ALGORITHMS IN AI & DATA SCIENCE 1 (AKIDS 1)

Graphs

Prof. Dr. Goran Glavaš

23.11.2023

Content

- ADS: Graph
- Graph traversals
  - Breadth-First-Search
  - Depth-First-Search
  - Topologic sorting

From Sets to **Sets with Relations**

Image from: https://tinyurl.com/2an89h3m

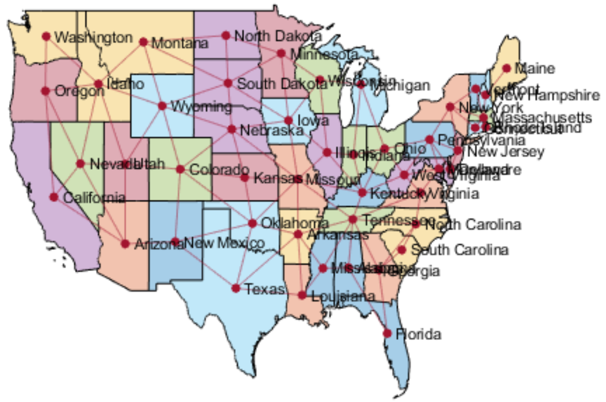


Image from: https://tinyurl.com/dk8hxy95

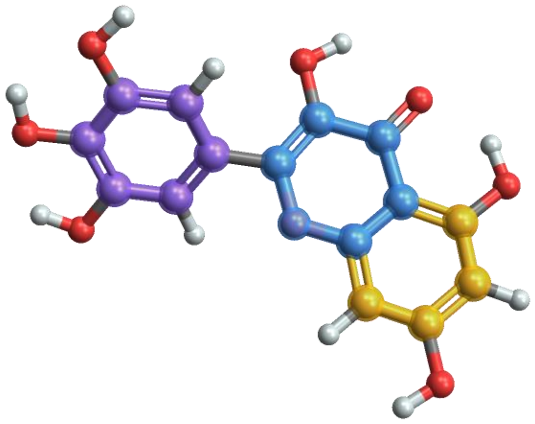


Image from: https://tinyurl.com/mxs38hpx

- In many **real-world problems and applications**, it is important not just to model the elements in a set, but also **relations/connections between the elements**
- **Graphs:** an ADS for modeling **relational data**
  - Social (and other) networks
  - Maps and geography
  - Chemistry
  - …

Graphs

Graphs

Graph is an abstract data structure for representing dynamic sets with relations: graphs consist of a (finite number of) **nodes** (also called **vertices**) representing set elements and (a finite number of) **edges** capturing relations between the elements.

- **Graph** is a more general data structure than **list** and **tree**
  - Lists and Trees can be seen as **special, reduced graphs**
    - In the same manner in which lists can be seen as a special reduced type of tree

  - **List**: a directed graph in which each node (except the last one) has exactly one outgoing edge and exactly one incoming edge (except the first one)

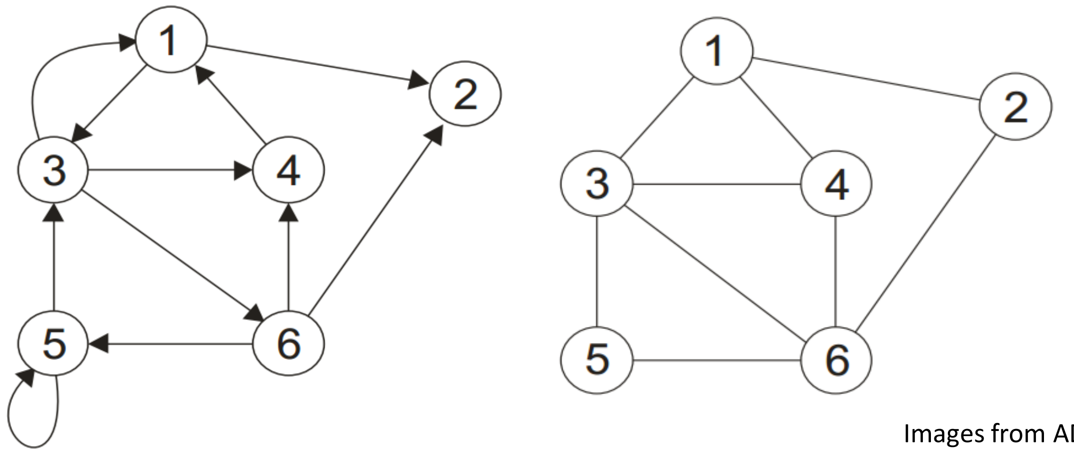
  - **Binary tree**: a graph in which each node (except the root) has exactly one incoming edge and at most two outgoing edges

Graph: definition, types

Graph: formal definition

A **graph G = (V, E)** is a pair of sets, with **V** as a set of vertices, and **E** a set of edges between the vertices **E** ⊆ { (u,v) | u, v ∈ **V**}. If the graph is **undirected**, the relation defined by an edges is symmetric, or E ⊆ {{u,v} | u, v ∈ V}, that is, edges are sets of two vertices rather than ordered pairs.

- **Directed** (gerichteter) graph – edges have directions: (u, v) ≠ (v, u)
- **Undirected** (ungerichteter) graph – edges don't have directions: {u, v} = {v, u}



