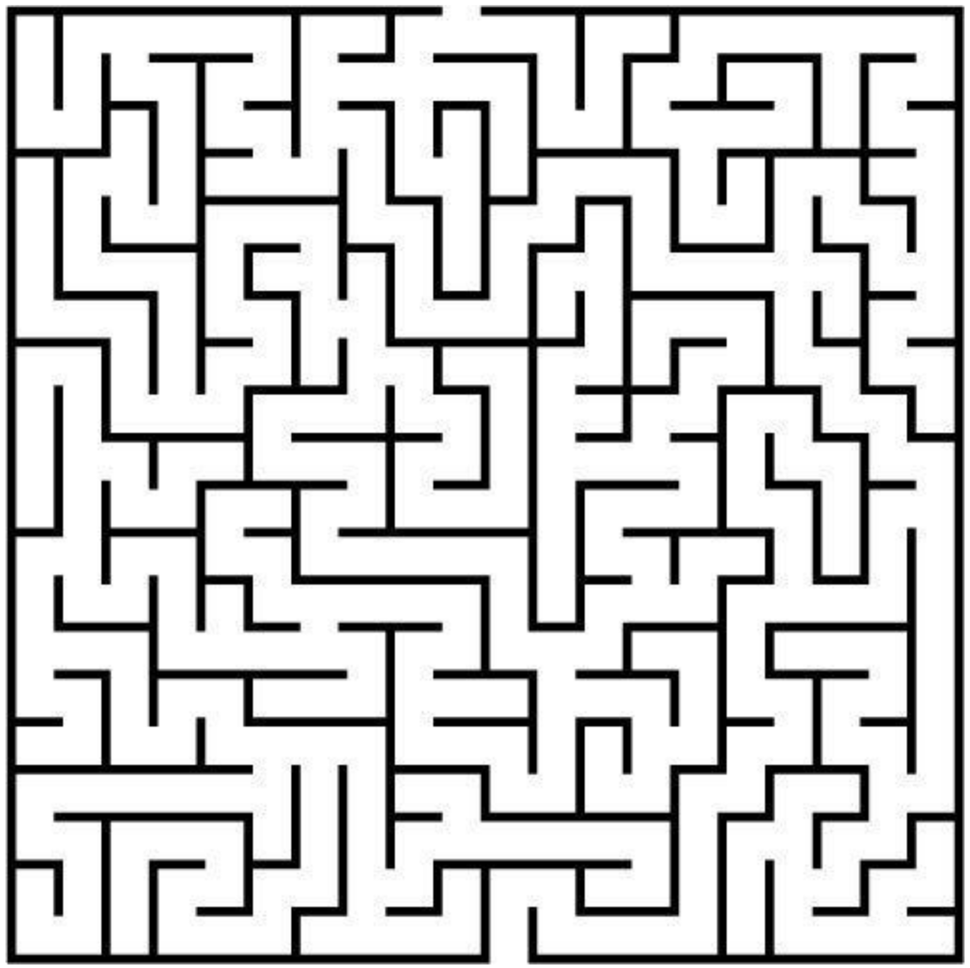
Graphtraversierung

- Systematisches Durchlaufen eines Graphen
- Viele Möglichkeiten,
  zwei Spezialfälle (*Tiefensuche* und *Breitensuche*)

Beispiel Labyrinth

Einzelne Person ~ Tiefensuche

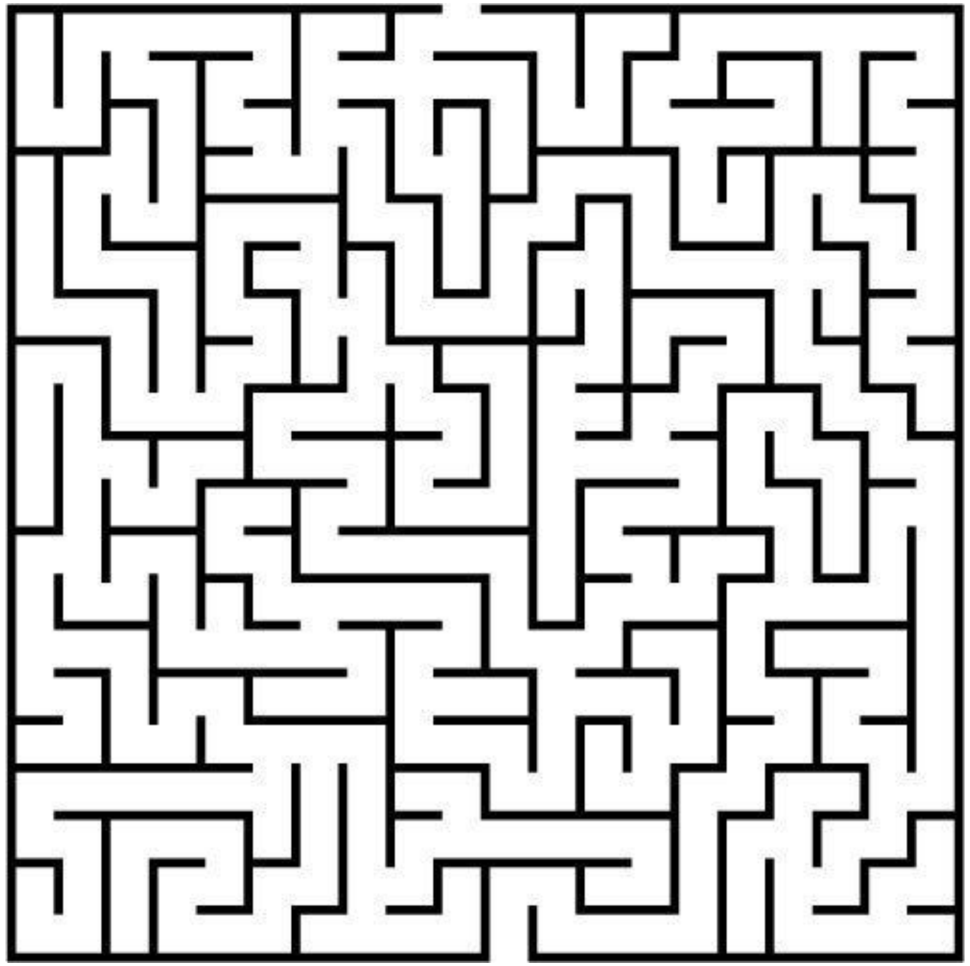
Suchtrupp ~ Breitensuche



Graphtraversierung

Wozu?

