Quiz

If a directed graph has no directed cycles then there is no back edge in any DFS tree of it.

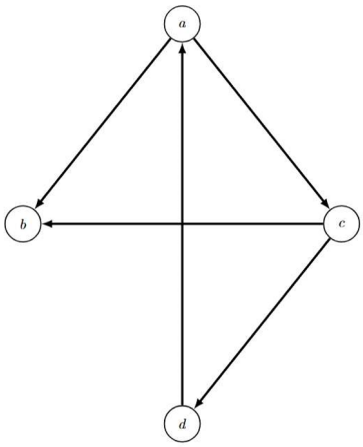
(True)
False

∃ back edge <=> ∃ directed cycle

shortest path 3

10 min

What is a valid adjacency list for vertex c?



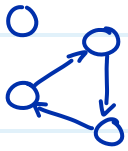
a. $[a]$

(b.) $[b, d]$

c. $[a, b, d]$

Decide whether the following are true or false.

True	False	
	X	If a directed graph has a sink, then it has no directed cycle.
X		If a directed graph has no directed cycle, then it has a sink.



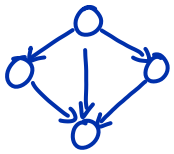
Let u, v be two vertices connected via an edge in an **undirected graph**. Suppose we compute a DFS tree of this graph. It is possible that $pre(u) < post(u) < pre(v) < post(v)$.

True
(False)

|—||—| not possible

Suppose we have a directed graph with 2 different directed cycles. We need to remove at least 2 edges for a topological order to exist in the graph.

True
(False)



Recall the notation S_i^u used to refer to the set of vertices at distance i from some vertex u, in some directed graph D. Assume that there exists a vertex w with $w \in S_2^u$ and $w \in S_3^v$. Then, the graph distance from u to v in D is at most 5.

True

False

Suppose that we run a BFS on a directed graph and after sorting by the enter-time the vertices are d, e, b, a, c, f. What would the order be if we sorted by leave-time instead?

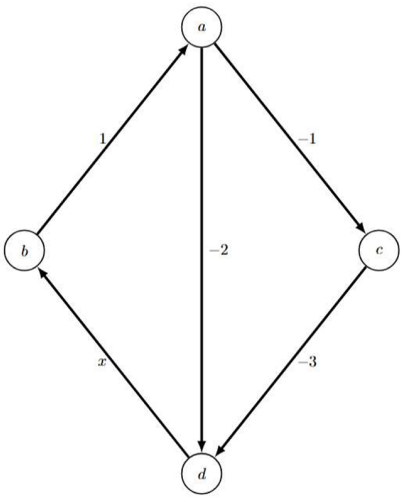
a. $f, c, a, b, e, d.$

b. $d, e, b, a, c, f.$

c. $a, b, c, d, e, f.$

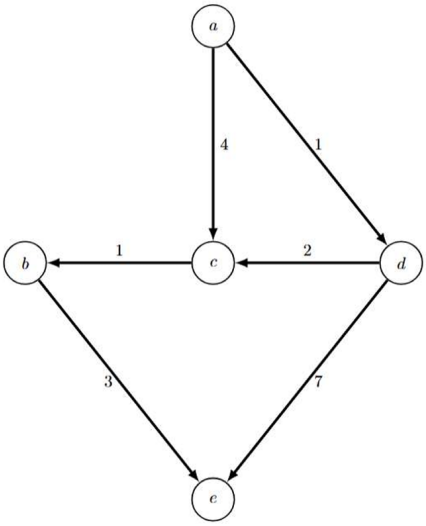
d. $f, e, d, c, b, a.$

e. Impossible to specify with the current information.



What is the minimum value of x for which there is no negative directed cycle?

Answer: 3



What is the length of the shortest path from a to e?

