8. Elementare Algorithmen für Graphen

- Tiefensuche
- Breitensuche
- Tiefen- und Breitensuche bei mehreren Zusammenhangskomponenten
- Bipartiter Graph
- Topologisches Sortieren

Rekursive Tiefensuche (depth-first search)

```
void visitDF(Vertex v, Graph g)  {
      Set<Vertex> besucht = ∅;
      visitDF(v, g, besucht);
}


void visitDF(Vertex v, Graph g, Set<Vertex> besucht) {
      besucht.add(v);

      // Bearbeite v:
      println(v);

      for ( jeden adjazenten Knoten w von v )
            if ( ! besucht.contains(w) )  // w noch nicht besucht
                  visitDF(w, g, besucht);
}
```

visitDF(v,g) startet von Knoten v eine Tiefensuche im Graph g.

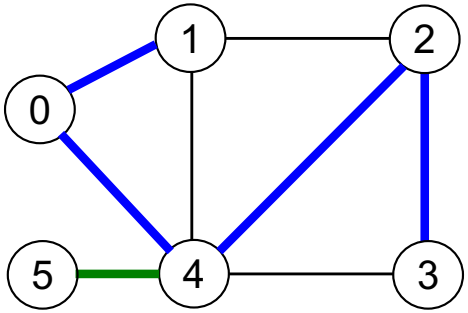
besucht ist die Menge aller bereits besuchten Knoten.

Wichtig zur Vermeidung von Endlosschleifen.

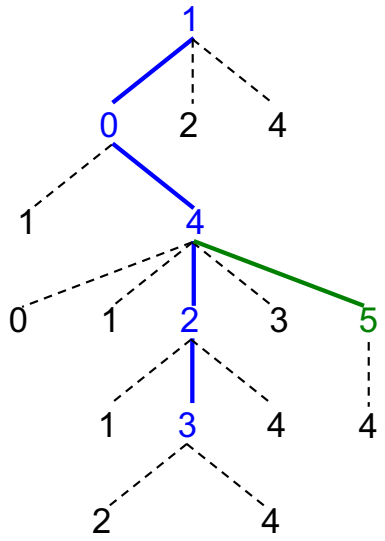
visitDF(v, g, besucht)
besucht Knoten v im Graphen g und ist rekursiv.

Beispiel für rekursive Tiefensuche

- Tiefensuche mit Start bei Knoten 1



Aufrufstruktur

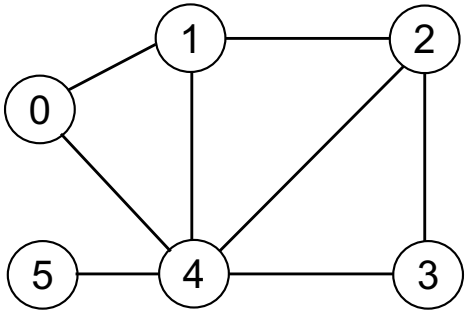


Besuchsreihenfolge:
1, 0, 4, 2, 3, 5

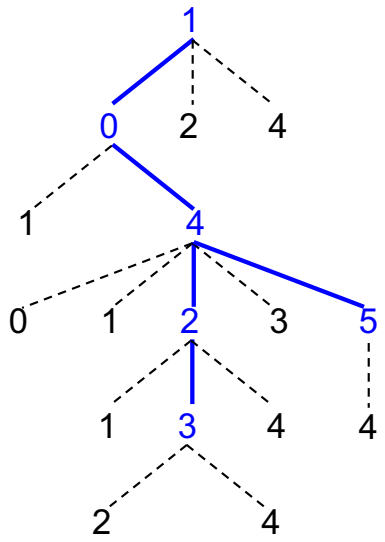
- die Nachbarn eines Knoten werden in numerischer Reihenfolge durchlaufen.
- Bereits besuchte Nachbarknoten sind durch eine gestrichelte Kante verbunden und gehören nicht zur Aufrufstruktur.

Tiefensuchbaum

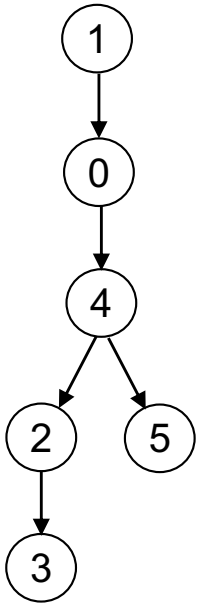
- Die Aufrufstruktur bildet mit den besuchten Knoten den sogenannten Tiefensuchbaum.