Images from ADS course of Andreas Hotho

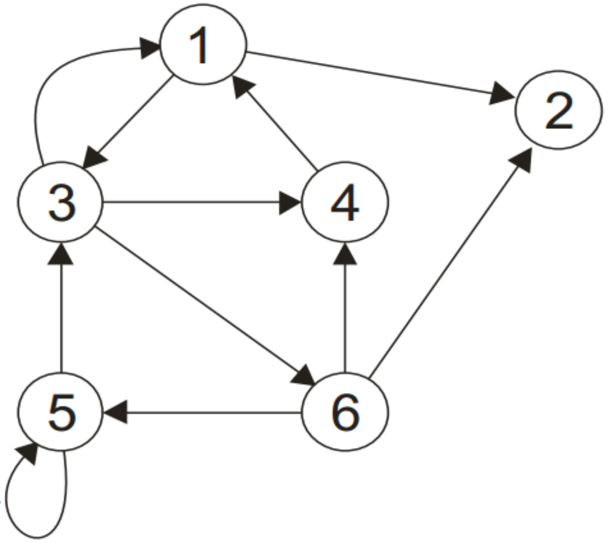
Graphs: types

- **Reflexive graphs**: nodes allowed to have an edge to itself

$G_D = (V_D, E_D)$

$V_D = \{1, 2, 3, 4, 5, 6\}$

$E_D = \{ (1, 2), (1,3), (3, 1), (3, 4), (3, 6)$
$(4, 1), (5, 3), (5, 5), (6, 2), (6, 4)$
$(6, 5) \}$



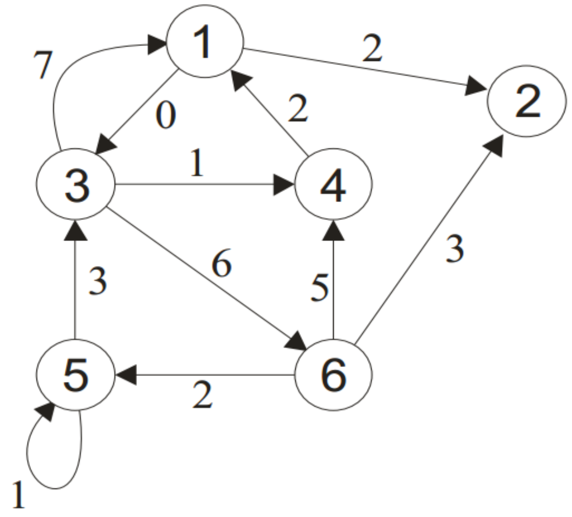
reflexive edge

Graphs: types

- **Weighted** (gewichtete) **edges** additionally have numeric weights on the edges
  - Example: distance between two cities

Weighted graph

A **weighted graph G = (V, E, γ)** is a triple with **V** as vertices, **E** as edges and γ as a function γ: **E** → $\mathbb{R}$ that assigns weights/scores to every edge (u, v) ∈ **E**.



Graphs: connectivity

- **Undirected graphs**
  - Vertices u and v **connected** if there exist a **path** (i.e., a sequence of edges) in G from u to v
  - Graph G is **connected** if any two vertices from V are connected

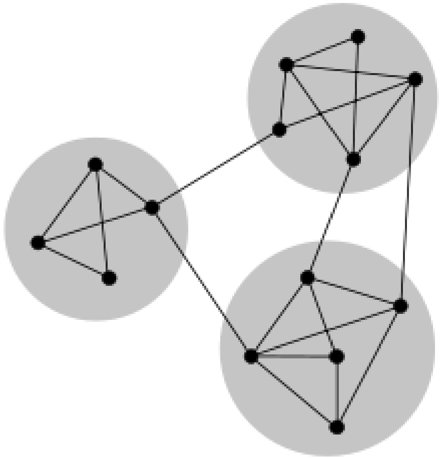
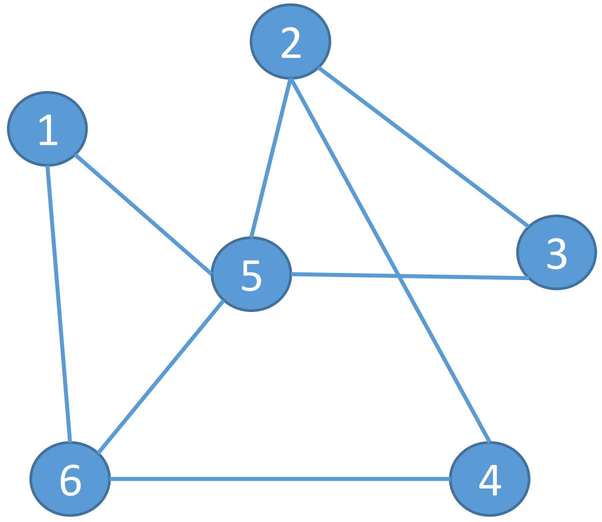


Image from Wikipedia

- **Directed graphs**
  - **Strongly connected**: if for every two nodes u, v both path from u to v and path from v to u exist
  - **Weakly connected**: if the corresponding undirected graph (make directed edges with undirected) is connected

Graphs: cycles

- **Cycle** is a **trail** (a path without repeating edges) that starts and ends in the same vertex
  - No other repetition of the vertices on the trail (otherwise it's a **circuit** and not a **cycle = simple circuit**)
  - Examples
    - 1-5-6-4-2-5-1 → circuit
    - 5-2-3-5-6-4-2-5 → circuit
    - 1-5-6-1 → cycle
    - 2-5-6-4-2 → cycle
    - 3-5-2-3 → cycle



Graphs as dynamic sets

- Graphs represent a **dynamic set** with relations between elements
- Three most common operations apply:
  - INSERT: add a node to a graph
  - DELETE: remove a node from a graph
  - SEARCH: find a node in a list
- **Runtime complexity** of algorithms on graphs
  - Runtime no longer dependent just on number of elements/ vertices (n = |**V**|), but also on the number of edges in the graph (|**E**|)
    - For simplicity, in **O-notation**, we'll write V for |**V**| and E for |**E**|
- **Pseudocode**: G – graph, G.V – set of graph vertices, G.E – set of graph edges

Graph representations

- Two common ways to represent graph
  - **Adjacency list**
  - **Adjacency matrix**

- Decision on **which one to use** is usually linked to the **density** of the graphs we're expected to represent
  - Density of a graph: $|E| / |V|^2$
  - Max. number of edges in a (reflexive directed) graph with $|V|$ nodes is $|V|^2$
  - **Q**: What is the maximal number of edges in a directed graph with $|V|$ vertices?

- **Adj. list** – more suitable for sparse graphs (most graphs are sparse)
- **Adj. matrix** – more suitable for dense graphs
  - Also convenient for graph operations that can be expressed as mathematical operations on the adjacency matrix of the graph, e.g., to compute G^k

Graph representations: **adjacency list**

- G(V, E) represented as an array/list of size |V|, each element of which corresponds to one vertex u ∈ **V** and is a pointer (head) to the list containing its neighbouring nodes {v ∈ **V** : (u, v) ∈ **E**}
- In pseudocode, we will indicate the adjacency list as G.Adj
  - **Q1:** if G is an (un)directed graph, what is the sum of lengths of all adj. lists?
  - **Q2:** what is the space (O notation) needed for storing G as adj. list?

