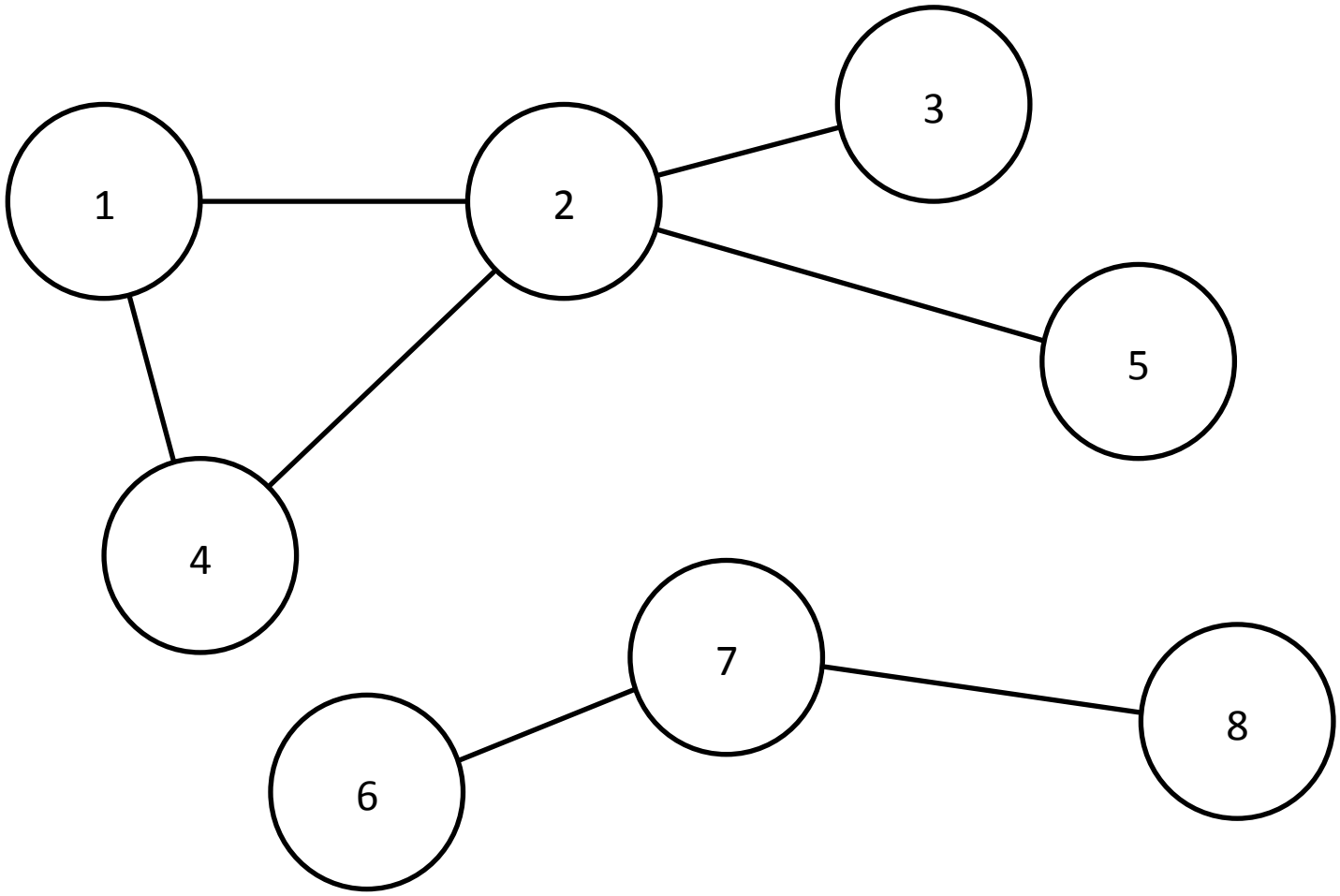
Graphen

BFS, DFS und Toposort

kurze Definition

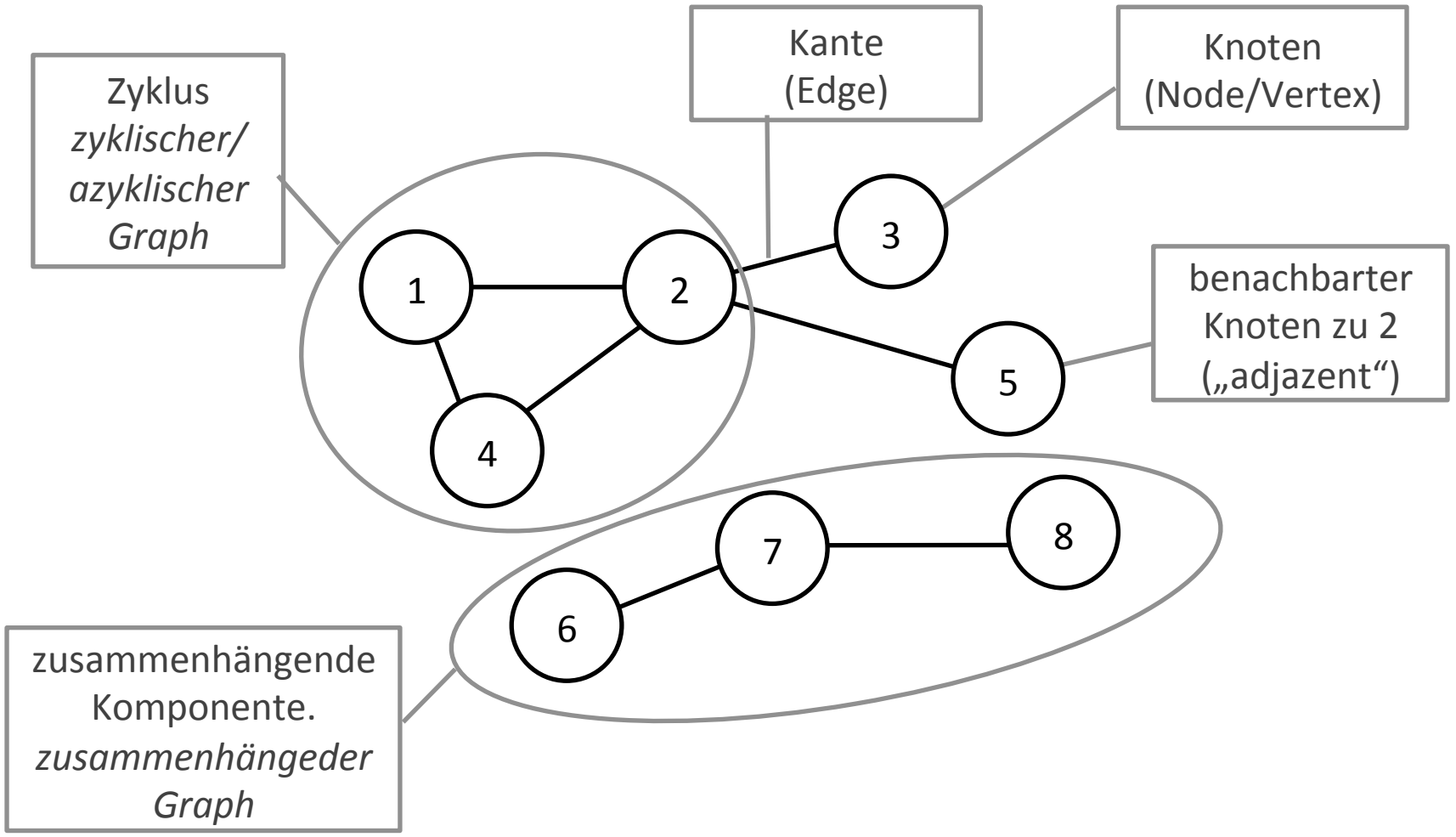


Für was braucht man das?

- Karten
- Computer-Netzwerke
- Verlinkte Webpages
- Kürzester Weg von A nach B?
- Kritische Verbindung suchen