Answer: 7

Consider the queue $Q = [3, 5, 2, 4, 1]$. Suppose we carry out the following operations:

1. dequeue
2. dequeue
3. enqueue(1)
4. enqueue(4)

What is the final state of the queue? The enqueue operation adds an element to the right/end of the queue.

a. $Q = [3, 5, 2, 4, 1]$

b. $Q = [3, 5, 2, 1, 4]$

c. $Q = [2, 4, 1, 1, 4]$

d. $Q = [2, 4, 1, 4, 1]$

Graph proofs

BFS

$dist(u,v)$ ist die kürzeste Länge eines Weges von u nach v (Anzahl Kanten)

$S_k = \{ v \in V \mid dist(s,v) = k \}$ ist die Menge aller Knoten, die genau Distanz k vom Startknoten s entfernt sind

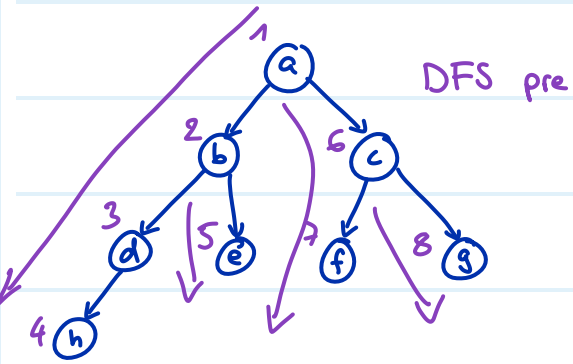
BFS berechnet die kürzesten Wege vom Startknoten s zu allen anderen Knoten

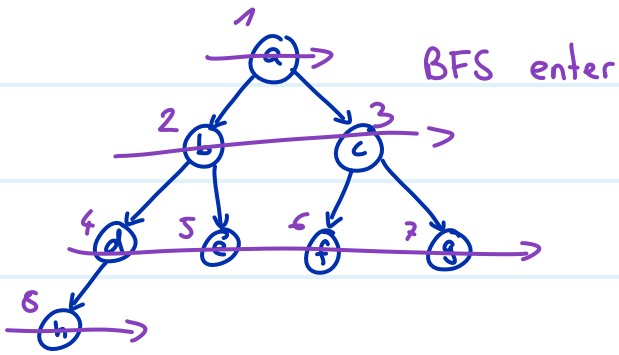
Algorithm 1 BFS(G, s) — Annahme G ungewichtet

```
1:  for each v ∈ V do
2:      visited[v] = false          ▷ Initialize visited status
3:      dist[v] ← ∞                 ▷ Initialize distance
4:  Q ← ∅                           ▷ Initialize empty queue Q
5:  visited[s] ← true               ▷ Mark start node as discovered
6:  dist[s] ← 0                     ▷ Distance to start node is 0
7:  ENQUEUE(s, Q)                   ▷ Add s to queue Q
8:
9:  while Q ≠ ∅ do
10:     u ← DEQUEUE(Q)              ▷ Current node
11:     for each (u, v) ∈ E do
12:         if visited[v] == false then   ▷ If v not yet discovered
13:             visited[v] ← true         ▷ Mark v as discovered
14:             dist[v] ← dist[u] + 1     ▷ BFS distance update
15:             ENQUEUE(v, Q)             ▷ Add v to queue Q
```

Initialisierungs-Teil

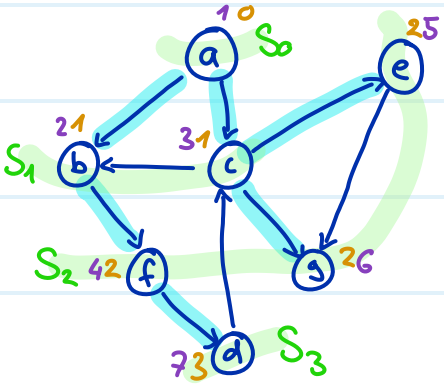
Laufzeit $O(n+m)$ mit Adjazenzliste





Achtung ob mit 0 oder 1 beginnen, und ob ihr nur enter berechnet oder auch leave

BFS mit Startknoten a



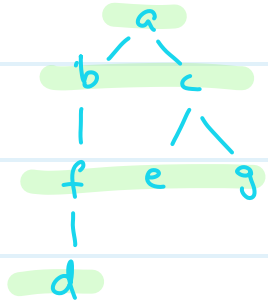
dist[]

enter[]