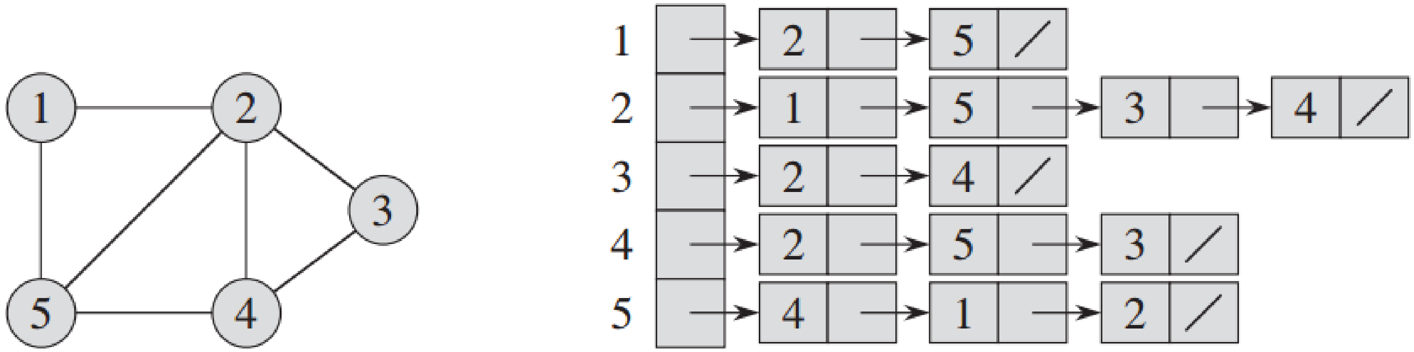
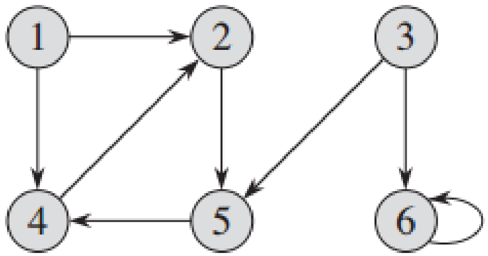


Image from Cormen et al.

Adjacency list representation of an undirected (unweighted) graph

Graph representations: **adjacency list**

- **Q:** How to represent weighted graphs as adjacency **lists**?
  - We just add the weight next to target node in each list element
- Search: is edge (u, v) in **E**?
  - Runtime? What's the average length of an adjacency list?



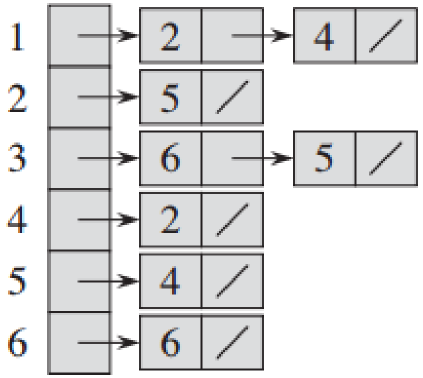
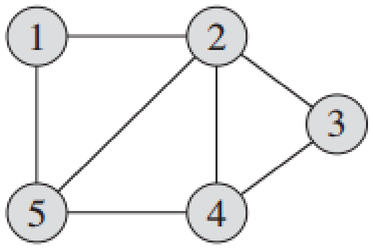


Image from Cormen et al.

Adjacency list representation of a directed (unweighted) graph

Graph representations: **adjacency matrix**

- G(V, E) represented as a matrix (2D-array) of size $|V|^2$, each element of which corresponds to one potential edge (u, v).
- **G.A** – the adjacency matrix (in pseudocode and algorithms)
  - a_{ij} – the element at the i-th row and j-th column of **A** – the value of that matrix element indicates if an edge between i-th and j-th vertex in **V**
  - **Q1:** if G is an (un)directed graph, what is the number of non-zero elements in **A**?
  - **Q2:** what is the space (O notation) needed for storing G as adj.matrix?



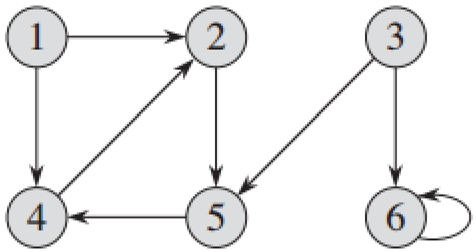
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Image from Cormen et al.

Adjacency matrix representation of an undirected (unweighted) graph

Graph representations: **adjacency matrix**

- **Q:** How to represent a weighted graph as an adjacency **matrix**?
  - We just replace binary values in the matrix with weights
- Search: is edge (u, v) in **E**?
  - Runtime? Assuming we know the indices of u and v in **V**



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Image from Cormen et al.

Adjacency matrix representation of a directed (unweighted) graph

Graph representations

- **Adjacency lists** vs. **adjacency matrix**

	Space	Edge search (runtime)
Adj. list	O(V + E)	O(E/V)
Adj. matrix	O(V^2)	O(1)

- Many applications deal with very large and very sparse graphs
  - $|E| \ll |V|^2$
  - For example, **social networks**
  - Storing a **matrix** with $|V|^2$ elements not feasible
    - Example: Facebook graph has **2.797 billion** (Miliarden) nodes (users)
    - **Adj. matrix** would have ca $8*10^{18}$ elements → need **exabytes** of memory
- If graphs are reasonably small (memory not an issue), we prefer **adj. matrix**
  - Not just for faster edge search, it is also a conceptually simpler graph representation

Content

- ADS: Graph
- Graph traversals
  - Breadth-First-Search
  - Depth-First-Search
  - Topologic sorting

Graph traversals

Graph traversal

Given a graph G = (**V**, **E**) and a source/starting vertex $s \in$ **V**, discover all vertices that can be reached from s.

