QUIZ-NACHBESPRECHUNG

Sei $Q = [2, 4, 3, 8, 9]$ eine Warteschlange (Queue). Betrachten Sie die folgenden Operationen:

1. $\mathrm{fügehinzu/enqueue}(6)$
2. $\mathrm{entferne/dequeue}$
3. $\mathrm{entferne/dequeue}$
4. $\mathrm{fügehinzu/enqueue}(7)$

~~2~~ ~~4~~ 3 8 9 6 7

Welche der folgenden Warteschlangen stellt Q richtig dar, nachdem die oben genannten Operationen in der angegebenen Reihenfolge ausgeführt wurden?
(Die Operation $\mathrm{fügehinzu/enqueue}$ *fügt ein Element an das* **rechte** *Ende der Warteschlange hinzu)*

Wählen Sie eine Antwort:

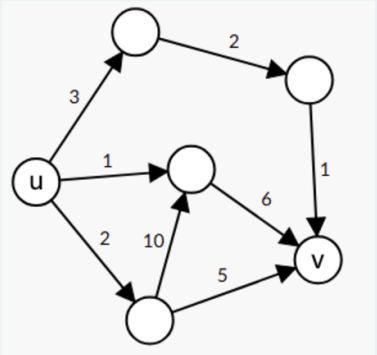
- ~~a.~~ $Q = [3, 8, 9, 6, 7]$
- b. $Q = [2, 4, 3, 8, 7]$
- c. $Q = [2, 4, 3, 8, 9]$

Betrachten Sie die Standardimplementierung der Breitensuche (BFS) mit einer Warteschlange (Queue) Q welche Sie in der Vorlesung gesehen haben.
Angenommen wir führen diesen Algorithums auf einem (ungewichteten) gerichteten Graph $G = (V, E)$ aus, beginnend bei einem Knoten $s \in V$.

Welche der folgenden Aussagen über Q muss direkt nachdem ein Knoten $v \in V$ entfernt (dequeued) wurde richtig sein?

Wählen Sie eine Antwort:

- a. $\mathrm{dist}(s, v) < \mathrm{dist}(s, w)$ für alle w in Q
- b. $\mathrm{dist}(s, v) > \mathrm{dist}(s, w)$ für alle w in Q.
- c. $\mathrm{dist}(s, v) = \mathrm{dist}(s, w)$ für alle w in Q.
- ~~d.~~ Keine der oben genannten



6

Was ist die Länge eines kürzesten Pfades zwischen den Knoten u und v im oben dargestellten gewichteten, gerichteten Graph?

Sei $G = (V, E)$ ein gewichteter, gerichteter Graph mit positiven Gewichten $c : E \to \mathbb{R}_{>0}$.
Erinneren Sie sich, dass $d(v, w)$ die Länge eines kürzesten Pfades von v nach w in G bezeichnet.

Seien $v \neq w$ zwei Knoten in G mit $d(v, w) < \infty$. Welche der folgenden Formeln sind korrekt?

Wählen Sie eine Antwort:

- a. $d(v, w) = \min_{(u,w) \in E} \{d(v, u) + c(u, w)\}$
- b. $d(v, w) = \min_{u \in V} \{d(v, u) + d(u, w)\}$
- ~~c.~~ (a) und (b) sind beide richtig.
- d. (a) und (b) sind beide nicht richtig.

Sei $G = (V, E)$ ein gewichteter, gerichteter Graph mit positiven Gewichten $c : E \to \mathbb{R}_{>0}$.
Erinneren Sie sich, dass $d(v, w)$ die Länge eines kürzesten Pfades von v nach w in G bezeichnet.

Sei $s \in V$. In der Vorlesung haben wir die folgende rekursive Formel gesehen:

$$d(s, v) = \begin{cases} 0 & \text{falls } v = s, \\ \infty & \text{falls } v \neq s,\ \deg_{\text{in}}(v) = 0, \\ \min_{(u,v) \in E} d(s, u) + c(u, v) & \text{sonst.} \end{cases}$$

In welcher Reihenfolge sollten die Knoten in G verarbeitet werden, damit diese rekursive Formel $d(s, v)$ für alle Knoten $v \in V$ korrekt berechnet?