Level Sets

Shortest Path tree edges

Shortest Path tree



enter order = leave order

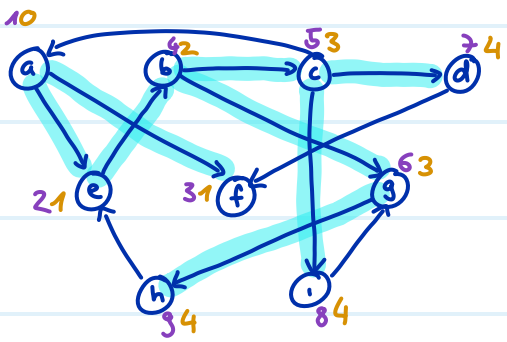
BFS mit Startknoten a

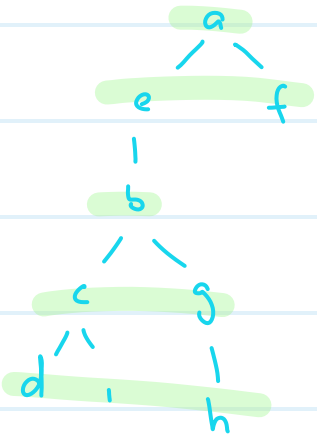
dist[]

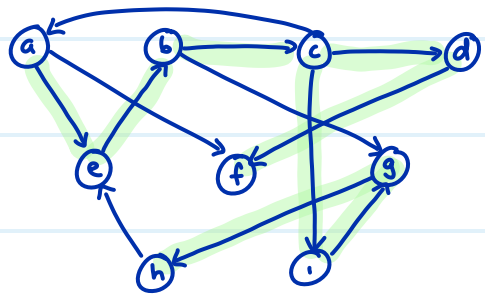
enter[]

Level Sets

Shortest Path tree edges







Zum Vergleich: Tiefensuchbaum

BFS	DFS
$O(n+m)$	$O(n+m)$
gerichtet / ungerichtet	gerichtet / ungerichtet
findet alle erreichbaren Knoten	findet alle erreichbaren Knoten (ohne Rahmenprogramm)
—	umgekehrte post order ist top Sortierung (falls G ein DAG ist)
Berechnet Distanzen vom Starthnoten (gerichtet/ungerichtet, falls es keine oder uniforme (alle gleich) positive Kantengewichte gibt)	—
verwendet eine Queue	verwendet Rekursion oder einen Stack

Max Landschoof (mlandschoof@student.ethz.ch)	Mario Malis (mmalis@student.ethz.ch)
Samuel Burkhardt (sburkhard@student.ethz.ch)	Moritz Becker (mobecker@student.ethz.ch)
Jonas Lüthi (joluethi@student.ethz.ch)	Tobias Hagen (tohagen@student.ethz.ch)
Benjamin Stupf (bstupf@student.ethz.ch)	Dominic Aschmann (daschmann@student.ethz.ch)
Nicolas Holzmann (nholzmann@student.ethz.ch)	Jolanda Bellert (jbellert@student.ethz.ch)
Matthias Lüthi (luethm@student.ethz.ch)	Danijel Jovanovic (danijelj@student.ethz.ch)
Francisco Zanatta Janzen (fzanatta@student.ethz.ch)	Patrick Back (pback@student.ethz.ch)
Marc Leonard Overbeck (moverbeck@student.ethz.ch)	Luca Schnellmann (lschnellman@student.ethz.ch)
Haochen Zhu (haoczhu@student.ethz.ch)	Roman Huber (romahuber@student.ethz.ch)
Lucien Gees (lugees@student.ethz.ch)	Mattia Niggli (nigglima@student.ethz.ch)
Siqi Wang (siqwang@student.ethz.ch)	Yannis Grimm (ygrimm@student.ethz.ch)
Timo Bruhin (tbruhin@student.ethz.ch)	Gaeacrist Salmo (gsalmo@student.ethz.ch)