- **Breadth-first search** (BFS): traverses the graph in a **First-In-First-Out order** – the vertex reached last is placed at the end of the exploration list
  - **Q:** what data structure is suitable for this?
- **Depth-first search** (DFS): traverses the graph in **Last-In-First-Out** order – the vertex reached last is the first to be explored next
  - **Q:** what data structure is suitable for this?
- **Topological sorting**: linearizes an (acyclic directed) graph via DFS
  - Kind of like creating a sorted array from (in-order traversal of) binary search tree

Graph traversals: breadth-first search

- Arguably the simplest algorithm for searching/traversing a graph
  - Its principle is also used in many more complex graph algorithms
  - Same algorithm applicable to both directed and undirected graphs
- For graph G and **source vertex** *s*, return all nodes reachable from *s*
  - Additionally, computes the (minimal) distance of each node from *s*
- BFS also creates a „breadth first search tree" of the node *s*
  - Shortest path from each node *v* is the path from *v* to the root (*s*) in that tree
  - **Uniform expansion** of the „search frontier" – all vertices at distance k from source *s* will be visited before any of the vertices at distance k+1

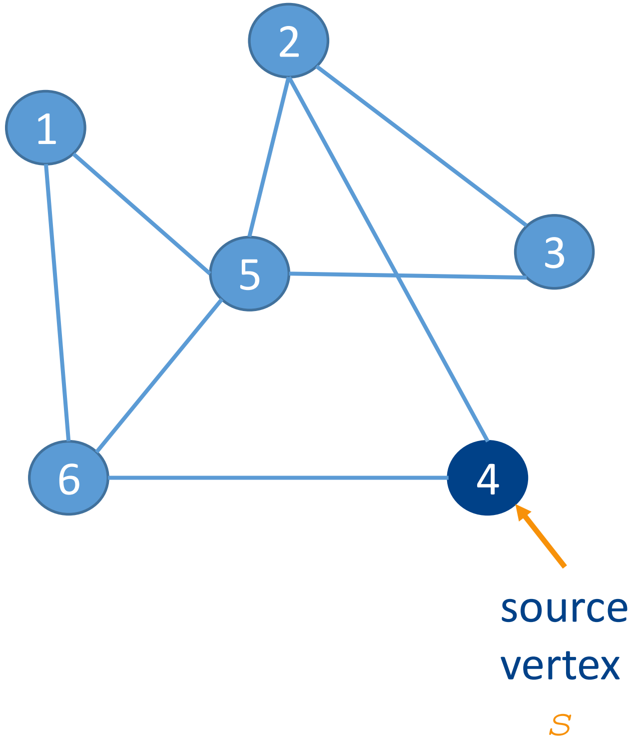
Graph traversals: breadth-first search

- Vertices of the node can be in three different „states" (for vertex *u*, `u.state`)
  - **undiscovered** (value 0; initially all except source *s*)
  - **discovered** (value 1): reached but not expanded
  - **expanded** (value 2): all vertices directly reachable from that vertex have been discovered
- We use the **Queue** data structure
  - to make sure that the vertices are expanded **in the same order** in which they are discovered
- We will assume the **adjacency list** implementation of the graph

```
bfs(G, s)
  for each vertex u in G.V-{s}
    u.state = 0
    u.dist = inf # big int
    u.parent = null

  qd = [] # empty queue
  s.state = 1 # discovered
  enqueue(qd, s)
  while not is_empty(qd)
    u = dequeue(qd)
    for vertex v in G.Adj[u]
      if v.state == 0 # so far undiscovered
        v.state == 1 # discovered
        v.dist == u.dist + 1
        v.parent = u
        enqueue(qd, v)
    u.state = 2
```

Graph traversals: breadth-first search

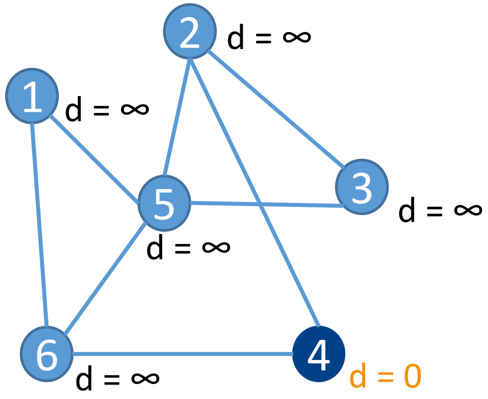
- Illustration: colors indicate states
  - undiscovered
  - discovered
  - expanded



- Initially, only `s` is discovered
- **Step 1**:
  - `s` is expanded
  - all vertices directly reachable from `s` are discovered (and queued)

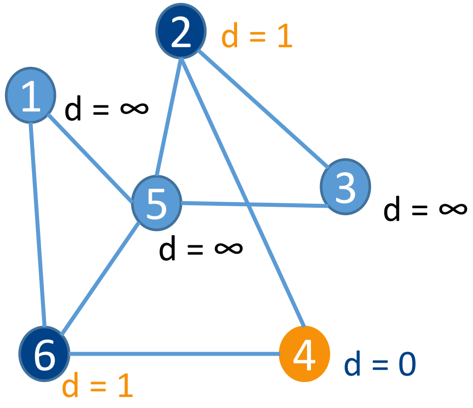
Graph traversals: breadth-first search

qd = [4]



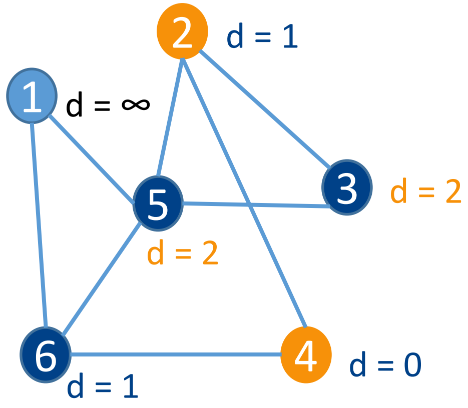
After initialization

qd = [2, 6]



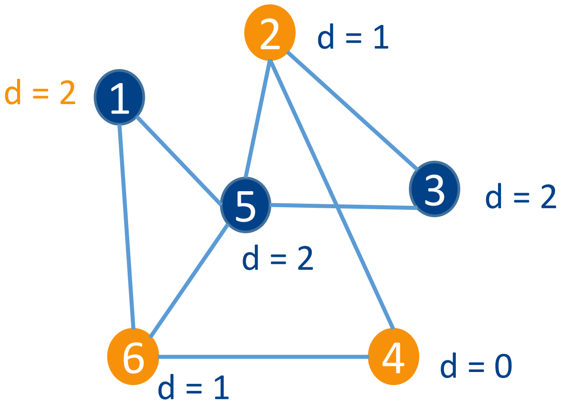
After **1.** iteration (of while loop)

qd = [6, 3, 5]



