Woche 10 – Übersicht & Tricks

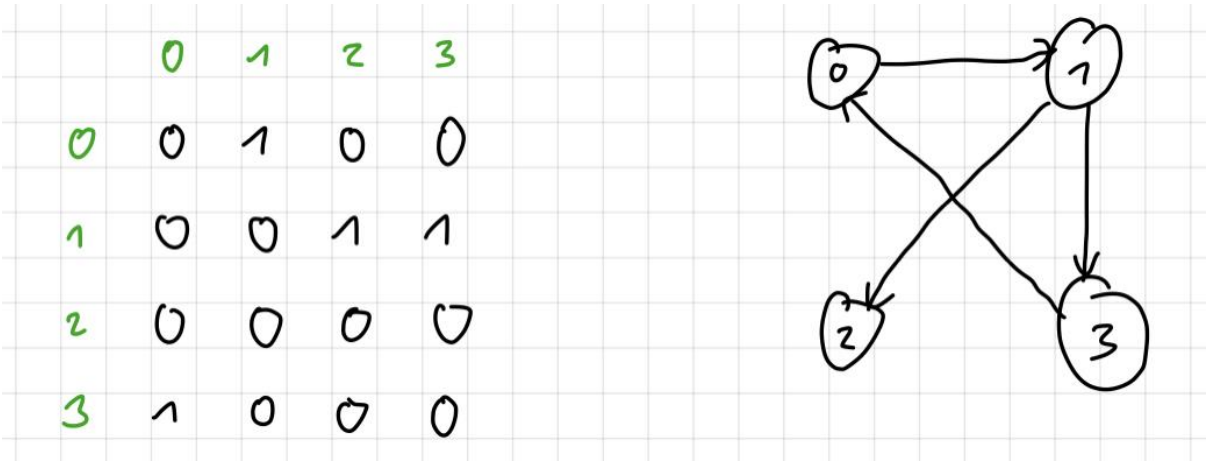
(Disclaimer: Die hier vorzufindenden Notizen haben keinerlei Anspruch auf Korrektheit oder Vollständigkeit und sind nicht Teil des offiziellen Vorlesungsmaterials. Alle Angaben sind ohne Gewähr.)

Anmerkungen zu Serie 8: Allgemeines

-

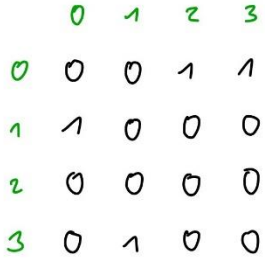
Adjazenzmatrizen n mal potenzieren für Pfade der Länge n:

Im Skript wird auf den Seite 102-103 ein interessantes Konzept vorgestellt: In einer Adjazenzmatrix sagt $a_{ij} = 1$ aus, dass es eine Kante von i nach j gibt. Beispielsweise sagt:



$a_{01} = 1$, dass es im Graphen rechts eine (gerichtete) Kante von Knoten 0 zu Knoten 1 gibt.

Wenn wir nun die Matrix quadrieren, erhalten wir folgende Matrix:



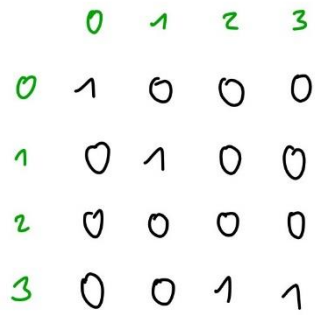
Wir nennen diese neue Matrix $B = A^2$, wobei A unsere ursprüngliche Matrix war. Aus der linearen Algebra wissen wir nun folgendes: $b_{ij} = \sum_{k=0}^{n-1} a_{ik} * a_{kj}$.

Das bedeutet: Jeder Eintrag der Matrix B ist ungleich 0 genau dann wenn mindestens ein $k \in \{0, \dots, n-1\}$ existiert, sodass $a_{ik} = 1$ und $a_{kj} = 1$. Aber was genau bedeutet das eigentlich?