- Tiefensuche kann testen, ob in einem Computer-netzwerk der Ausfall einer Verbindung das ganze Netzwerk “disconnectet”.
- Breitensuche kann die kürzesten Wege von einem Knoten zu allen anderen Knoten finden.
- Traversierungen können auch eingesetzt werden, um einen Graph überhaupt erst zu erforschen.



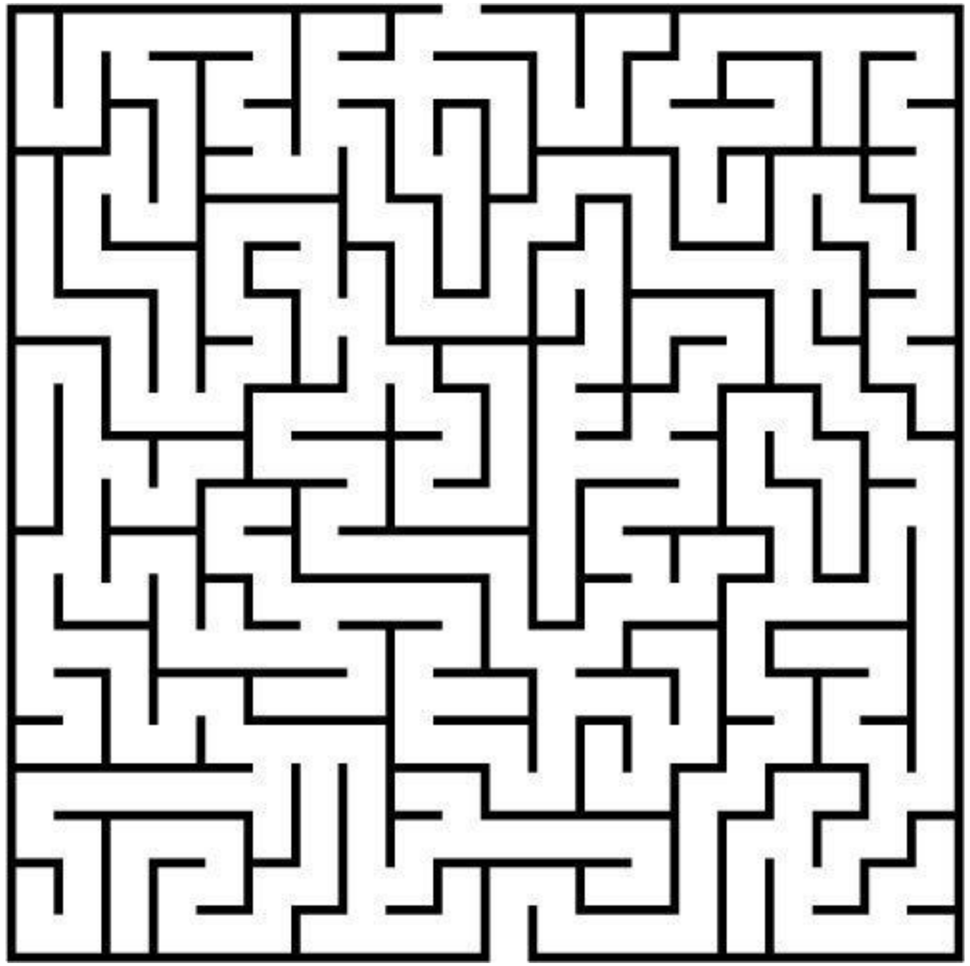
Graphtraversierung

Was ist anders als bei Bäumen?

- Ein Graph kann Zyklen enthalten
- Es gibt oft mehrere Pfade zwischen zwei Knoten

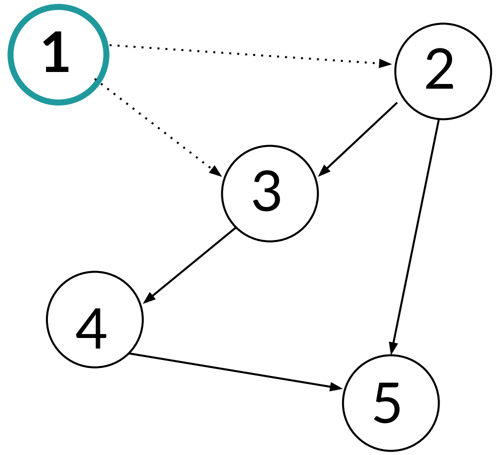
➡ Aufpassen, dass wir:
keine Knoten mehrfach besuchen und
keine Kanten mehrfach benutzen!

z.B. in Java: boolean *visited* in der Klasse Vertex



Fachhochschule
Nordwestschweiz

s =



B	1
R	1
O	1

Grundprinzip

s Startknoten

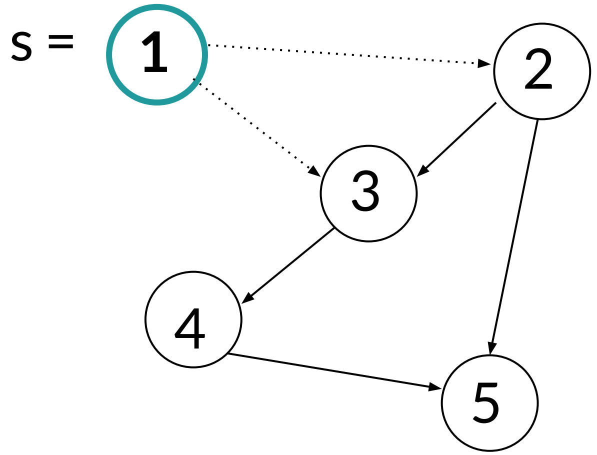
B Menge aller besuchten (gefundenen) Knoten