After **2.** iteration

qd = [3, 5, 1]

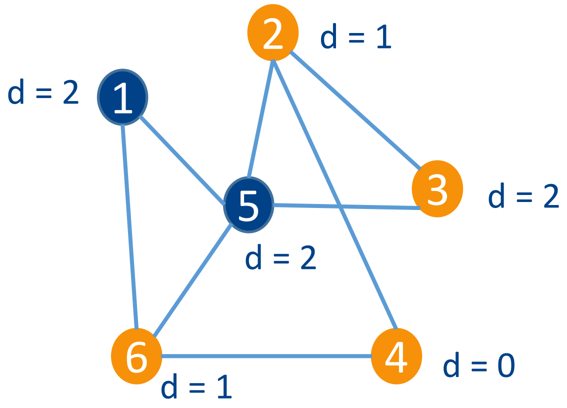


After **3.** iteration

```
bfs(G, s)
  for each vertex u in G.V-{s}
    u.state = 0
    u.dist = inf
    u.parent = null
  qd = []
  s.state = 1
  enqueue(qd, s)
  while not is_empty(qd)
    u = dequeue(qd)
    for vertex v in G.Adj[u]
      if v.state == 0
        v.state == 1
        v.dist == u.dist + 1
        v.parent = u
        enqueue(qd, v)
    u.state = 2
```

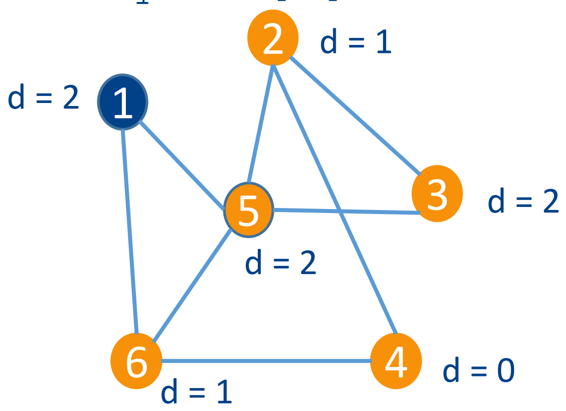
Graph traversals: breadth-first search

`qd = [5, 1]`



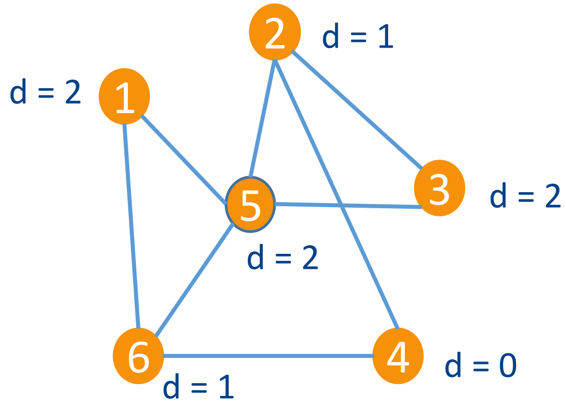
After **4.** iteration

`qd = [1]`



After **5.** iteration

`qd = [] → end`

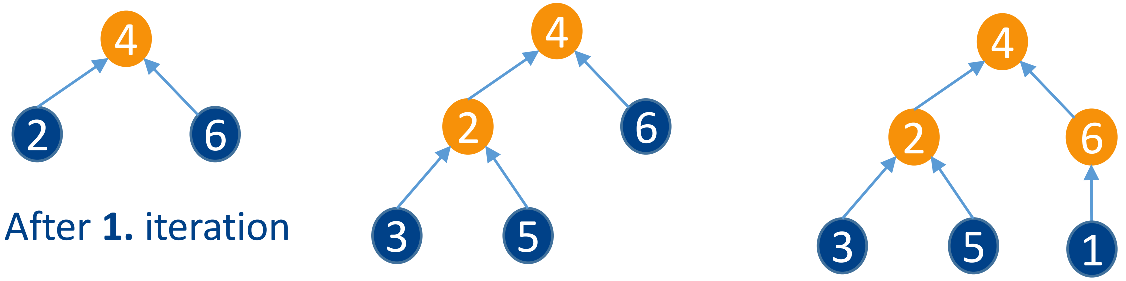


After **6.** iteration

```
bfs(G, s)
  for each vertex u in G.V-{s}
    u.state = 0
    u.dist = inf
    u.parent = null

  qd = []
  s.state = 1
  enqueue(qd, s)
  while not is_empty(qd)
    u = dequeue(qd)
    for vertex v in G.Adj[u]
      if v.state == 0
        v.state == 1
        v.dist == u.dist + 1
        v.parent = u
        enqueue(qd, v)
    u.state = 2
```

Graph traversals: breadth-first search

- How is BFS building a tree?



After **1.** iteration

After **2.** iteration

After **3.** iteration

- Going from each node `v` to the root `s`
  - Gives the shortest path in the graph from `v` to `s`
- **Q**: what is the runtime of BFS?
  - Each vertex is (en/de)queued **at most** once
    - **Q:** can a vertex **not be queued at all**?
  - **O(V+E)**: why?
    - How many iterations of the for loop will you have in total (summed across all iterations of the while loop)?

```
bfs(G, s)
  for each vertex u in G.V-{s}
    u.state = 0
    u.dist = inf
    u.parent = null

  qd = []
  s.state = 1
  enqueue(qd, s)
  while not is_empty(qd)
    u = dequeue(qd)
    for vertex v in G.Adj[u]
      if v.state == 0
        v.state == 1
        v.dist == u.dist + 1
        v.parent = u
        enqueue(qd, v)
    u.state = 2
```

Content

- ADS: Graph
- Graph traversals
  - Breadth-First-Search
  - Depth-First-Search
  - Topologic sorting

Graph traversals: depth-first search

- Together with BFS, depth-first search is the most basic/common algorithm for searching/traversing a graph
  - Its principle is also used in many more complex graph algorithms
  - Same algorithm applicable to both directed and undirected graphs
- For graph G and **source vertex** *s*, return all nodes reachable from *s*
  - Additionally, can compute the (minimal) distance of each node from *s*
  - Though less suitable for „shortest paths” than BFS
- DFS deepens the search by always expanding the first newly discovered vertex (last in, first out :))
  - In contrast to BFS, **distance at discovery** is not necessarily the shortest distance from the source s