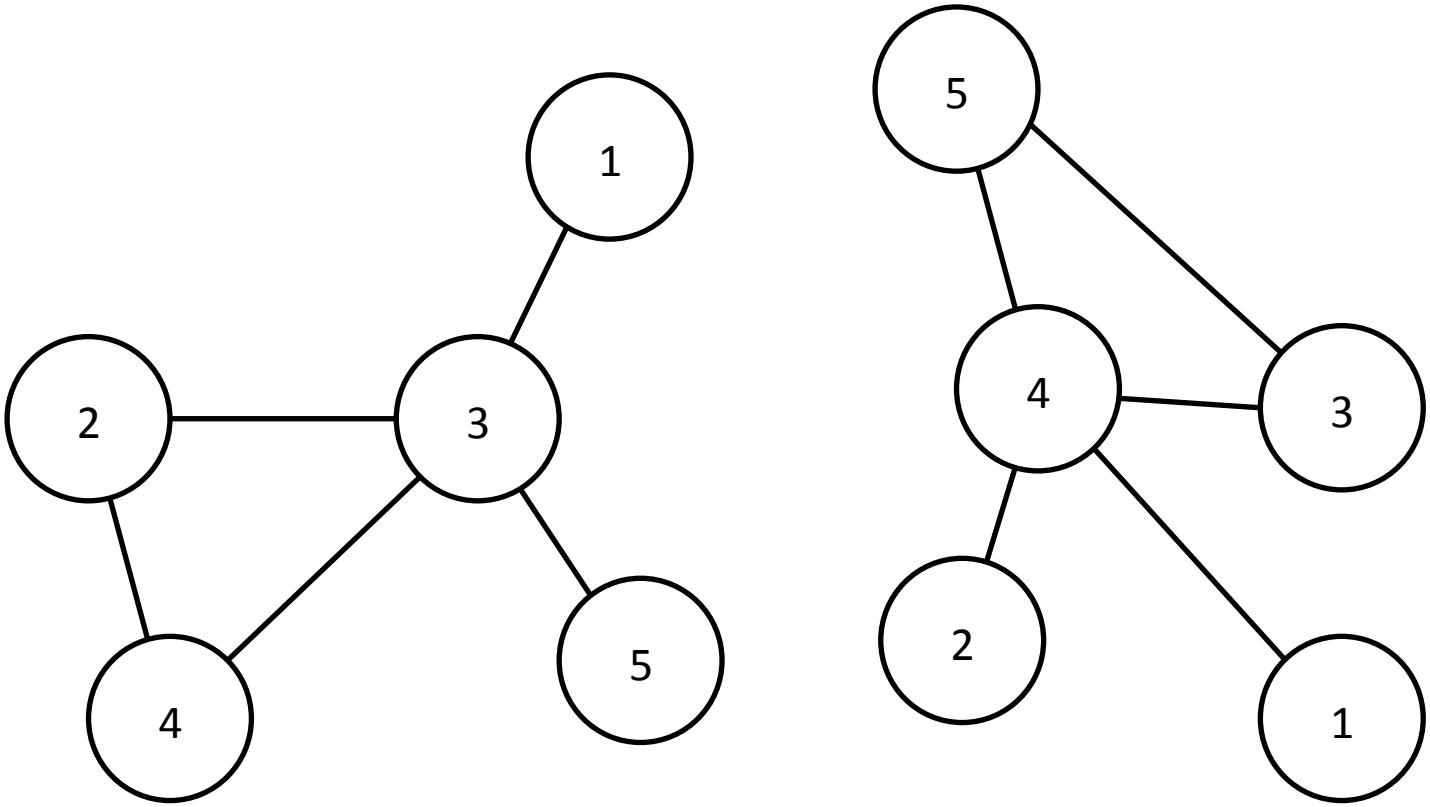
NOMENKLATUR UND TYPEN

Nomenklatur



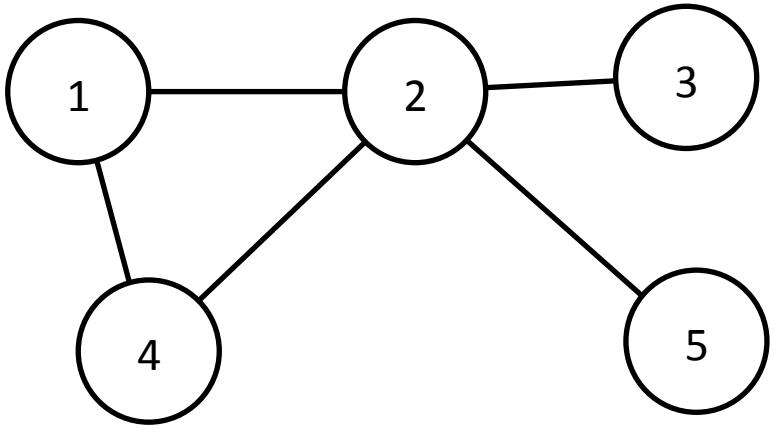
Isomorphie

= Gleiche Struktur

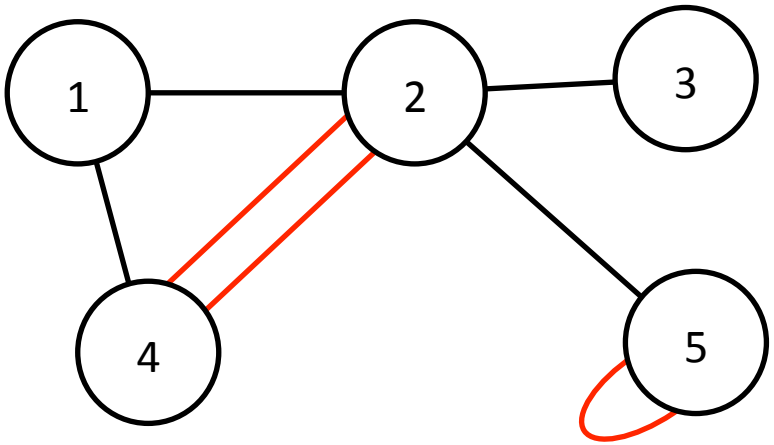


Typen

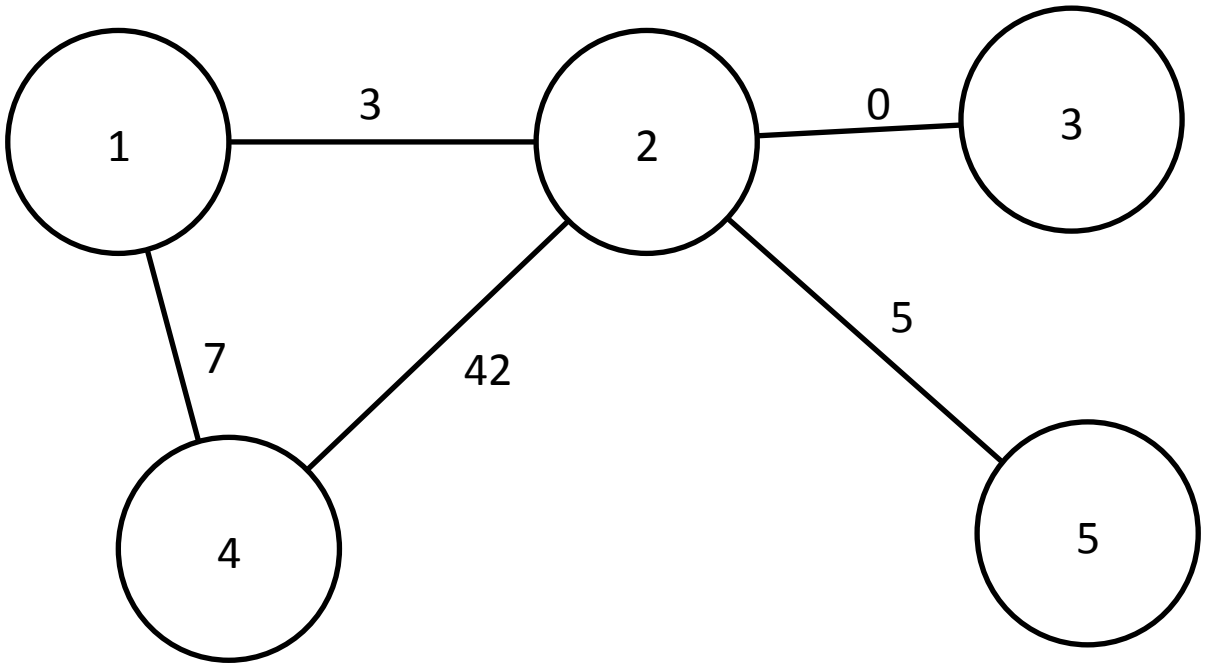
- Einfacher Graph:



- Multigraph:

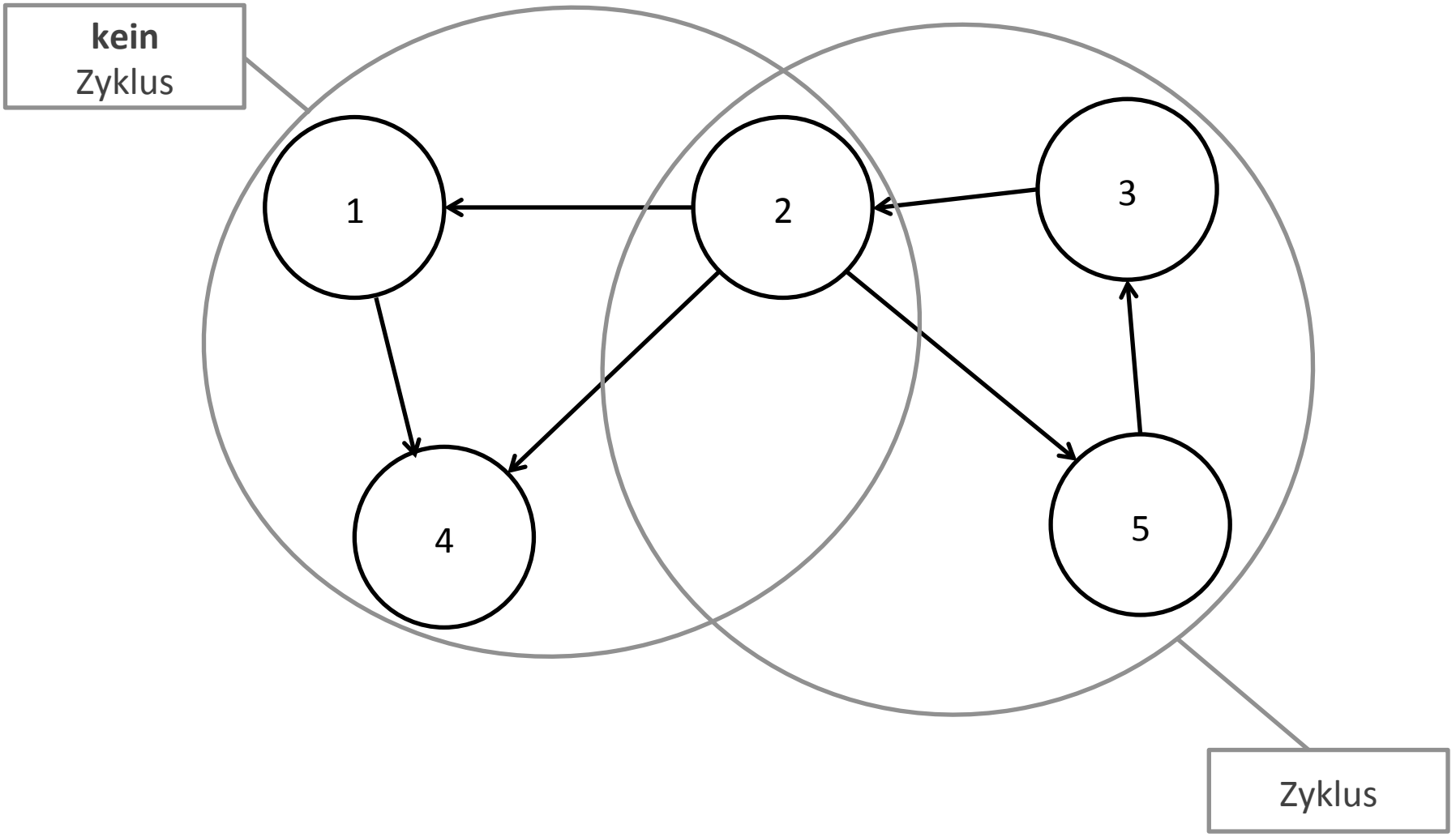


Gewichteter Graph

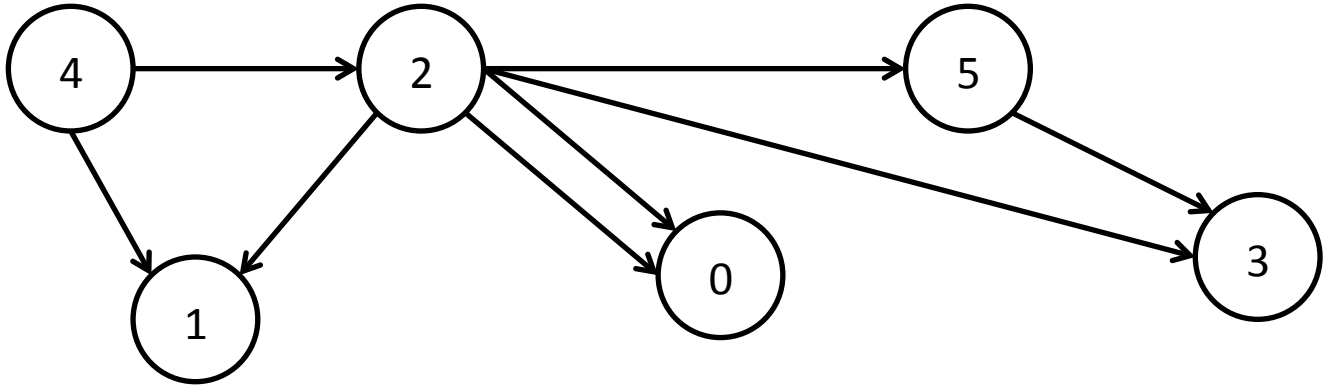


- z.B. Längen von Strassen

Gerichteter Graph (Digraph)



Repetition: Multiple Choice



Dieser Graph hat...