Kürzeste Wege in Gewichteten Graphen

$c(u,v)$ ist das Gewicht der Kante von u nach v, es darf negativ sein

$c(W)$ ist das Gewicht des Weges W und ist die Summe der Kantengewichte von W

$d(u,v)$ ist das kleinstmögliche Gewicht eines Weges von u nach v

Annahme: es gibt keinen negativen Zyklus

Für gerichtete azyklische Graphen:

Finde topologische Sortierung $v_1, v_2, \ldots, v_n$ mit DFS

Sei der Startknoten $s = v_k$

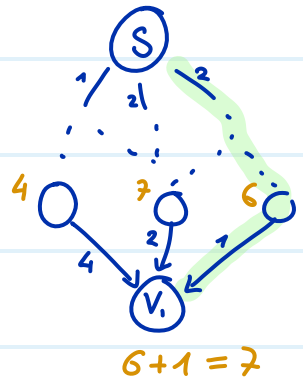
Dimension: $DP[1 \ldots n]$

Teilproblem: $DP[i] = d(s, v_i)$ = Distanz vom Startknoten s zu v_i

Base Case: $DP[k] = 0$

Alle anderen DP Einträge werden mit ∞ initialisiert

Rekursion: $DP[i] = \underbrace{\min \{ DP[j] + c(v_j, v_i) \mid (v_j, v_i) \in E \}}_{= \infty \text{ falls } v_i \text{ keine Vorgänger hat, ie } deg_{in}(v_i) = 0}$



Berechnungsreihenfolge: For $i = (k+1) \ldots n$:
compute $DP[i]$

Lösung: $\forall 1 \leq i \leq n: d(s, v_i) = DP[i]$

Laufzeit: $O(n+m)$

Dijkstra

Bedingung: es gibt keine negativen Kantengewichte

DIJKSTRA$(G = (V, E), s)$ in $O((n+m) \cdot \log n)$

```
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
```

S ist die Menge aller abgeschlossenen Knoten für die wir wissen dass $d[v] = d(s,v)$

$d[v]$ ist zu jeder Zeit eine obere Schranke für $d(s,v)$, d.h es kann nur noch kleiner werden

Invariante: (gilt in Zeile 7)

$$\forall v \in V: \; d[v] = \begin{cases} d(s,v) & \text{falls } v \in S \\ \min\{d(s,u) + c(u,v) \mid u \in S \text{ und } (u,v) \in E\} & \text{sonst} \end{cases}$$