Aufrufstruktur



Tiefensuchbaum



Iterative Tiefensuche mit einem Keller

```
void visitDF(Vertex v, Graph g)  {
        Set<Vertex> besucht = ∅;
        visitDF(v, g, besucht);
}

void visitDF(Vertex v, Graph g, Set<Vertex> besucht) {
        Stack<Vertex> stk;
        stk.push(v);

        while( ! stk.empty() )  {
                v = stk.pop();
                if ( besucht.contains(v) )
                        continue;

                besucht.add(v);

                // Bearbeite v:
                println(v);

                for ( jeden adjazenten Knoten w von v )
                        if ( ! besucht.contains(w) )
                                stk.push(w);
        }
}
```

visitDF startet von Knoten v eine Tiefensuche.

visitDF besucht Knoten v im Graphen g, wobei die bereits besuchten Knoten in besucht abgespeichert werden.

Im Keller stk werden alle Knoten verwaltet, die als nächstes zu besuchen sind.

Die Tiefensuche wird erreicht durch die LIFO-Organisation des Kellers.

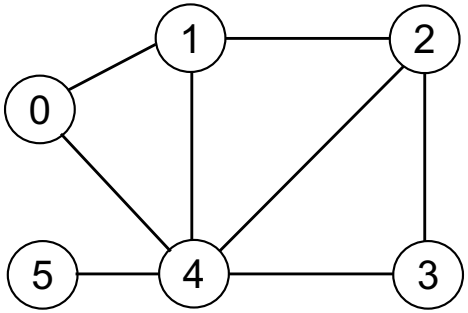
Um die gleiche Besuchsreihenfolge wie bei der rekursiven Funktion zu erreichen, müssen in der for-Schleife die Nachbarn in umgekehrter Reihenfolge bearbeitet werden.

Beachte: Gleiche Knoten können mehrfach eingekellert werden. Das ließe sich durch eine weitere Knotenmarkierung verhindern:

- nicht besucht und nicht im Keller,
- nicht besucht und im Keller,
- besucht.

Allerdings würde sich eine andere Besuchs-reihenfolge wie bei der rekursiven Funktion ergeben.

Tiefensuche mit Stack



Keller

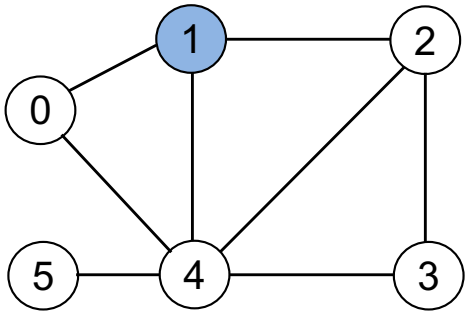
Besuchsreihenfolge:
1, 0, 4, 2, 3, 5

Bitte ausfüllen

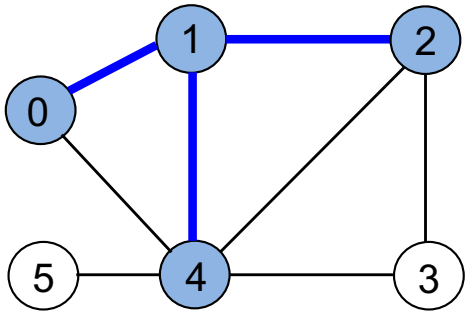
8. Elementare Algorithmen für Graphen

- Tiefensuche
- Breitensuche
- Tiefen- und Breitensuche bei mehreren Zusammenhangskomponenten
- Bipartiter Graph
- Topologisches Sortieren

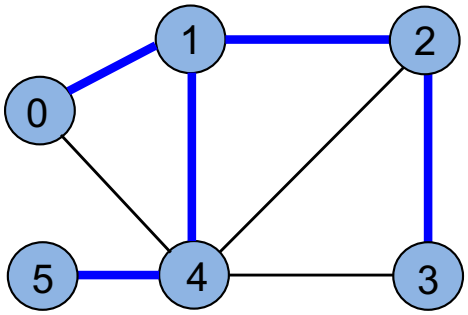
Breitensuche - Beispiel



1) Beginne mit Knoten 1.