R Teilmenge aller Knoten in B, von denen unbenutzte Kanten ausgehen (*Rand von B*)

O Reihenfolge, in der die Knoten besucht werden

1. Füge Startknoten **s** zum Rand **R** und setze **s.visited = true**
2. Solange **R** nicht leer ist,
   betrachte einen (beliebigen) Knoten **v** in **R**
   a. Falls aus **v** keine unbenutzte Kante führt, lösche **v** aus **R**.
   b. Sonst, folge einer noch unbenutzten Kante **<v,w>**.
      i. Falls !**w.visited**, füge **w** zu **R** hinzu und setze **w.visited=true**

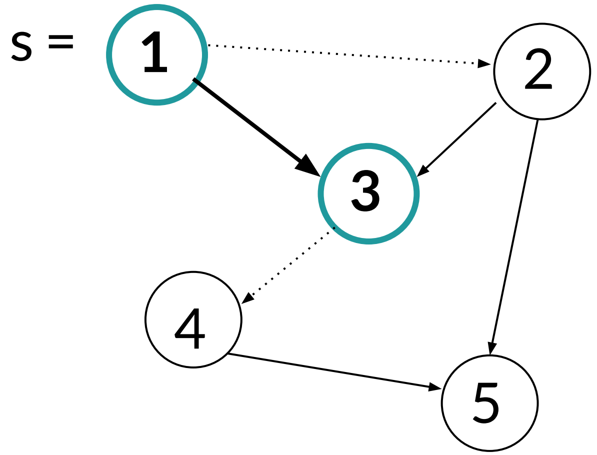
Grundprinzip



B	1
R	1
O	1

s Startknoten

B Menge aller besuchten (gefundenen) Knoten



B	1, 3
R	1, 3
O	1, 3

R Teilmenge aller Knoten in B, von denen unbenutzte Kanten ausgehen (*Rand von B*)

O Reihenfolge, in der die Knoten besucht werden

Fachhochschule
Nordwestschweiz

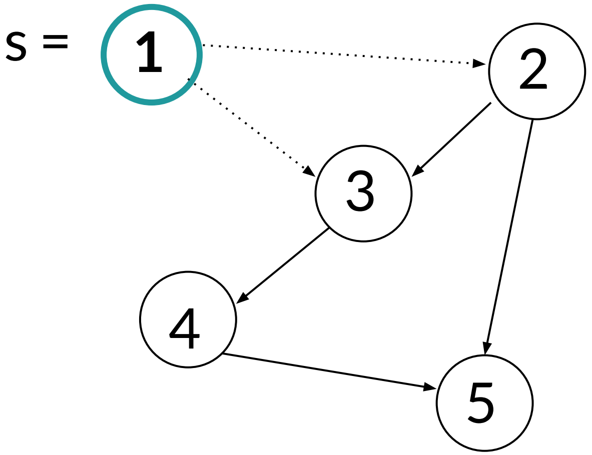
Grundprinzip

s Startknoten

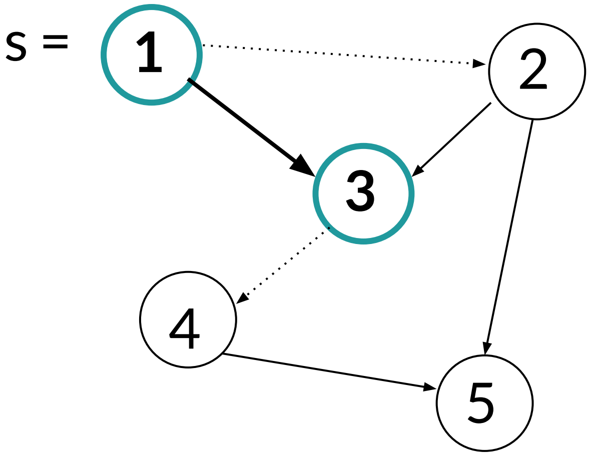
B Menge aller besuchten (gefundenen) Knoten

R Teilmenge aller Knoten in B, von denen unbenutzte Kanten ausgehen (*Rand von B*)

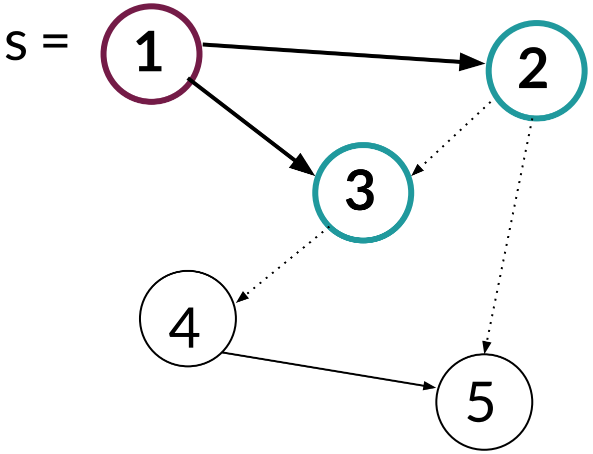
O Reihenfolge, in der die Knoten besucht werden