→ d.h $v \notin S$ aber benachbart zu einem Knoten $u \in S$

$= \infty$ falls es kein solches u gibt

$p[v]$ speichert den Vorgänger von v

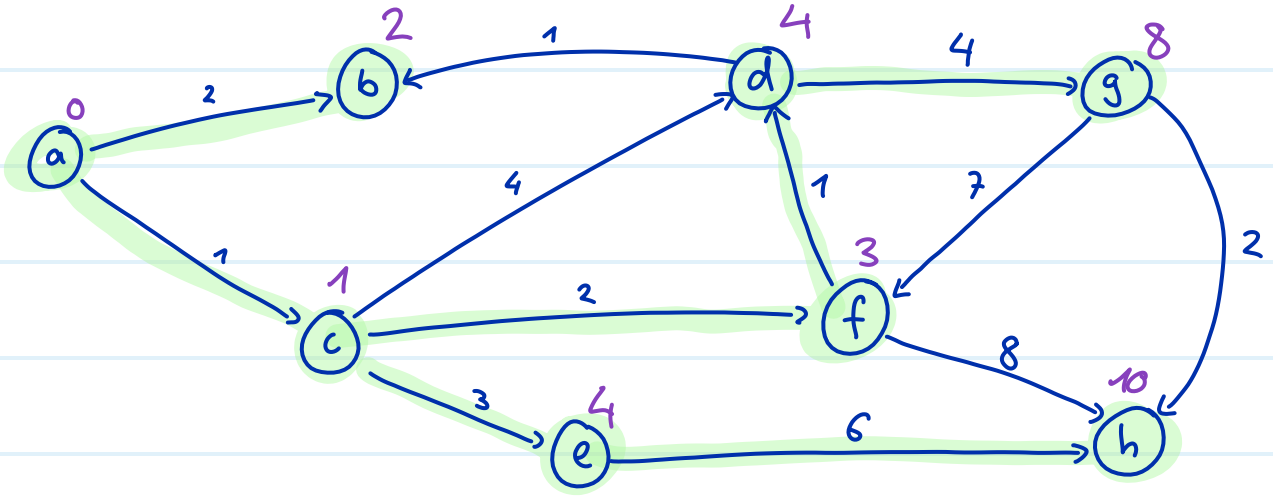
Schritt: 1) wir wählen von allen Kanten $(u,v) \in (S \times V \setminus S) \cap E$, das heisst $u \in S$ und $v \notin S$, diejenige aus, sodass $d[u] + c(u,v)$ minimal ist.