2) Besuche alle Knoten, die über genau eine Kante von Knoten 1 erreichbar sind.



3) Besuche alle Knoten, die über genau 2 Kanten von Knoten 1 erreichbar sind.

Besuchsreihenfolge:
1, 0, 2, 4, 3, 5

Breitensuche (breadth-first search) mit einer Schlange

```
void visitBF(Vertex v, Graph g)  {
        Set<Vertex> besucht = ∅;
        visitBF(v, g, besucht);
}

void visitBF(Vertex v, Graph g, Set<Vertex> besucht)
{
        Queue<Vertex> q;
        q.add(v);

        while( ! q.empty() )  {
                v = q.remove();
                if ( besucht.contains(v))
                        continue;

                besucht.add(v);

                // Bearbeite v:
                println(v);

                for ( jeden adjazenten Knoten w von v )
                        if ( ! besucht.contains(w) )
                                q.add(w);
        }
}
```

visitBF startet von Knoten v eine Breitensuche im Graphen g.

visitBF besucht Knoten v im Graphen g, wobei die bereits besuchten Knoten im Feld besucht abgespeichert werden.

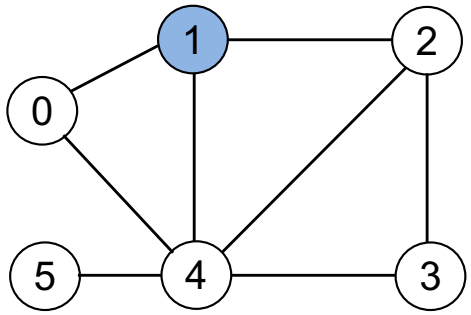
In der Schlange q werden alle Knoten verwaltet, die als nächstes zu besuchen sind.

Die Breitensuche wird erreicht durch die FIFO-Organisation der Schlange

Beachte: Gleiche Knoten können mehrfach in die Schlange eingereiht werden.
Das ließe sich durch eine weitere Knotenmarkierung verhindern:

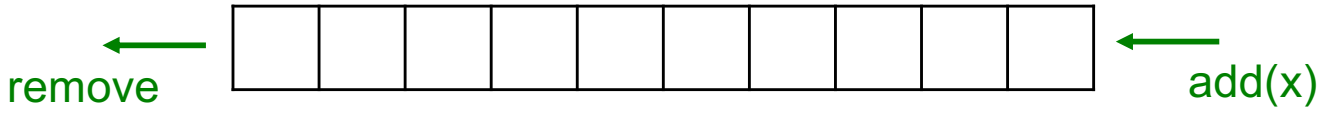
- nicht besucht,
- nicht besucht und in der Schlange,
- besucht.

Breitensuche mit Queue



Bitte ausfüllen

Schlange



Besuchsreihenfolge:
1, 0, 2, 4, 3, 5

8. Elementare Algorithmen für Graphen

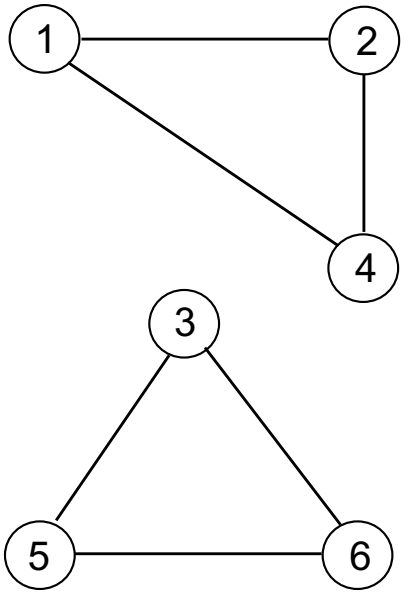
- Tiefensuche
- Breitensuche
- Tiefen- und Breitensuche bei mehreren Zusammenhangskomponenten
- Bipartiter Graph
- Topologisches Sortieren

Tiefen- bzw. Breitensuche bei mehreren Zusammenhangskomponenten

- Hat ein Graph mehrere Zusammenhangskomponenten, dann ist nicht garantiert, dass es von jedem Knoten v einen Weg zu jedem anderen Knoten w gibt.
- Sollen alle Knoten besucht werden, dann muss visitDF (Tiefensuche) bzw. visitBF (Breitensuche) mehrere Male mit allen noch nicht besuchten Knoten als Startknoten aufgerufen werden.

```
void visitAllNodes() {

    Set<Vertex> besucht = ∅;

    for (jeden Knoten v)
        if (! besucht.contains(v) )
            visit(v, g, besucht);
}
```

visit kann visitDF oder visitBF sein.