B	1
R	1
O	1



B	1, 3
R	1, 3
O	1, 3



B	1, 3, 2
R	3, 2
O	1, 3, 2

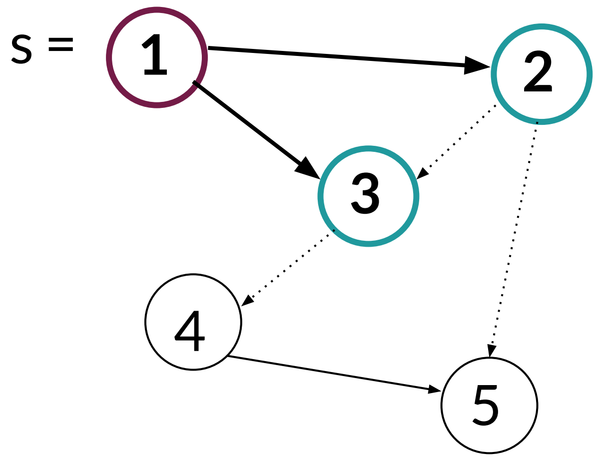
Grundprinzip

s Startknoten

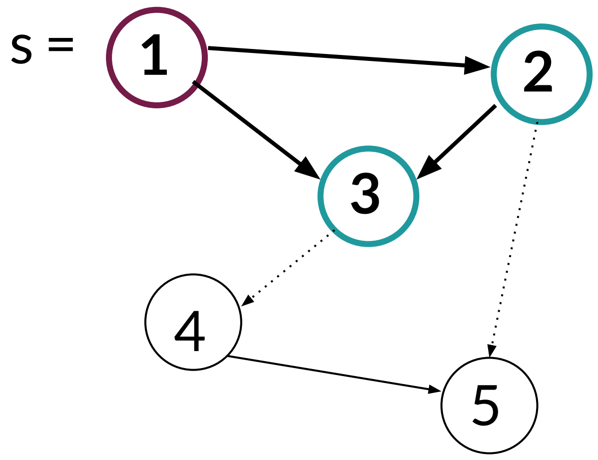
B Menge aller besuchten (gefundenen) Knoten

R Teilmenge aller Knoten in B, von denen unbenutzte Kanten ausgehen (*Rand von B*)

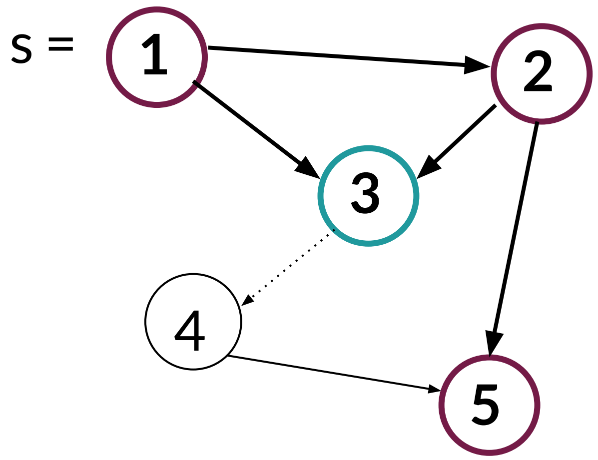
O Reihenfolge, in der die Knoten besucht werden




```
B   1, 3, 2
R      3, 2
O   1, 3, 2
```




```
B   1, 3, 2
R      3, 2
O   1, 3, 2
```




```
B   1, 3, 2, 5
R      3,    5
O   1, 3, 2, 5
```

Grundprinzip - “nimm einen (beliebigen) Knoten v im Rand, folge einer unbenutzten Kante <v,w>**”**

1. Füge Startknoten **s** zum Rand **R** und setze **s.visited = true**
2. Solange **R** nicht leer ist, betrachte einen (beliebigen) Knoten **v** in **R**
   a. Falls aus **v** keine unbenutzte Kante führt, lösche **v** aus **R**.
   b. Sonst, folge einer noch unbenutzten Kante **<v,w>**.
      i. Falls !**w.visited**, füge **w** zu **R** hinzu und setze **w.visited=true**

Aufwand (n Knoten, m Kanten)

mit Adjazenzlisten

- Jeder Kante wird 1x gefolgt: O(m)
- Jeder Knoten wird 1x in R eingefügt und einmal aus R gelöscht: O(n)

Total: O(n + m)

Aufwand mit Adjazenzmatrix

- Die ganze Matrix muss durchlaufen werden, um alle Kanten zu finden: mindestens O(n^2)

Grundprinzip - “nimm einen (beliebigen) Knoten v im Rand, folge einer unbenutzten Kante <v,w>”

1. Füge Startknoten **s** zum Rand **R** und setze **s.visited = true**
2. Solange **R** nicht leer ist, betrachte einen (beliebigen) Knoten **v** in **R**
   a. Falls aus **v** keine unbenutzten Kanten führt, lösche **v** aus **R**.
   b. Sonst, folge einer noch unbenutzten Kante **<v,w>**.
      i. Falls !**w.visited**, füge **w** zu **R** hinzu und setze **w.visited=true**

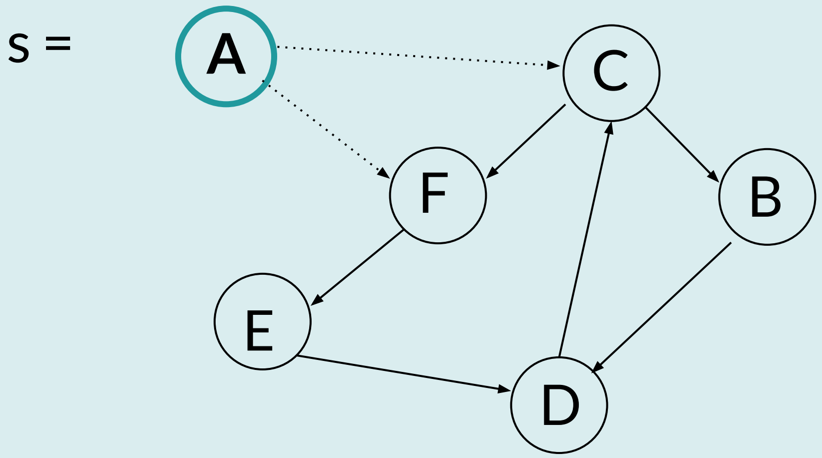
Tiefensuche
- ***Stack*** für das Speichern der Randknoten
- Setzt Traversierung im ***zuletzt*** gefundenen Knoten fort