Wenn $a_{ik} = 1$ und $a_{kj} = 1$ für irgendein $k \in \{0, \dots, n-1\}$, heisst das, dass es eine Kante vom Knoten i zum Knoten k gibt und, dass es eine Kante vom Knoten k zum Knoten j gibt (per unserer Definition). Also gibt es einen Weg von Knoten i nach Knoten j der Länge 2, der als "Zwischenknoten" k hat. Also gilt: $b_{ij} \neq 0 \Leftrightarrow \exists$ Weg der Länge 2 von i nach j.

Genauer gilt sogar: $b_{ij} =$ Anzahl der Wege von Knoten i zu Knoten j. Denn b_{ij} wird jedes Mal um 1 erhöht, wenn es ein k gibt, sodass $a_{ik} = 1$ und $a_{kj} = 1$.

Wenn wir nun A noch einmal potenzieren, also $C = B * A = A^3$ berechnen, erhalten wir folgende Matrix:



Hier gilt nun für alle Einträge: $c_{ij} = \sum_{k=0}^{n-1} b_{ik} * a_{kj}$. Jeder Eintrag der Matrix C ist also ungleich 0, gdw. ein $k \in \{0, \dots, n-1\}$ existiert, sodass $b_{ik} > 0$ und $a_{kj} = 1$. Das heisst, nun Zählen wir die Wege der Länge 3. Der Eintrag c_{ij} besagt dabei: $c_{ij} =$ Anzahl Wege der Länge 3 von i nach j.

Jetzt können wir diese Matrix auch noch weiterverwenden, um die Anzahl der Dreiecke zu finden, ein übliches Problem (und vor einigen Jahren auch mal eine Klausuraufgabe). Wir haben an einem Knoten i ein Dreieck gdw. es einen Weg der Länge 3 von i zu i gibt, also wenn $c_{ii} > 0$.

Das heisst, die Anzahl der Dreiecke ist einfach: $D = \sum_{k=0}^{n-1} c_{ii}$.

Allerdings müssen wir jetzt noch etwas beachten: Ein Dreieck, bspw. $< 0,1,3,0 >$ kann auch folgendermassen gezählt werden: $< 1,3,0,1 >$ oder $< 3,0,1,3 >$ und in einem ungerichteten Graphen sogar auch noch in umgekehrter Richtung. Das heisst für einen gerichteten Graphen müssen wir D noch einmal durch 3 teilen, um die Mehrfachzählungen zu vermeiden und für gerichtete Graphen durch 6 (da es noch mehr Möglichkeiten gibt, da ein Dreieck in beide Richtungen gezählt werden kann).

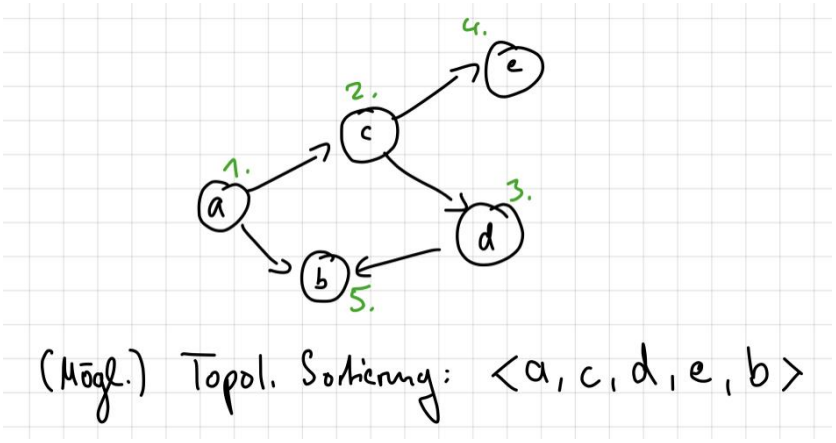
Wir können das auch verwenden, um kürzeste Pfade zu finden: Einfach die Matrix $n-1$ mal (höchstens – denn Pfade länger als $n-1$ sind automatisch Wege, da sie Knoten mehrfach verwenden **müssen**) potenzieren und dann für jeden Eintrag schauen, wann er das erste Mal ungleich 0 wird. Das ist dann die Länge eines kürzesten Pfades von i nach j. Das Problem ist hier allerdings, dass dieser Algorithmus n Matrixmultiplikationen benötigt und damit in $O(n * n^3) = O(n^4)$ läuft, was eher ungünstig ist. Im weiteren Verlauf der Vorlesung werdet ihr effizientere Algorithmen für dieses Problem kennenlernen.