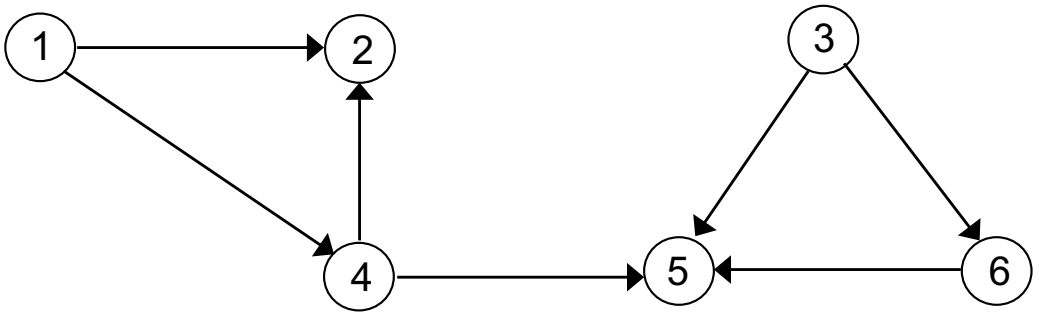


Tiefensuchwald bei Tiefensuche mit mehreren Zusammenhangskomponenten

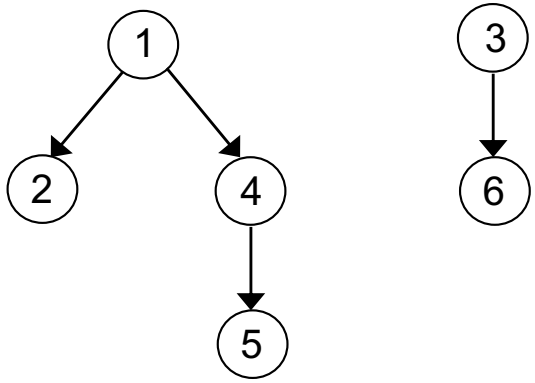
- Bei der Tiefensuche in einem Graphen mit mehreren Zusammenhangskomponenten entsteht dann der sogenannte Tiefensuchwald.

```
void visitDFAllNodes() {

    Set<Vertex> besucht = ∅;

    for (jeden Knoten v)
        if (! besucht.contains(v))
            visitDF(v, g, besucht);
}
```

Gerichteter Graph (mit mehreren starken Zusammenhangskomponenten):



Tiefensuchwald (Knoten werden nach ihrer Nummerierung aufsteigend besucht):



Analyse

- Hier: Analyse der Tiefensuche über den kompletten Graphen.
  Bei der Breitensuche ergibt sich dieselbe Komplexität.
- Jeder Knoten wird genau 1-mal besucht.
- Jede Kante wird genau 2-mal bei ungerichteten Graphen bzw. genau 1-mal bei gerichteten Graphen in einer der for-Schleifen betrachtet.
  Wird die besucht-Menge besucht als boolsches Feld implementiert und werden für den Graph Adjazenzlisten verwendet, ist der Aufwand für den Durchlauf aller for-Schleifen höchtens O(|E|).
- Damit ergibt sich insgesamt:

$$T = O(|V| + |E|)$$

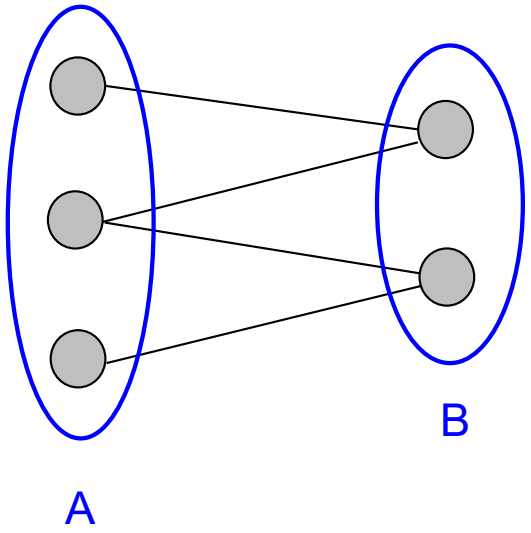
```
void visitDFAllNodes() {
        Set<Vertex> besucht = ∅;

        for (jeden Knoten v)
                if (! besucht.contains(v) )
                        visitDF(v, g, besucht);
}

void visitDF(Vertex v, Graph g, Set<Vertex> besucht) {
        besucht.add(v);

        // Bearbeite v:
        println(v);

        for ( jeden adjazenten Knoten w von v )
                if ( ! besucht.contains(w) )
                        visitDF(w, g, besucht);
}
```

8. Elementare Algorithmen für Graphen

- Tiefensuche
- Breitensuche
- Tiefen- und Breitensuche bei mehreren Zusammenhangskomponenten
- Bipartiter Graph
- Topologisches Sortieren

Bipartiter Graph

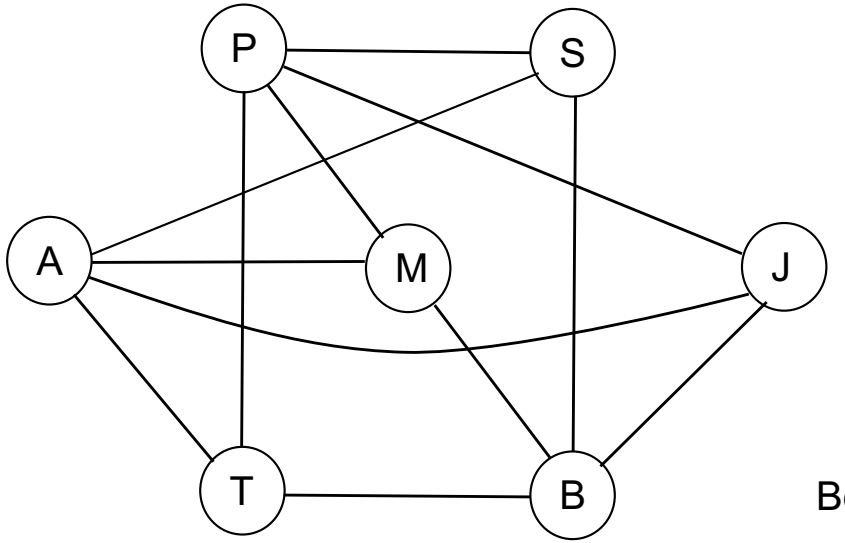
- Ein ungerichteter Graph G = (V, E) heißt bipartit, falls sich V disjunkt in A und B zerlegen lässt, so dass für jede Kante (u,v) $\in$ E gilt:
  - u $\in$ A und v $\in$ B oder
  - u $\in$ B und v $\in$ A.



Es gibt nur Kanten zwischen A und B.

Beispiel Sympathiegraph

- Für eine Menge von Personen V ist ein Sympathiegraph G = (V, E) gegeben.
- (u,v) ist eine Kante in E gdw. sich u und v gegenseitig sympathisch finden.



Beispiel aus [Vöcking], Kap. 7

- Gesucht ist (für eine Show) eine Zerlegung von V in zwei disjunkte Personengruppen A und B, die sich nicht untereinander sympathisch finden.
- Also: Ist G ein bipartiter Graph?
  Wenn ja, wie sieht die disjunkte Zerlegung in A und B aus?