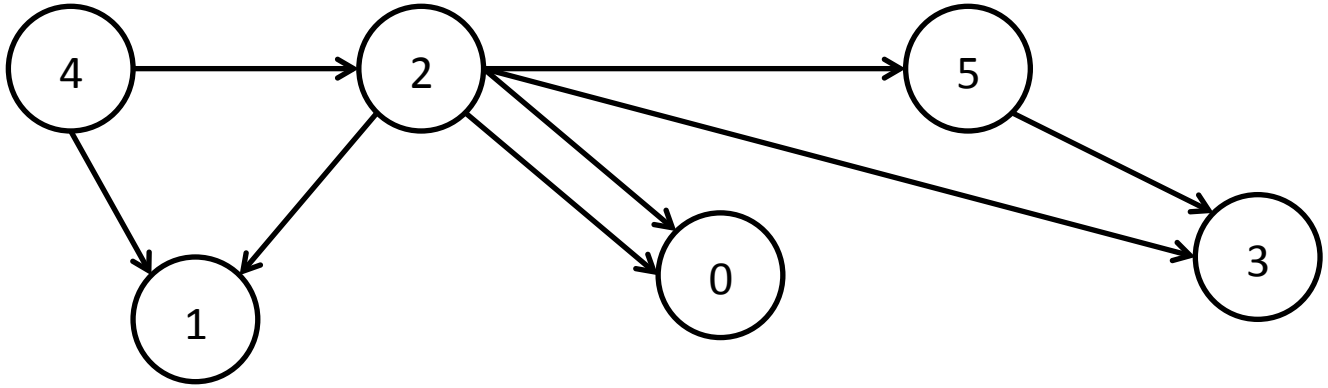
a) 6 Knoten und 8 Kanten

b) 8 Knoten und 6 Kanten

c) 6 Knöpfe und 8 Kanten

d) 8 Knöpfe und 6 Kanten

Repetition: Multiple Choice



Das ist ein...

a) zyklischer, gewichteter, zusammenhängender Multigraph

b) zyklischer, gerichteter, zusammenhängender einfacher Graph

c) azyklischer, gerichteter, zusammenhängender Multigraph

d) azyklischer, gerichteter, gewichteter, nicht zusammenhängender Multigraph

e) azyklischer, gewichteter, nicht zusammenhängender einfacher Graph

Wie viele Kanten hat ein einfacher Graph maximal?

V = #Knoten (Vertices)

a) $O(V)$

b) $O(V \log V)$

c) $O(V\sqrt{V})$

d) $O(V^2)$

e) $O(V^2 \log V)$

f) $O(V^3)$

$V(V\text{-}1)/2$

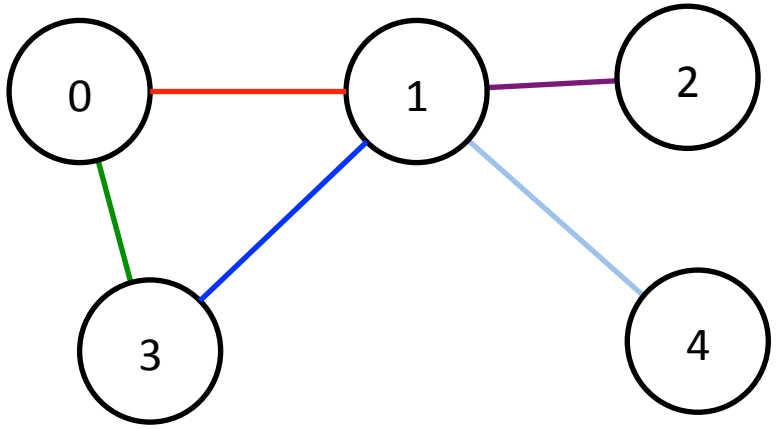
Zyklen

- Ein ungerichteter Graph ist ab V Kanten immer zyklisch
- Ein gerichteter Graph ist ab $V\,(V\text{-}1)/2 + 1$ Kanten immer zyklisch

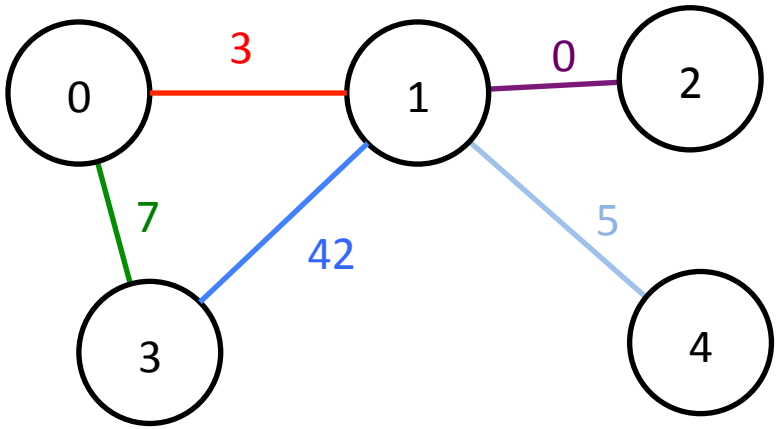
IMPLEMENTIERUNGEN

Adjazenzmatrix

	0	**1**	**2**	**3**	**4**
0	0	1	0	1	0
1	1	0	1	1	1
2	0	1	0	0	0
3	1	1	0	0	0
4	0	1	0	0	0

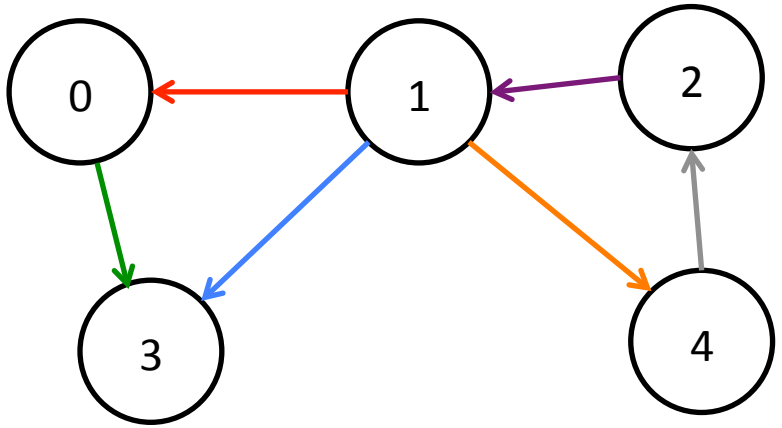


	0	**1**	**2**	**3**	**4**
0	-1	3	-1	7	-1
1	3	-1	0	42	5
2	-1	0	-1	-1	-1
3	7	42	-1	-1	-1
4	-1	5	-1	-1	-1



Adjazenzmatrix

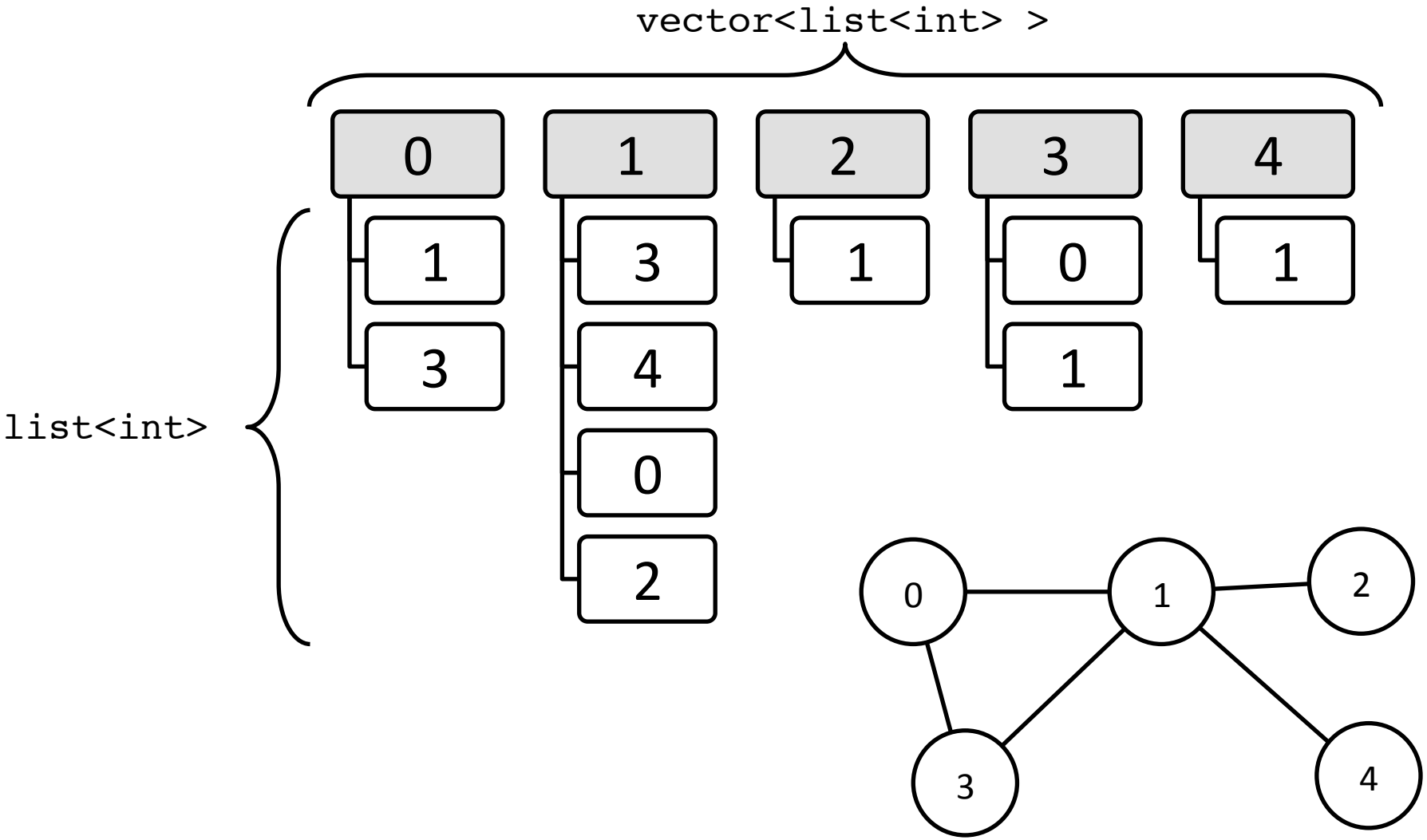
nach \ von	**0**	**1**	**2**	**3**	**4**
0	0	1	0	1	0
1	0	0	1	0	0
2	0	0	0	0	1
3	0	1	0	0	0
4	0	1	0	0	0