Topologische Sortierungen:

Wird formal durch einen gerichteten Graphen $G = (V, E)$ beschrieben: Die Knoten repräsentieren Aufgaben/Tätigkeiten die in einer bestimmten Reihenfolge, also mit bestimmten Abhängigkeiten erledigt werden sollen. Das heisst: $\exists(u, v) \in E \Leftrightarrow u$ muss vor v erledigt werden.

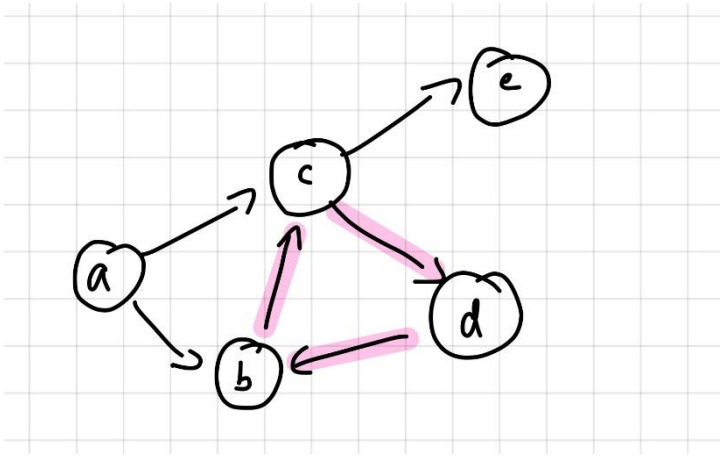
Wir wollen nun feststellen, ob es möglich ist, eine globale Reihenfolge für alle Aufgaben/Tätigkeiten zu bestimmen, also zu sagen, welches das erste, zweite, … Element in der Gesamtreihenfolge ist. Formal drücken wir das aus, indem wir sagen, dass wir versuchen, eine Bijektion $ord: V \to \{1, \dots, |V|\}$ zu bestimmen, die jedem Knoten $v \in V$ einen Index, der seine Position in der globalen Reihenfolge, der "topologischen Sortierung", beschreibt. Oftmals geben wir diese topologische Sortierung dann als Sequenz von Knoten $< v_1, \dots, v_{|V|} >$ in der korrekten Reihenfolge an. Achtung: Das soll keinen Pfad/Weg beschreiben!

Hier ist ein Beispiel:



Offensichtlich gibt es eine topologische Sortierung (hier einmal als Sequenz und einmal in Form von Einträgen (in grün) an den einzelnen Knoten notiert).

Nun fragen wir uns: Welche Voraussetzung(en) müssen erfüllt sein, damit ein gerichteter Graph eine topologische Sortierung enthält? Logisch kann man das folgendermassen betrachten: Jede Aufgabe hat eine gewisse Anzahl an Aufgaben (mindestens 0), die vorher erledigt werden müssen, bevor die Aufgabe erledigt werden kann – also in der topologischen Sortierung vor der Aufgabe stehen müssen. Das heisst, damit es eine topologische Sortierung geben kann, muss es **immer** möglich sein, alle Bedingungen einer Aufgabe zu erledigen, bevor man eine Aufgabe erledigt. Im Beispiel oben sehen wir das bspw. an der Aufgabe b: Die Aufgaben, die vorher erledigt werden müssen, sind a und d. Und es ist möglich, diese Aufgabe vor b zu erledigen. Hätten wir nun diese Situation, wäre dem nicht mehr so, da das Lösen von d das Lösen von b vorrausetzen würde:



Das Problem ist der Kreis. Aus dieser Feststellung folgt folgendes Theorem, das sich leicht per Induktion (siehe Skript) beweisen lässt:

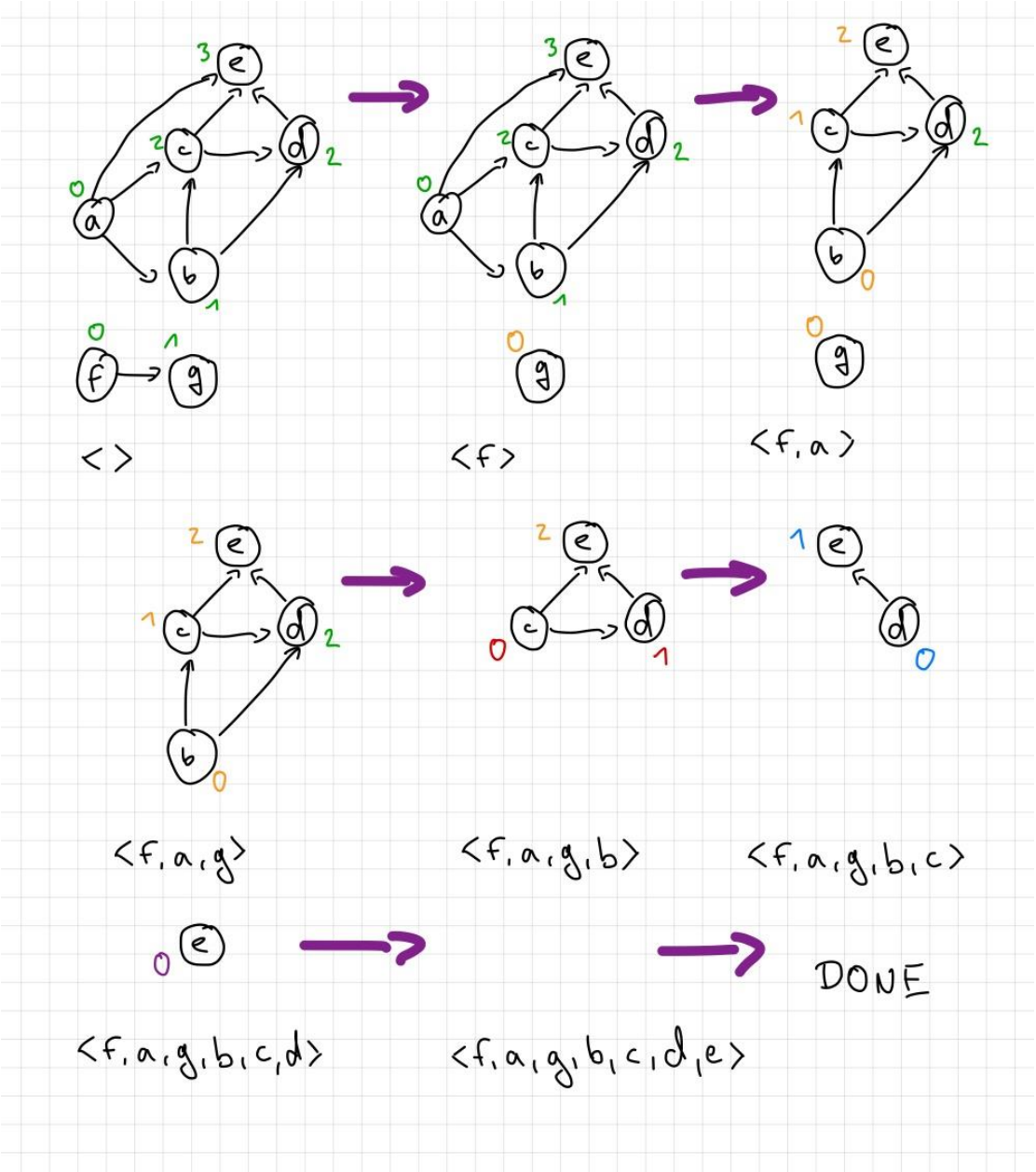
Ein gerichteter Graph $G = (V, E)$ enthält eine topologische Sortierung

$\Leftrightarrow$

Der Graph $G = (V, E)$ ist kreisfrei

Das heisst, wir müssen nur prüfen (was mit DFS sehr leicht ist), ob unser Graph einen Kreis hat und können so schnell (in $O(|E| + |V|)$) entscheiden, ob eine topologische Sortierung für ihn bestimmt werden kann.