Tiefensuche zur Prüfung der Bipartitheit

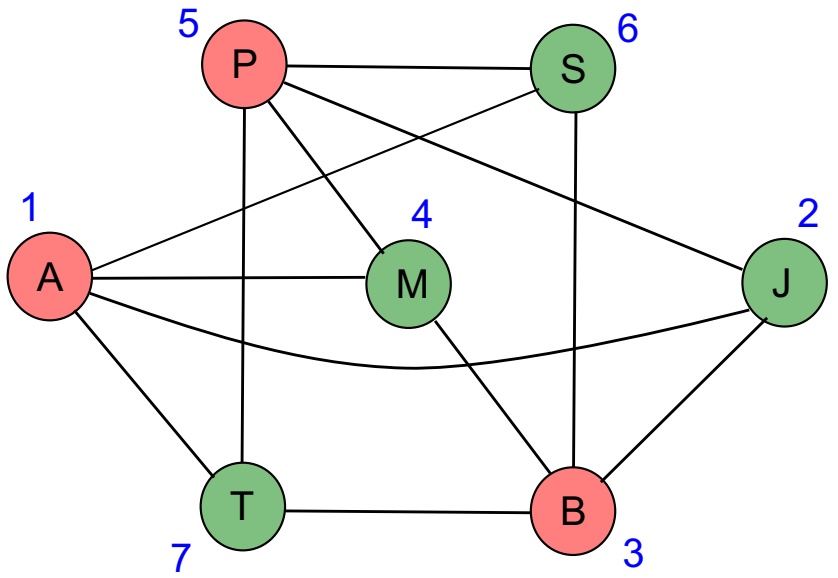
- Besuche mit Tiefensuche alle Knoten und färbe abwechselnd mit rot oder grün ein bzw. prüfe ob Färbung OK.

```
void pruefeBipartheit() {
        for (jeden Knoten v)
                if (v hat noch keine Farbe)
                        färbe(v, "rot"))
        print("Graph ist bipartit");
}
```

```
void färbe(Vertex v, Farbe f) {
        if (v bereits gefärbt)  {
                if (Farbe von v ist nicht f) {
                        print("Graph ist nicht bipartit");
                        beende Prüfung;
                }
        } else {
                färbe v mit f ein;
                for ( jeden Nachbarn w von v )
                        färbe(w, flip(f));
        }
}
```

flip("rot") = "grün" und
flip("grün") = "rot" .

Tiefensuche mit alphabetischer Reihenfolge.
Reihenfolge der Färbung in blau.



8. Elementare Algorithmen für Graphen

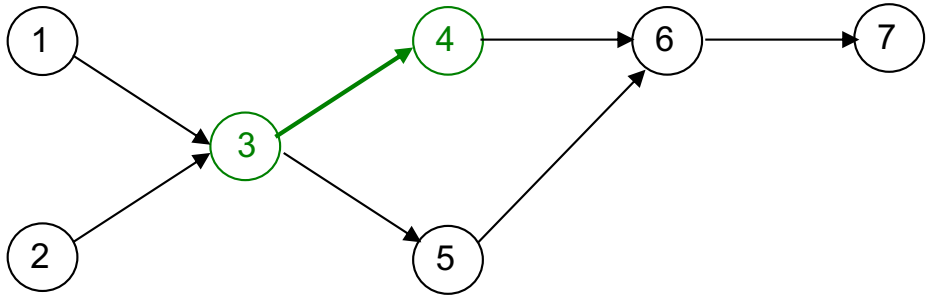
- Tiefensuche
- Breitensuche
- Tiefen- und Breitensuche bei mehreren Zusammenhangskomponenten
- Bipartiter Graph
- Topologisches Sortieren

Definition topologische Sortierung

- Eine Folge $v_0, v_1, v_2, \ldots, v_{n-1}$ aller Knoten eines gerichteten Graphen G heißt topologische Sortierung, falls für alle Knoten u, v folgende Bedingung gilt:

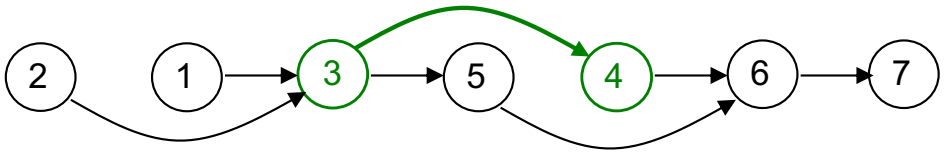
  falls (u,v) eine Kante in G ist,
  dann steht u vor v in der Folge $v_0, v_1, v_2, \ldots, v_{n-1}$.

- Beispiel:



(3,4) ist Kante

topologisch sortiert:



3 steht vor 4

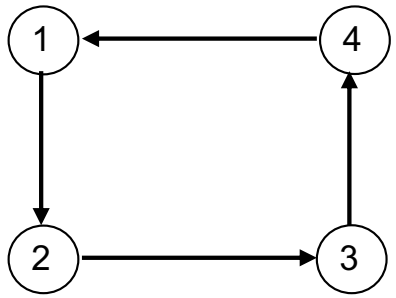
- Anschaulich:

  Der Graph lässt sich so auf eine Linie „verbiegen“,
  dass die Pfeile nur nach rechts gehen.