Breitensuche
- ***Queue*** für das Speichern der Randknoten
- Setzt Traversierung im ***zuerst*** gefundenen Knoten fort

Fachhochschule Nordwestschweiz

Aufgabe

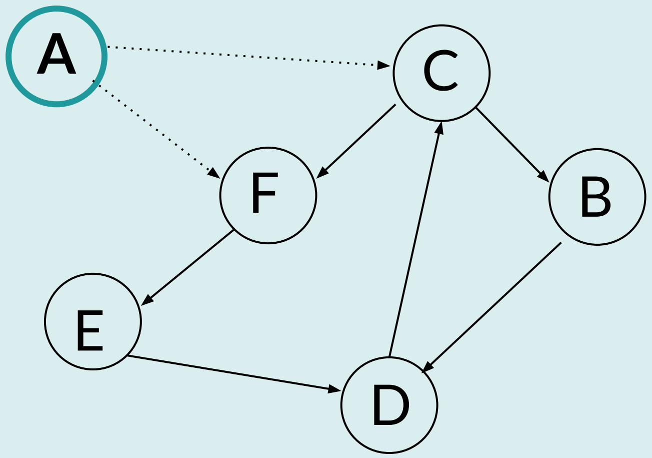
Führen Sie auf dem folgenden Graphen jeweils eine Tiefensuche und eine Breitensuche durch und notieren Sie sich die Reihenfolge, in der Sie die Knoten besuchen. (Für eine eindeutige Lösung: Besuchen Sie die Nachbarn eines Knotens jeweils in alphabetischer Reihenfolge.)



Aufgabe - Lösung

Führen Sie auf dem folgenden Graphen jeweils eine Tiefensuche und eine Breitensuche durch und notieren Sie sich die Reihenfolge, in der Sie die Knoten besuchen. (Für eine eindeutige Lösung: Besuchen Sie die Nachbarn eines Knotens jeweils in alphabetischer Reihenfolge.)

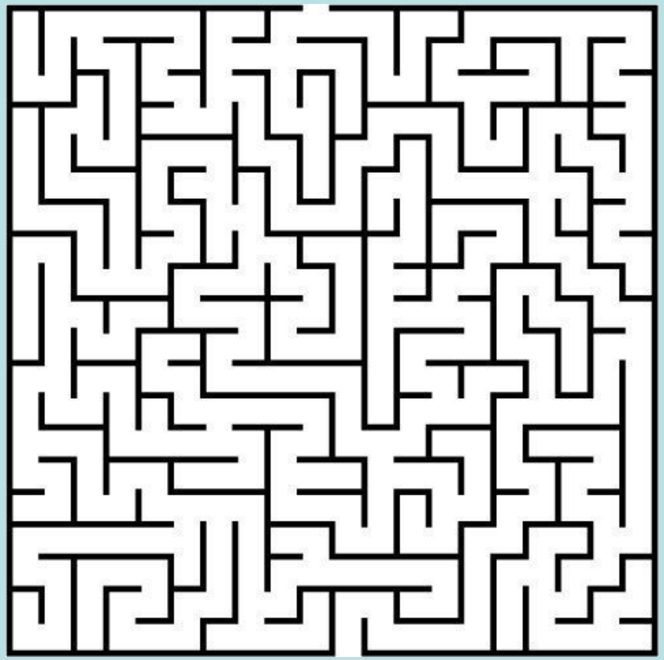
s =



Tiefenordnung: A - C - B - D - F - E

Breitenordnung: A - C - F - B - E - D

Graphtraversierungen - Breitensuche (BFS)



Breitensuche

- ***Queue*** für das Speichern der Randknoten
- Setzt Traversierung im ***zuerst*** gefundenen Knoten fort
- Es werden immer zuerst alle Nachbarn eines Knotens besucht, bevor vom nächsten Knoten aus weitergesucht wird.

Vereinfachung Grundprinzip:

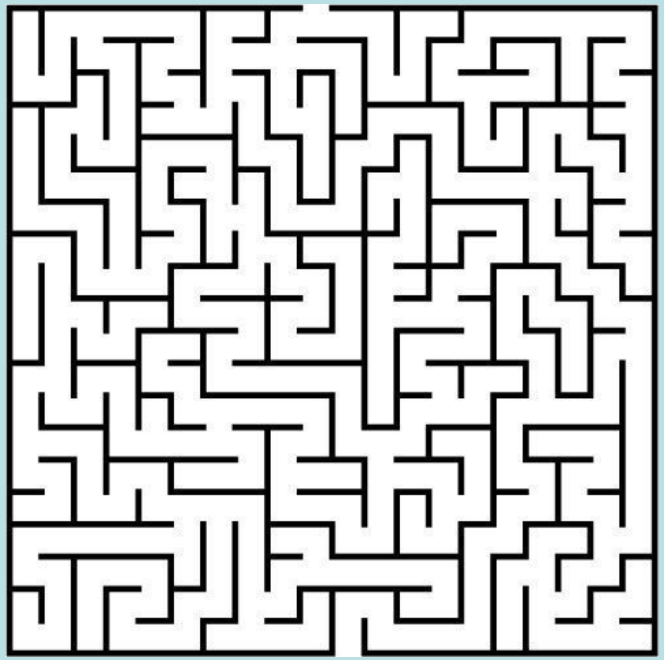
- Statt: “nimm einen (beliebigen) Knoten v im Rand, folge einer unbenutzten Kante <v,w>”
- **Neu: “nimm den *ersten* Knoten v im Rand, folge nacheinander allen Kanten <v,w>”**

Breitensuche

```
void BFS(Vertex s) {
     Queue<Vertex<K>> R = new LinkedList<Vertex<K>>();
     print(v); s.visited = true;
     R.add(s);

     while(!R.isempty()) {
       Vertex v = R.remove();
          for(Vertex w : v.adjList) { // benutze alle Kanten
          if(!w.visited) {
             print(w); w.visited = true;
             R.add(w);
}  }  }  }
```

Graphtraversierungen - Tiefensuche (DFS)



Tiefensuche

- ***Stack*** für das Speichern der Randknoten
- Setzt Traversierung im ***zuletzt*** gefundenen Knoten fort

Stack kann durch rekursive Aufrufe “ersetzt” werden.