Idee für einen Algorithmus, der eine topologische Sortierung berechnet: Bestimme die Eingangsgrade aller Knoten und speichere sie in Array $A[0, \dots, n-1]$. Dann entfernen wir einen der Knoten mit Eingangsgrad $N^-(v) = 0$ und dekrementieren den Eingangsgrad aller Knoten u mit $(v, u) \in E$. Das wiederholen wir anschliessend für alle Knoten mit Eingangsgrad 0. Diesen Prozess wiederholen wir nun, bis wir alle Knoten entfernt haben. Hier ein Beispiel:

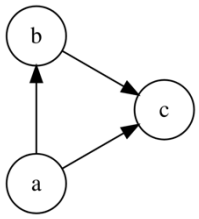


Aufgabe (FS21):

Let G_1 and G_2 be two directed acyclic graphs with the same vertex set and such that all the edges of G_1 are also edges of G_2, i.e., G_1 is a (directed) subgraph of G_2. You get 1P for a correct answer, -1P for a wrong answer, 0P for a missing answer. You get at least 0 points in total.

Claim	**true**	**false**
Every topological sorting of G_1 is also a topological sorting of G_2.	☐	☐
Every topological sorting of G_2 is also a topological sorting of G_1.	☐	☐

Aussage 1: Falsch, Gegenbeispiel: $G_1 =$



$G_2 = (V, E\backslash\{(a,b)\}) \rightarrow [b,a,c]$ ist eine topol. Sortierung, aber in G_1 nicht

Aussage 2: Wahr: Durch das Entfernen von Kanten aus G_2, um auf G_1 zu kommen gehen keine topologischen Sortierungen verloren.

Depth-First-Search:

Bei der Depth-First-Search (DFS) handelt es sich um einen Algorithmus zum systematischen Durchlaufen von Graphen. Die Idee ist eigentlich recht simpel: Wir starten in einem Knoten und gehen zu einem (falls es welche gibt) Nachfolger. Wenn wir bei dem Nachfolger sind, wiederholen wir entweder den Prozess (falls der Nachfolger selbst Nachfolger hat) oder kehren zu unserem vorherigen Knoten zurück, falls der Nachfolger keine Nachfolger mehr hat. Von dem vorherigen Knoten aus gehen wir nun zu seinem nächsten Nachfolger und wiederholen den Prozess, bis wir alle Knoten einmal besucht haben. Daher nennt man den Algorithmus Tiefensuche: Wir versuchen jedes Mal, von einem Knoten aus in einer Tour so tief wie möglich in dem Graph zu kommen, bis es nichtmehr geht.

Im Skript wird der Algorithmus folgendermassen beschrieben:

DFS-VISIT(G, v)

```
1 Markiere v als besucht
2 for each (v, w) ∈ E do
3     if w ist noch nicht besucht then
4         DFS-VISIT(G, w)
```

Wir müssen uns offensichtlich merken, ob wir einen Knoten schon einmal besucht haben (um Dopplungen zu vermeiden) und ansonsten starten wir für jeden Nachfolger, den ein Knoten hat, eine Tiefensuche.