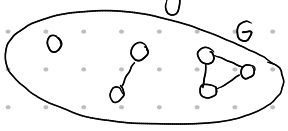
Wählen Sie eine Antwort:

- ~~a.~~ In einer topologischen Reihenfolge (sofern eine solche existiert).
- b. In einer umgekehrten topologischen Reihenfolge (sofern eine solche existiert).
- c. In aufsteigender Reihenfolge der enter-Zahl einer Breitensuche, die bei s beginnt.
- d. In aufsteigender Reihenfolge der leave-Zahl einer Breitensuche, die bei s beginnt.

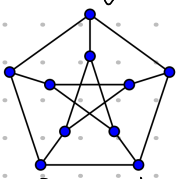
SERIE 8 - COMMON MISTAKES

Graphen allgemein

- bei Graphenmodellierung klar beschreiben, ob der Graph zusammenhängend, gerichtet, etc. ist
- G nicht zusammenhängend $\not\Rightarrow$ #ZHKs = 2



- ZHKs können unbalanciert sein, ihr könnt nicht annehmen, dass alle gleich gross sind
- $\exists$ Hamiltonkreis $\not\Leftrightarrow \forall v \in V \exists$ Hamiltonpfad startend in v
- path vs. walk



Beweise bei Graphen

- einen kurzen Satz wie „wir nehmen ... an und beweisen per Widerspruch"
- Ihr könnt keine eigenen Einschränkungen/Annahmen bei Beweisen treffen
- versucht Gegenbeispiele so simpel wie möglich zu halten und erklärt, warum sie die Aussage widerlegen
- Gegenbeispiele müssen Annahmen in der Aufgabenstellung erfüllen, z.B. zusammenhängend sein

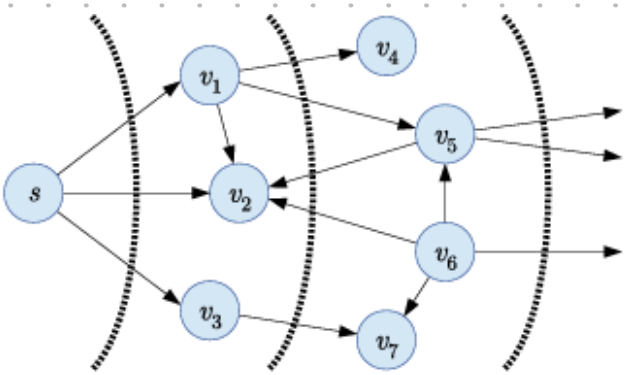
BREADTH-FIRST SEARCH (BFS)

ähnlich zu DFS, nur suchen wir jetzt in der Breite und nicht in der Tiefe des Graphen → sozusagen Level-by-Level

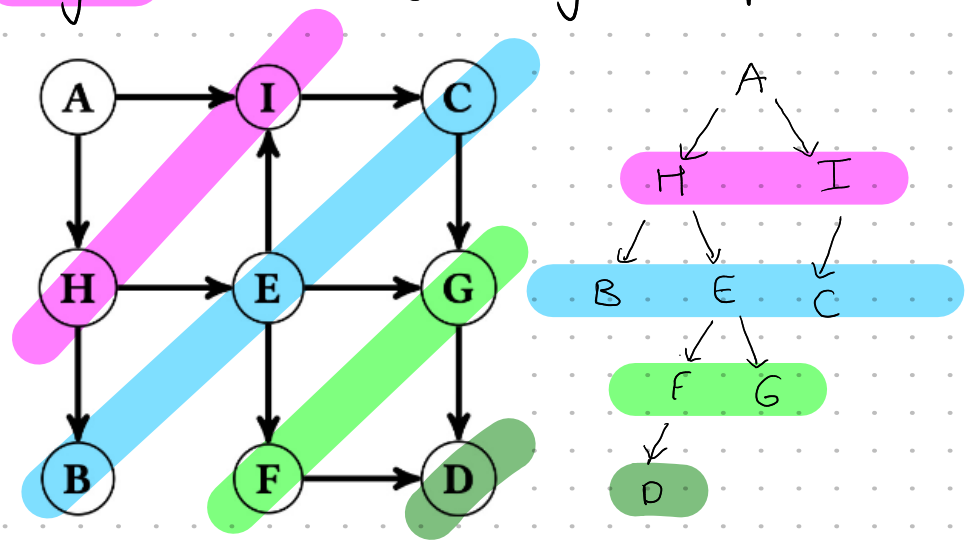
$O(|V|+|E|)$

BFS-VISIT-ITERATIVE(G, v)

```
1  Q ← ∅                                   ▷ Initialisiere leere Schlange Q
2  Markiere v als aktiv
3  ENQUEUE(v, Q)                           ▷ Füge v zur Schlange Q hinzu
4  while Q ≠ ∅ do
5      w ← DEQUEUE(Q)                      ▷ Aktueller Knoten
6      Markiere w als besucht
7      for each (w, x) ∈ E do
8          if x nicht aktiv und x noch nicht besucht then
9              Markiere x als aktiv
10             ENQUEUE(x, Q)               ▷ Füge x zur Schlange Q hinzu
```



