Data Analytics with Graph Algorithms – A Hands-on Tutorial with Neo4J

March 4th 2019

Lecturer: Dr. Lena Wiese

Agenda

- Short CV of speaker
- Graph Theory Basics
- The Neo4J Database
- Graph Algorithms
  - Centralities
  - Path Finding
  - Community Detection

Agenda

- **Short CV of speaker**
- Graph Theory Basics
- The Neo4J Database
- Graph Algorithms
  - Centralities
  - Path Finding
  - Community Detection

Short CV Dr. Lena Wiese

- **University of Göttingen**
  (Group Leader Knowledge Engineering)
- **University of Salzburg**
  (Guest Lecturer)
- **University of Hildesheim**
  (Visiting Professor for Databases)
- **National Institute of Informatics**, Tokyo
  (funded by DAAD)
- **Robert Bosch India Ltd.**, Bangalore, India
- **TU Dortmund** (Master/PhD)
- **Universidad Complutense**, Madrid

- Web: http://wiese.free.fr/

Short CV Dr. Lena Wiese

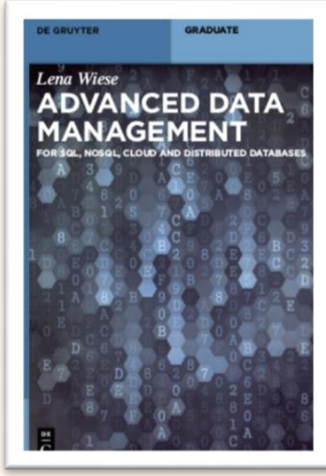
Short CV Dr. Lena Wiese

Teaching and Research:

- **NoSQL Databases**
  (in particular Graph Databases)

- **Intelligent Data Management**
  (in particular, analytics of biomedical data for example, patient similarity analysis, disease prediction, etc.)

- **Intelligent Information Systems**
  (ontologies, recommender systems, etc.)



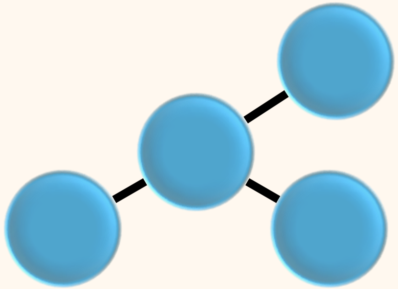
Agenda

- Short CV of speaker
- **Graph Theory Basics**
- The Neo4J Database
- Graph Algorithms
  - Centralities
  - Path Finding
  - Community Detection

Graphs, graphs, graphs

There are lots of graphs in the real world:

- The Internet: a graph of web pages
- Social network: a graph of people
- Geographic Information System: a graph of locations
- Gene-Regulatory Network: a graph of genomic components

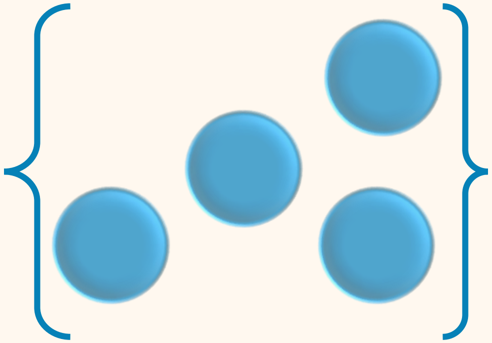


Graphs in Mathematics

Mathematical definition of a graph: **G=(V,E)**

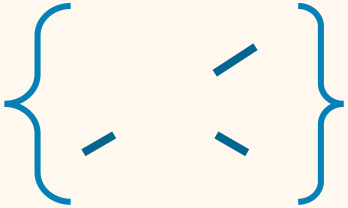
V is a set of nodes
(also called „vertices“)

V=

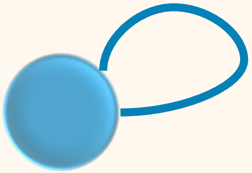


E is a set of edges
(„links“/“relationships“)

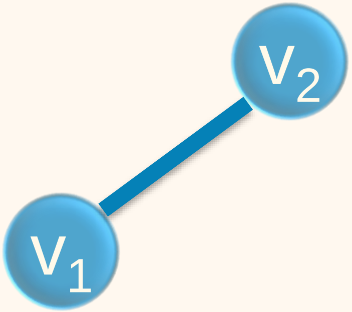
E=



Special case:
Self-loop



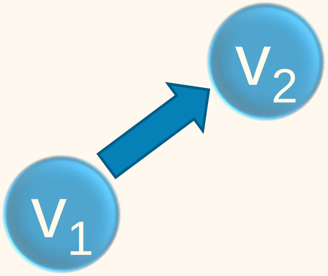
Direction of edges



An edge can be

- **undirected**: $\mathbf{e=\{v_1,v_2\}}$
  - can be traversed in both directions

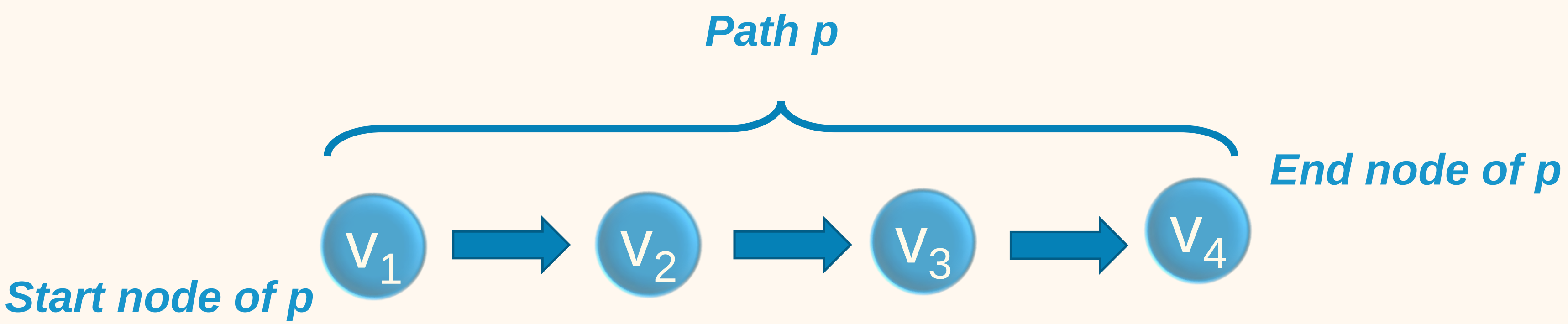
- **directed**: $\mathbf{e=(v_1,v_2)}$
  - can be traversed in one direction



Traversal and path

Traversal:
Go from one node to another by following an edge

Path:
A concatenation of nodes and edges that can be traversed in the graph

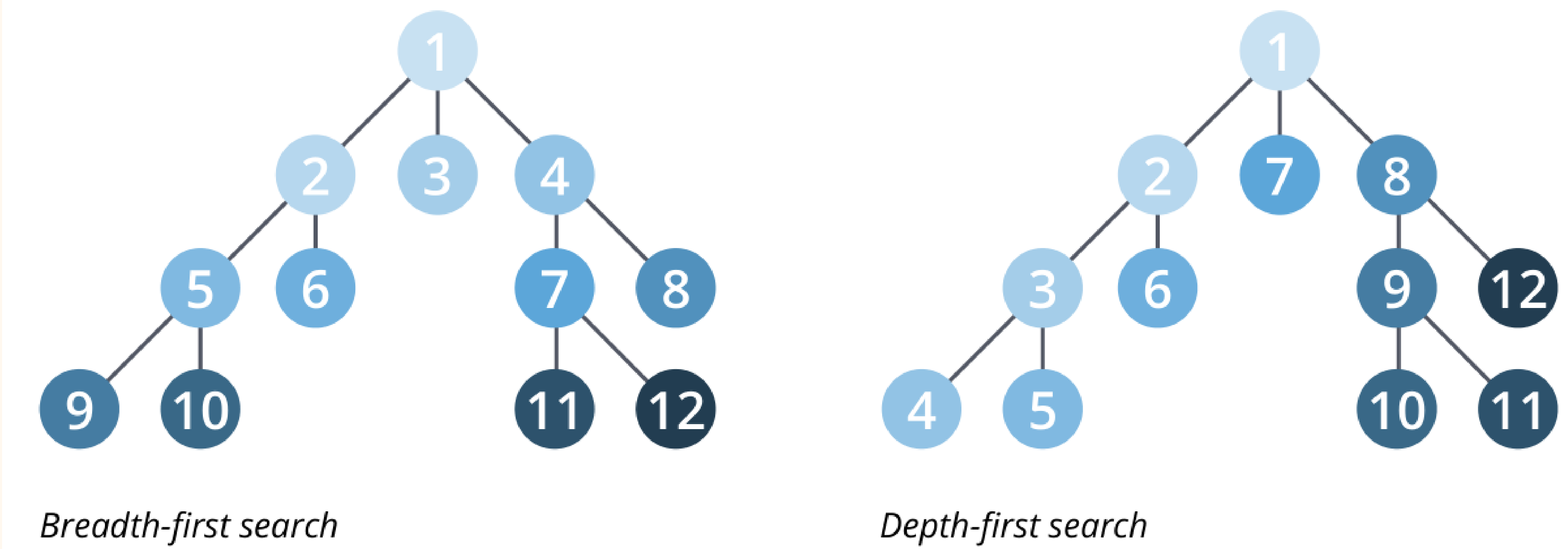


Exhaustive Graph Traversals

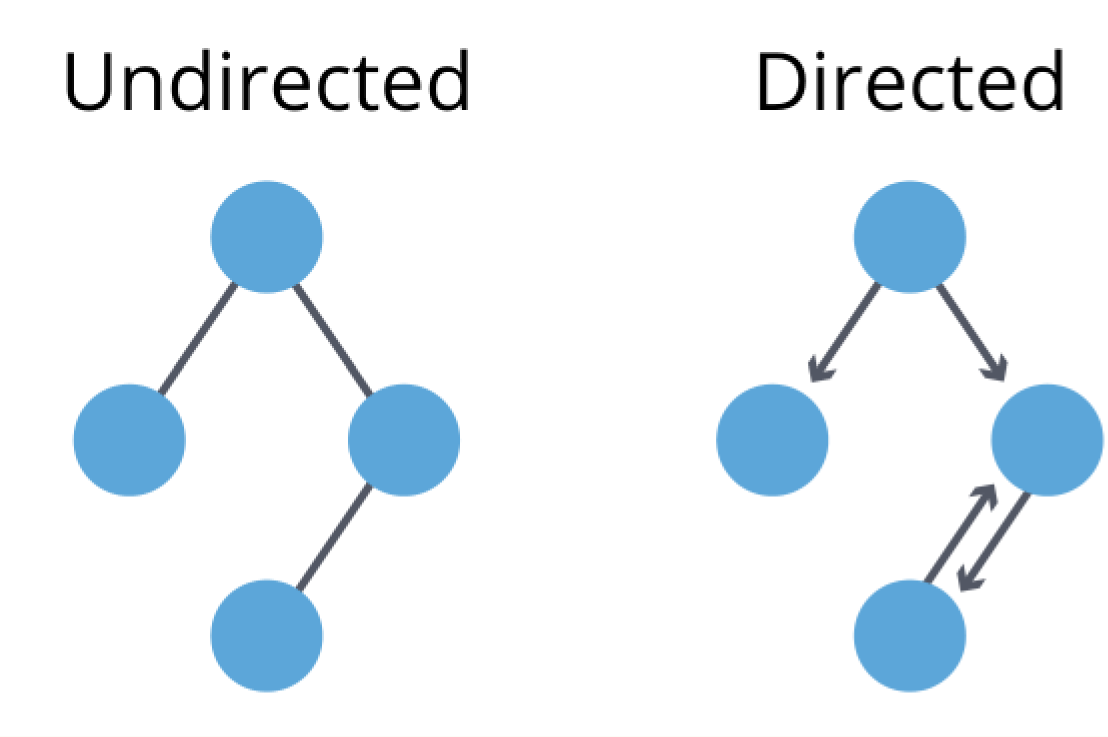
BFS: From the start node visit all direct neighbors first before visiting a neighbor‘s neighbor

DFS: From the start node choose one neighbor then visit the neighbor‘s neighbor

Picture source: [1, page 18]



Direction in graphs

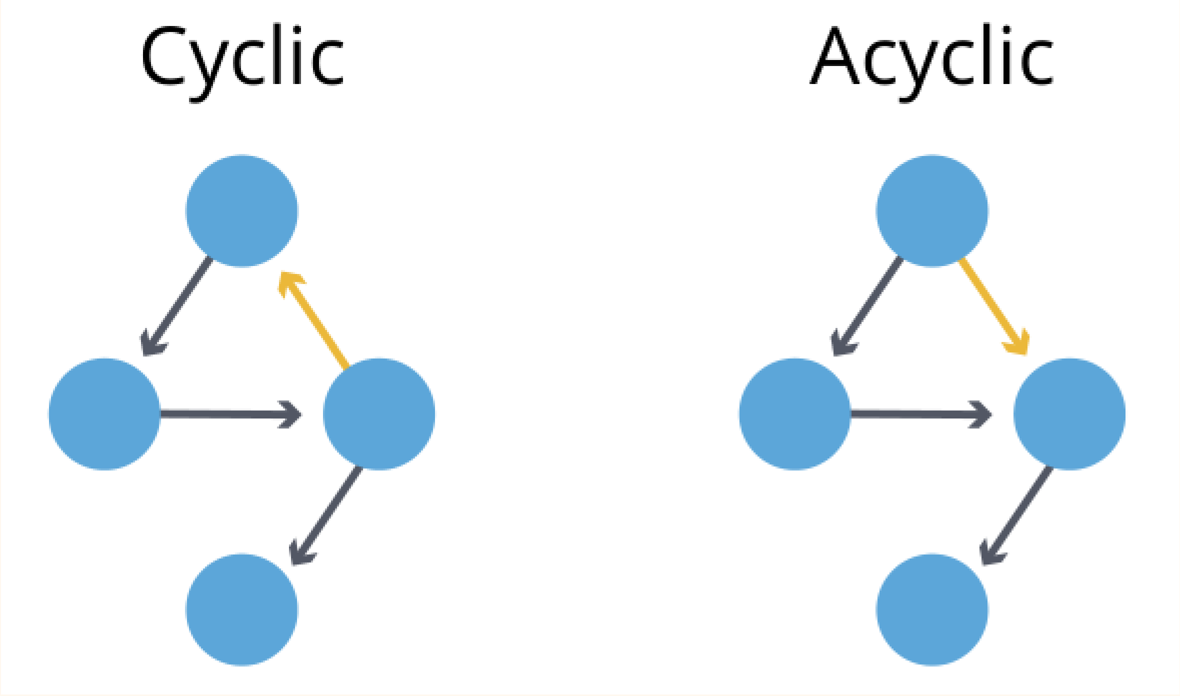


Picture source:
[1, page 19]

