Graph Representation

Die Aufgabe Grepr diente dazu, den Stoff der Vorlesung in einem Code umzusetzen. Die Musterlösung sieht so aus:

```
n, m = map(int, input().split())
g=[[] for _ in range(n)]
for _ in range(m):
    a, b=map(int, input().split())
    g[a].append(b)
    g[b].append(a)
print(g)
```

Illuminati

Die Limits für Illuminati sind so gewählt, dass jede "sinnvolle" Lösung im Zeitlimit durchkommt.

Gedacht war folgende Idee: Starte bei beliebigem Knoten x. Gehe über alle seine Nachbarn, d.h. finde alle Kanten $x - y$. Dann gehe über alle Nachbarn von y, also finde alle Wege $x - y - z$. Zuletzt überprüfe, ob x mit z verbunden ist. Wir finden damit alle möglichen Dreiecke. Das Dreieck der Knoten a, b, c finden wir, da wir irgendwann mit a als x starten, da b Nachbar ist, haben wir irgendwann y als b und analog dazu z als c.

```
from sys import exit
n, m = map(int, input().split())
graph = [[] for _ in range(m)]
for _ in range(m):
    a, b = map(int, input().split())
    graph[a].append(b)
    graph[b].append(a)
for x in range(n):
    for y in graph[x]:  # Pfade x-y
        for z in graph[y]:  # Pfade x-y-z
            if x in graph[z]:  #  Pfad x-y-z und Kante z-x
                print("Illuminati confirmed!")
                exit()
print("That is too far fetched.")
```

Die Laufzeit des obigen Codes ist $O(n^4)$ (n Möglichkeiten für x, y und z, und dann nochmal etwa n Durchläufe um zu testen, ob z mit x verbunden ist).

Man könnte dies mit einer Set-Datenstruktur auf $O(n^3)$ verbessern, wir untersuchen nun aber einen anderen Ansatz.

Wir starten mit einer Kante $x - y$ und testen, ob es einen Knoten z gibt, der in der gemeinsamen Nachbarschaft von x und y liegt. Die gemeinsame Nachbarschaft ist die Schnittmenge der Nachbaren von x und y. In Python kann man die Schnittmenge zweier Sets mit dem &-Operator berechnen.

```
from sys import exit
n, m = map(int, input().split())
graph = [set() for _ in range(m)]
edges = []
for _ in range(m):
    a, b = map(int, input().split())
    graph[a].add(b)
    graph[b].add(a)
    edges.append((a, b))
for x, y in edges:
    if graph[x] & graph[y]:  # x and y have common neighbour => triangle
        print("Illuminati confirmed!")
        exit()
print("That is too far fetched.")
```

Die Laufzeit davon? Gemäss Python-Wiki ist die Laufzeit von `s&t` gleich $O(min(len(s), len(t))$. Da das im Worst-Case $O(n)$ entspricht, ist die Gesamtlaufzeit des Codes $O(nm)$.

Components

Hier die Musterlösung, sie entspricht in etwa dem Code der Vorlesung. Kleinere Kommentare weiter unten:

```python
n, m = map(int, input().split())
graph = [[] for _ in range(n)]
for _ in range(m):
    a, b = map(int, input().split())
    graph[a].append(b)
    graph[b].append(a)

visited = [False for _ in range(n)]

def dfs(v):
    if visited[v]:
        return
    visited[v] = True
    for w in graph[v]:
        dfs(w)

components = 0
for v in range(n):
    if not visited[v]:
        dfs(v)
        components += 1

print(components)
```

- Anstatt *i* wird bei Loops, dessen Index nicht benötigt wird, _ benutzt. Dies ist so allgemeiner Standard bei Python.
- Es gibt zwei Arten, dfs zu implementieren: Entweder geht man davon aus, dass visited false ist und überprüft das vor jedem Aufruf. Oder man erlaubt beides und macht die Fallunterscheidung in der Funktion. Meine persönliche Vorliebe ist zweiteres, so wie es im Code oben steht.

Kürzester Weg

Wie vorher, hier der Code und Kommentare weiter unten:

```
from collections import deque
from sys import exit
n, m = map(int, input().split())
graph = [[] for _ in range(n)]
for _ in range(m):
    a, b = map(int, input().split())
    graph[a].append(b)
    graph[b].append(a)

start, target = map(int, input().split())
dist = [None for _ in range(n)]
q = deque([start])
dist[start] = 0
while q:
    v = q.popleft()
    d = dist[v]
    if v == target:
        print(d)
        exit(0)
    for w in graph[v]:
        if dist[w] is None:
            dist[w] = d + 1
            q.append(w)
print(-1)
```

- In den DFS-Slides wird appendleft/pop benutzt, hier append/popleft. Beides läuft aufs Gleiche hinaus und ist wieder Vorliebe.
- Die Distanz-Liste wird mit `None` initialisiert und später auf Zahlen gesetzt. Um zu überprüfen, ob ein Eintrag `None` ist, kann man entweder `== None` oder `is None` benutzen. Ersteres überprüft auf Wertegleichheit, zweiteres ob es das gleiche Objekt ist. Allgemein wird empfohlen, `is None` zu benutzen.
- Man könnte auch `[None]*n` schreiben anstatt `[None for _ in range(n)]`.