Eigenschaften und Anwendungen

Eigenschaften:

- Falls der Graph einen Zyklus enthält, dann existiert keine topologische Sortierung und umgekehrt.

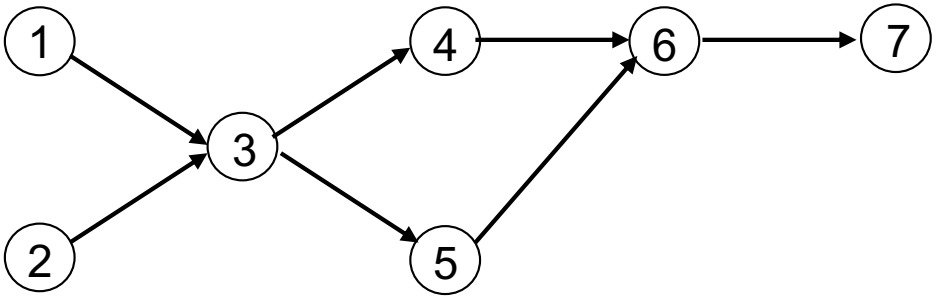


- Falls ein Graph topologisch sortiert werden kann, dann ist im allgemeinen die topologische Sortierung nicht eindeutig.

Beispiele für Anwendungen:

- Stelle fest, ob es eine Durchführungsreihenfolge für die Aktivitäten oder auch Prozesse in einem Vorranggraphen gibt.
- Prüfe, ob Vererbungsgraph oder include-Graph zyklenfrei ist.

Idee für topologische Sortierung



inDegree und
Queue einzeichnen!

- Speichere für jeden Knoten v:
  inDegree[v] = Anzahl der noch nicht besuchten Vorgänger
- Halte alle noch nicht besuchten Knoten v mit inDegree[v] == 0
  als Kandidat in einer Queue q.
- Besuche immer nächsten Knoten aus der Queue q.

Algorithmus für topologische Sortierung

```
List<Vertex> topSort(DiGraph g) {
    List<Vertex> ts;  // topologisch sortierte Folge
    int[ ] inDegree;   // Anz. noch nicht besuchter Vorgänger
    Queue<Vertex> q;

    for (jeden Knoten v) {
        inDegree[v] = Anzahl der Vorgänger;
        if (inDegree[v]  == 0)
            q.add(v);
    }

    while (! q.empty() ) {
        v = q.remove();
        ts.add(v);
        for ( jeden Nachfolger w von v )
            if (--inDegree[w] == 0)
                q.add(w);
    }

    if (ts.size() != Anzahl Knoten in g)
        return null;       // Graph zyklisch;
    else
        return ts;
}
```

Analyse

- Für jeden Knoten w kann inDegree[w] höchstens einmal gleich 0 werden. Damit kommen in die Queue q höchstens |V| Knoten.
  Die while-Schleife läuft damit höchstens |V|-mal.
- Jede Kante wird höchstens einmal in einer der for-Schleifen betrachtet. Verwendet man die Adjazenzlistendarstellung, ist der Aufwand für den Durchlauf aller for-Schleifen höchtens O(|E|).
- Damit ergibt sich mit Adjazenzlistendarstellung insgesamt:

  $T = O(|V| + |E|)$

```
List<Vertex> topSort(DiGraph g) {
    List<Vertex> ts;  // topologisch sortierte Folge
    int[ ] inDegree;   // Anz. noch nicht besuchter Vorgänger
    Queue<Vertex> q;

    for (jeden Knoten v) {
        inDegree[v] = Anzahl der Vorgänger;
        if (inDegree[v]  == 0)
            q.add(v);
    }

    while (! q.empty() ) {
        v = q.remove();
        ts.add(v);
        for ( jeden Nachfolger w von v )
            if (--inDegree[w] == 0)
                q.add(w);
    }

    if (ts.size() != Anzahl Knoten in g)
        return null;       // Graph zyklisch;
    else
        return ts;
}
```