- Statt: “füge w in den Rand R”
- **Neu: “rufe rekursiv dfs(w) auf”**

Tiefensuche

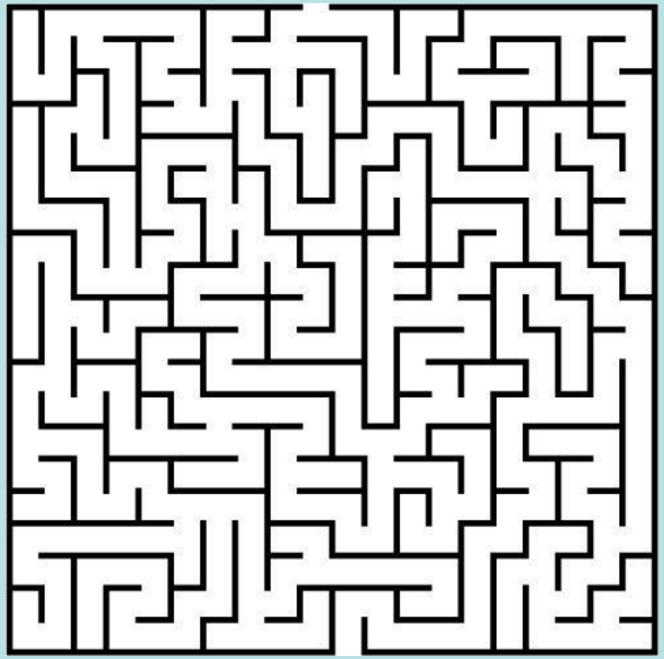
```
void dfs(Vertex v) {
   print(v); v.visited=true;
     for (Vertex w : v.adjList) {
          if (!w.visited) {
               dfs(w);
}  }  }


void dfs_variante(Vertex v) {
     if (!v.visited) {
          print(v); v.visited = true;
          for (Vertex w : v.adjList) {
             dfs_variante(w);
}  }  }
```

Aufgabe

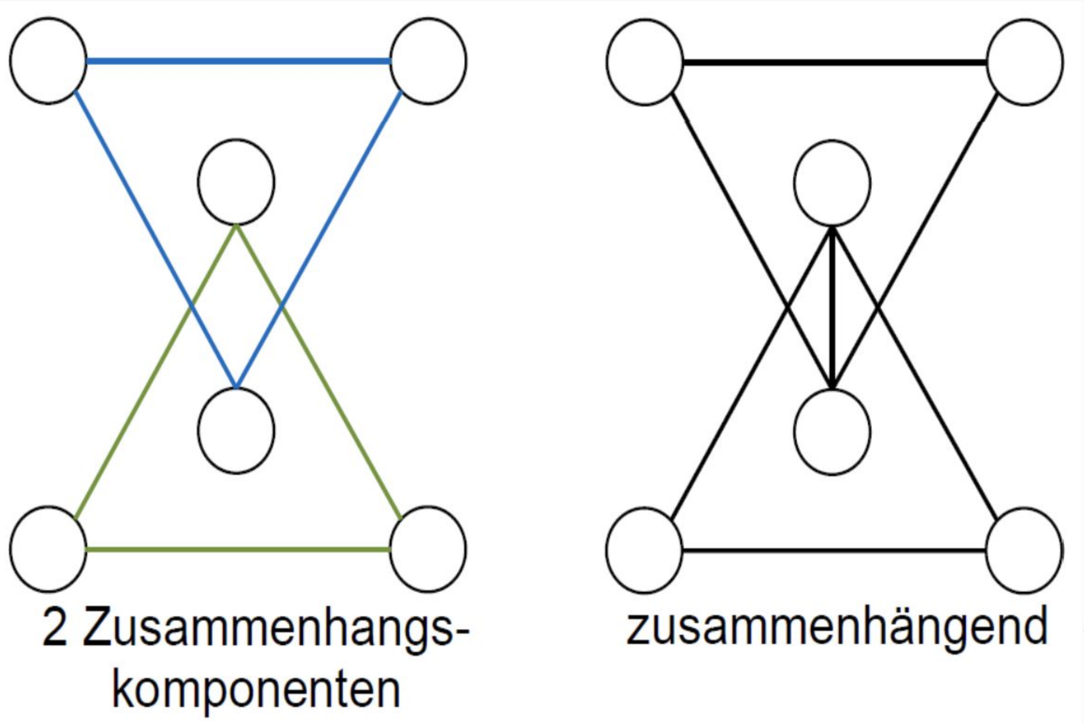
Lösen Sie die Aufgabe 4 auf Arbeitsblatt 1.

Anwendungen für ungerichtete Graphen



Zusammenhangskomponenten

Definition: Ein ungerichteter Graph ist zusammenhängend, falls es für jedes Paar von verschiedenen Knoten einen verbindenden Pfad gibt.