2) aktualisiere $d[v] = d[u] + c(u,v)$

3) füge v zu S hinzu

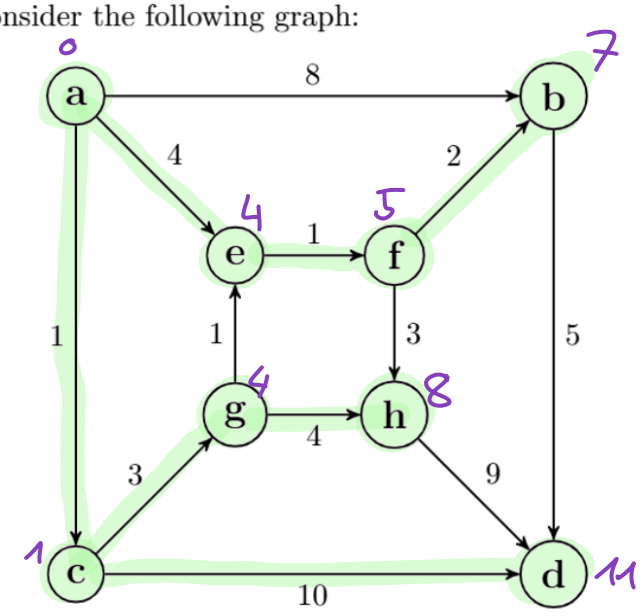
shortest path tree

$d[v]$

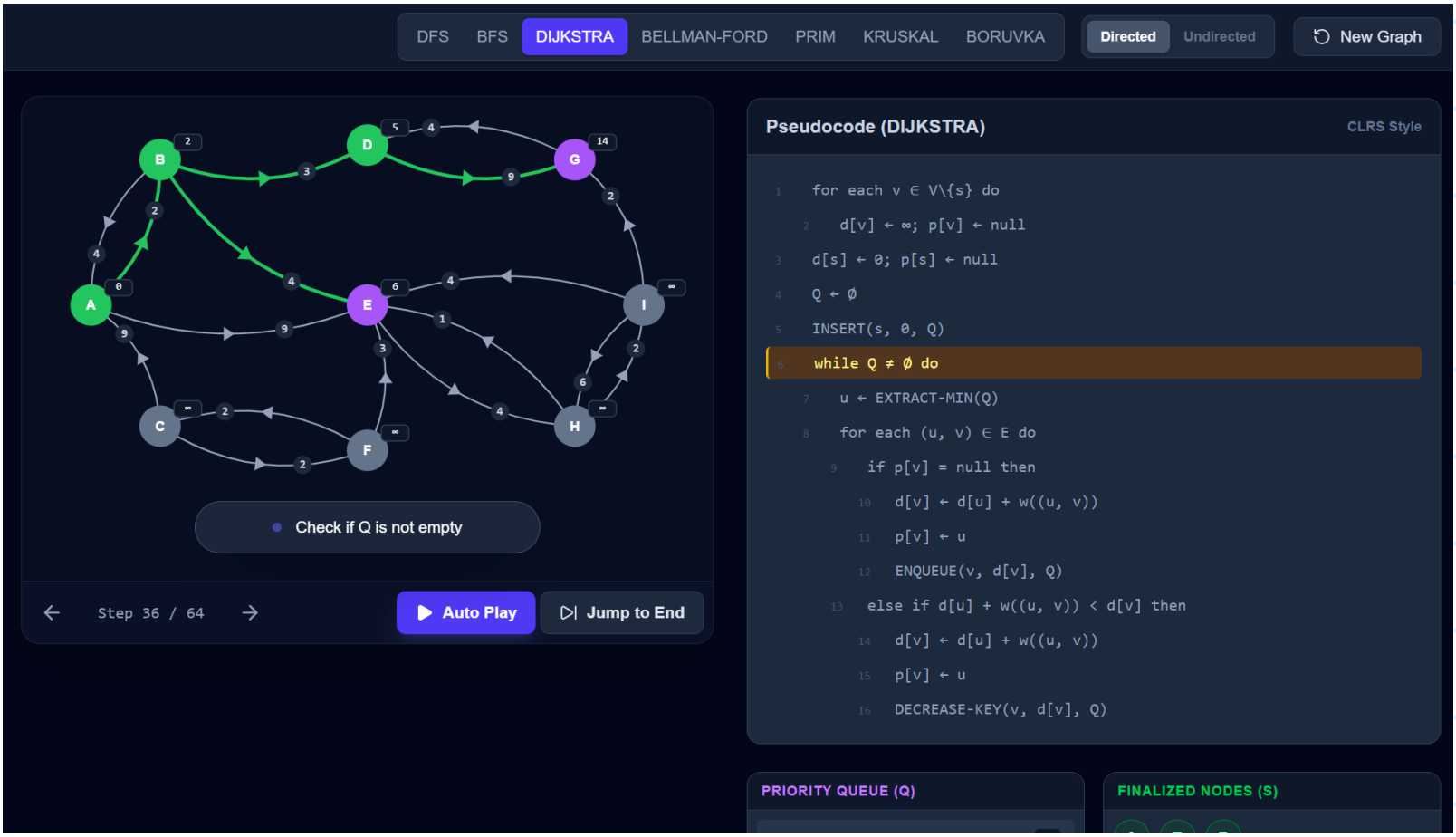


/ 2 P

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



i) Find a shortest-path tree rooted at vertex a (e.g., the set of edges selected by Dijkstra's algorithm starting from vertex a). You can either highlight its edges in the picture above, or draw the shortest-path tree below. (If there is more than one shortest-path tree, it suffices to just find one of them.)