C++ Code

```
vector<vector<int> > matrix;

void initGraph(int n){
     matrix = vector<vector<int> >(n, vector<int>(n, 0));
}
void addEdge(int nodeA, int nodeB){
     matrix[nodeA][nodeB] = 1;
     matrix[nodeB][nodeA] = 1;
}
bool hasEdge(int nodeA, int nodeB){
     return matrix[nodeA][nodeB] == 1;
}
void removeEdge(int nodeA, int nodeB){
     matrix[nodeA][nodeB] = 0;
     matrix[nodeB][nodeA] = 0;
}
```

Adjazenzliste



C++ Code

```
vector<list<int> > nodes;

void initGraph(int n){
  nodes = vector<list<int> >(n, list<int>());
}
void addEdge(int nodeA, int nodeB){
  nodes[nodeA].push_back(nodeB);
  nodes[nodeB].push_back(nodeA);
}
bool hasEdge(int nodeA, int nodeB){
  return count(nodes[nodeA].begin(),
    nodes[nodeA].end(), nodeB) != 0;
}
void removeEdge(int nodeA, int nodeB) {
  remove(nodes[nodeA].begin(), nodes[nodeA].end(), nodeB);
  remove(nodes[nodeB].begin(), nodes[nodeB].end(), nodeA);
}
```

Speicherplatz Adjazenzliste

- $\mathrm{O}(V + E)$
  - V = Knoten, E = Kanten

Vergleich Adjazenzliste/matrix

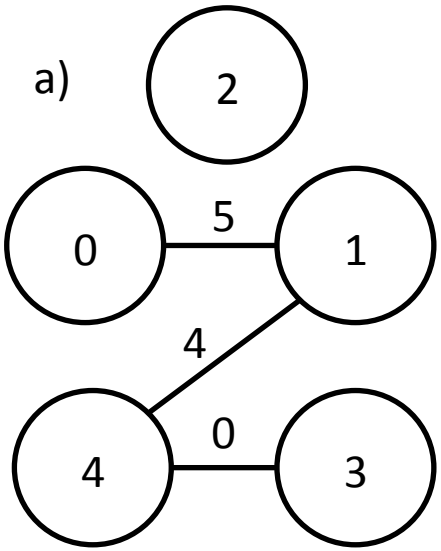
	Adjazenzliste	**Adjazenzmatrix**
Speicherplatz	V+E	V^2
Kante einfügen	1	1
Kante löschen	E	1
Existiert Kante?	E	1
Finde alle Kanten eines Knoten	E	V
Traversieren	V+E	V^2

Multiple Choice

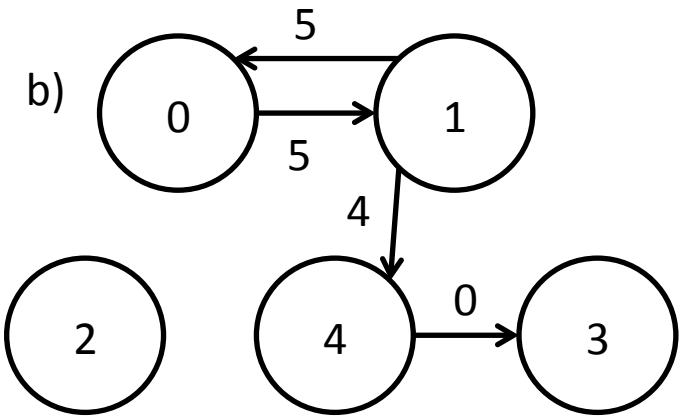
- Wie sieht der Graph dieses Programms aus?

```
int main(){
  vector<list<pair<int, int> > > nodes(5);
  nodes[0].push_back(make_pair(1, 5));
  nodes[1].push_back(make_pair(0, 5));
  nodes[4].push_back(make_pair(2, 1));
  nodes[3].push_back(make_pair(4, 0));
  nodes[4].push_back(make_pair(1, 4));
  nodes[4].remove(make_pair(2, 1));
}
```

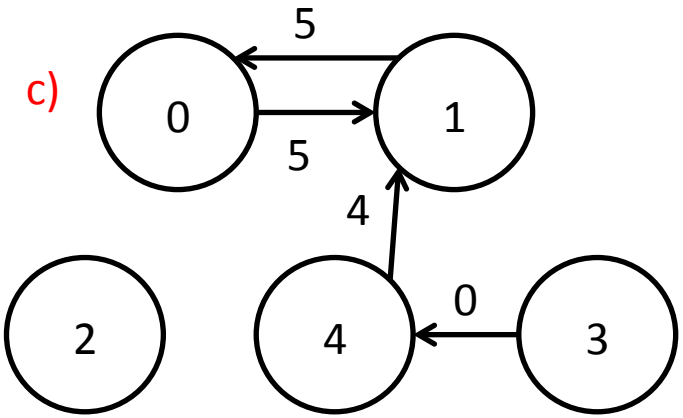
a)



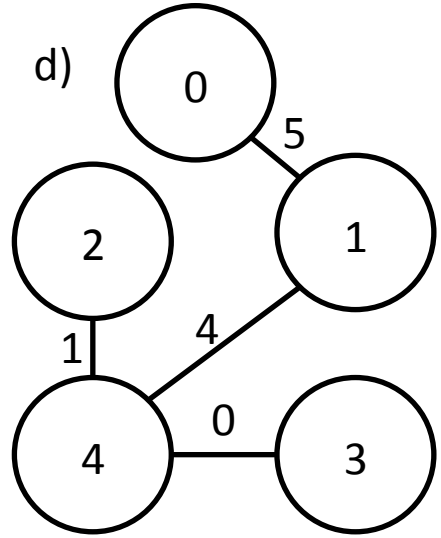
b)



c)

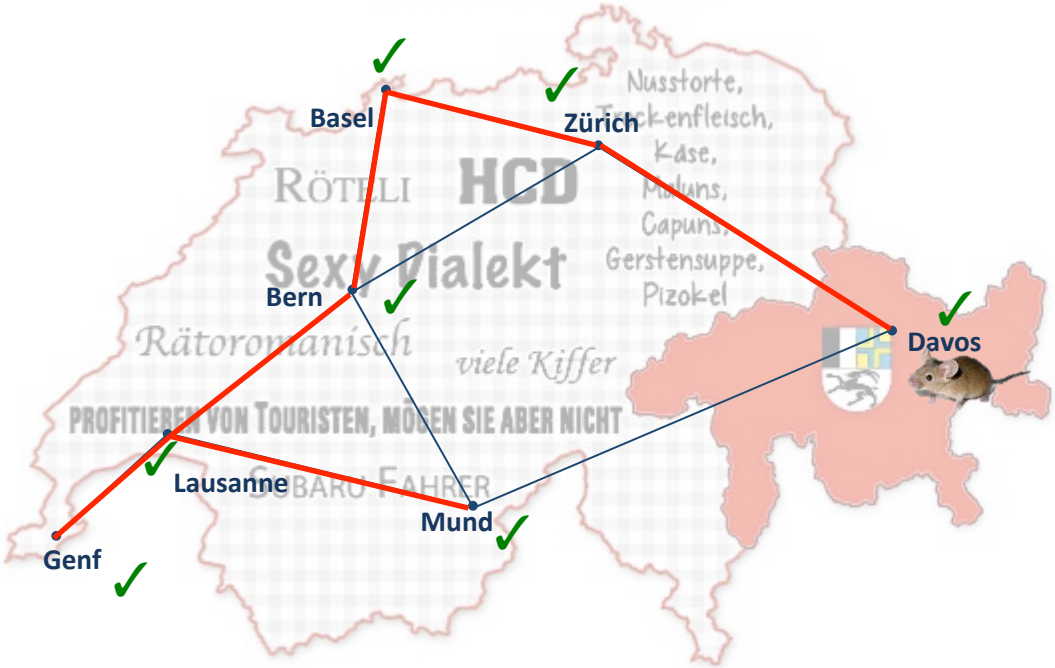


d)