Graph traversals: depth-first search

- Vertices of the node can be in three different „states" (for vertex u, `u.state`)
  - **not visited** (value 0; initially all except source s)
  - **visited** (value 1): reached (and then immediately expanded)
  - If we want **shortest distances**, then nodes may be **revisited**
- We use the **Stack** data structure
  - to make sure that the most recently discovered vertix is expanded first
- We assume the **adjacency list** implementation of the graph

```
dfs(G, s) # non-recursive
  for each vertex u in G.V-{s}
    u.state = 0
    u.dist = 0
    u.parent = null

  stack = [] # empty stack
  s.state = 1 # visited
  push(stack, (s, 0))

  while not is_empty(stack)
    u, time = pop(stack)
    for vertex v in G.Adj[u]
      if v.state == 0 or time+1 < v.dist
        v.state = 1 # visited
        v.dist = time + 1
        v.parent = u
        push(stack, (v, time+1))
```

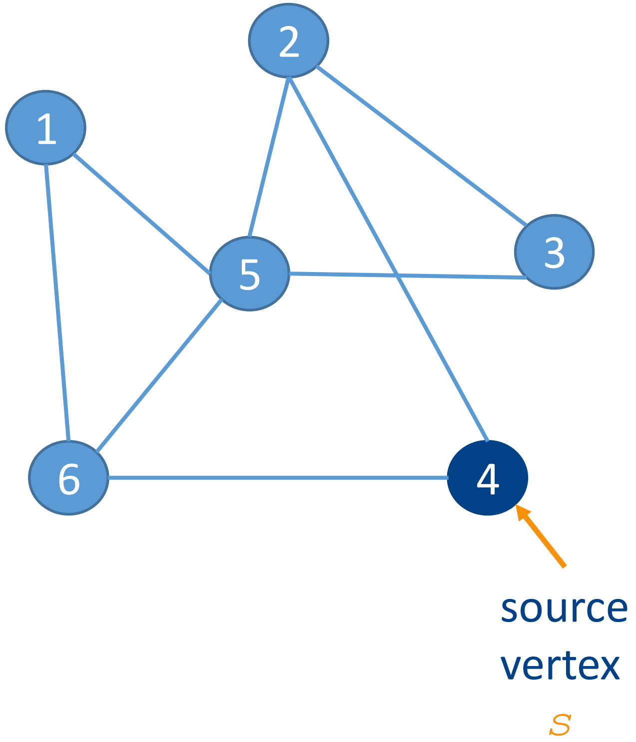
Graph traversals: depth-first search

- The pseudocode of the DFS is iterative
- But DFS naturally lends itself to **recursion**
  - Why?

- **Exercise**
  - Write the **recursive** DFS algorithm
  - Recursive DFS doesn't require (an explicit) **stack**
  - Where is the **stack** hidden in that case?

Graph traversals: depth-first search

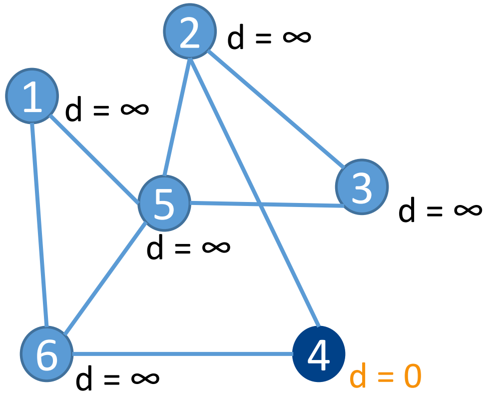
- Illustration: colors indicate states
  - unvisited
  - visited
  - revisited



- Initially, only `s` is **visited**
- **Step 1**:
  - `s` is visited
  - All vertices directly reachable from `s` are **pushed to the stack**

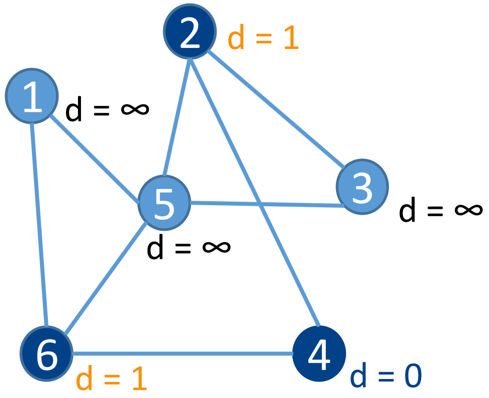
Graph traversals: depth-first search

stack = [(4, t=0)]



After initialization (time 0)

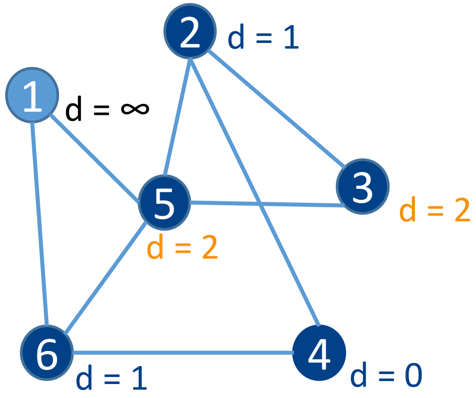
stack = [(2,t=1), (6,t=1)]



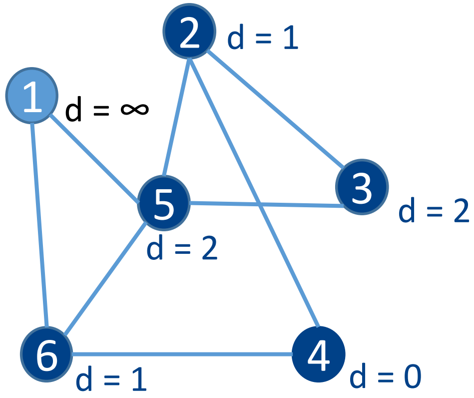
After **1**. iteration (while loop)

```
dfs(G, s) # non-recursive
  for each vertex u in G.V-{s}
    u.state = 0
    u.dist = 0
    u.parent = null

  stack = [] # empty stack
  s.state = 1 # visited
  push(stack, (s, 0))

  while not is_empty(stack)
    u, time = pop(stack)
    for vertex v in G.Adj[u]
      if v.state == 0 or time+1 < v.dist
        v.state = 1 # visited
        v.parent = u
        v.dist = time + 1
        push(stack, (v, time+1))
```

stack = [(3,t=2), (5,t=2),(6,t=1)]