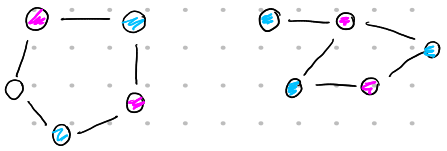
Aufgabe: Finde SP-Tree von folgendem Graphen (startend bei A)



Anwendungen:

- Distanzen in ungewichteten Graphen
- Bipartitheit von Graphen feststellen

↑ Zweifärbbarkeit: kann man einen Graphen mit zwei Farben färben, sodass ein Knoten einer Farbe nur zu Knoten der anderen Farbe benachbart ist?



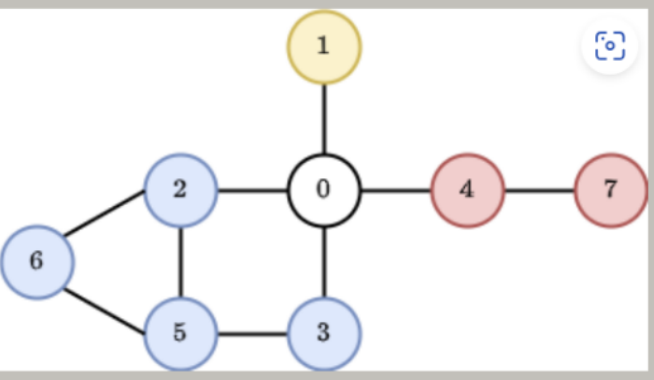
Beispielaufgabe

Graph Sets

You are given an undirected connected graph with n nodes numbered from 0 to $n-1$ and m edges between them.

The nodes of the graph are partitioned into sets in the following way. Node 0 forms a set of its own. Suppose that node 0 is removed from the graph. Then, the nodes in each connected subgraph of the resulting graph forms another set.

An example is shown below, in which the nodes are colored according to the set they are in. Specifically, the sets in this example are $\{0\}$, $\{1\}$, $\{2,3,5,6\}$, and $\{4,7\}$.



Given such a graph, you have to answer queries of the following type:

1. **hasCycle()**: Return 1 if the graph has a cycle, and 0 otherwise.
2. **hasCycleWithoutNodeZero()**: Suppose that node 0 is removed from the graph. Return 1 if the resulting graph has a cycle, and 0 otherwise.
3. **isSameSet(x, y)**: Given two nodes x and y, return 1 if they are in the same set, and 0 otherwise.
4. **getShortestPath(x, y)**: Given two nodes x and y **that are in different sets**, return the shortest-path distance between them.

DIJKSTRA

kürzeste Wege in gewichteten Graphen finden (one-to-all)
→ warum reicht hier nicht BFS

Einschränkung: Der Graph darf keine negativen Gewichte enthalten

$\forall v \in V$ und einen Startknoten s gilt dann: $d(s,v) = \min_{\substack{u \in V \\ (u,v) \in E}} \{d(s,u) + c(u,v)\}$

Idee:
- initialisiere alle Distanzen von s nach v mit ∞
  Ausnahmen: $d(s,s) = 0$ und $d(s,v) = c(s,v)$ falls $(s,v) \in E$
- finde Knoten mit minimaler Distanz und markiere ihn als besucht