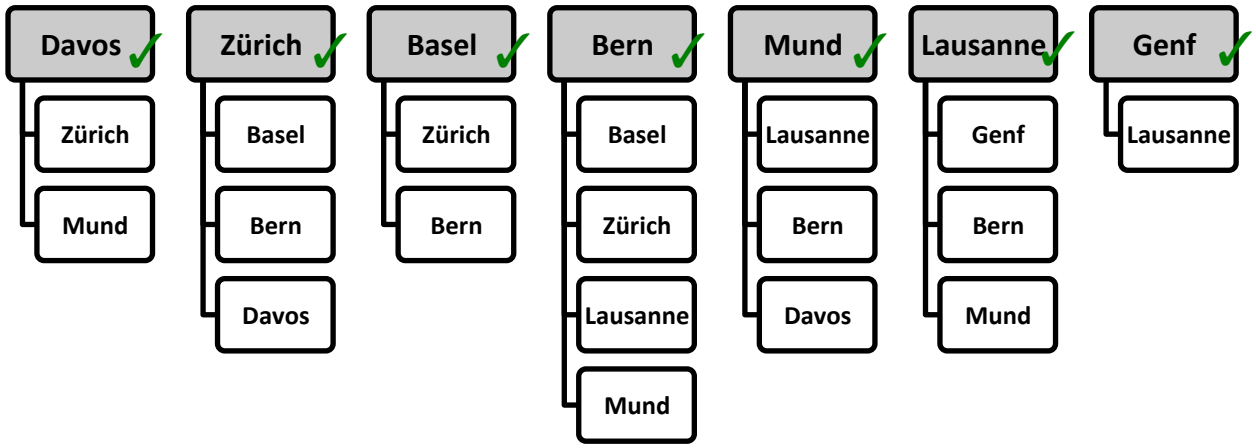


DEPTH-FIRST-SEARCH

DFS




```
vector<list<int> > nodes;
vector<bool> visited;

void dfs(int node){
  visited[node] = true;
  for(int adjacent : nodes[node]){
    if(!visited[adjacent])
      dfs(adjacent);
    }
  }
int main(){
  visited = vector<bool>(n, false);
  //Fuelle den Graphaen
  dfs(0);
}
```



DFS: Laufzeit mit Adjazenzliste

a) $O(V)$

b) $O(V \log V)$

c) $O(V + E)$

d) $O(V^2)$

e) $O(VE)$

f) $O(V^2 + E)$

>> Linear in der Speichergrösse

DFS mit Adjazenzmatrix

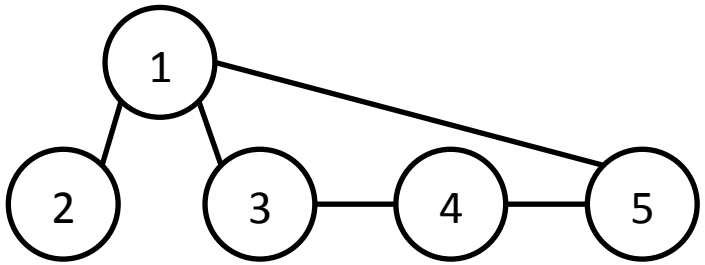
```
vector<vector<int> > matrix;
vector<bool> visited;

void dfs(int node) {
  visited[node] = true;
  for(int i = 0; i < matrix.length(); i++){
    if(matrix[node][i] == 1 && !visited[i])
      dfs(i);
  }
}

int main(){
  visited = vector<bool>(n, false);
  //Fuelle den Graphaen
  dfs(0);
}
```

DFS: Implementierung mit Stack

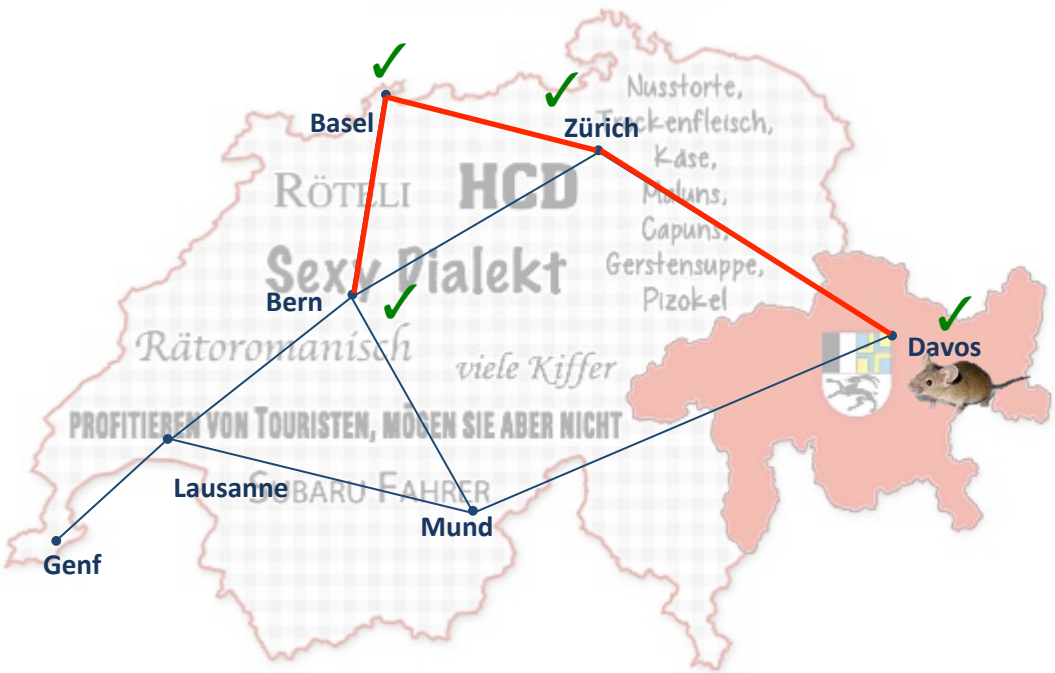
```
vector<list<int> > nodes;
vector<bool> visited;
int main(){
  visited = vector<bool>(n, false);
  //Fuelle den Graphaen
  stack<int> st;
  st.push(0);
  visited[0] = true;
  while(!st.empty()){
    int node = st.top();
    st.pop();
    if(!visited[node]){
      visited[node] = true;
        for(int adjacent : nodes[node])
          if(!visited[adjacent])
            st.push(adjacent);
    }
  }
}
```

DFS: Anwendungen

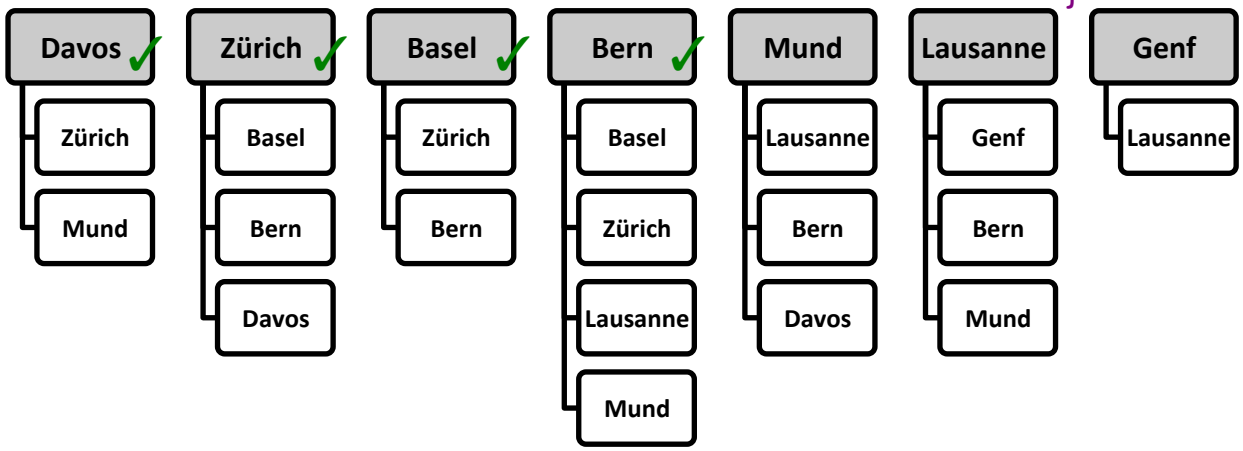
- Simple path: Existiert ein Pfad zwischen zwei Knoten?
- Simple connectivity: Ist der Graph zusammenhängend?
  - `visited`-Vektor am Schluss prüfen
- Cycle-Detection



DFS: Cycle-Detection




```
vector<list<int> > nodes;
vector<bool> visited;

void dfs(int node, int parent){
  visited[node] = true;
  for(int adjacent : nodes[node]){
    if(!visited[adjacent])
      dfs(adjacent, node);
    else
      if(adjacent != parent)
        //Cycle detected!
  }
}

int main()
{
  visited = vector<bool>(n, false);
  //Fuelle den Graphaen
  dfs(0, 0);
}
```

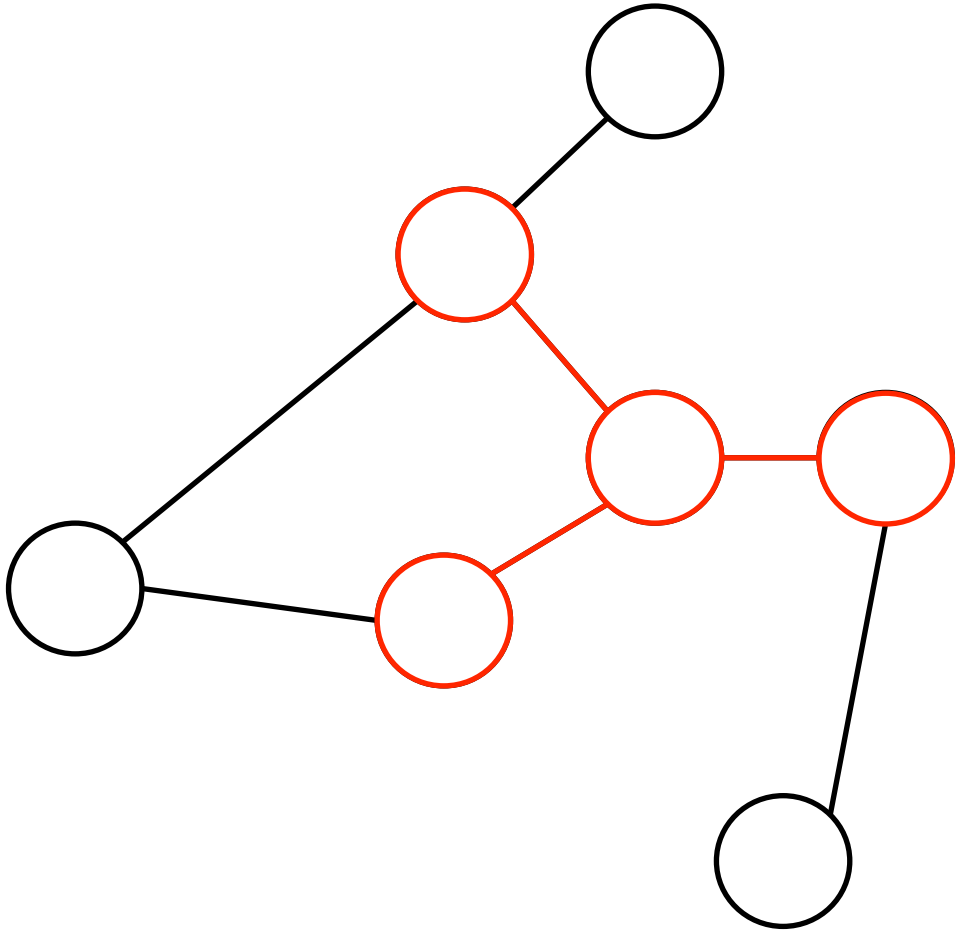


DFS: Anwendungen

- Simple path: Existiert ein Pfad zwischen zwei Knoten?
- Simple connectivity: Ist der Graph zusammenhängend?
  - `visited`-Vektor am Schluss prüfen
- Cycle-Detection
  - Braucht O(V), denn Graph mit $E \geq V$ hat sicher einen Zyklus