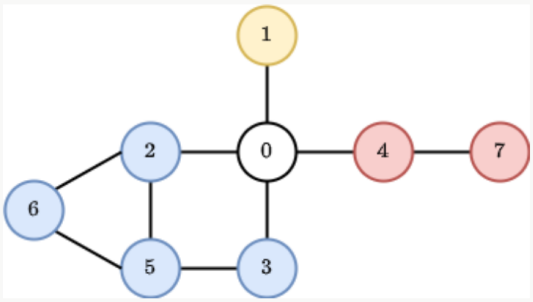

DFS
BFS
DIJKSTRA
BELLMAN-FORD
PRIM
KRUSKAL
BORUVKA
Directed
Undirected
New Graph
Pseudocode (DIJKSTRA)
CLRS Style
for each v ∈ V\{s} do
d[v] ← ∞; p[v] ← null
d[s] ← 0; p[s] ← null
Q ← ∅
INSERT(s, 0, Q)
while Q ≠ ∅ do
u ← EXTRACT-MIN(Q)
for each (u, v) ∈ E do
if p[v] = null then
d[v] ← d[u] + w((u, v))
p[v] ← u
ENQUEUE(v, d[v], Q)
else if d[u] + w((u, v)) < d[v] then
d[v] ← d[u] + w((u, v))
p[v] ← u
DECREASE-KEY(v, d[v], Q)
Check if Q is not empty
Step 36 / 64
Auto Play
Jump to End
PRIORITY QUEUE (Q)
FINALIZED NODES (S)

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.

You have to implement the four methods described above. In addition, we provide an **intialize()** method which we guarantee to run before any of the queries. Feel free to use this method, for example, to initialize information that you want available in all of the queries (of course, the time consumed by **intialize()** counts toward the time limit of the problem).

For an easier implementation of the queries, **recall and make use of the fact** that the graph is connected.

Grading (24 points):

1. **hasCycle()** (4 points): This query will be run at most once. An $O(n+m)$-time implementation gets 4 points.
2. **hasCycleWithoutNodeZero()** (4 points): This query will be run at most once. An $O(n+m)$-time implementation gets 4 points.
3. **isSameSet(x, y)** (8 points): If q is the number of queries of this type, an $O(n+m+q)$-time implementation gets 8 points and an $O(q(n+m))$-time implementation gets 4 points.
4. **getShortestPath(x, y)** (8 points): If q is the number of queries of this type, an $O(n+m+q)$-time implementation gets 8 points and an $O(q(n+m))$-time implementation gets 4 points.

```
public void initialize() {

  Queue<Integer> Q = new LinkedList<Integer>();
  visited = new boolean[n];
  dist = new int[n];
  Q.add(0);
  visited[0] = true;
  dist[0] = 0;
  while(!Q.isEmpty()) {
    int x = Q.poll();
    for (int i = 0; i<degree[x]; i++) {
      int y = edges[x][i];
      if (!visited[y]) {
        visited[y] = true;
        dist[y] = dist[x] + 1;
        Q.add(y);
      }
    }
  }

  set = new int[n];
  visited = new boolean[n];
  int setCount = 1;

  for (int i = 0; i<degree[0]; i++) {
    int x = edges[0][i]; // i-th neighbour of 0
    if (!visited[x]) {
      set[x] = setCount++;
      DFS(x); // finde alle erreichbaren Knoten und weise ihnen
              // dasselbe set zu
    }
  }

}

public int hasCycle() {
  return m > n-1 ? 1 : 0;
}

public int hasCycleWithoutNodeZero() {
  int zhkCount = 0;
  visited = new boolean[n];
  for (int i = 1; i<n; i++) { // starting at i = 1 to ignore 0
    if (!visited[i]) {
      zhkCount++;
      DFS(i);
    }
  }

  return (m - degree[0] > n - 1 - zhkCount) ? 1 : 0;
}

public int isSameSet(int x, int y) {
  return set[x] == set[y] ? 1 : 0;
}
```

```java
public int getShortestPath(int x, int y) {
  return dist[x] + dist[y];
}

private void DFS(int x) {
  visited[x] = true;
  for (int i = 0; i < degree[x]; i++) {
    int y = edges[x][i];
    if (y == 0) continue;   // ignore 0
    if (!visited[y]) {
      set[y] = set[x];      // y is in same set as its parent x
      DFS(y);
    }
  }
}
```