DIJKSTRA$(G = (V, E), s)$

```
1  for each v ∈ V\{s} do                  ▷ Initialisiere für alle Knoten die
2      d[v] ← ∞; p[v] ← null              ▷ Distanz zu s sowie Vorgänger
3  d[s] ← 0; p[s] ← null                  ▷ Initialisierung des Startknotens
4  Q ← ∅                                  ▷ Leere Prioritätswarteschlange Q
5  INSERT(s, 0, Q)                        ▷ Füge s zu Q hinzu
6  while Q ≠ ∅ do
7      u ← EXTRACT-MIN(Q)                 ▷ Aktueller Knoten
8      for each (u, v) ∈ E do             ▷ Inspiziere Nachfolger
9          if p[v] = null then            ▷ v wurde noch nicht entdeckt
10             d[v] ← d[u] + w((u, v))    ▷ Berechne obere Schranke
11             p[v] ← u                   ▷ Speichere u als Vorgänger von v
12             ENQUEUE(v, d[v], Q)        ▷ Füge v zu Q hinzu
13         else if d[u] + w((u, v)) < d[v] then ▷ Kürzerer Weg zu v entdeckt
14             d[v] ← d[u] + w((u, v))    ▷ Aktualisiere obere Schranke
15             p[v] ← u                   ▷ Speichere u als Vorgänger von v
16             DECREASE-KEY(v, d[v], Q)   ▷ Setze Priorität von v herab
```

Zeilen 1–5: $O(|V|)$
Zeilen 9–16: $O(\log|V|)$

insgesamt $O((|V|+|E|) \cdot \log|V|)$

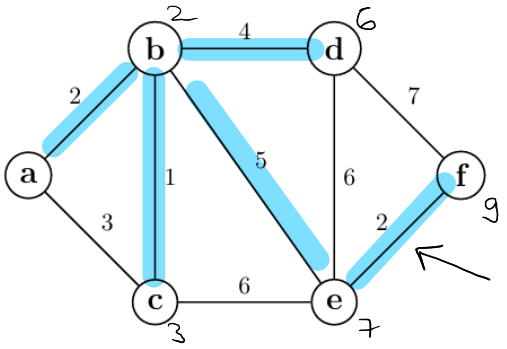
mit Fibonacci-Heaps: $O(|V|\log|V| + |E|)$

```java
//DIJKSTRA
int[] d = new int[n];
for(int i=0; i<n; i++){
  d[i] = Integer.MAX_VALUE;
}
PriorityQueue<Tuple> pq = new PriorityQueue<>();
pq.add(new Tuple(0, s));
d[s] = 0;
boolean[] visited = new boolean[n];
visited[s] = true;
while(!pq.isEmpty()){
  Tuple u = pq.poll();
  if(u.c > d[u.v]) continue;
  for(int i=0; i<Ev.get(u.v).size(); i++){
    if(d[Ev.get(u.v).get(i)] > d[u.v] + Ew.get(u.v).get(i)){
      d[Ev.get(u.v).get(i)] = d[u.v] + Ew.get(u.v).get(i);
      visited[Ev.get(u.v).get(i)] = true;
      pq.add(new Tuple(d[Ev.get(u.v).get(i)], Ev.get(u.v).get(i)));
    }
  }
}
```

alte Prüfungsaufgabe (FS23)

/ 2 P

f) *Shortest Path Tree:* Consider the following graph:



i) Highlight the edges that are part of the shortest-path tree rooted at vertex a (i.e., the output of Dijkstra's algorithm if we were to start from vertex a).

	a	b	c	d	e	f
distance	0	2	3	∞	∞	∞
	0	2	3	6	7	∞
	0	2	3	6	7	∞
	0	2	3	6	7	13
	0	2	3	6	7	9

ii) Write out all positive integers x such that we could replace the weight 2 of the edge $\{e, f\}$ in the above graph by x, such that the edge would be in at least one shortest-path tree rooted at a of the resulting graph.

1 – 6

alte Prüfungsaufgabe (FS23)

Theory Task T4. / 15 P

An online travel company is developing a new feature that allows customers to search for the cheapest flight under constraints of maximum number of transfers and arrival time. A transfer refers to a change of flight at an intermediate city. For example, a trip from New York to Los Angeles with a transfer in Chicago would include one transfer.

Imagine a scenario where you have n cities, each labeled with a unique number from 1 to n. You are provided with a list that contains details for m flights, where m is greater than n. For each flight, there is a five-positive-integer tuple containing information about the flight $(i_t, j_t, w_t, b_t, l_t)$.

- i_t: The city where the flight originates.
- j_t: The city where the flight lands.
- w_t: The cost of the flight.
- b_t: The time when the flight departs.
- l_t: The time when the flight arrives.