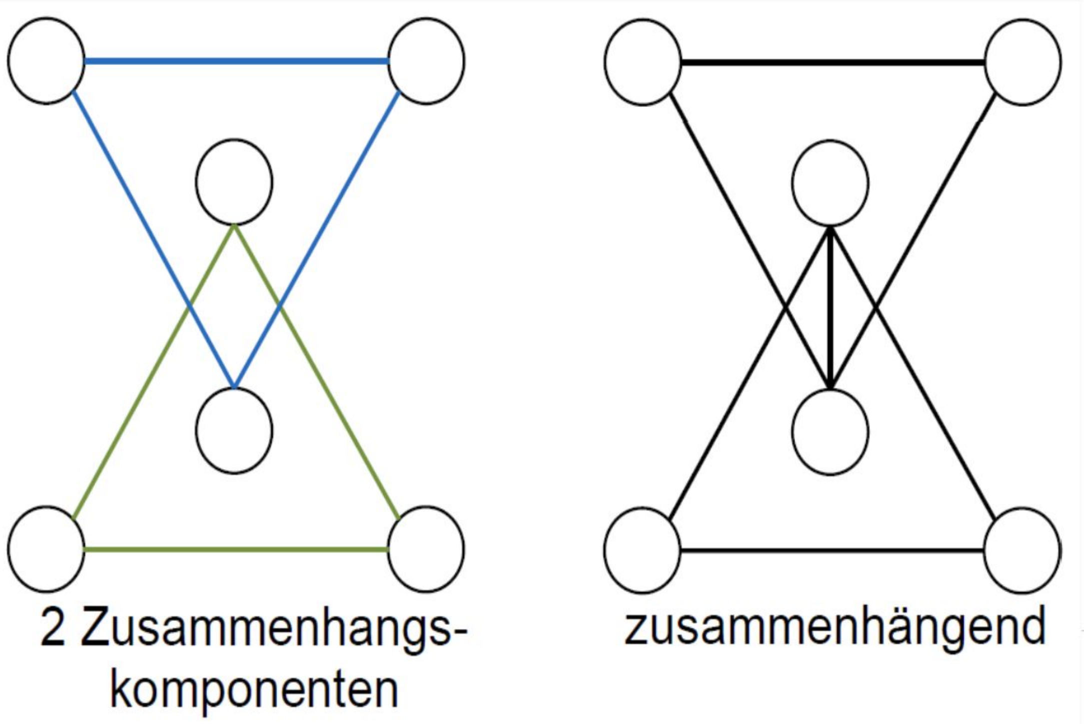


Zusammenhangskomponenten

Zusammenhängigkeit prüfen mit DFS:

1. Alle Knoten als bisher nicht besucht markieren
2. dfs(v) für irgendeinen Knoten v aufrufen
3. Sequenzielle Suche in der Menge aller Knoten nach dem ersten, der nicht besucht wurde

Wird in Schritt 3 kein solcher Knoten gefunden, ist der Graph zusammenhängend.



Fachhochschule Nordwestschweiz

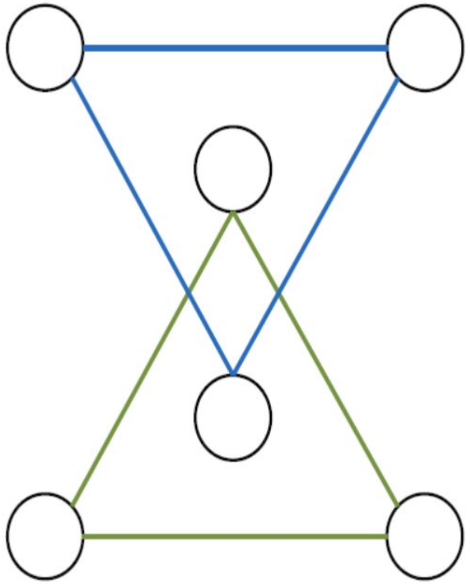
Zusammenhangskomponenten

Alle Komponenten finden mit DFS:

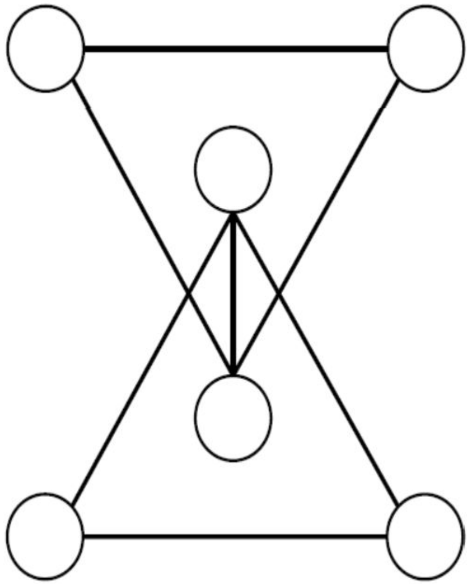
1. Alle Knoten als bisher nicht besucht markieren
2. dfs(v) für irgendeinen Knoten v aufrufen
3. Sequenzielle Suche in der Menge aller Knoten nach dem ersten, der nicht besucht wurde

Findet die Suche in Schritt 3 einen noch nicht besuchten Knoten w, wird erneut dfs(w) gestartet.

Die Anzahl solcher dfs-Aufrufe bis alle Knoten besucht sind, ergibt die Anzahl der Zusammenhangskomponenten des Graphen.