After **2**. iteration

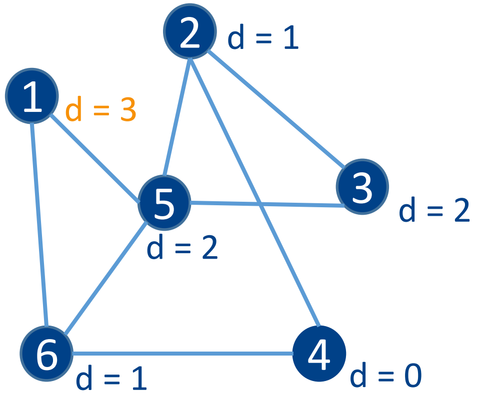


After **3**. iteration

stack = [(5,t=2),(6,t=1)]

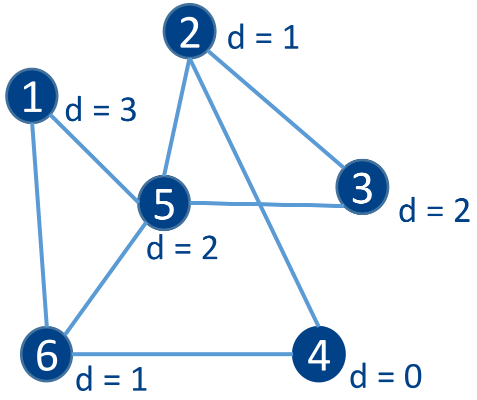
Graph traversals: depth-first search

stack = [(1,t=3),(6,t=1)]



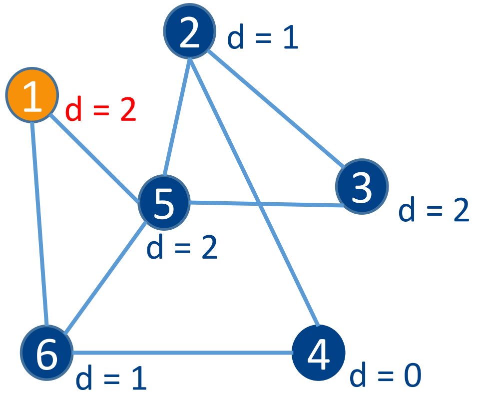
After **4**. iteration

stack = [(6,t=1)]



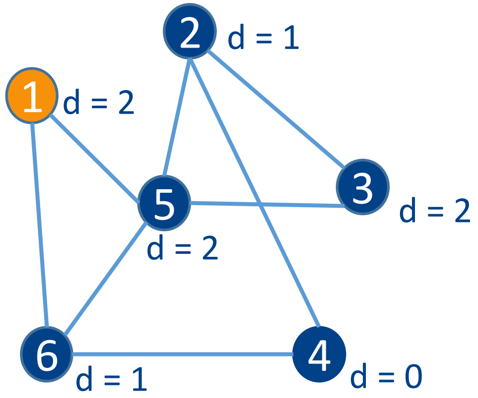
After **5**. iteration

stack = [(1,**t=2**)]



After **6**. iteration

stack = [] → **end**



After **7**. iteration

```
dfs(G, s) # non-recursive
  for each vertex u in G.V-{s}
    u.state = 0
    u.dist = 0
    u.parent = null

  stack = [] # empty stack
  s.state = 1 # visited
  push(stack, (s, 0))

  while not is_empty(stack)
    u, time = pop(stack)
    for vertex v in G.Adj[u]
      if v.state == 0 or time+1 < v.dist
        v.state = 1 # visited
        v.parent = u
        v.dist = time + 1
        push(stack, (v, time+1))
```

Graph traversals: depth-first search

- DFS is, however, not really used for computing shortest distances
  - To get shortest distances correctly, DFS may **revisit** the vertices → **slower**
  - If we need shortest distances, use BFS
- No distances → the DFS algorithm is simpler
  - **Exercise**: write the recursive version of this simplified DFS too
- **Q:** Runtime of DFS (without shortest distances)? Compare the execution to BFS

```
dfs(G, s) # non-recursive
  for each vertex u in G.V-{s}
    u.state = 0

  stack = [] # empty stack
  s.state = 1 # visited
  push(stack, s)

  while not is_empty(stack)
    u = pop(stack)
    # print(u)
    for vertex v in G.Adj[u]
      if v.state == 0
        v.state = 1 # visited
        push(stack, v)
```

Content

- ADS: Graph
- Graph traversals
  - Breadth-First-Search
  - Depth-First-Search
  - Topologic sorting

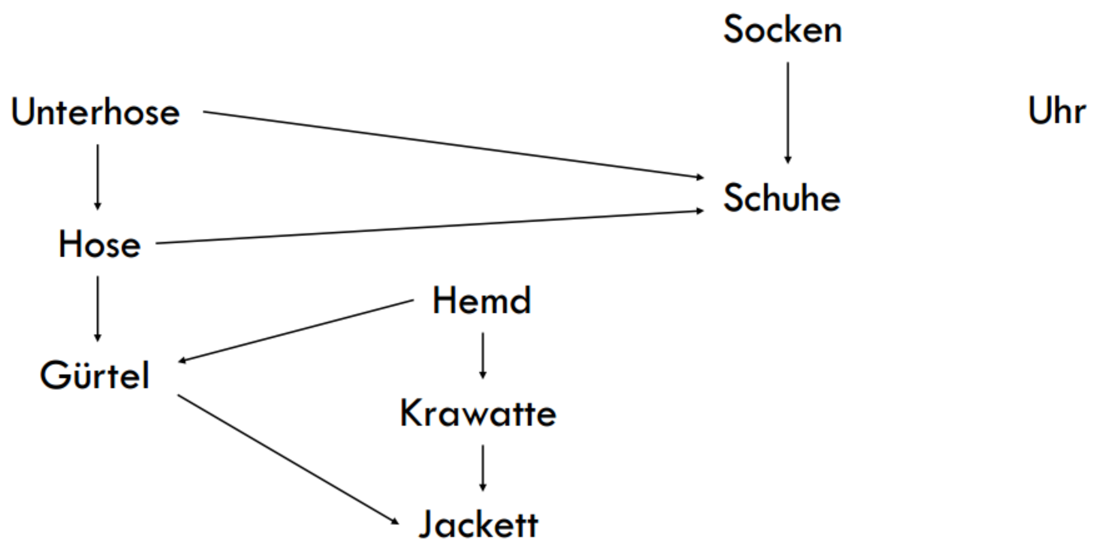
DFS for Topological Sort

- **Q:** How do we sort/linearize a graph?
  - Something **analogous** to `inorder_walk` for binary search trees
  - Is sorting even a meaningful operation for graphs?
- Linear ordering for directed acyclic graphs (DAGs)
  - Graphs „closest” to a tree
  - Edges are directed and there are no cycles
  - But each node can have multiple „parents”
- **Topological sorting**: linearization of DAGs with DFS