We make a few assumptions for the problem setting:

- Assume that the process of transferring between flights is instantaneous, implying that it takes no time. This means that if you arrive at city j at time T, you can catch any subsequent flight departing from city j at any time that is later than T.
- Assume that there is only one flight between each pair of the cities(which implies $m \leq O(n^2)$).

You need to address the following aspects in your algorithm description:

- the graph algorithm used to solve this problem;
- the construction of the graph that you run this algorithm on;
- the total running time of your algorithm;
- the short justification of your solution.

/ 5 P a) Assume that you are allowed to make any number of transfers between flights. Starting from city i at time 0, you want to arrive at city j before some time $T > 0$. Devise an algorithm that determines whether there exists such a flight route. The algorithm should run in time $O(m \log n)$.

Hint: Only consider flights which arrive before time T.

Hint: For each flight $(i_t, j_t, w_t, b_t, l_t)$, create two nodes indexed by (i_t, b_t) and (j_t, l_t). Then connect a directed edge between them. You might also want to connect edges between nodes indexed by the same city, but different time.

/ 5 P b) Starting from city i at time 0, for some integer C, you want to arrive at city j before time $T > 0$ with total flight cost at most C. Suppose you can transfer arbitrarily number of times. Devise an algorithm which determines whether there exists such a flight route. The algorithm should run in time $O(m \log n)$.

/ 5 P c) Starting from city i at time 0, for some integer C, you want to arrive at city j with total flight cost at most C. Suppose you only want to transfer at most 2 times. Devise an algorithm which determines whether there exists such a flight route. The algorithm should run in time $O(m \log n)$.

an der Tafel besprochen,

siehe Lösungen auf exams.vis.ethz.ch

An Undirected, Unweighted Graph

During the semester, you have learned basic graph theory and many graph algorithms. This assignment considers an **undirected**, **unweighted** graph $G = (V, E)$ with adjacency lists. You need to implement the following tasks:

1. **IsPath()**: Check if G is a ***path*** provided that G is connected, i.e., G contains a path connecting all the vertices in V, but G does not conain any other edge.
2. **EdgeOfTriangle(u, v)**: For a given edge $e = (u, v)$, check if G contains a ***triangle*** that includes e, i.e., V contains a vertex w such that E contains all the three edges (u, w), (u, v) and (v, w).
3. **NumberOfComponents()**: Calculate the number of connected components in G.
4. **LargestPerimeter(v)**: Given a vertex v, calculate the ***largest perimeter*** from v in G provided that G is connected, i.e., the largest number of vertices equidistant from v in G.

Grading (24 points):

1. **IsPath()** (4 points): An $O(|V| + |E|)$-time implementation gets 4 points.
2. **EdgeOfTriangle(u, v)** (6 points): An $O(|V|)$-time implementation gets 6 points, while an $O(|V|^2)$-time implementation only gets 3 points.
3. **NumberOfComponents()** (6 points): An $O(|V| + |E|)$-time implementation gets 6 points.
4. **LargestPerimeter(v)** (8 points): An $O(|E|)$-time implementation gets 8 points, while an $O(|V| \cdot |E|)$-time implementation only gets 4 points.