2 Zusammenhangs-komponenten

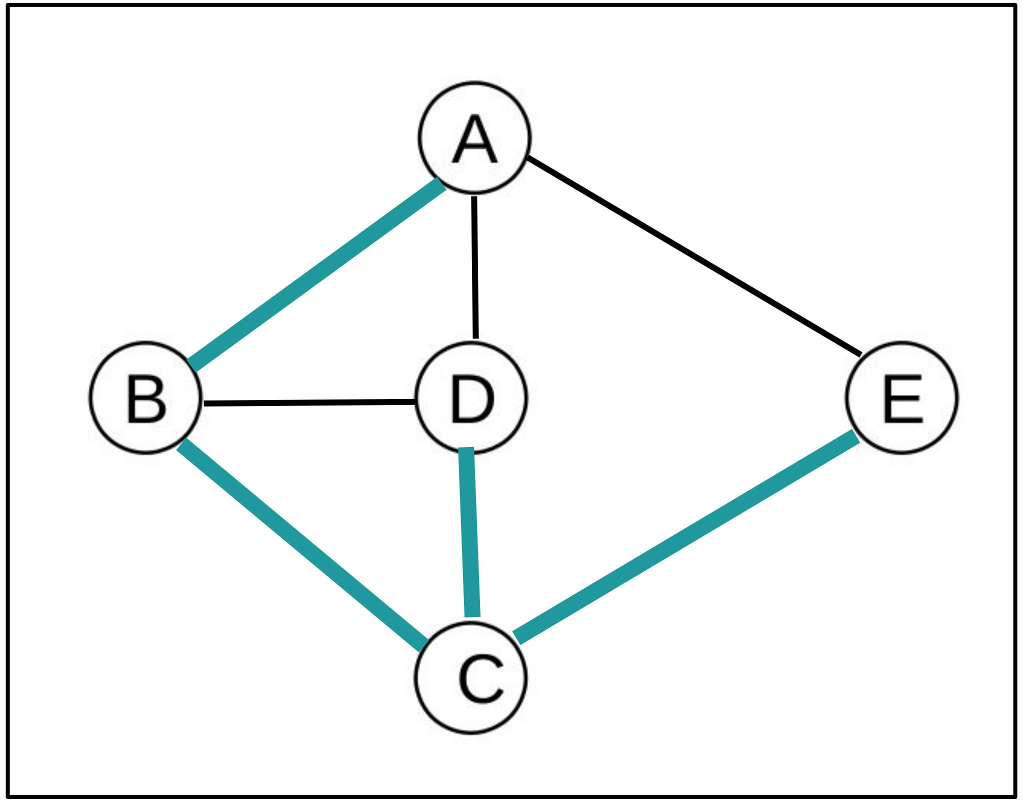


zusammenhängend

Spannbaum

Definition: Ein Spannbaum ist ein Teilgraph eines Graphen, der ein Baum ist und alle Knoten dieses Graphen enthält.

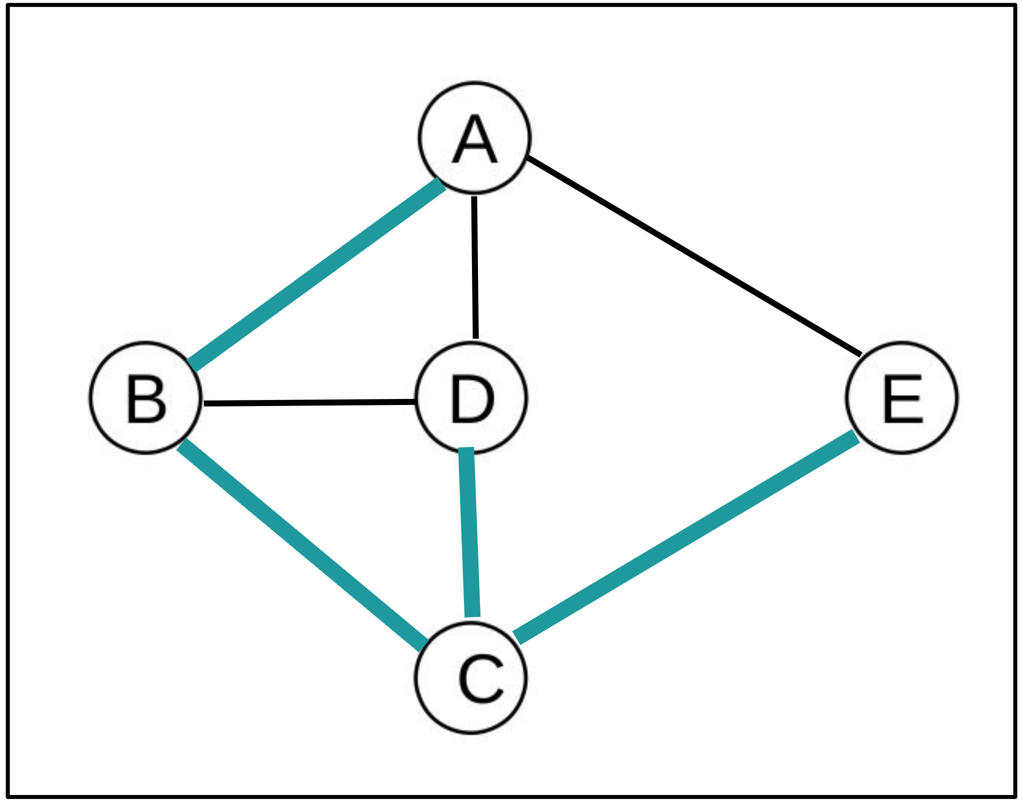
Spannbäume existieren nur in zusammenhängenden Graphen.



Spannbaum

Spannbaum finden mit mit DFS:

Speichert man alle Kanten, die DFS benutzt, um zu bisher nicht besuchten Knoten zu kommen, erhält man einen Baum, der alle vom Startknoten aus erreichbaren Knoten des Graphen enthält, sowie die kleinstmögliche Menge an Kanten um diese zu verbinden.



Spannbaum

```
void dfs(Vertex v) {
   v.visited=true;
     for (Vertex w : v.adjList) {
          if (!w.visited) {
              tree.add(“<v,w>”);
               dfs(w);
}  }  }
```

Aufgabe

Programmieren 2, Aufgabe 1 (DFS + Spannbaum)

- Benützen Sie dafür das Gerüst: “AdjListGraph - mit DFS Gerüst”

Hausaufgaben

- Repetieren und Fragen mitbringen.