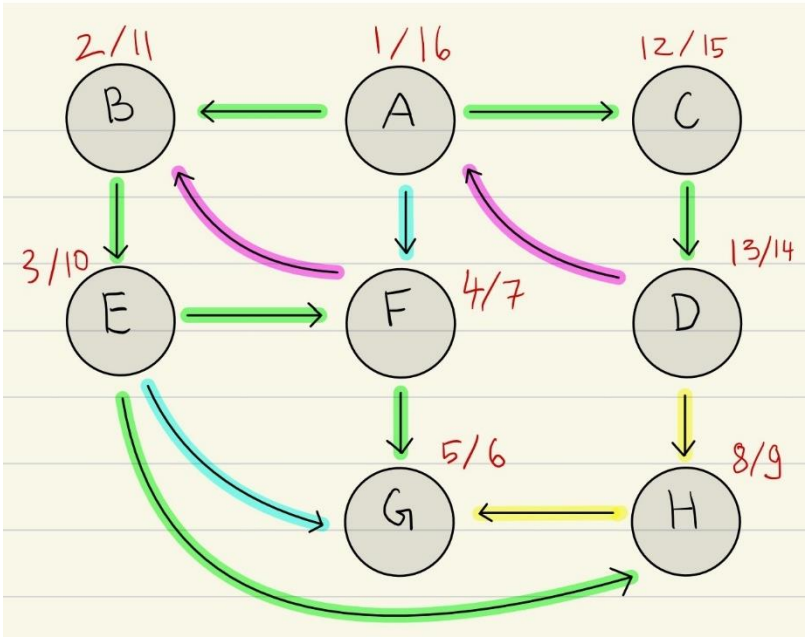
In Java bedarf es dabei offensichtlich einer Datenstruktur mit der wir uns merken, welche Knoten wir bereits besucht haben. Ein einfaches $visited[]$ array ist für diese Zwecke allerdings vollkommen ausreichend. So könnte eine rekursive Implementation in Java aussehen (iterativ auch möglich):

```java
class Graph {
    void dfs(int[][] graph, int start, boolean[] visited){
      if(visited[start])
        return;
      visited[start] = true;
      for(int i = 0; i<graph[0].length; i++){
        if(graph[start][i] == 1 && i != start)
           dfs(graph, i, visited);
      }
    }
}
```

In dem Fall würde der Algorithmus in $O(|V|^2)$ laufen. Der DFS-Algorithmus ist auch effizienter, in $O(|V| + |E|)$ implementierbar, falls der Graph (anders als hier nicht als Adjazenzmatrix, sondern:) als Adjazenzliste gegeben ist. Die Implementation ist an dieser Stelle dem Leser überlassen (da sie eventuell eine Übungsaufgabe sein könnte). Hilfreich ist hierbei folgendes: $Iterator < Integer > iterator = adj[start].listIterator()$; Was hier was bedeutet und in welchem Kontext dieser Iterator bei einer Java-Implementation des DFS wichtig sein könne ist auch dem Leser überlassen, kann aber leicht hergeleitet (oder zumindest via Google herausgefunden) werden.

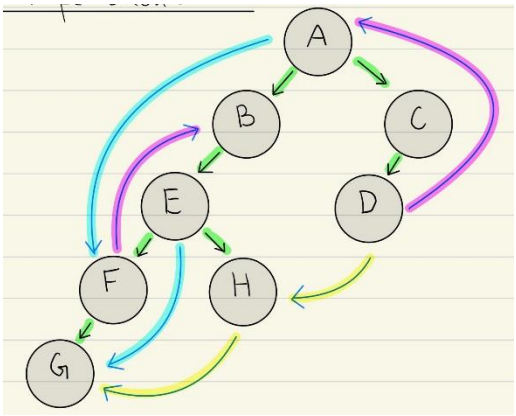
Nun gibt es eine Möglichkeit, DFS zu modifizieren, um weitere Probleme mit DFS leichter zu lösen: Wir merken uns für jeden Knoten, wann wir ihn besuchen und wann unser Besuch wieder “endet”. Das klingt eventuell erst einmal ein wenig abstrakt: Sobald wir einen Knoten besuchen, inkrementieren wir einen Counter und speichern sofort ab, zu welchem Counter-Wert wir diesen Knoten besucht haben, bspw. in einem Array $Pre[]$. Sobald wir alle Nachfolgerknoten des Knotens per DFS besucht haben, speichern wir den (bis dahin mindestens genau so hohen, in den meisten Fällen höheren) Counter-Wert erneut ab, bspw, in einem Array $Post[]$.

Hier das Beispiel aus der Vorlesung:



Für jeden Knoten entsteht also ein **Intervall:** $\boldsymbol{I_v = \{Pre[v], \dots, Post[v]\}}$

Nun gibt es in einem, bei der Tiefensuche entstehenden **Tiefensuchbaum** verschiedene Arten von Kanten: In **grün**: Normale Baumkanten. In **blau:** Forward-Kanten. In **gelb:** Cross-Kanten. In **pink:** Back-Kanten.