DFS for Topological Sort

- Applications with **precedencies** between events
  - Edges in a directed graph naturally specify a **precedence relation**
  - Problem: **find an order** in which to execute all tasks (vertices in the graph) such that it is **consistent** with the expressed precedencies (edges)

- **Example**: dressing up
  the „confused" professor
- **Solution:**
  - Iteratively run DFSs from vertices that have no incoming edges



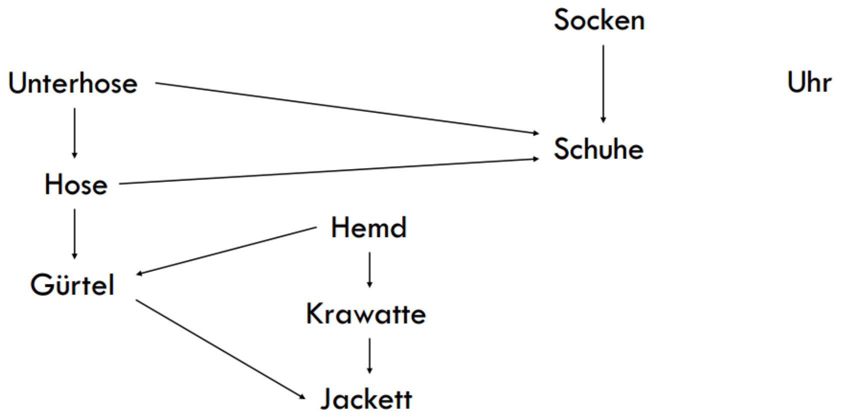
DFS for Topological Sort

- We're working with single-source DFS (DFS that starts from a given source node)
- For **topological sort**, we add two more properties to the nodes
  - v.num_in – number of unvisited incoming edges
  - v.has_in – indicates whether the vertex has any incoming edges
- We slightly modify DFS to adjust the counter of unvisited incoming edges when we reach the node

```
dfs(G, s) # non-recursive
  for each vertex u in G.V-{s}
    u.state = 0

  stack = [] # empty stack
  s.state = 1 # visited
  push(stack, s)

  while not is_empty(stack)
    u = pop(stack)

    if u.num_in == 0
      print(u)

    for vertex v in G.Adj[u]
      if v.state == 0
        v.state = 1 # visited
        v.num_in = v.num_in - 1

        if v.num_in == 0
          push(stack, v)
```

DFS for Topological Sort

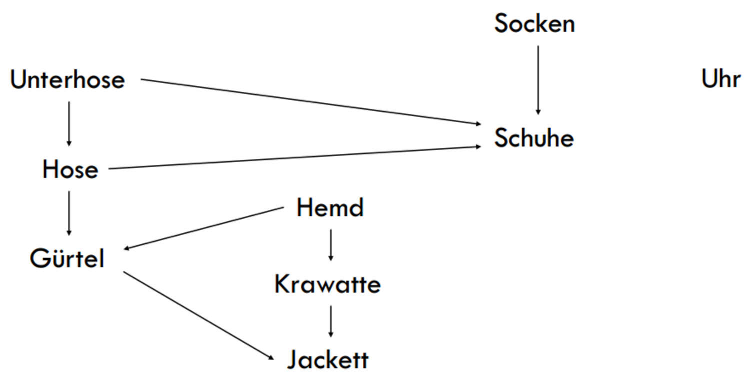
```
topological_sort(G)
  for each vertex u in G.V
    u.num_in = 0

  for each edge (u, v) in G.E
    v.num_in = v.num_in + 1

  for each vertex u in G.V
    if u.num_in == 0
       u.has_in = 0
    else
       u.has_in = 1

  for each vertex u in G.V
    if u.has_in == 0
      dfs(G, u)
```




```
dfs(G, s) # non-recursive
  for each vertex u in G.V-{s}
    u.state = 0

  stack = [] # empty stack
  s.state = 1 # visited
  push(stack, s)

  while not is_empty(stack)
    u = pop(stack)

    if u.num_in == 0
      print(u)

    for vertex v in G.Adj[u]
      if v.state == 0
        v.state = 1 # visited
        v.num_in = v.num_in - 1

        if v.num_in == 0
          push(stack, v)
```

DFS for Topological Sort

```
topological_sort(G)
  for each vertex u in G.V
    u.num_in = 0

  for each edge (u, v) in G.E
    v.num_in = v.num_in + 1

  for each vertex u in G.V
    if u.num_in == 0
       u.has_in = 0
    else
       u.has_in = 1

  for each vertex u in G.V
    if u.has_in == 0
      dfs(G, u)
```



- V = [Hose, Socken, Unterhose, Gürtel, Schuhe, Jackett, Krawatte, Uhr, Hemd]
- Q: On which „source“ vertices will DFS be called?

- DFS #1 (Socken): prints Socken
  - Why not Schuhe?
- DFS #2 (Unterhose): prints Unterhose -> Hose -> Schuhe
  - Why not Gürtel?
- DFS #3 (Uhr): prints Uhr
- DFS #4 (Hemd): prints Hemd -> Gürtel -> Krawatte -> Jackett

Questions?


Awọn ibeere?
ਸਵਾਲ?
Küsimusi?
Turite klausimų?
Pitanja?
Sorusu olan?
Dúvidas?
質問は？
有问题吗？
Fragen?
¿Preguntas?
Frågor?
Questions?
Pytania?
Domande?
Питання?
Vragen?
Ερωτήσεις;
Porandukuéra?
คำถาม?
أسئلة؟