BREADTH-FIRST-SEARCHH

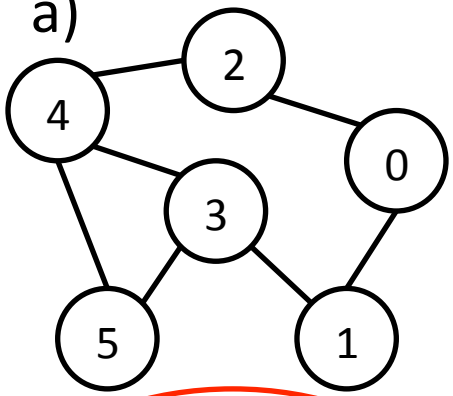
BFS



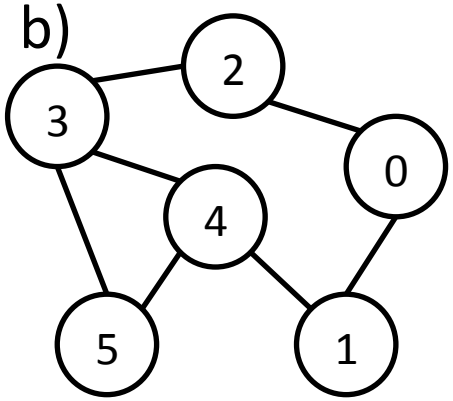
Multiple Choice

- Welches ist keine BFS-Nummerierung?

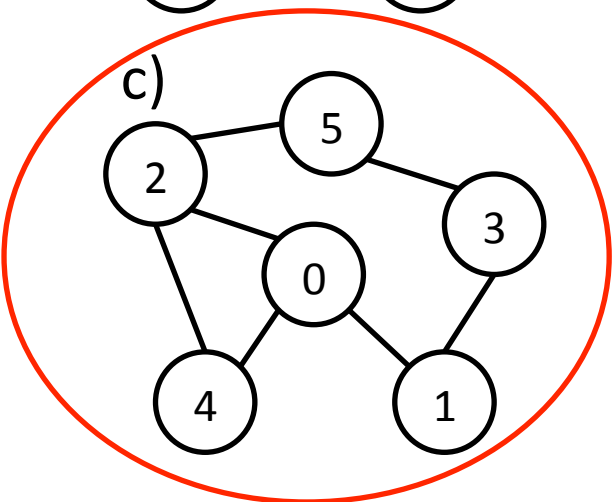
a)



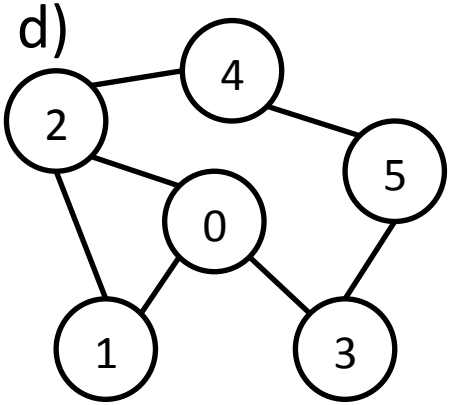
b)



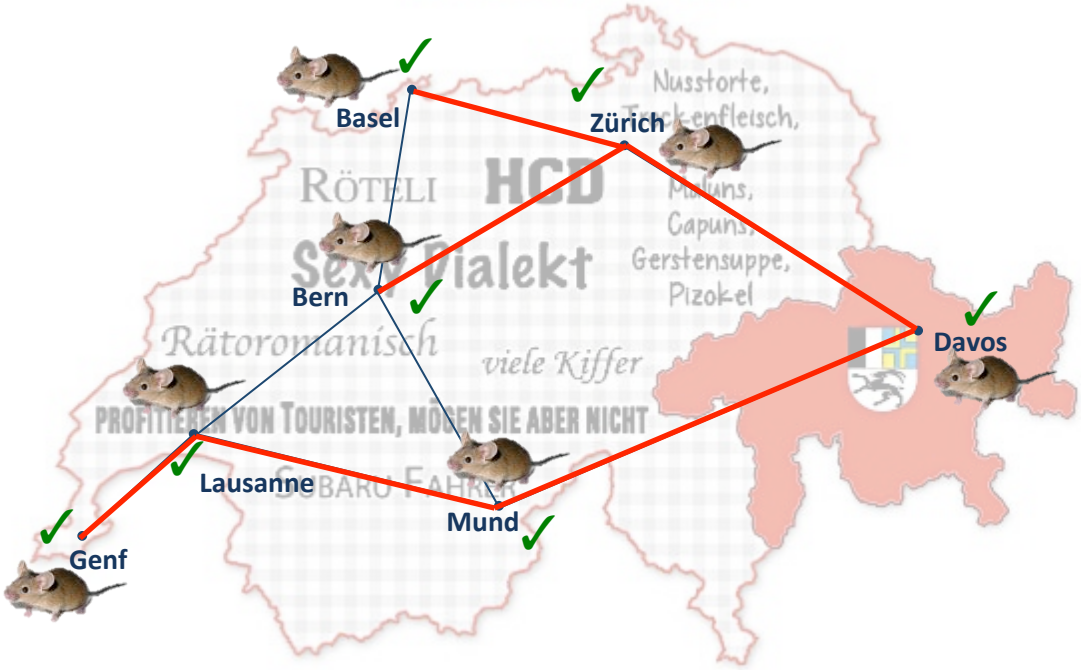
c)



d)



BFS



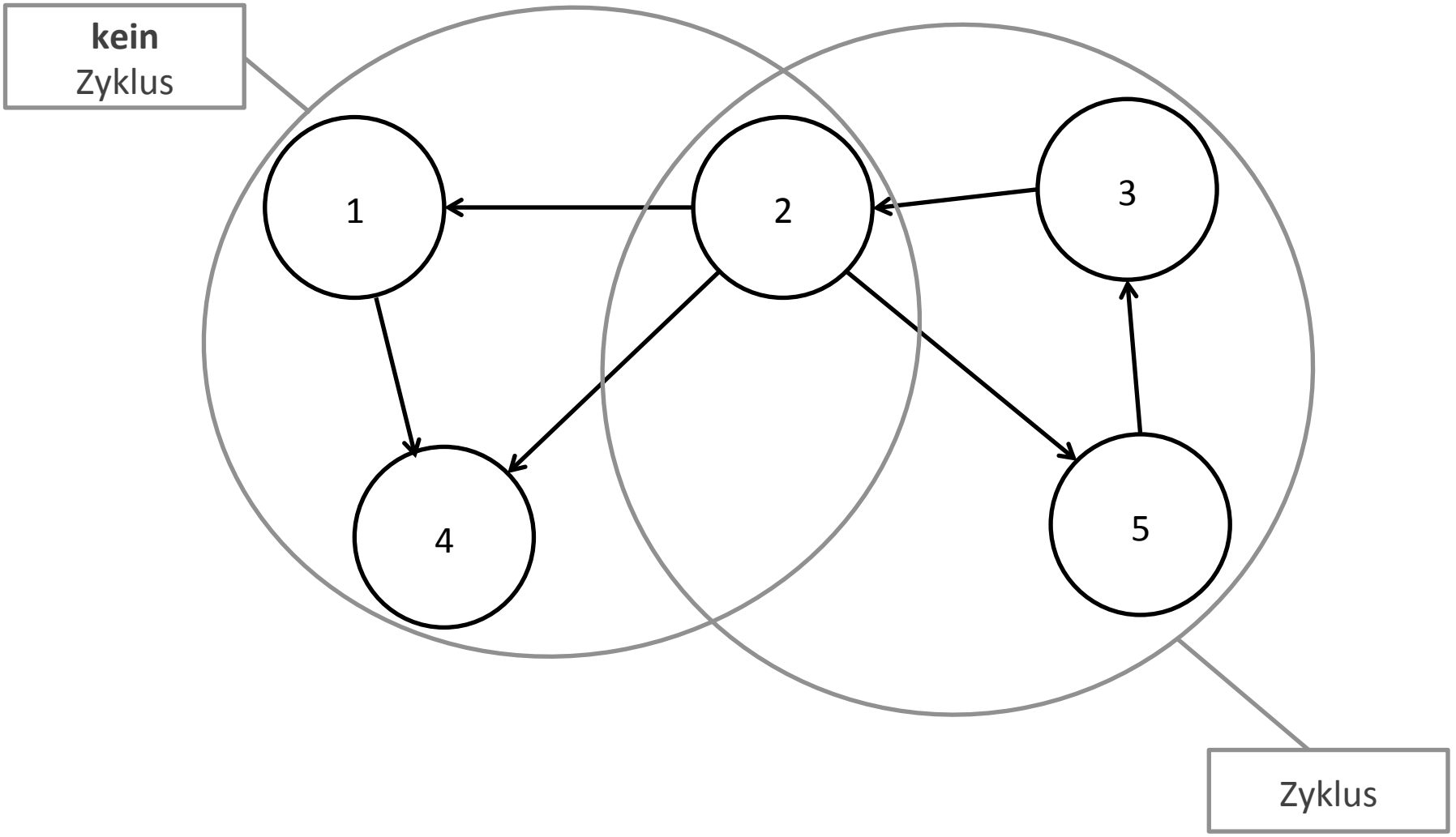

```
vector<list<int> > nodes;
vector<bool> visited;

int main(){
  visited = vector<bool>(n, false);
  //Fuelle den Graphaen
  queue<int> q;

  q.push(0);
  visited[0] = true;
  while(!q.empty()){
    int node = q.top();
    q.pop();

    for(int adjacent : nodes[node]){
      if(!visited[adjacent]){
        visited[adjacent] = true;
        q.push(adjacent);
      }
    }
  }
}
```

Queue:

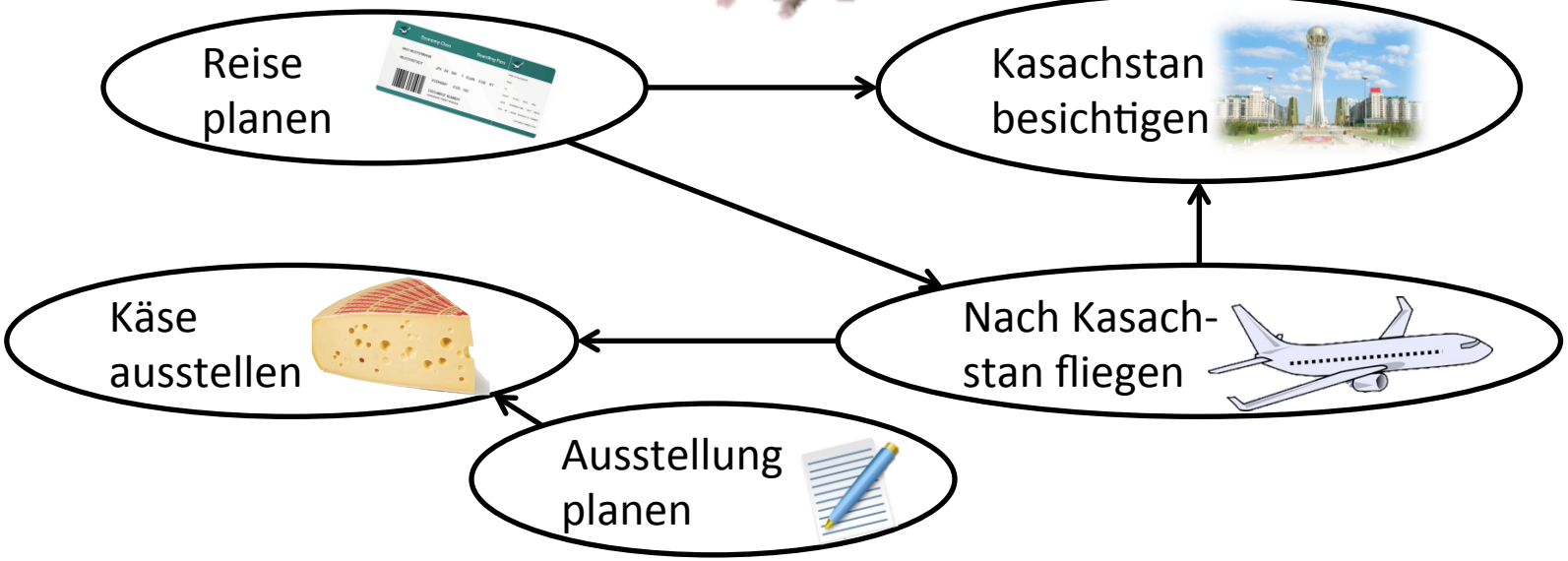
Genf	Lausanne	Bern	Basel	Mund	Zürich	Davos

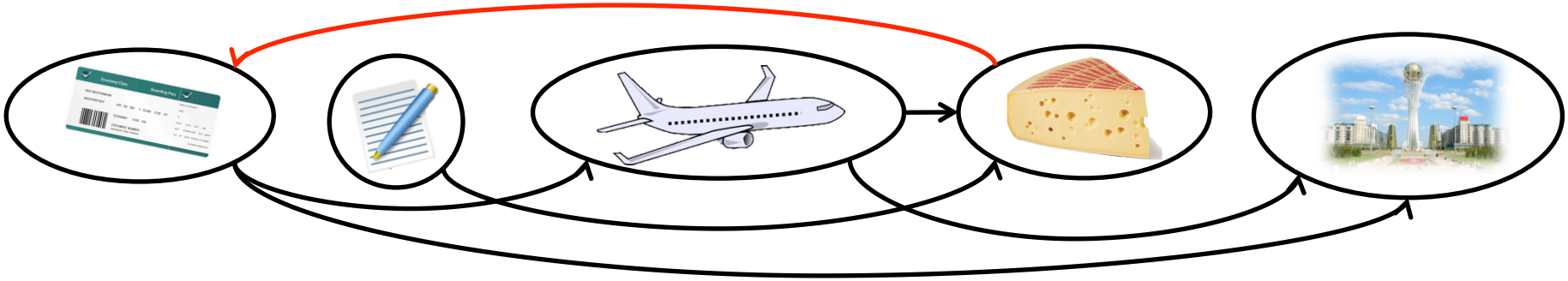
DIGRAPHEN & TOPOSORT

Gerichteter Graph (Digraph)

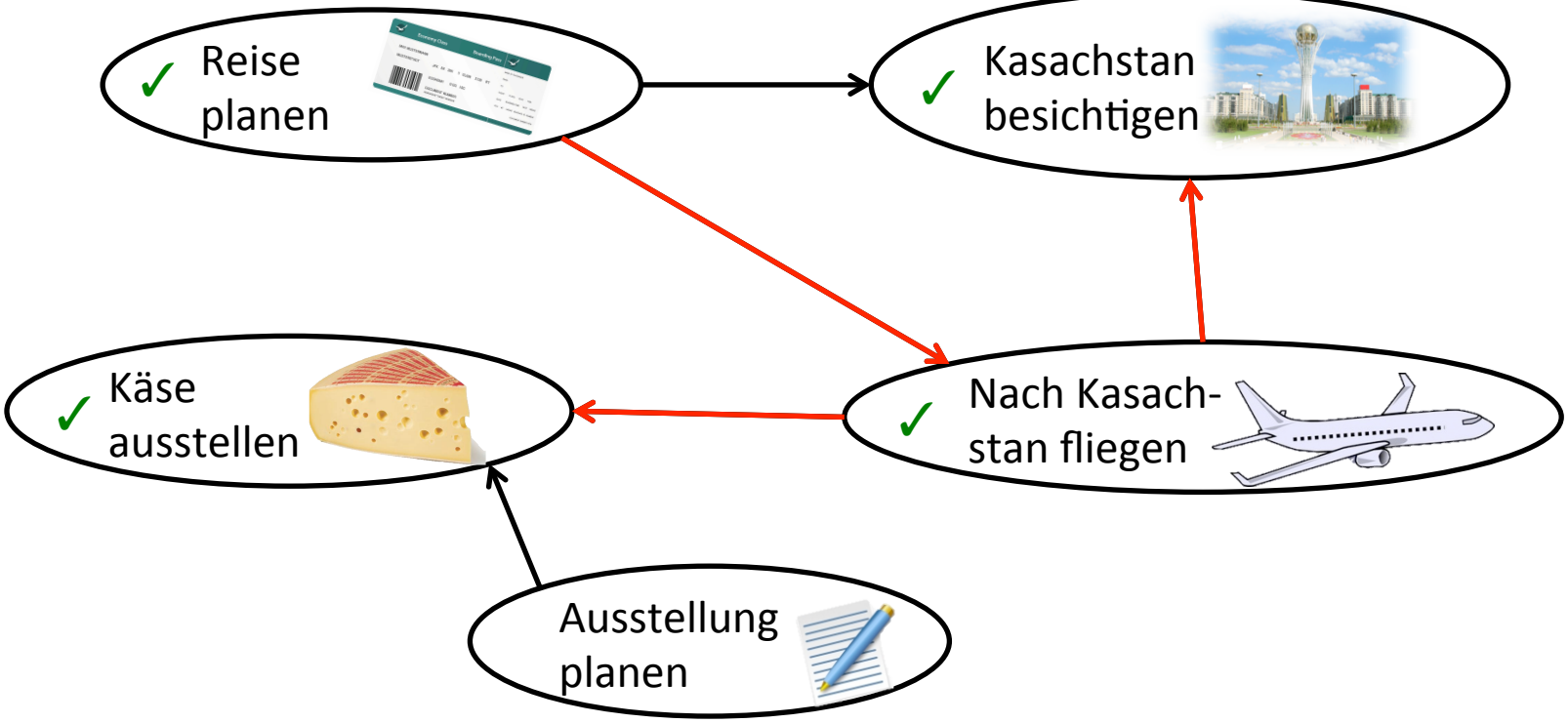


Aufgaben-Planung (Schedueling)