Um zu klassifizieren um welche Art der Kante es sich bei einer Kante $(u, v) \in E$ handelt, können wir das Intervall verwenden. Es gilt für eine Kante $(u, v) \in E$:

1. Pre[v] < Pre[u] und Post[v] > Post[u]: Back- Kante
2. Pre[u] < Pre[v] und Post[u] > Post[v]: Forward- Kante
3. Pre[v] < Pre[u] und Post[v] < Post[u]: Cross-Kante

Diese Klassifizierungsmöglichkeit für Kanten können wir nun verwenden, um ein eingangs beschriebenes Problem zu lösen: Bestimmen, ob ein gerichteter Graph eine topologische Sortierung besitzt. Dazu müssen wir nun prüfen, dass es keine Back-Kanten gibt. Dann existiert kein Kreis und damit auch eine topologische Sortierung.

Breadth-First-Search:

Anstatt, dass wir jedes Mal versuchen, im Graphen so weit wie möglich vorzudringen, bewegen wir uns bei der Breadth-First-Search (BFS) “in Schüben”, das heisst, in jeder Iteration bewegen schauen wir uns Knoten an, die um eine Kante weiter von unserem Startknoten entfernt sind, als in der Iteration davor. Da wir dafür also in jedem “Schub” die Nachfolger von mehreren verschiedenen Nachfolgern besuchen wollen, müssen wir eine geeignete Datenstruktur verwenden. Hier bietet sich eine Queue an:

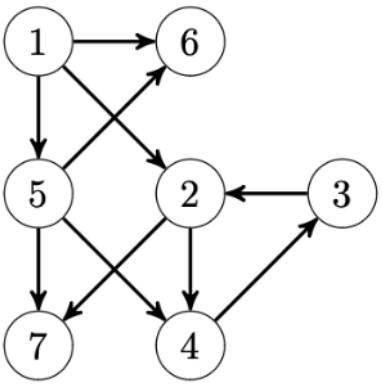
```java
class Graph {
    void bfs(int[][] graph, int start){
      boolean[] visited = new boolean[graph[0].length];
      LinkedList<Integer> queue = new LinkedList<Integer>();
      visited[start] = true;
      queue.add(start);
      while(queue.size() != 0){
        int s = queue.poll(); //holen uns Knoten aus Queue
        for(int  i = 0; i<n; i++){ //Betrachten alle seine Nachfolger
          if(graph[s][i] == 1 && s!=i && !visited[i]){
              visited[i] = true;
              queue.add(i); //Fügen betrachtete Nachfolger in Queue ein
          }
        }
      }
    }
}
```

Wieder habe ich den Algorithmus bloss für eine Adjazenzmatrix implementiert. Dabei läuft er in $O(|V|^2)$. Eine effizientere Implementation (in $O(|V| + |E|)$) ist mit Adjazenzlisten möglich. Die Implementation ist hier aus gleichen Gründen wie bei DFS dem Leser überlassen.

Der Algorithmus BFS kann, falls es keine Kantengewichte gibt, verwendet werden, um die kürzesten Pfade von einem Startknoten zu allen anderen Knoten zu berechnen. In diesem Fall ist der Algorithmus BFS mit einer Optimal-Laufzeit von $O(|V| + |E|)$ der effizienteste bekannte Algorithmus.

Aufgabe (HS20):

Depth-first search / breadth first search: Consider the following directed graph:

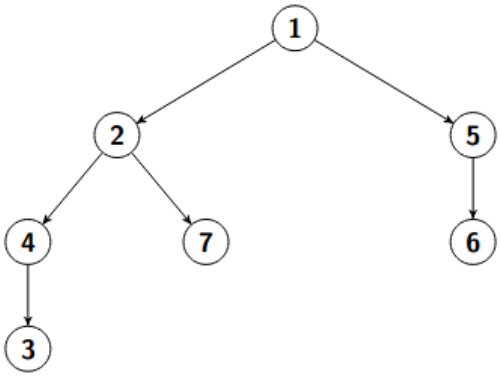


i) Draw the depth-first tree resulting from a depth-first search starting from vertex 1. When processing the neighbors of a vertex, process them in increasing order.

ii) Draw the breadth-first search tree resulting from a breadth-first search starting from vertex 1. When processing the neighbors of a vertex, process them in increasing order.

Solution

i)



ii)