Hier ist die Lösung, die wir zusammen implementiert haben

```java
import java.io.*;
import java.util.LinkedList;
import java.util.Scanner;
import java.util.Queue;
import java.lang.Math;
import java.lang.Integer;
import java.lang.String;


class Main {
  public static void main(String[] args) {

    char operator = In.readChar();  // type of operation
    int N;                          // number of graphs
    int n;                          // number of vertices
    int m;                          // number of edges
    int q;                          // number of querie
    int[][] edge_array;             // array of edges
    Graph G;                        // Graph

    switch (operator) {
      case 'P':
        // IsPath
        N = In.readInt();           // number of graphs
        for (int l = 0; l < N; l++) {
          n = In.readInt();         // number of vertices
          m = In.readInt();         // number of edges

          edge_array = new int[m][2];
```

```
      for (int i = 0; i < m; i++) {
        edge_array[i][0] = In.readInt();
        edge_array[i][1] = In.readInt();
      }

      G = new Graph(n, m, edge_array);
      Out.println(G.IsPath());
    }
    break;
  case 'T':
    // EdgeOfTriangle
    n = In.readInt();           // number of vertices
    m = In.readInt();           // number of edges

    edge_array = new int[m][2];
    for(int i = 0; i < m; i++) {
      edge_array[i][0] = In.readInt();
      edge_array[i][1] = In.readInt();
    }

    G = new Graph(n, m, edge_array);
    q = In.readInt();           // number of querie
    for (int i = 0; i < q; i++) {
      int u = In.readInt();
      int v = In.readInt();
      Out.println(G.EdgeOfTriangle(u, v));
    }
    break;
  case 'C':
    // NumberOfComponents
    N = In.readInt();           // number of graphs
    for(int l = 0; l < N; l++) {
      n = In.readInt();         // number of vertices
      m = In.readInt();         // number of edges

      edge_array = new int[m][2];
      for (int i = 0; i < m; i++) {
        edge_array[i][0] = In.readInt();
        edge_array[i][1] = In.readInt();
      }

      G = new Graph(n, m, edge_array);
      Out.println(G.NumberOfComponents());
    }
    break;
  case 'L':
    // LargestPerimeter
    n = In.readInt();           // number of vertices
    m = In.readInt();           // number of edges

    edge_array = new int[m][2];
    for (int i = 0; i < m; i++) {
      edge_array[i][0] = In.readInt();
      edge_array[i][1] = In.readInt();
    }

    G = new Graph(n, m, edge_array);
```

```java
        q = In.readInt();           // number of querie
        for(int i = 0; i < q; i++) {
          int v = In.readInt();
          Out.println(G.LargestPerimeter(v));
        }
        break;
      }
    }
  }

class Graph{
  private int n;              // number of vertices
  private int m;              // number of edges
  private int[] degrees;      // degrees[i]: the degree of vertex i
  private int[][] edges;      // edges[i][j]: the endpoint of the j-th edge of vertex i
  private boolean[] visited;  // visited[i]: whether vertex i has been visited

  Graph(int n, int m, int[][] edge_array) {
    this.n = n;
    this.m = m;
    degrees = new int[n];
    edges = new int[n][];
    visited = new boolean[n];

    for (int i = 0; i < n; i++) {
      degrees[i] = 0;
    }
    for (int i = 0; i < m; i++) {
      degrees[edge_array[i][0]]++;
      degrees[edge_array[i][1]]++;
    }
    for(int i = 0; i < n; i++) {
      if (degrees[i] != 0) {
        edges[i] = new int[degrees[i]];
        degrees[i] = 0;
      } else {
        edges[i] = null;
      }
    }
    for (int i = 0; i < m; i++) {
      edges[edge_array[i][0]][degrees[edge_array[i][0]]++] = edge_array[i][1];
      edges[edge_array[i][1]][degrees[edge_array[i][1]]++] = edge_array[i][0];
    }
  }

  public boolean IsPath() {
    int deg1 = 0;
    int deg2 = 0;
    for(int i=0; i<n; i++){
      if(degrees[i] == 1) deg1++;
      else if(degrees[i] == 2) deg2++;
      else return false;
    }
    if(deg1 == 2) return true;
    return false;
  }

  public boolean EdgeOfTriangle(int u, int v) {
    visited = new boolean[n];
    for(int i=0; i<edges[u].length; i++) visited[edges[u][i]] = true;
    for(int i=0; i<edges[v].length; i++){
      if(visited[edges[v][i]]) return true;
    }
    return false;
  }

```

```java
  public int NumberOfComponents() {
    visited = new boolean[n];
    int components = 0;
    for(int i=0; i<n; i++){
      if(!visited[i]){
        components++;
        DFS(i);
      }
    }
    return components;
  }

  public int LargestPerimeter(int v) {
    int[] d = new int[n];
    for(int i=0; i<n; i++) d[i] = -1;
    Queue<Integer> q = new LinkedList<Integer>();
    d[v] = 0;
    q.add(v);
    while(!q.isEmpty()){
      int u = q.poll();
      for(int i=0; i<edges[u].length; i++){
        if(d[edges[u][i]] == -1){
          d[edges[u][i]] = d[u] + 1;
          q.add(edges[u][i]);
        }
      }
    }
    int[] dist = new int[n];
    for(int i=0; i<n; i++){
      dist[d[i]]++;
    }
    int maxD = 0;
    for(int i=0; i<n; i++){
      maxD = Math.max(maxD, dist[i]);
    }
    return maxD;
  }

  private void DFS(int v) {
    visited[v]=true;
    for (int i = 0; i < degrees[v]; i++) {
      if(visited[edges[v][i]] == false) {
        DFS(edges[v][i]);
      }
    }
  }
}
```