Example:
Mutual friendship relation

Example:
One-way streets

Cycles in graphs

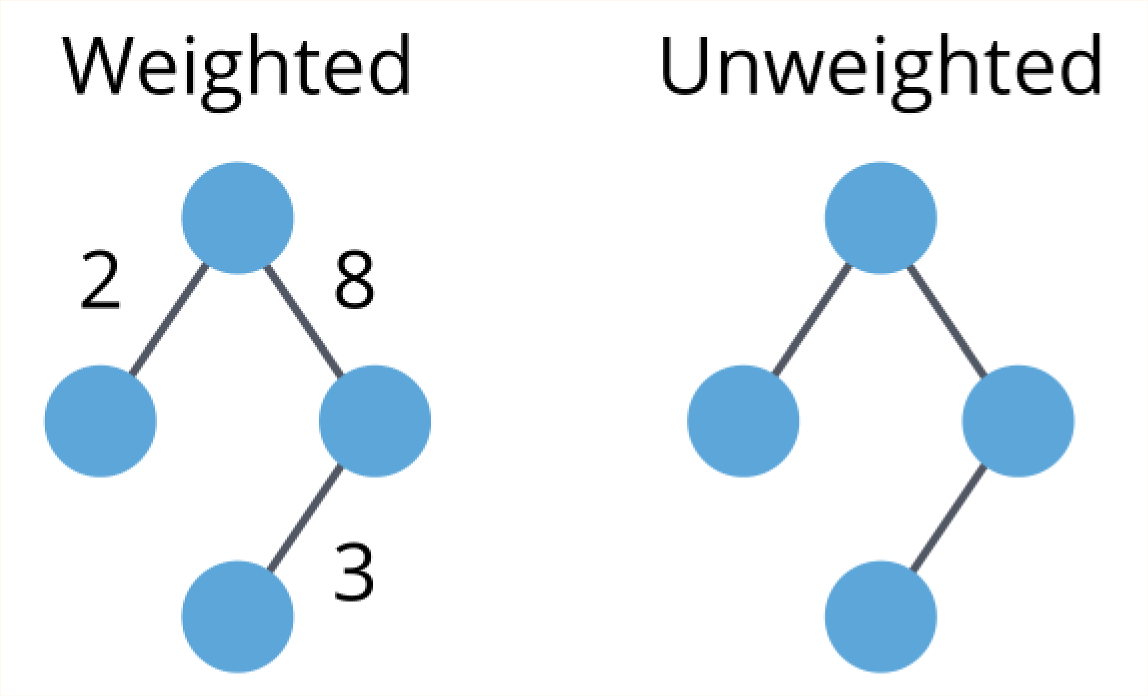


Picture source:
[1, page 19]

Cycle: A path containing the same node as start and end node

Triangle: A cycle with three nodes

Weights in graphs

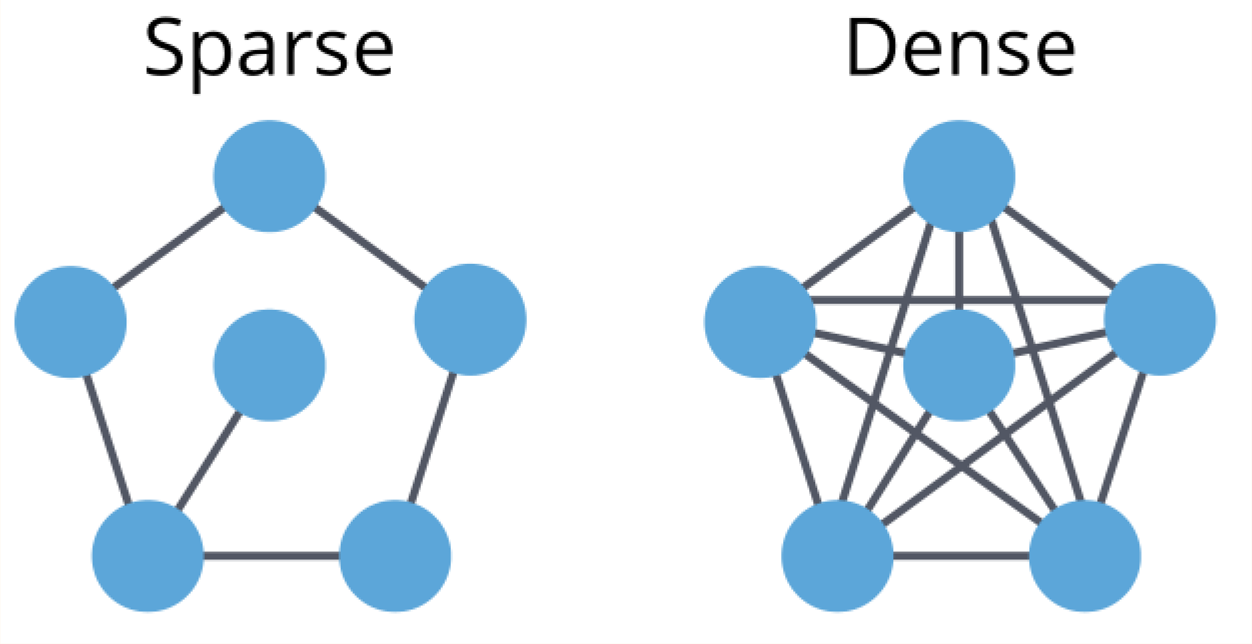


Picture source:
[1, page 19]

Example „shortest path“:
Find a path between two nodes with minimum cost

Example „shortest path“:
Find a path with minimum number of edges

Edge count in graphs



Picture source:
[1, page 19]

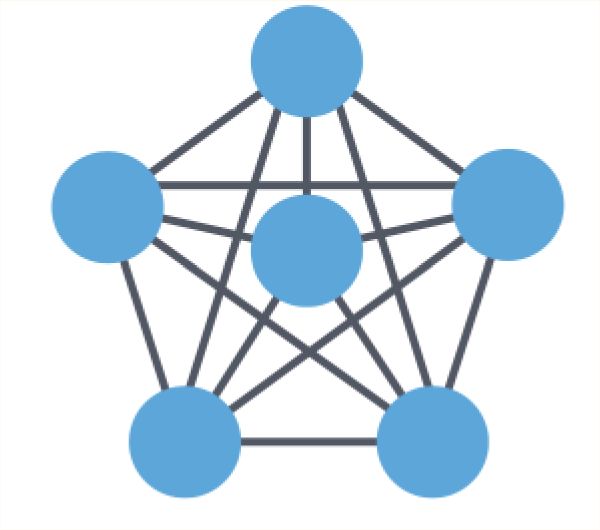
Only few edges exist

Many edges exist

Complete graph:
All edges exist

Quiz

What is the **edge count** in a complete undirected graph without self-loops?



Picture source: [1, page 19]

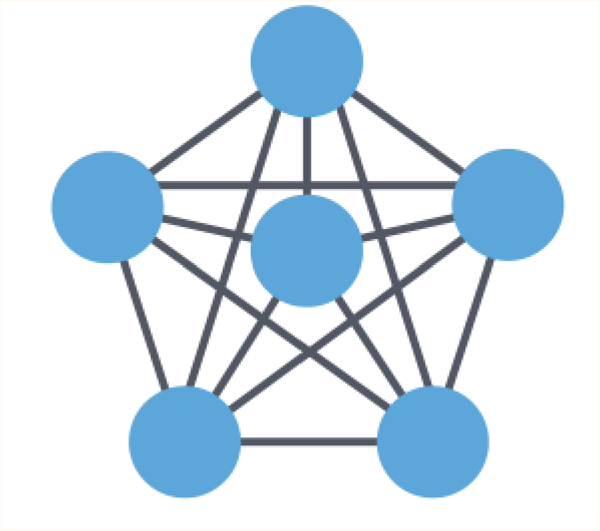
Answer

What is the edge count in a
complete undirected graph without self-loops?

$$\binom{|\boldsymbol{V}|}{2} = \frac{|\boldsymbol{V}|!}{2!\,(|\boldsymbol{V}|-2)!} = \frac{|\boldsymbol{V}|(|\boldsymbol{V}|-1)}{2}$$

Example: $|\boldsymbol{V}|=6$

$$\binom{6}{2} = \frac{6!}{2!4!} = \frac{6\cdot 5}{2} = 15$$



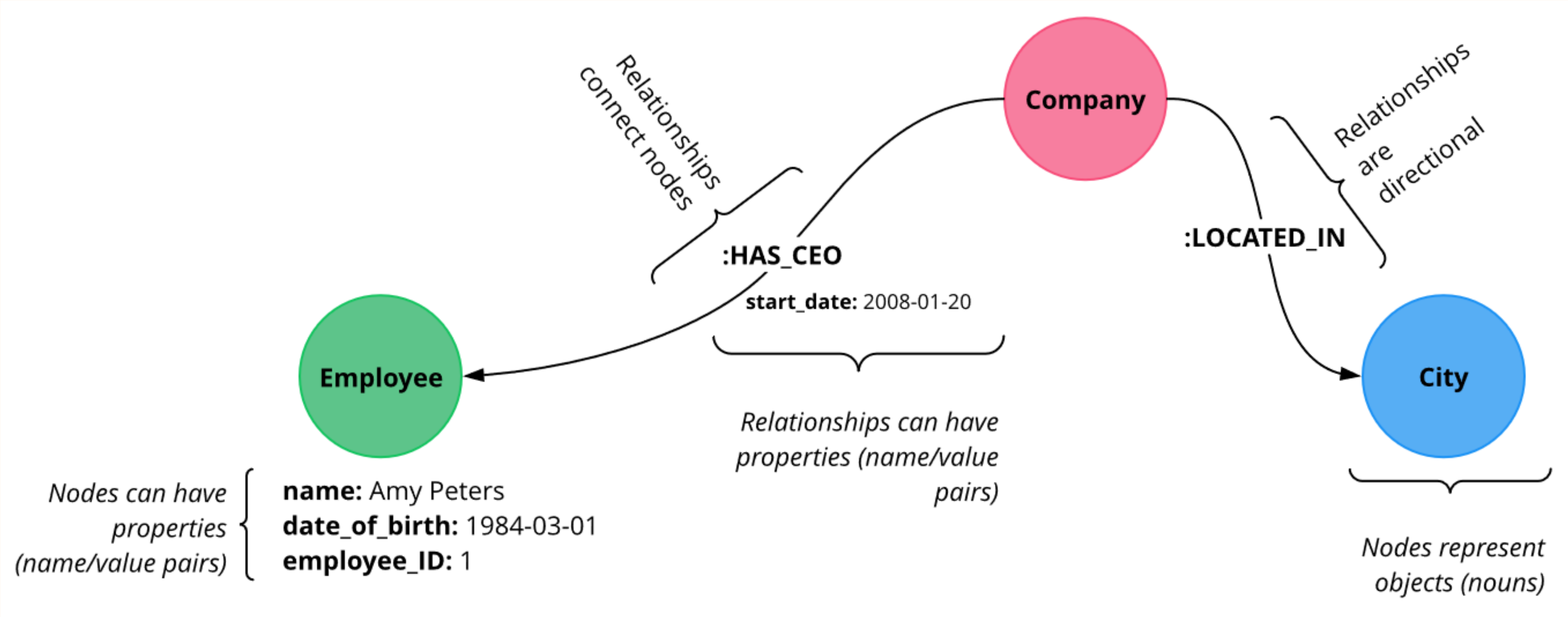
Picture source:
[1, page 19]

Agenda

- Short CV of speaker
- Graph Theory Basics
- **The Neo4J Database**
- Graph Algorithms
  - Centralities
  - Path Finding
  - Community Detection

Property Graph Model

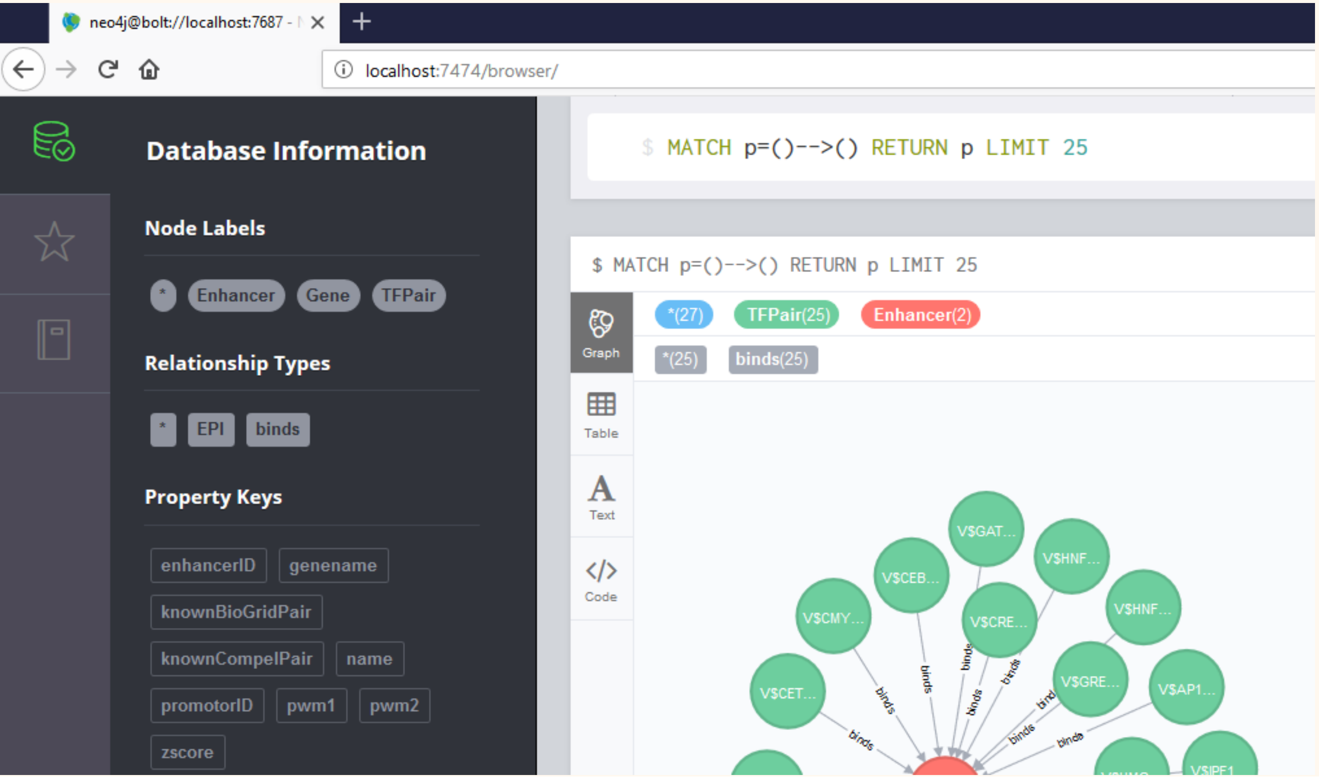
Nodes have **labels** (e.g. Employee)
Edges have **types** (e.g. :HAS_CEO)
Information is stored in ***name:value*** pairs (**„properties“**)



Picture source: https://neo4j.com/developer/graph-database/

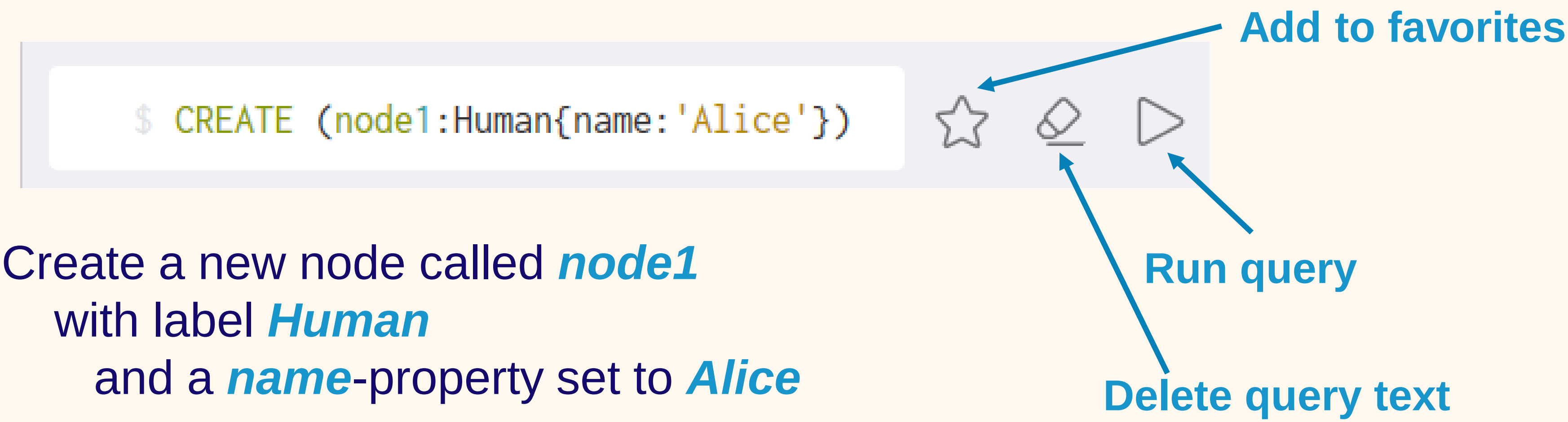
Neo4J Graph Database

- Open source graph database written in Java
- Query language called Cypher
- Interface called Neo4J Browser



- Visualization of query results
- Support for graph algorithms

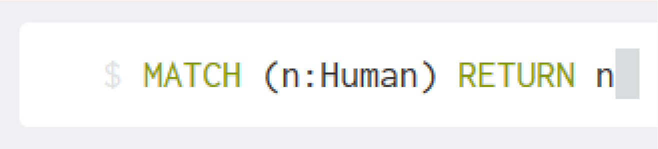
Create Nodes with Cypher



Create a new node called ***node1***
with label ***Human***
and a ***name***-property set to ***Alice***

Task 1:
Create a new node called ***node2***
with label ***Human***
and a ***name***-property set to ***Bob***

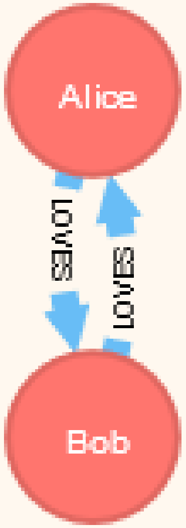
Task 2:
Find all nodes matching the label ***Human***
and return them



Create Edges with Cypher

```
MATCH (node1:Human{name:'Alice'}) MATCH (node2:Human{name:'Bob'})
CREATE (node1)-[:LOVES{until:'forever'}]->(node2)
```

Find two nodes **matching** the label ***Human*** and **create** a new edge with type ***:LOVES*** and an ***until***-property set to ***forever***



Task 3:
Create a new edge pointing from ***Bob*** to ***Alice***

Task 4:
Find all paths ***p*** with edges of type ***:LOVES*** and return them

```
$ MATCH p=()-[r:LOVES]->() RETURN p
```

MERGE

Avoid duplicates:

If node / edge does **not** exist: CREATE it as new

If node / edge exists: MATCH and return the node / edge

```
$ MERGE (node1:Human{name:'Alice'})
```

Deletion with Cypher

Delete edge between two nodes with certain **properties**

```
MATCH (n1:Human)-[e:LOVES]->(n2:Human) WHERE n1.name='Alice' AND n2.name='Bob'
DELETE e
```

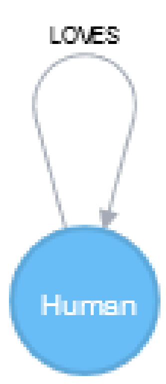
Or delete edge by **ID**

```
$ MATCH ()-[e:LOVES]->() WHERE id(e)=14 DELETE e
```

Graph schema with Cypher

Show schema information (all **types** and **labels**)

```
$ CALL db.schema
```



Agenda

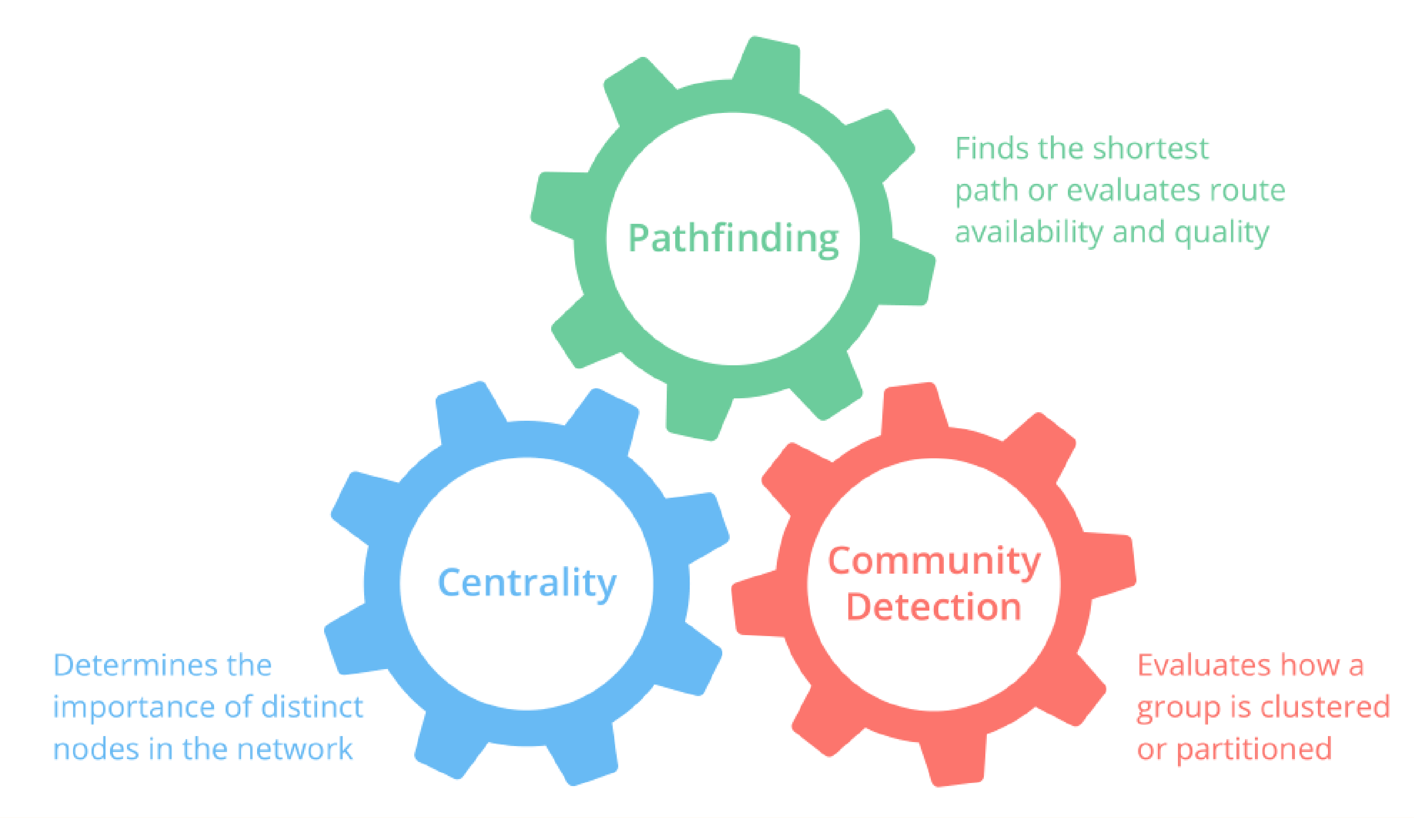
- Short CV of speaker
- Graph Theory Basics
- The Neo4J Database
- **Graph Algorithms**
  - Centralities
  - Path Finding
  - Community Detection

Graph Algorithms

Centrality: Find one or more nodes with an optimal score

Pathfinding: Find one or more optimal paths in a graph

Community Detection: Find densely connected subgraphs



Picture source: [1, page 15]

Agenda

- Short CV of speaker
- Graph Theory Basics
- The Neo4J Database
- Graph Algorithms
  - **Centralities**
  - Path Finding
  - Community Detection

Centrality: PageRank

Centrality: PageRank

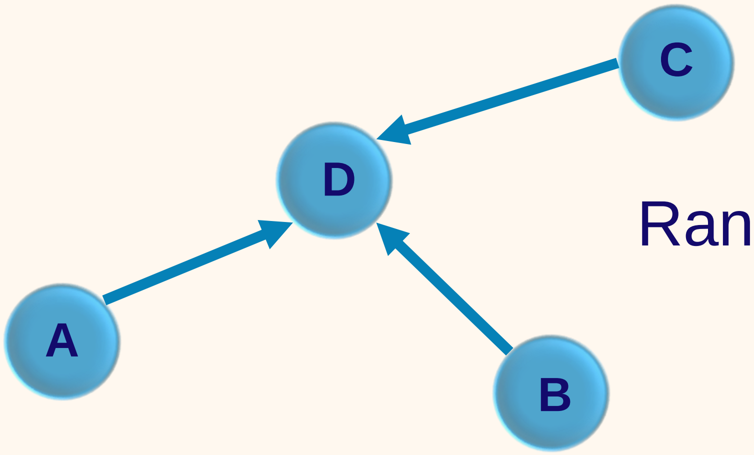
Google's PageRank score represents the importance of web pages in the Internet

- Rank of a page depends on the **in-links** to a page
- PageRank is transitive
  - PageRank of a node is influenced by the **neighbors'** PageRank
- In each iteration a node distributes its rank to its neighbors along its **out-links**

Reference: Page, L., Brin, S., Motwani, R., & Winograd, T. (1999). The PageRank citation ranking: Bringing order to the web. Stanford InfoLab

Centrality: PageRank

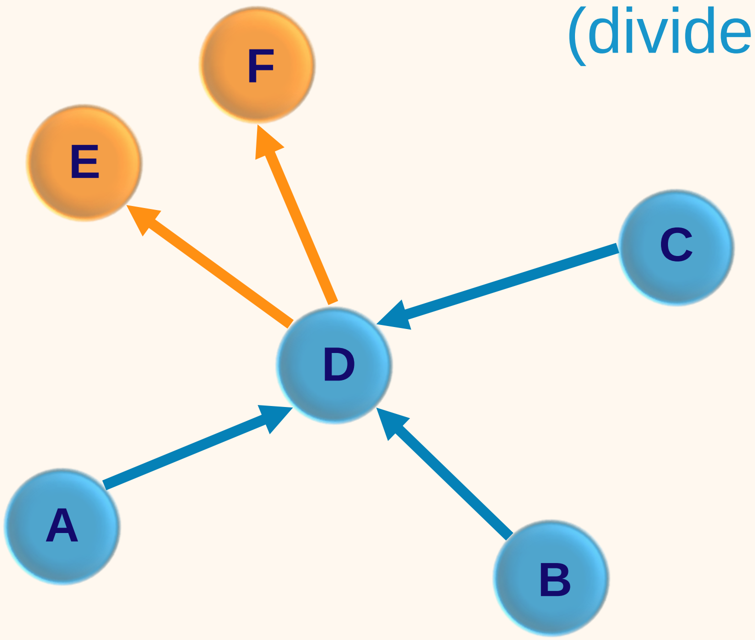
Step 1: Sum up the ranks of your in-links



Rank(**D**)=
Rank(**A**)+Rank(**B**)+Rank(**C**)

Centrality: PageRank

Step 2: Distribute your rank among your out-links (divide by amount of out-links)



Rank(**E**)=½ Rank(**D**)

Rank(**F**)=½ Rank(**D**)

Centrality: PageRank

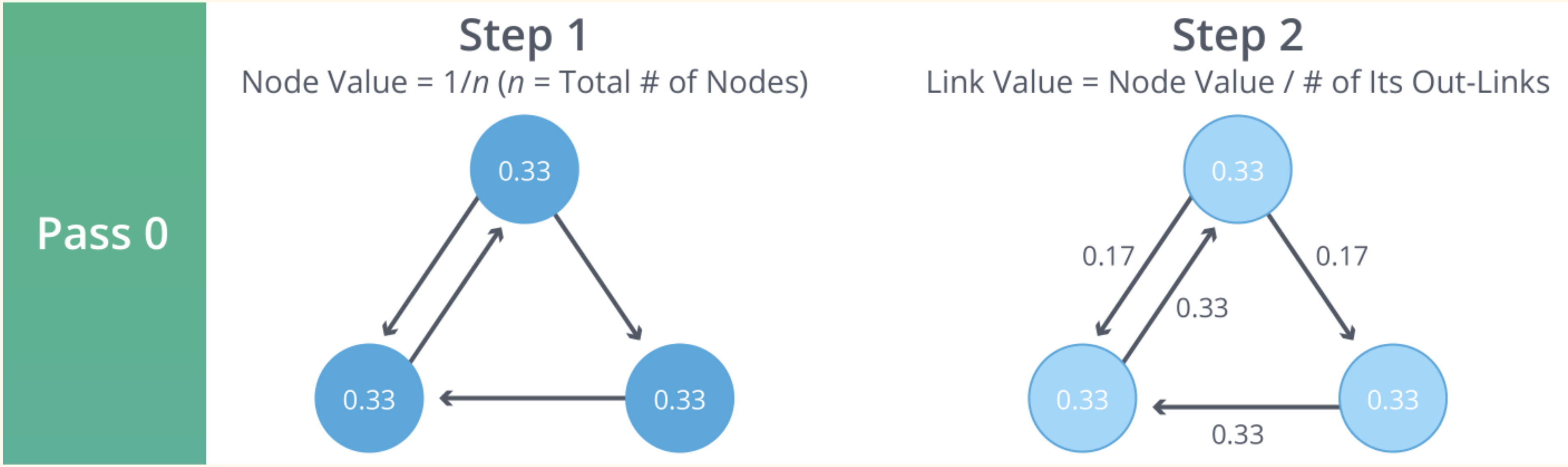
Random surfer model:

- A web surfer randomly follows **out-links** of pages
- **Uniform** probability distribution: if a page has **m** out-links, each link is followed with probability $^{1}/_{m}$
- The higher the PageRank, the higher the likelihood that a random surfer will be at this page at an arbitrary point of time

Note: A few in-links from very **important** pages raise your PageRank more than many in-links from **unimportant** pages.

Centrality: PageRank

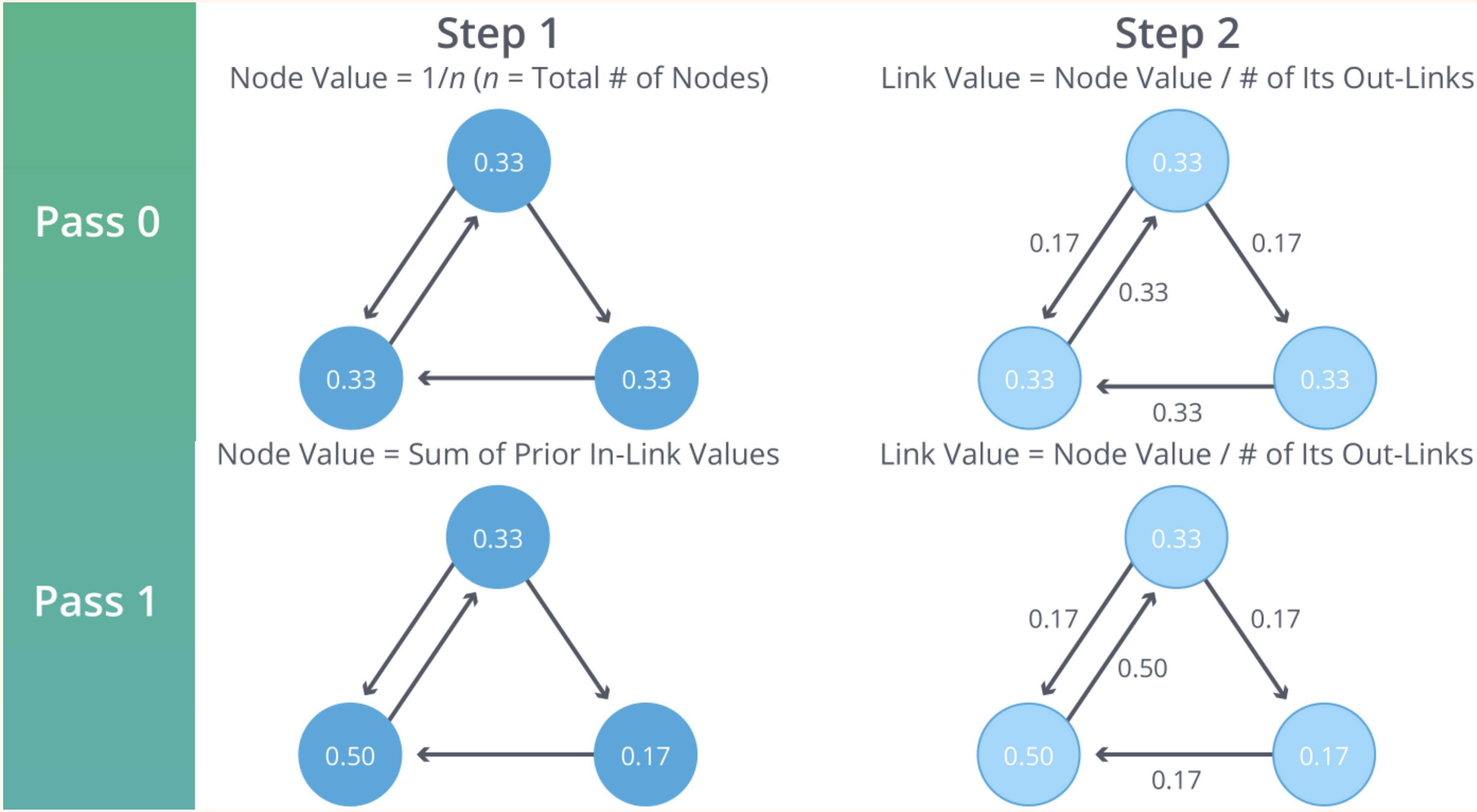
Example: start with equal PageRank for all pages, then iterate for a certain number of rounds



Picture source: [1, page 35]

Centrality: PageRank

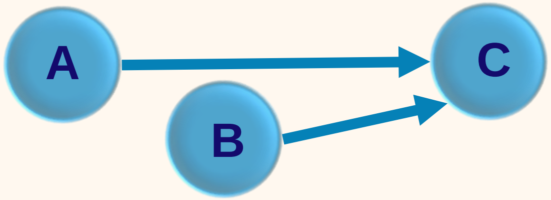
Picture source: [1, page 35]



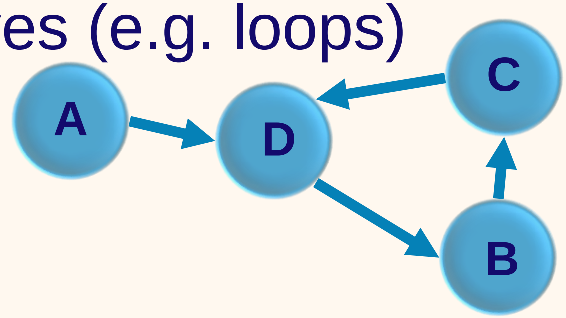
Centrality: PageRank

Problem Cases:

- **Dead end:** a node that has no out-link



- **Rank sink:** a group of nodes that have in-links from other nodes but out-links just among themselves (e.g. loops)



- **Disconnected subgraphs:** subgraphs without links between each other

These problem cases disturb the **distribution** of PageRank

Solution:

- add a constant **rank source** to each page
- corresponds to random **jumps** (instead of following links)

Centrality: PageRank

rank source for each page (for **random jumps**)

The final formula:

$$\text{Rank}(u) = d \cdot \sum_{v \epsilon\, In(u)} \frac{\text{Rank}(v)}{|Out(v)|} + (1 - d)$$

Damping factor (reduces the likelihood of following a link by **random surfer**)

v has an **in-link** to **u**

Rank of **v** **divided by amount of out-links** (for **random surfer**)

Possibly: **Normalization** of ranks so that all ranks sum up to 1

Graph Algorithms in Cypher

- Two options:
  1. **Streaming** of result

     ```
     CALL algo.pageRank.stream
     ```

     CALL algo.<name>.stream
     - Immediately outputs the result

  2. **Writing** result into a property

     ```
     CALL algo.pageRank
     ```

     CALL algo.<name>
     - Properties can be queried in a second query

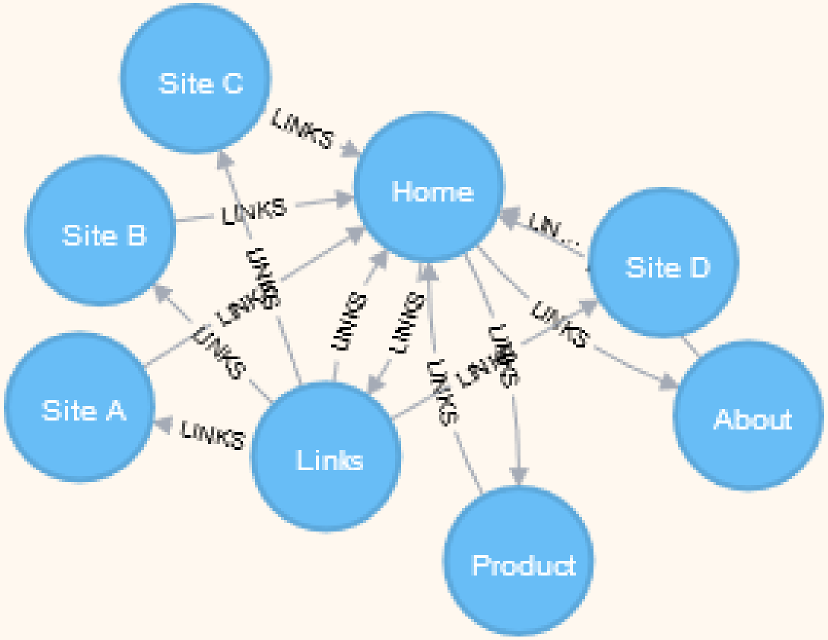
     ```
     write: true,writeProperty:"pagerank"
     ```

- Clause **YIELD** defines which statistics to print out

```
YIELD nodeId, score
```

PageRank in Cypher

Task 5: Create sample graph of 8 web pages (see https://neo4j.com/docs/graph-algorithms/current/ ➡ PageRank)



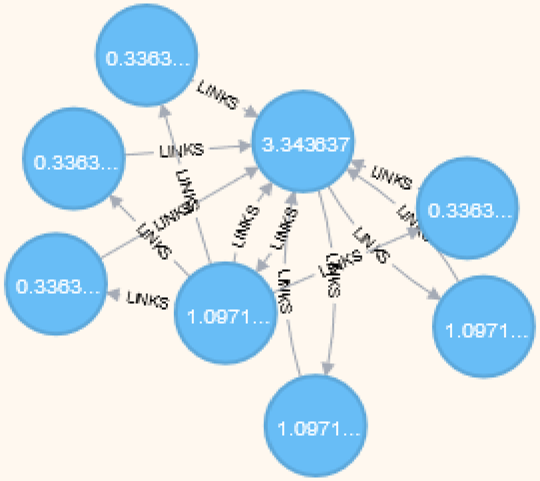
PageRank in Cypher

Task 6:

- Run PageRank with 20 iterations

```
CALL algo.pageRank.stream("Page", "LINKS",
{iterations:20})
YIELD nodeId, score
```

- Increase the amount of iterations and observe the effect on the PageRank
- Which page is the most important one?



- **Optional:** Write the PageRank into a node property called ***pagerank*** and display the page rank for each node

```
CALL algo.pageRank('Page', 'LINKS',
  {iterations:20, dampingFactor:0.85, write: true,writeProperty:"pagerank"})
```

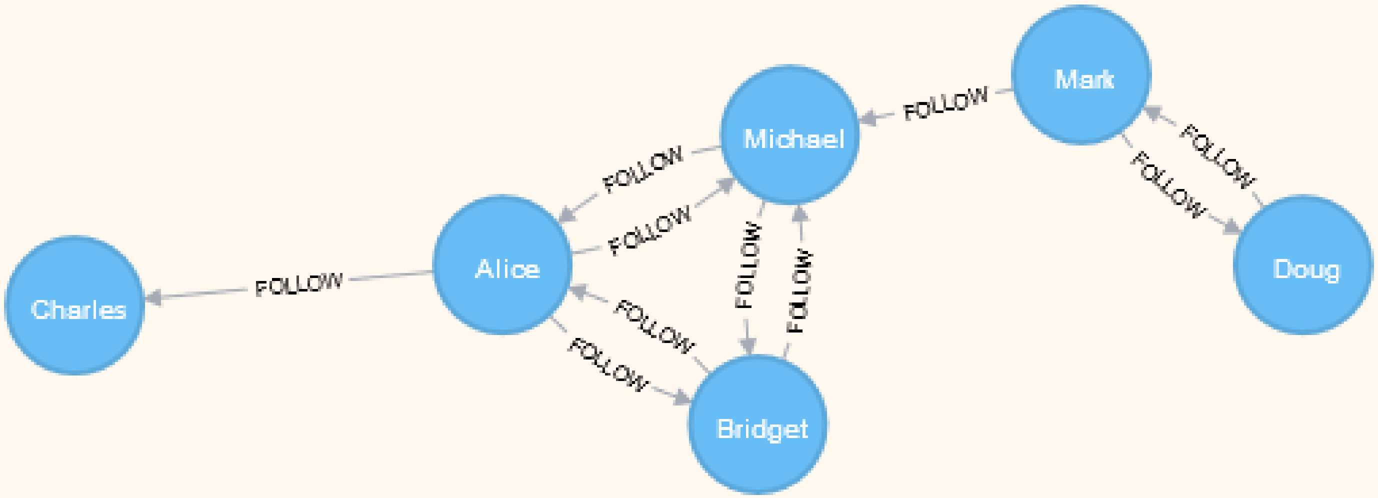
Centrality: Degree

Centrality: Degree

Degree centrality is the amount of in-links and out-links of each node

Task 7:

- Create sample graph of 6 users with a **:FOLLOW** relationship (see https://neo4j.com/docs/graph-algorithms/current/ ➡ The Strongly Connected Components algorithm)



Centrality: Degree

Degree centrality is the amount of in-links and out-links of each node

Task 8:

- For each user return the amount of out-links and the amount of in-links

```
MATCH (u:User)
RETURN u.id AS name,
size((u)-[:FOLLOW]->()) AS follows,
size((u)<-[:FOLLOW]-()) AS followers
```

- Which user has the highest degree centrality?

- **Optional:** Return the total amount of links of each node

Centrality: Betweenness

Centrality: Betweenness

Betweenness centrality is the fraction of shortest paths going through a node

$$C_B(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

The amount of shortest paths between s and t going through v

The total amount of shortest paths between s and t

Betweenness centrality for node v

Sum over all node pairs s,t (different from v)

Nodes with high betweenness centrality ensure crucial connections in the graph: „Bridge“ between different subgraphs

Centrality: Betweenness

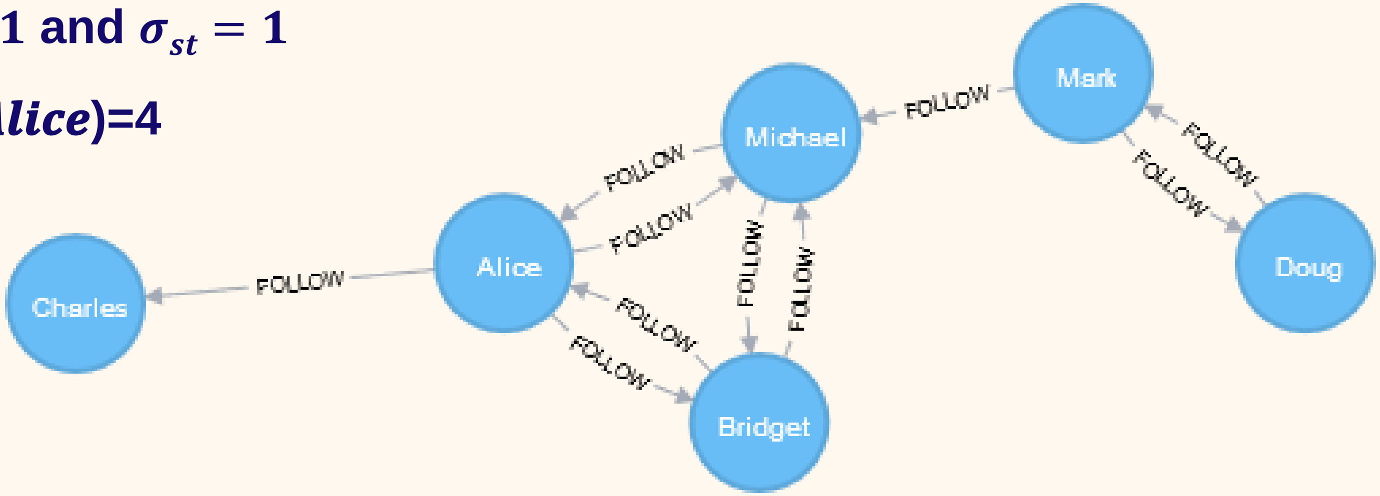
$$C_B(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

Let v = Alice
Shortest paths via Alice:
1. **Michael-Charles**
2. **Mark-Charles**
3. **Doug-Charles**
4. **Bridget-Charles**

For all these paths:
Only one shortest path (and only through Alice)

$\sigma_{st}(Alice) = 1$ **and** $\sigma_{st} = 1$

Hence $C_B(Alice)=4$



Centrality: Betweenness

Task 9:

- For each user return the betweenness centrality along the **:FOLLOW** relationship

```
CALL algo.betweenness.stream('User','FOLLOW',{direction:'out'})
YIELD nodeId, centrality
MATCH (user:User) WHERE id(user) = nodeId
RETURN user.id AS user,centrality
ORDER BY centrality DESC;
```

- **Optional:** remove the edges between **Michael** and **Bridget** and observe the effect on the betweenness centrality

Agenda

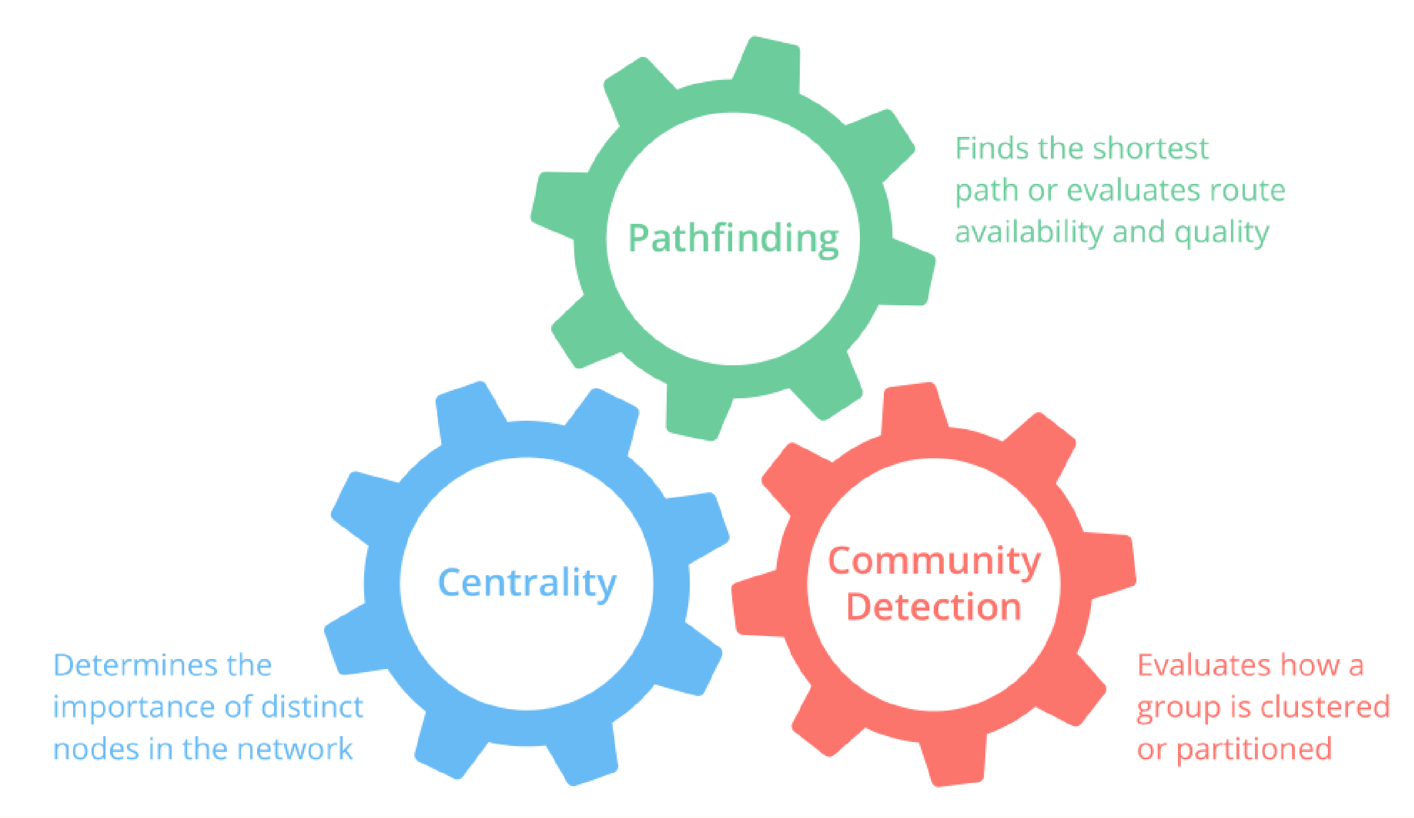
- Short CV of speaker
- Graph Theory Basics
- The Neo4J Database
- Graph Algorithms
  - Centralities
  - **Path Finding**
  - Community Detection

Graph Algorithms

Centrality: Find one or more nodes with an optimal score

Pathfinding: Find one or more optimal paths in a graph

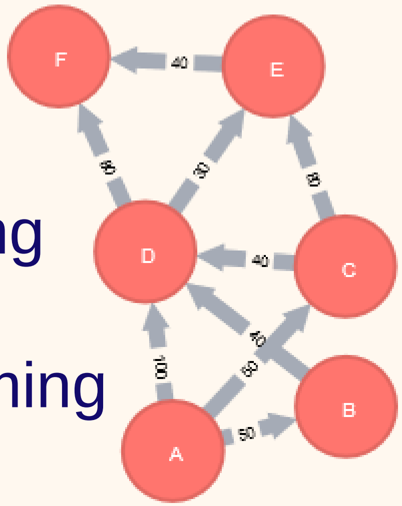
Community Detection: Find densely connected subgraphs



Picture source: [1, page 15]

Pathfinding:
Minimum Weight Spanning Tree

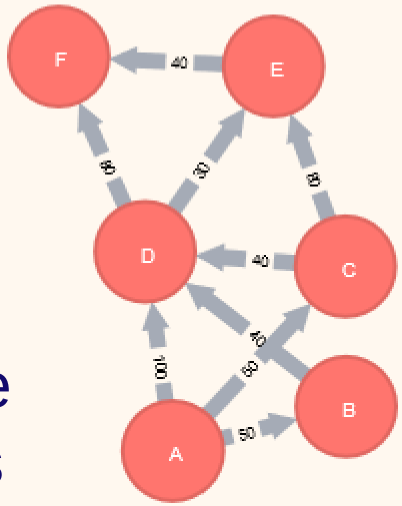
Pathfinding:
Minimum Weight Spanning Tree



- **Minimum Spanning Tree:**
  - If a graph has |**V**| nodes, then a spanning tree has |**V**|***-1*** edges
  - The tree has a **root** node (with no incoming edges)
  - From the root node we can reach **all other** nodes in the graph with minimal **total** cost by using the edges of the spanning tree

Use case:
Distribute information from the root node to all other nodes for example in communication networks or social networks

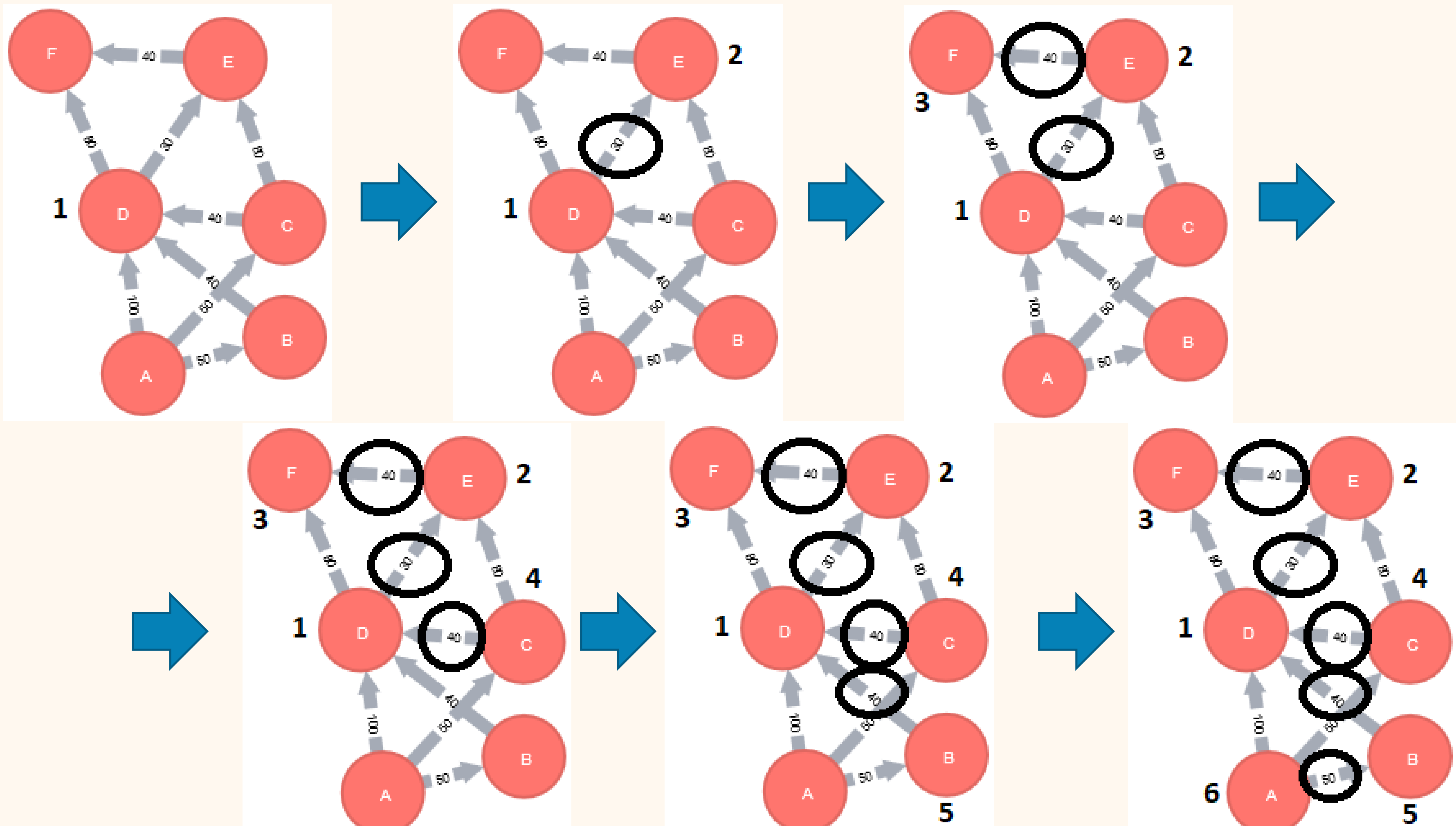
Pathfinding:
Minimum Weight Spanning Tree



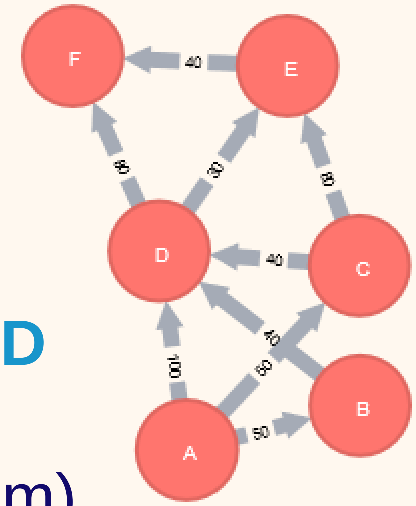
- **Prim‘s algorithm**
  - Maintain a list of unvisited nodes
  - Start with the **root** node
  - From the unvisited nodes select the one that can be connected to the tree nodes with **minimal** cost
  - **Repeat** until all nodes are visited

For undirected graphs:
We ignore direction of edges (traversal in both directions)

Pathfinding: Minimum Weight Spanning Tree



Pathfinding:
Minimum Weight Spanning Tree



Task 10:

- Create sample graph of 6 locations with a **:ROAD** relationship (see https://neo4j.com/docs/graph-algorithms/current/ ➡ The Shortest Path algorithm)

- **Optional:** Display the cost of each road on the edge

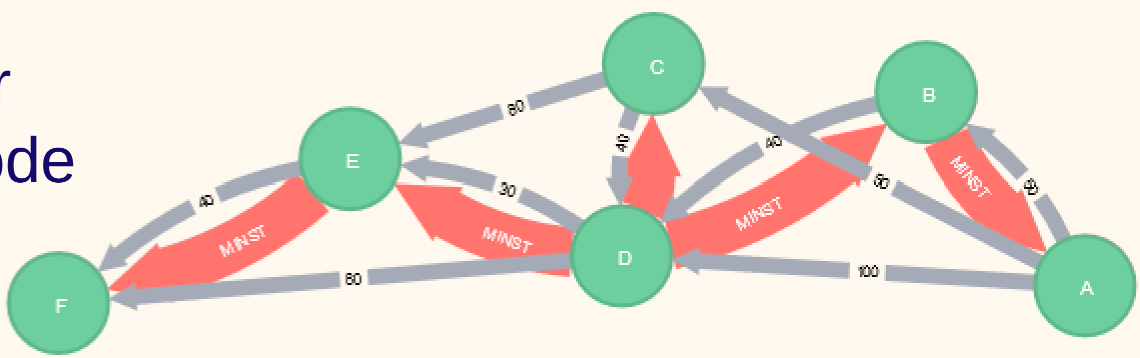
Pathfinding:
Minimum Weight Spanning Tree

Task 11:

- Choose location **D** as the root node and compute a minimum spanning tree by creating new edges of type **:MINST**

```
MATCH (n:Loc {name:"D"})
CALL algo.spanningTree.minimum('Loc', 'ROAD', 'cost', id(n),
  {write:true, writeProperty:"MINST"})
YIELD loadMillis, computeMillis, writeMillis, effectiveNodeCount
RETURN loadMillis, computeMillis, writeMillis, effectiveNodeCount
```

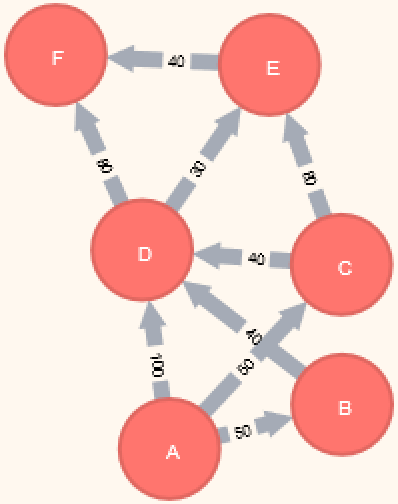
- **Optional:** use another location as the root node



Pathfinding: Shortest Path

Pathfinding: Shortest Path

- **Shortest Path with minimum cost:**
  - Provide a start and an end node
  - Find the minimum-cost path between these two nodes
  - In unweighted graph: edge cost 1



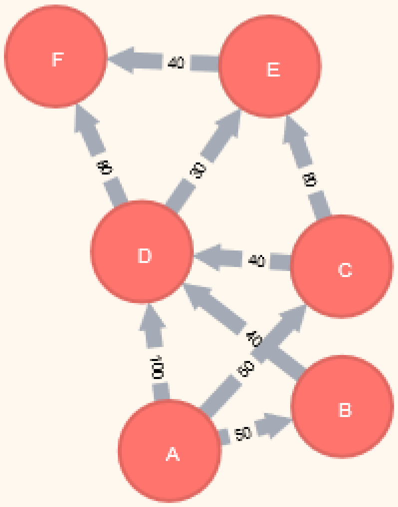
- **Example:**
  - Start node **A** and end node **F**
  - One minimum-cost shortest path:

    A – C – D – E – F

    50 + 40 + 30 + 40 = 160

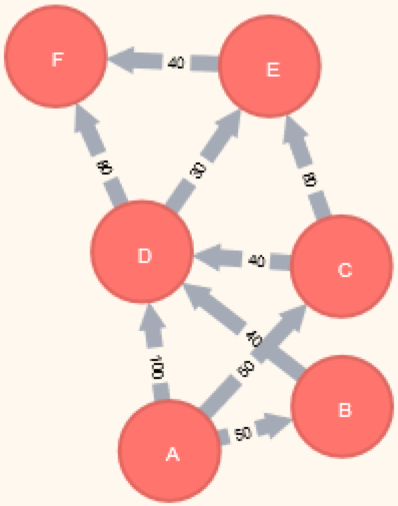
Use case:
Travel route planning

Pathfinding: Shortest Path



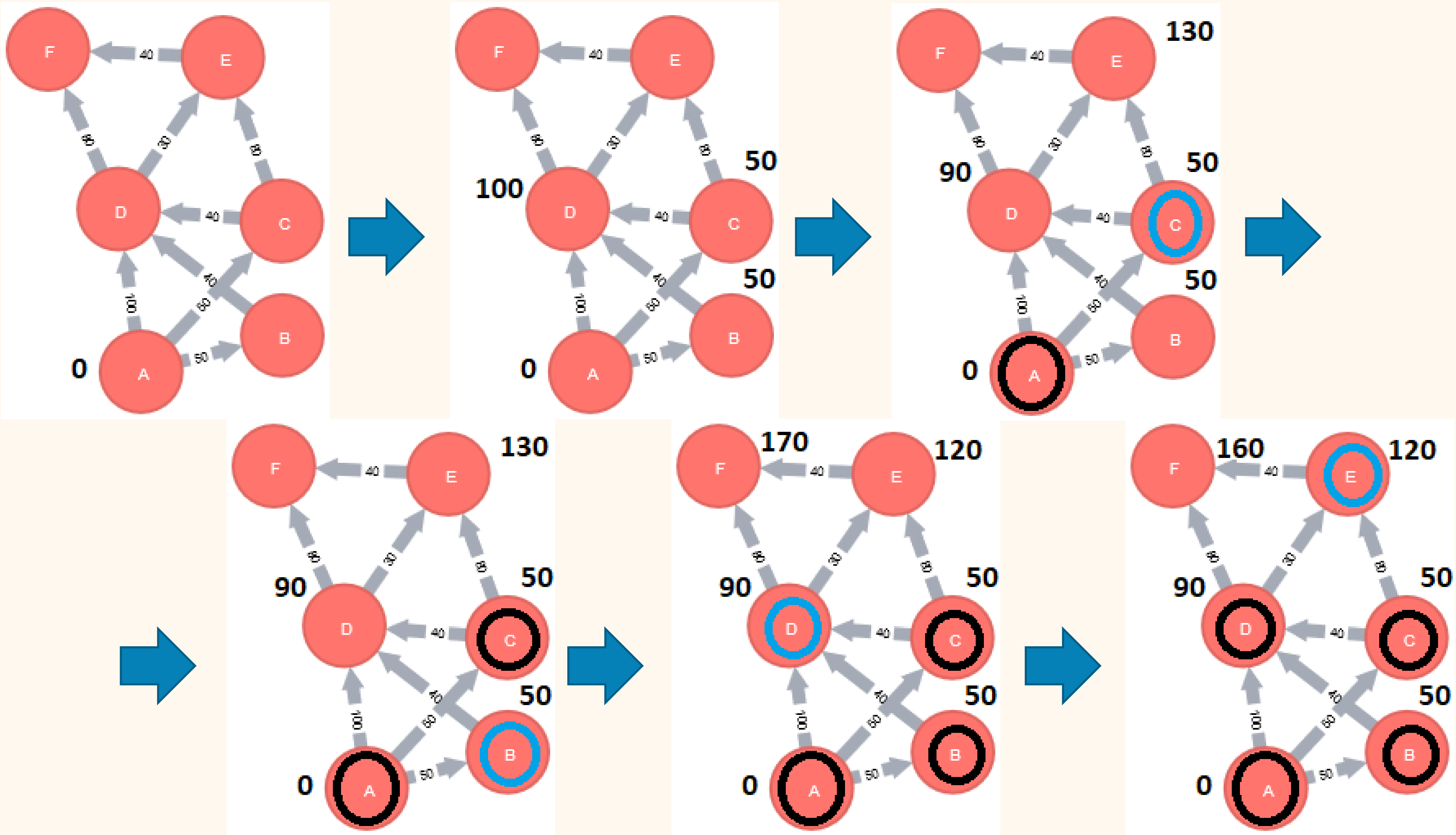
- **Dijkstra‘s algorithm:**
  - Maintain a list of unvisited nodes
  - For each node maintain its distance to the start node (initially: ∞)
  - Start with the **start** node and set its distance to 0
  - For **all** unvisited **neighbor** nodes: set their distance to be the edge cost to the start node
  - Remove the start node from the list of unvisited nodes

Pathfinding: Shortest Path



- **Dijkstra‘s algorithm (continued):**
  - Select one unvisited node with currently **smallest** distance
    - Make it the **current node**
  - For all **unvisited** neighbors of the current node:
    - Sum up the edge cost to the neighbor and the distance of the current node
    - If smaller than current distance of the neighbor: update distance (shorter path found)
  - **Remove** current node from list of unvisited nodes
  - Repeat until **end node** is visited

Pathfinding: Shortest Path



Pathfinding: Shortest Path

Task 12:

- Choose location **A** as the start node and location **F** as the start node and compute the shortest path

```
MATCH (start:Loc{name:'A'}), (end:Loc{name:'F'})
CALL algo.shortestPath.stream(start, end, 'cost')
YIELD nodeId, cost
RETURN algo.getNodeById(nodeId).name AS name, cost
```

- **Optional:** use other locations as start and end nodes

Agenda

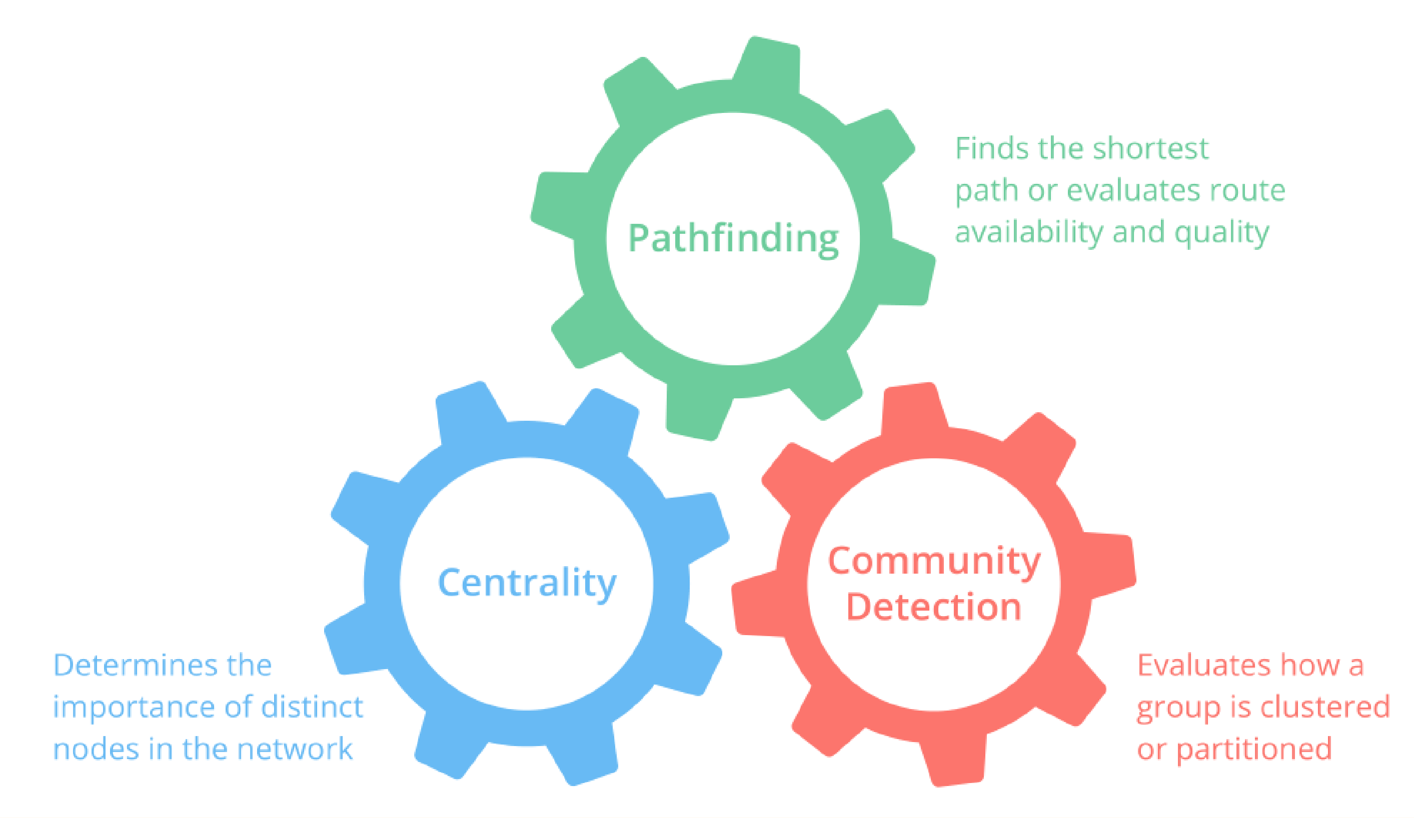
- Short CV of speaker
- Graph Theory Basics
- The Neo4J Database
- Graph Algorithms
  - Centralities
  - Path Finding
  - **Community Detection**

Graph Algorithms

Centrality: Find one or more nodes with an optimal score

Pathfinding: Find one or more optimal paths in a graph

Community Detection: Find densely connected subgraphs

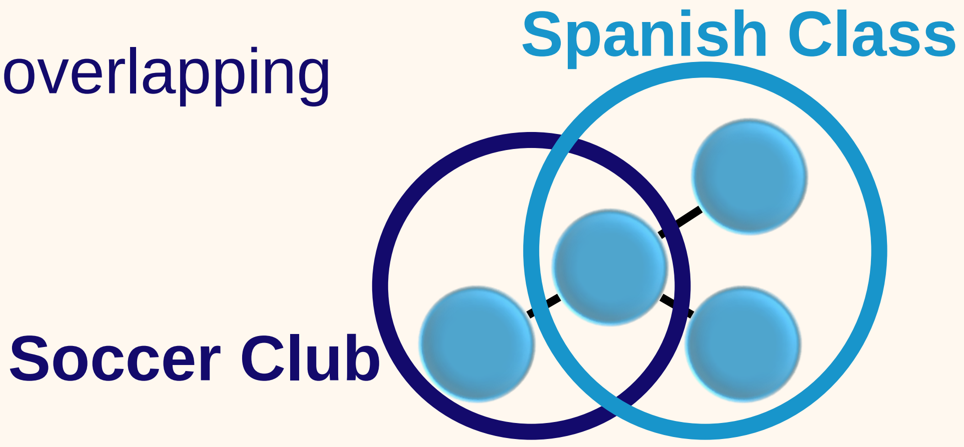


Picture source: [1, page 15]

Community Detection

Community: Densely connected subgraph

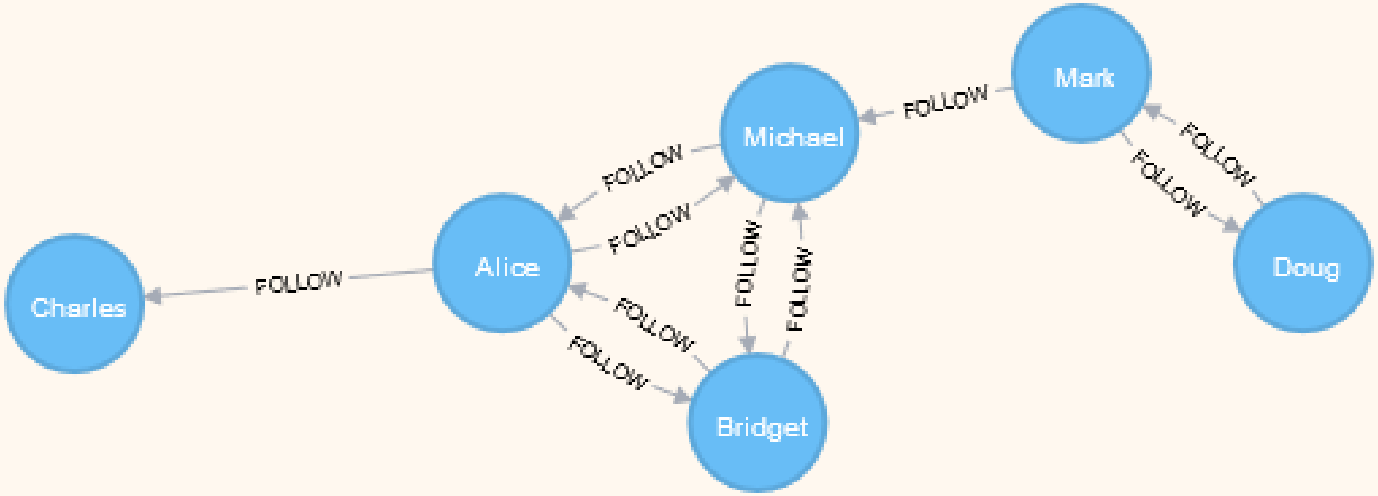
- More communication between the nodes of a community than to other nodes of the graph
- Communities may be overlapping



Community Detection: Strongly Connected Components

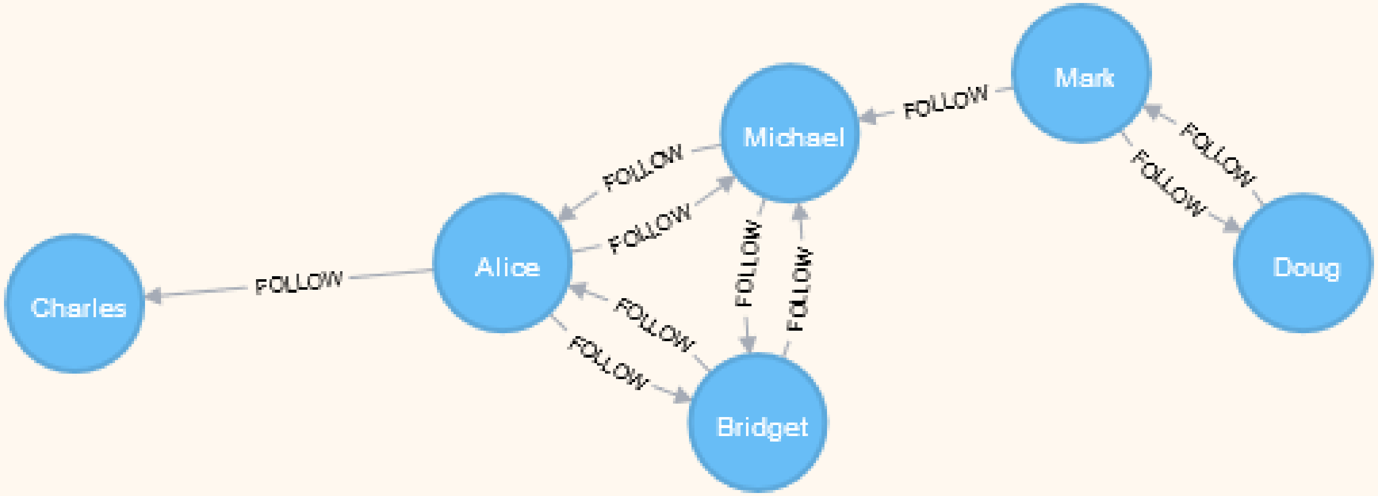
Community Detection: Strongly Connected Components

- **Strongly Connected Components:**
  - Apply to directed graphs
  - Two nodes **A**, **B** are in the same strongly connected component if there is a path from **A** to **B** **and** a path from **B** to **A**



Community Detection: Strongly Connected Components

- **Example:**
  - **Charles** has no out-links and is a component on his own
  - **Michael**, **Alice** and **Bridget** can all reach each other
  - **Mark** and **Doug** can reach each other

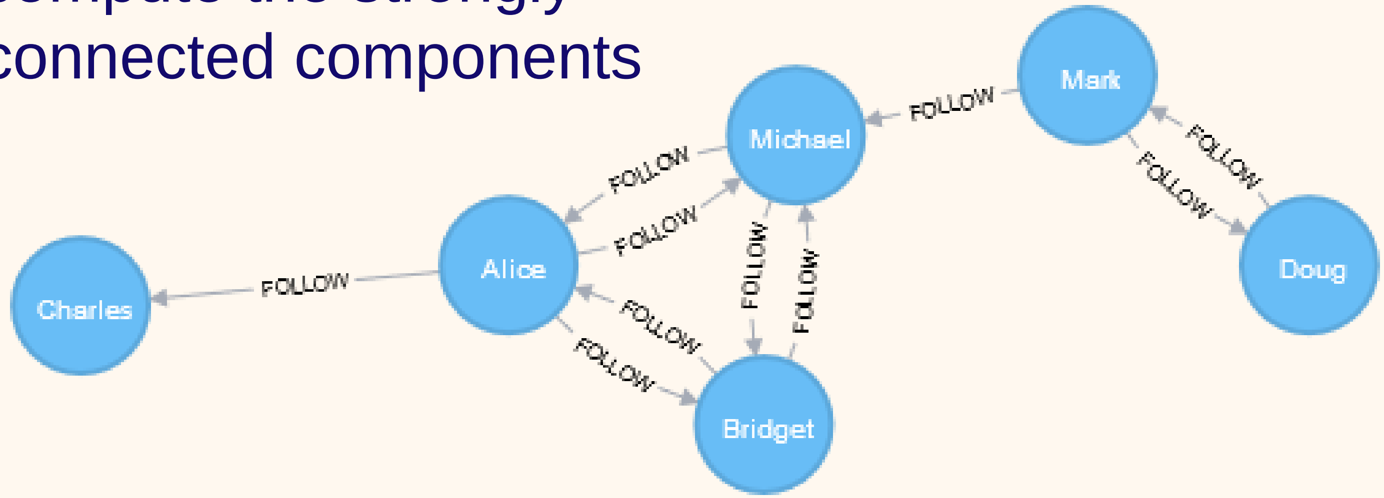


Community Detection: Strongly Connected Components

Task 13:

- For **User** nodes and the **:FOLLOW** relationship compute the strongly connected components

```
CALL algo.scc.stream("User","FOLLOW")
YIELD nodeId, partition
MATCH (u:User) WHERE id(u) = nodeId
RETURN u.id AS name, partition
```

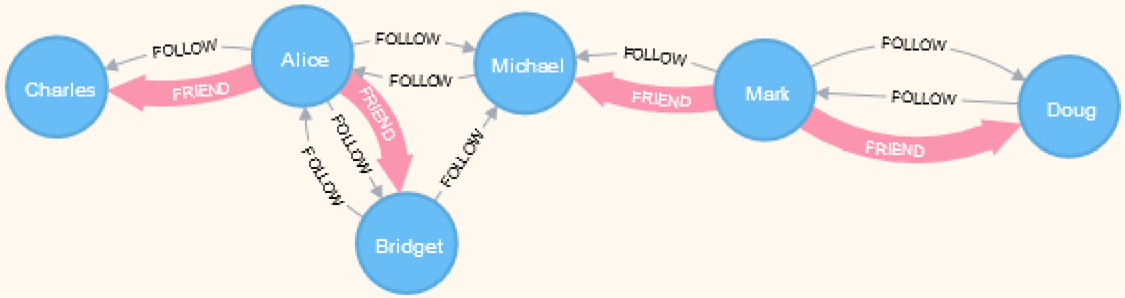


- **Optional:** add a **:FOLLOW** edge from **Charles** to **Alice** and observe the effect

Community Detection: Weakly Connected Components

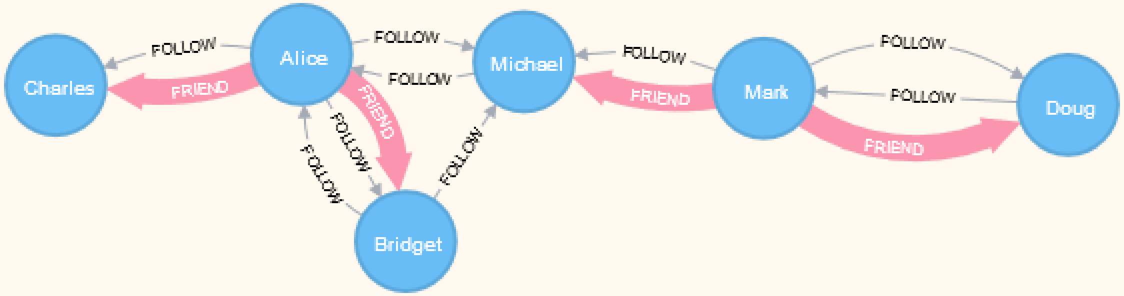
Community Detection: Weakly Connected Components

- **Weakly Connected Components:**
  - We interpret the graph as undirected
  - Two nodes **A**, **B** are in the same weakly connected component if there is a path from **A** to **B** **or** a path from **B** to **A**



Community Detection: Weakly Connected Components

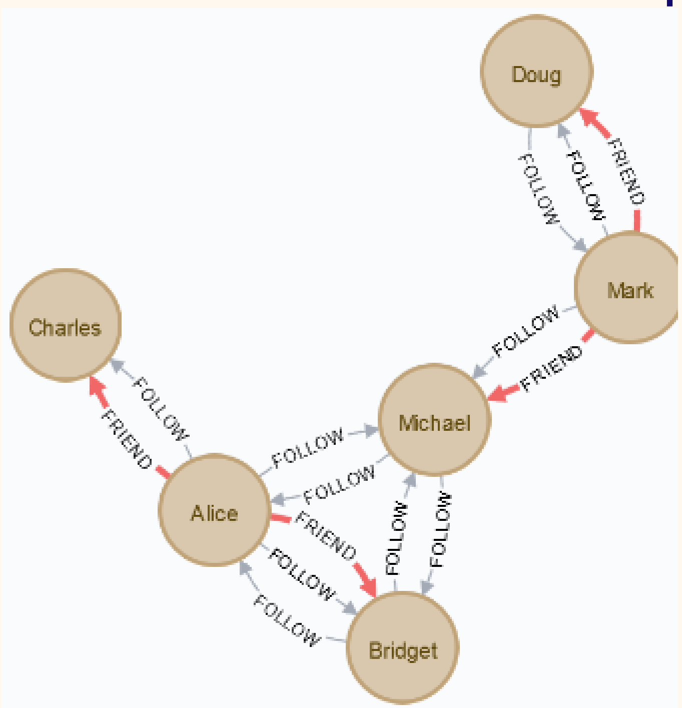
- **Example**
  - New **:FRIEND** relationship
  - **Charles**, **Alice** and **Bridget** are weakly connected
  - **Michael**, **Mark** and **Doug** are weakly connected



Community Detection: Weakly Connected Components

Task 14:

- Add the **:FRIEND** relationship to the graph
  (see https://neo4j.com/docs/graph-algorithms/current/
  ➡ The Connected Components algorithm)




```
MERGE (nAlice:User {id:'Alice'})
MERGE (nBridget:User {id:'Bridget'})
MERGE (nCharles:User {id:'Charles'})
MERGE (nDoug:User {id:'Doug'})
MERGE (nMark:User {id:'Mark'})
MERGE (nMichael:User {id:'Michael'})

MERGE (nAlice)-[:FRIEND]->(nBridget)
MERGE (nAlice)-[:FRIEND]->(nCharles)
MERGE (nMark)-[:FRIEND]->(nDoug)
MERGE (nMark)-[:FRIEND]->(nMichael);
```

Community Detection: Weakly Connected Components

Task 15:

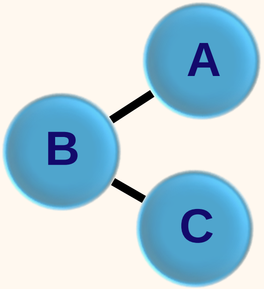
- Find the weakly connected components according to the **:FRIEND** relationship

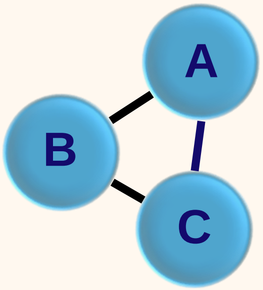
```
CALL algo.unionFind.stream('User', 'FRIEND', {})
YIELD nodeId,setId
RETURN algo.getNodeById(nodeId).id AS user, setId
```

- **Optional:** add a **:FRIEND** edge from **Michael** to **Alice** and observe the effect

Community Detection: Triangle Counting

Community Detection: Triangle Counting

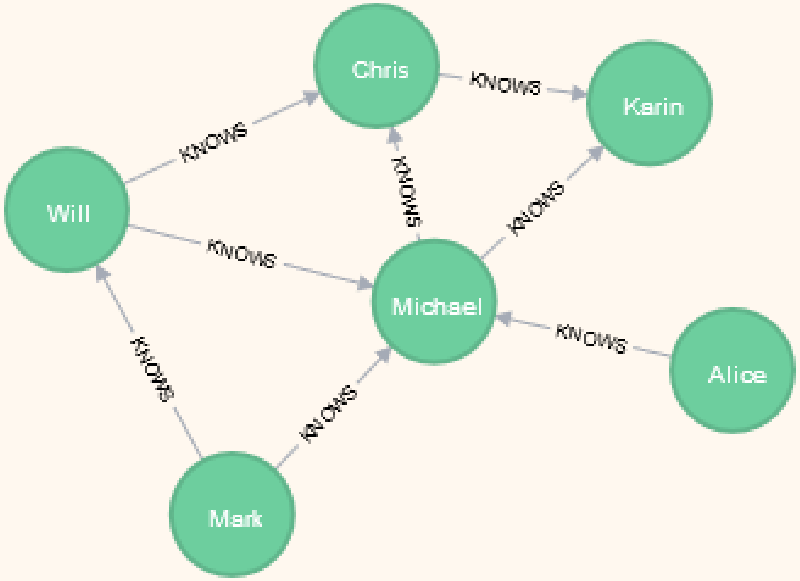
- **Triplet:** 3 connected nodes
  - **Open** triplet 

  - **Closed** triplet = triangle 

- The **older** a community, the **more** triangles are present:
  - if **A** and **B** are friends and **B** and **C** are friends
  - high probability that **A** and **C** become friends, too

Community Detection: Triangle Counting

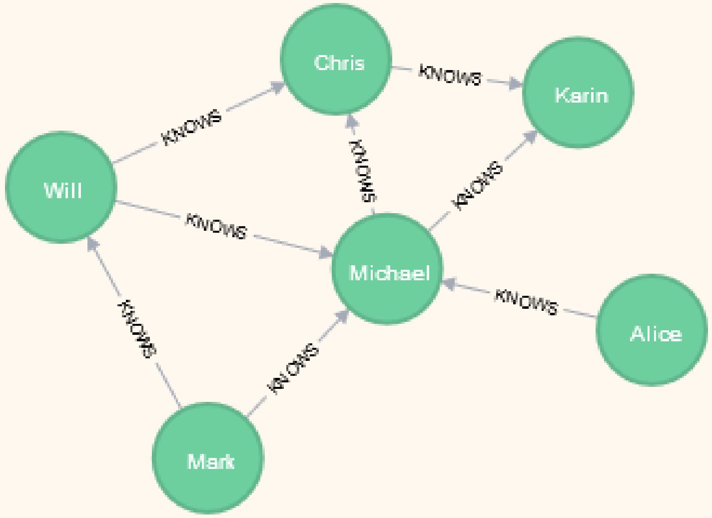
- **Example:**
  - **Michael** participates in 3 triangles
  - **Alice** participates in no triangle



Community Detection: Triangle Counting

Task 16:

- Create a new graph with 6 persons and a **:KNOWS** relationship (see https://neo4j.com/docs/graph-algorithms/current/ ➡ The Triangle Counting / Clustering Coefficient algorithm)



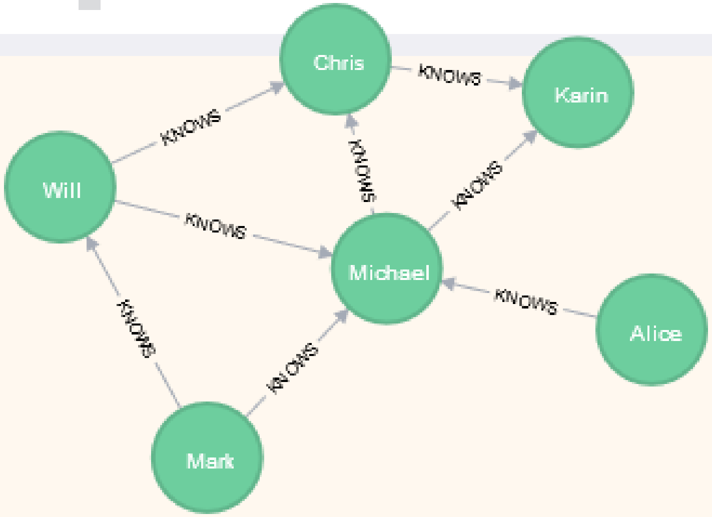
Community Detection: Triangle Counting

Task 17:

- Return all triangles in the graph

```
CALL algo.triangle.stream('Person','KNOWS')
YIELD nodeA,nodeB,nodeC

RETURN algo.getNodeById(nodeA).id AS nodeA, algo.getNodeById(nodeB).id AS nodeB,
algo.getNodeById(nodeC).id AS nodeC
```



Community Detection: Triangle Counting

The local clustering coefficient of a node determines how well connected the node's neighbors are

Local clustering coefficient:

- For a node **v**, count the edges among its neighbors
- Divide by the amount of edges in a complete graph among its neigbors

- **Example:** a person in a social network who is good at connecting his/her friends has a high coefficient

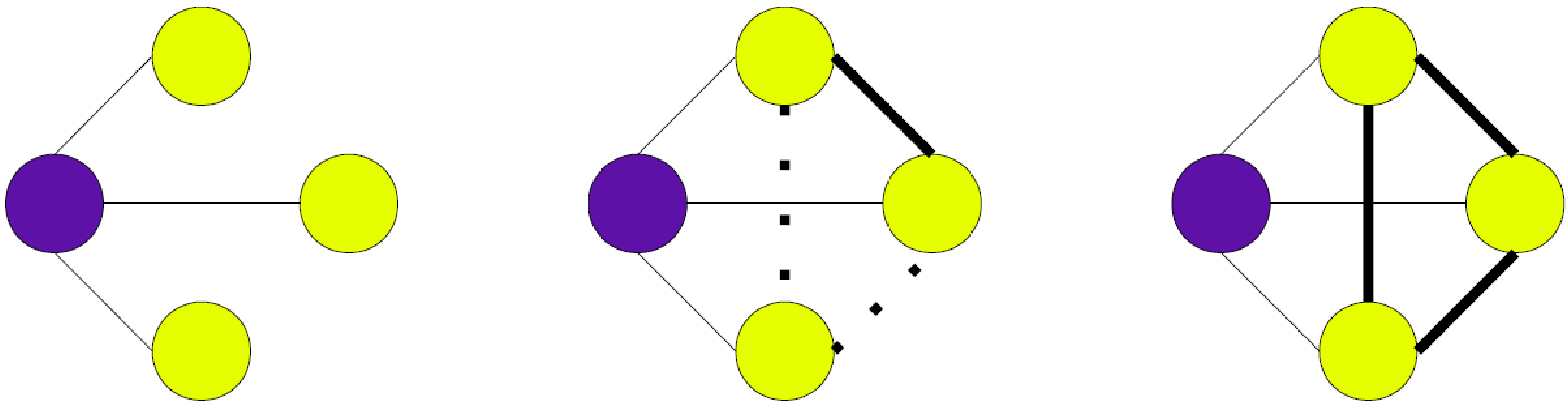
Community Detection: Triangle Counting

The local clustering coefficient of a node determines how well connected the node‘s neighbors are

Local clustering coefficient for undirected graph:

- Recall our quiz from the beginning (edge count in complete undirected graph)
- If a node **v** has **k** neighbors, the **complete** graph among the neighbors has $c=(k \cdot (k-1))/2$ edges
- Let the **actual** edge count among the neighbors be **r**
- **LCC**(**v**) = $^{r}/_{c}$

Community Detection: Triangle Counting



(a) No pairs formed among neighbors: $C = 0$

(b) One pair formed among neighbors: $C = 1 / 3$

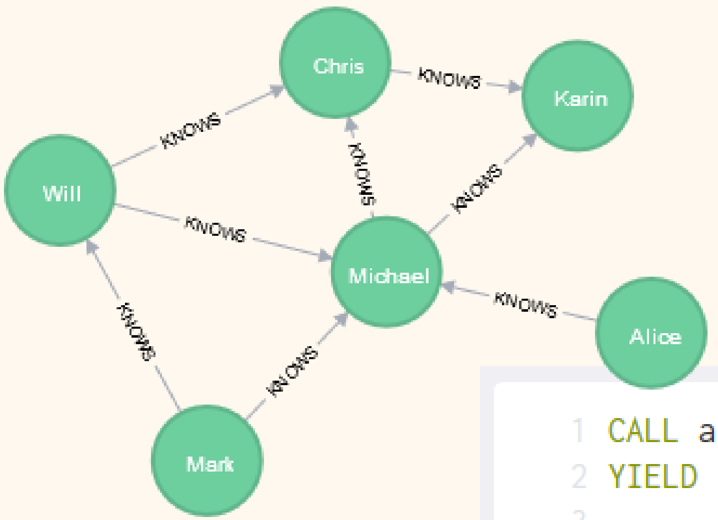
(c) Three pairs formed among neighbors: $C = 3 / 3$

Picture source: Luis Casillas Santillán/Alonso Castillo Pérez
http://www.revistascientificas.udg.mx/index.php/REC/article/viewFile/5091/4754/16111

Community Detection: Triangle Counting

Task 18:

- Compute the amount of triangles and the local clustering coefficient for each node




```
CALL algo.triangleCount.stream('Person', 'KNOWS')
YIELD nodeId, triangles, coefficient

RETURN algo.getNodeById(nodeId).id AS name, triangles, coefficient
ORDER BY coefficient DESC
```

- Which person has the best-connected friends in the graph?

Agenda

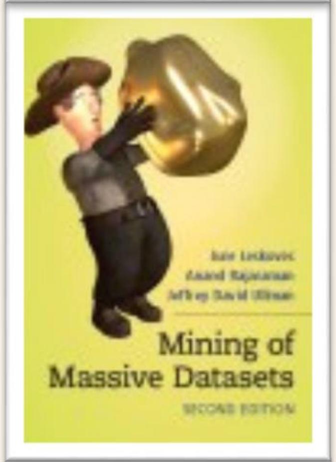
- Short CV of speaker
- Graph Theory Basics
- The Neo4J Database
- Graph Algorithms
  - Centralities
  - Path Finding
  - Community Detection

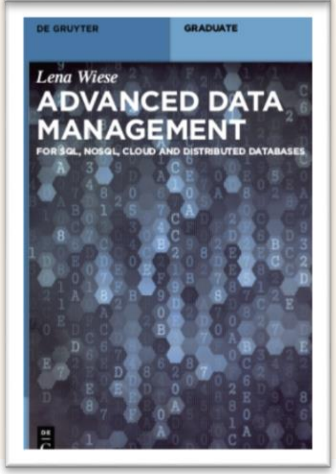
References

1. Mark Needham & Amy E. Hodler: A Comprehensive Guide to Graph Algorithms in Neo4j. Neo4j.com, 2018.

2. Anand Rajaraman, Jure Leskovec & Jeffrey D. Ullman: Mining of Massive Datasets. Mmds.org, 2014.

3. Lena Wiese: Advanced Data Management for SQL, NoSQL, Cloud and Distributed Databases. De Gruyter Graduate, 2015.





Extra exercises

„In this guide we'll learn how to use the Neo4j Graph Algorithms package using a Game of Thrones dataset."

```
$ :play https://guides.neo4j.com/sandbox/graph-algorithms/
```