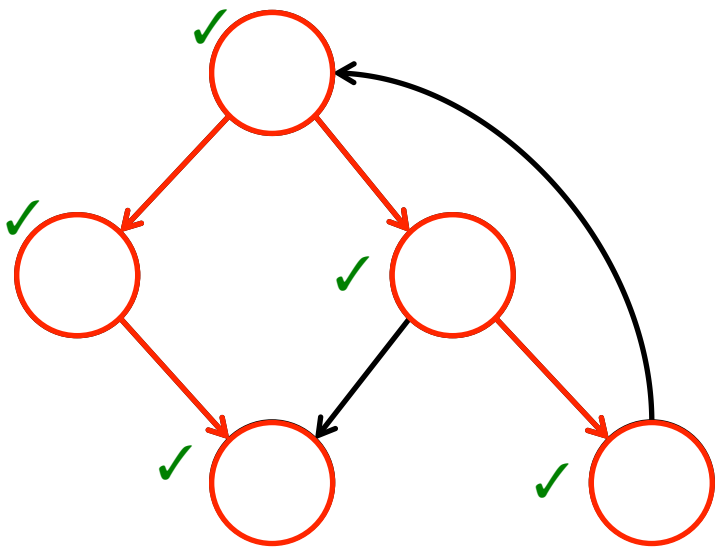




Cycle-Detection in Digraph

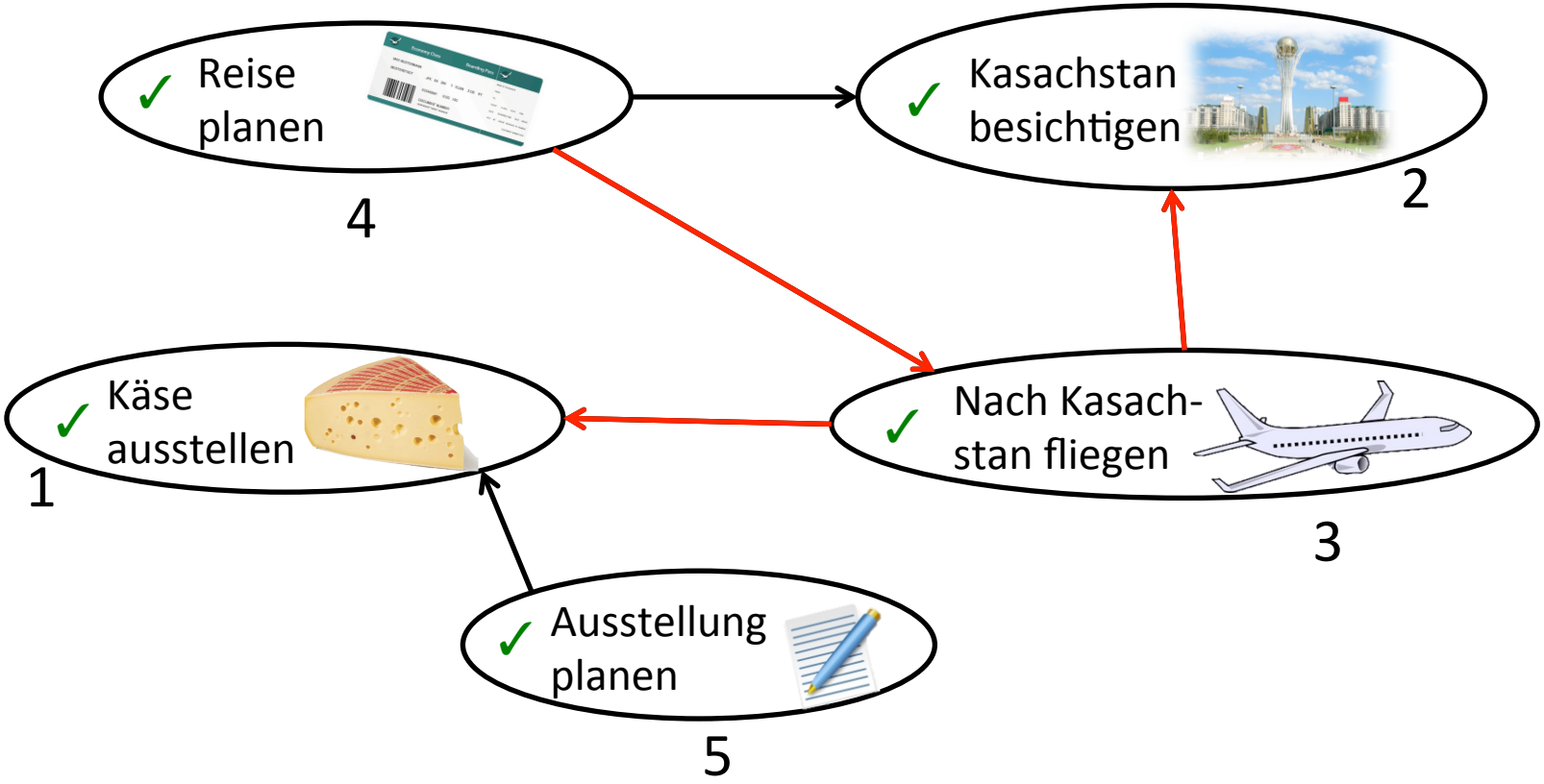


Cycle-Detection Digraph



```
vector<list<int> > nodes;
vector<bool> visited;
vector<bool> st;

void dfs(int node){
  visited[node] = true;
  st[node] = true;
  for(int adjacent : nodes[node]){
    if(!visited[adjacent])
      dfs(adjacent);
    if(st[adjacent])
      //Found Cycle!
  }
  st[node] = false;
}

int main(){
  visited = vector<bool>(n, false);
  st = vector<bool>(n, false);
  //Fuelle den Graphaen
  for(int i = 0; i < n; i++)
    if(!visited[i])
      dfs(i);
}
```

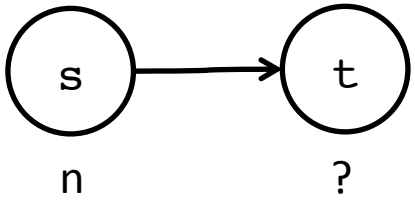
Topologische Sortierung (Toposort)



Umgekehrte Topologische Sortierung

```
vector<list<int> > nodes;
vector<bool> visited;
vector<int> sorted;
int counter = 0;

void dfs(int node){
  visited[node] = true;
  for(int adjacent :: nodes[node]){
    if(!visited[adjacent])
      dfs(adjacent);
  }
  sorted[counter++] = node;
}

int main(){
  visited = vector<bool>(n, false);
  sorted = vector<bool>(n);
  //Fuelle den Graphaen
  for(int i = 0; i < n; i++)
    if(!visited[i])
      dfs(i);
}
```

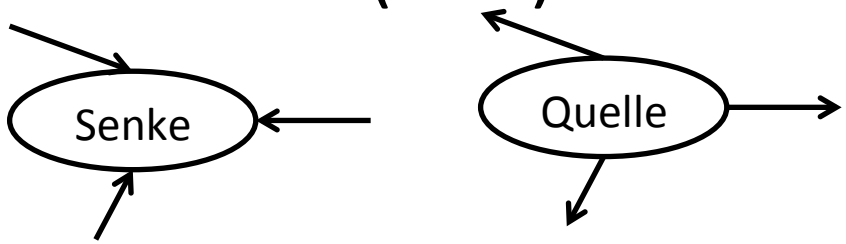
Wieso funktioniert das?



- Wenn wir `s` n zuweisen, haben wir alle Kanten, die von `s` abgehen abgearbeitet, also auch die Kante `s-t`.
- `dfs(t)` muss fertig sein.
- Also hat `t` eine kleinere Nummer als n.

2. Variante

- Jeder Digraph hat eine Quelle (source) und eine Senke (sink)



- Beweis für Quelle: Nehme an, es gäbe keine Quelle. Nehme einen Knoten und folge den Kanten: Spätestens nach V-1 Kanten muss man einen Knoten besuchen, bei dem man schon war => es ist ein zyklischer Graph => Widerspruch!

2. Variante

1. n = 0
2. Nehme eine Quelle und gib ihr die Nummer n. Erhöhe n um 1.
3. Entferne sie und gehe zu Schritt 2.

- Problem bei Brute-Force-Lösung: $O(V^2)$
  (V mal eine Quelle suchen)

2. Variante

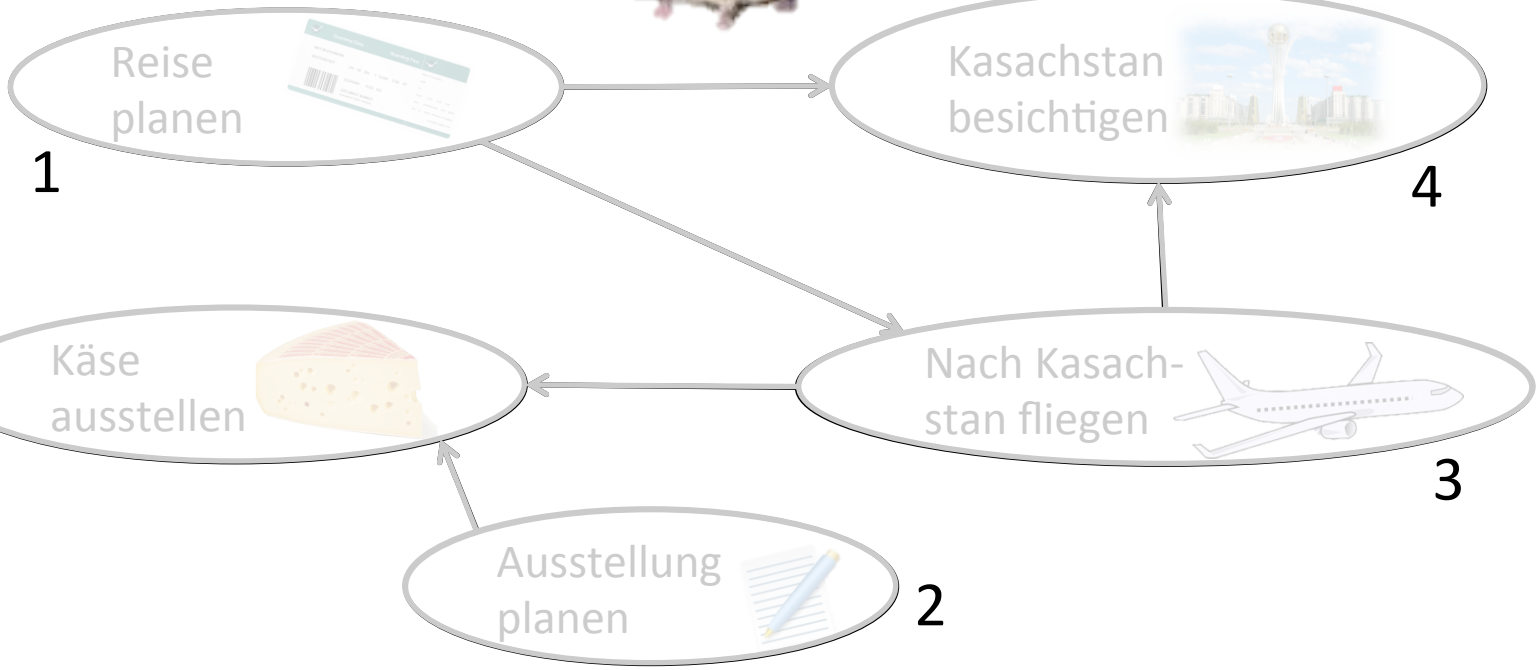
- Lösung: Array mit Ingrad (Indegree) der Knoten.
- Queue mit allen Knoten mit Ingrad 0.

1. Entferne einen Knoten von der Queue und gib ihm die Nummer n++.
2. Verkleinere die Ingrade der Nachbare des Knotens um 1.
3. Wenn der Ingrad eines solchen 0 wurde, füge ihn der Queue hinzu.

2. Variante

```
int main(){
  vector<list<int> > nodes;
  //Fuelle den Graphe
  vector<int> sorted(n);
  vector<int> indegree(n, 0);

  queue<int> q;
  for(list<int> adjacents : nodes)
    for(int node : adjacents)
      indegree[node]++;

  for(int i = 0; i < n; i++)
    if(indegree[i] == 0)
      q.push(i);

  for(int counter = 0; counter < n; counter++){
    int node = q.top();
    q.pop();
    sorted[counter] = node;
    for(int adjacent : nodes[node]){
      indegree[adjacent]--;
      if(indegree[adjacent] == 0)
        q.push(adjacent);
    }
  }
}
```

Topologische Sortierung (Toposort)



Queue: