Graph Algorithms for Engineering

Session 4

🧮 📐 🚀

Joseph Azar

CP58 - UTBM Sevenans

Quick Recap

What we know so far:

- ✅ Graph modeling (nodes, relationships)
- ✅ Cypher queries (CREATE, MATCH, traversals)
- ✅ Advanced patterns (properties, time-based, hierarchies)

Today: Use ALGORITHMS to solve real engineering problems!

The Story

A Crisis at LogisTech

🚛 ⚠️ 🔥

Meet LogisTech Industries

Company: LogisTech Industries

Business: Supply chain for automotive parts across Europe

Crisis: A major supplier just went bankrupt!

Critical Questions:

- Which factories are affected?
- What's the fastest way to reroute shipments?
- Which suppliers are most critical to our operations?
- Can we find alternative supply chains?

Why They Need Graph Algorithms

Manual analysis would take weeks—they need answers NOW!

Graph algorithms can:

- Find optimal routes in seconds
- Identify critical nodes (suppliers, warehouses)
- Detect communities (clustered operations)
- Calculate impact of failures

Let's learn these algorithms!

Algorithm 1

Shortest Path (Dijkstra)

🛣️ 📍 ⚡

Shortest Path: What Is It?

Find the path with MINIMUM total cost between two points

Real-world applications:

- **Logistics:** Fastest delivery route
- **Manufacturing:** Shortest material flow path
- **Robotics:** Optimal robot navigation
- **Network design:** Minimum cable length

Cost can be: Distance, time, money, energy, risk...

How Dijkstra's Algorithm Works

Simple explanation:

1. Start at source node
2. Visit nearest unvisited neighbor
3. Calculate total cost to reach it
4. Keep track of best path found so far
5. Repeat until destination is reached

Always finds the optimal path!

🛠️ Hands-On: Build Logistics Network

Scenario: Build a distribution network with delivery times

```
CREATE (paris:City {name: "Paris"})
CREATE (lyon:City {name: "Lyon"})
CREATE (marseille:City {name: "Marseille"})
CREATE (toulouse:City {name: "Toulouse"})
CREATE (bordeaux:City {name: "Bordeaux"})
CREATE (strasbourg:City {name: "Strasbourg"})

CREATE (paris)-[:ROAD {distance: 465, time_hours: 4.5}]->(lyon)
CREATE (lyon)-[:ROAD {distance: 315, time_hours: 3.0}]->(marseille)
CREATE (paris)-[:ROAD {distance: 679, time_hours: 6.5}]->(toulouse)
CREATE (toulouse)-[:ROAD {distance: 245, time_hours: 2.5}]->(bordeaux)
CREATE (bordeaux)-[:ROAD {distance: 584, time_hours: 5.5}]->(paris)
CREATE (paris)-[:ROAD {distance: 489, time_hours: 4.8}]->(strasbourg)
CREATE (lyon)-[:ROAD {distance: 505, time_hours: 5.0}]->(toulouse)
```

Find Shortest Path: Simple Method

Question: What's the shortest route from Paris to Marseille?

```
MATCH path = shortestPath(
  (start:City {name: "Paris"})
  -[:ROAD*]->
  (end:City {name: "Marseille"})
)
RETURN nodes(path) as cities,
       reduce(dist = 0, rel in relationships(path) |
              dist + rel.distance) as total_distance
```

Result: Paris → Lyon → Marseille (780 km)

🔍 Breaking Down: shortestPath()

Let's understand how shortestPath works with reduce()

```
MATCH path = shortestPath(
  (start:City {name: "Paris"})
  -[:ROAD*]->
  (end:City {name: "Marseille"})
)
RETURN nodes(path) as cities,
       reduce(dist = 0, rel in relationships(path) |
              dist + rel.distance) as total_distance
```

Step 1: Find Shortest Path

```
MATCH path = shortestPath(
  (start:City {name: "Paris"})
  -[:ROAD*]->
  (end:City {name: "Marseille"})
)
```

What this does:

- `shortestPath()` → Built-in Cypher function
- `-[:ROAD*]->` → Follow ROAD relationships, any depth
- `path =` → Save the complete path
- Finds path with FEWEST hops (not necessarily shortest distance!)

Found path: Paris → Lyon → Marseille (2 hops)

Step 2: Calculate Total Distance

```
RETURN nodes(path) as cities,
    reduce(dist = 0, rel in relationships(path) |
       dist + rel.distance) as total_distance
```

Breaking it down:

- `nodes(path)` → Extract cities from path
- `reduce(dist = 0, ...)` → Start with distance 0
- `rel in relationships(path)` → Loop through road segments
- `dist + rel.distance` → Add each segment's distance

Calculation:

```
Start: dist = 0
+ Paris→Lyon: 465 km = 465
+ Lyon→Marseille: 315 km = 780 km total
```

⚠️ Important: Fewest Hops ≠ Shortest Distance

shortestPath() finds path with FEWEST relationships, NOT minimum total distance!

Example:

- **Path A:** Paris → Lyon → Marseille (2 hops, 780 km)
- **Path B:** Paris → Bordeaux → Toulouse → Lyon → Marseille (4 hops, 650 km)

`shortestPath()` chooses Path A (fewer hops), even though Path B is shorter distance!

Solution: Use reduce() to calculate actual distance, then sort manually (next slide!)

Shortest Path by TIME (not distance)

Challenge: Find fastest route, not shortest!

```
MATCH path = shortestPath(
  (start:City {name: "Paris"})
  -[:ROAD*]->
  (end:City {name: "Marseille"})
)
WITH path,
     reduce(time = 0, rel in relationships(path) |
            time + rel.time_hours) as total_time
RETURN nodes(path) as route,
       total_time
ORDER BY total_time
LIMIT 1
```

✅ Might find a longer route that's faster!

Find ALL Possible Paths

Use case: Find backup routes in case of road closure

```
MATCH path = (start:City {name: "Paris"})
             -[:ROAD*1..4]->
             (end:City {name: "Marseille"})
WITH path,
     reduce(dist = 0, rel in relationships(path) |
            dist + rel.distance) as total_distance
RETURN nodes(path) as route,
       total_distance
ORDER BY total_distance
LIMIT 5
```

✅ Shows top 5 routes sorted by distance!

Algorithm 2

Impact Analysis (Connected Components)

💥 📊 🔍

Impact Analysis: What Is It?

Find everything AFFECTED when a node fails

Critical business questions:

- "If Supplier X fails, which factories stop?"
- "If this part is recalled, which products are affected?"
- "If this server goes down, which services fail?"
- "If this road closes, which deliveries are delayed?"

Also called: Downstream impact, cascade analysis, dependency analysis

🛠️ Hands-On: Build Supply Chain

Scenario: Multi-tier supply chain

```
CREATE (raw:Supplier {name: "RawMaterials Co", tier: 3})
CREATE (parts1:Supplier {name: "PartsMaker GmbH", tier: 2})
CREATE (parts2:Supplier {name: "Components Ltd", tier: 2})
CREATE (assembly:Supplier {name: "AssemblyPro SA", tier: 1})
CREATE (factory:Factory {name: "AutoMeca Plant"})
CREATE (product:Product {name: "Electric Motor EM-1000"})

CREATE (raw)-[:SUPPLIES]->(parts1)
CREATE (raw)-[:SUPPLIES]->(parts2)
CREATE (parts1)-[:SUPPLIES]->(assembly)
CREATE (parts2)-[:SUPPLIES]->(assembly)
CREATE (assembly)-[:SUPPLIES]->(factory)
CREATE (factory)-[:PRODUCES]->(product)
```

✅ A realistic multi-tier supply chain!

Find Downstream Impact

Crisis: RawMaterials Co just failed! What's affected?

```
MATCH (failed:Supplier {name: "RawMaterials Co"})
      -[:SUPPLIES*]->(affected)
RETURN DISTINCT labels(affected)[0] as entity_type,
                affected.name as affected_entity
ORDER BY entity_type
```

Result shows:

- 2 Tier-2 suppliers (PartsMaker, Components)
- 1 Tier-1 supplier (AssemblyPro)
- 1 Factory
- 1 Product

Complete production stops! 🚨

Quantify the Impact

```
MATCH (s:Supplier)
OPTIONAL MATCH (s)-[:SUPPLIES*]->(affected)
WITH s,
     COUNT(DISTINCT affected) as impact_count
RETURN s.name as supplier,
       s.tier as tier,
       impact_count
ORDER BY impact_count DESC
```

Shows: Which suppliers are most critical!

Higher tier suppliers have bigger impact

🔍 Breaking Down: OPTIONAL MATCH

This query uses OPTIONAL MATCH to handle suppliers with no downstream!

```
MATCH (s:Supplier)
OPTIONAL MATCH (s)-[:SUPPLIES*]->(affected)
WITH s,
     COUNT(DISTINCT affected) as impact_count
RETURN s.name as supplier,
       s.tier as tier,
       impact_count
ORDER BY impact_count DESC
```

Step 1: MATCH vs OPTIONAL MATCH

```
MATCH (s:Supplier)
OPTIONAL MATCH (s)-[:SUPPLIES*]->(affected)
```

Regular MATCH:

- Returns ONLY suppliers that have downstream
- Excludes end-of-chain suppliers
- Like SQL INNER JOIN

OPTIONAL MATCH:

- Returns ALL suppliers
- If no downstream, `affected = null`
- Like SQL LEFT JOIN

Use OPTIONAL MATCH when: Some nodes might not have the pattern, but you still want them in results

Step 2: COUNT(DISTINCT affected)

```
WITH s,
  COUNT(DISTINCT affected) as impact_count
```

Breaking it down:

- `COUNT(affected)` → Count how many downstream nodes
- `DISTINCT` → Don't count same node twice
- If `affected = null` (no downstream), COUNT = 0

Example results:

Supplier	Tier	Impact Count
RawMaterials Co	3	**5**
AssemblyPro SA	1	**2**

💡 Why Use OPTIONAL MATCH Here?

Without OPTIONAL, we'd miss important suppliers!

Without OPTIONAL:

```
MATCH (s:Supplier)
MATCH (s)-[:SUPPLIES*]->(affected)
...
```

❌ Excludes Tier-1 suppliers who supply directly to factories!

With OPTIONAL:

```
MATCH (s:Supplier)
OPTIONAL MATCH (s)-[:SUPPLIES*]->(affected)
...
```

✅ Includes ALL suppliers, even those at end of chain!

Deep Dive

Understanding OPTIONAL MATCH

🔍 ⚠️ 💡

Let's Build a Simple Example

We'll create 3 suppliers with different downstream connections:

```
// Create simple supply chain
CREATE (s1:Supplier {name: "RawMat Inc", tier: 3})
CREATE (s2:Supplier {name: "PartsMaker", tier: 2})
CREATE (s3:Supplier {name: "EndSupplier", tier: 1})
CREATE (factory:Factory {name: "Plant A"})

// Create supply chain: s1 → s2 → factory
CREATE (s1)-[:SUPPLIES]->(s2)
CREATE (s2)-[:SUPPLIES]->(factory)

// Note: s3 has NO downstream connections!
```

Graph structure:

RawMat Inc (tier 3) → PartsMaker (tier 2) → Plant A
EndSupplier (tier 1) → [nobody]

Test 1: Using Regular MATCH

Let's try to count downstream for ALL suppliers using regular MATCH:

```
MATCH (s:Supplier)
MATCH (s)-[:SUPPLIES*]->(downstream)
WITH s, COUNT(DISTINCT downstream) as impact
RETURN s.name as supplier,
       s.tier as tier,
       impact
ORDER BY impact DESC
```

Result: Regular MATCH

Problem: EndSupplier is MISSING!

Supplier	Tier	Impact
RawMat Inc	3	**2**
PartsMaker	2	**1**
EndSupplier	1	❌ **NOT SHOWN**

Why? Second MATCH requires pattern to exist. EndSupplier has no downstream, so it's excluded!

Test 2: Using OPTIONAL MATCH

Now let's try with OPTIONAL MATCH:

```
MATCH (s:Supplier)
OPTIONAL MATCH (s)-[:SUPPLIES*]->(downstream)
WITH s, COUNT(DISTINCT downstream) as impact
RETURN s.name as supplier,
       s.tier as tier,
       impact
ORDER BY impact DESC
```

Notice: Only changed MATCH to OPTIONAL MATCH!

Result: OPTIONAL MATCH

Success: ALL suppliers included!

Supplier	Tier	Impact
RawMat Inc	3	**2**
PartsMaker	2	**1**
EndSupplier	1	✅ **0**

Why? OPTIONAL MATCH returns NULL when pattern doesn't match. COUNT(NULL) = 0!

Side-by-Side: The Difference

❌ Regular MATCH

```
MATCH (s:Supplier)
MATCH (s)-[:SUPPLIES*]->(d)
...
```

Acts like SQL `INNER JOIN`

Result: 2 rows

- RawMat Inc (2)
- PartsMaker (1)
- EndSupplier MISSING!

✅ OPTIONAL MATCH

```
MATCH (s:Supplier)
OPTIONAL MATCH (s)-[:SUPPLIES*]->(d)
...
```

Acts like SQL `LEFT JOIN`

Result: 3 rows

- RawMat Inc (2)
- PartsMaker (1)
- EndSupplier (0) ✓

When to Use OPTIONAL MATCH?

Use OPTIONAL MATCH when some nodes might NOT have the pattern

✅ Use OPTIONAL MATCH for:

- Counting connections (some may have 0)
- Finding all nodes even if isolated
- Calculating impact (includes zero impact)
- Left join scenarios

⚠️ Use Regular MATCH for:

- Pattern MUST exist
- Only interested in connected nodes
- Inner join scenarios
- Filtering by relationship existence

🛠️ Practice: Try Both!

> **Exercise:** Run both queries and compare results

Query 1 (Regular MATCH):

```
MATCH (s:Supplier)
MATCH (s)-[:SUPPLIES]->(customer)
RETURN s.name, COUNT(customer) as customers
ORDER BY customers DESC
```

Query 2 (OPTIONAL MATCH):

```
MATCH (s:Supplier)
OPTIONAL MATCH (s)-[:SUPPLIES]->(customer)
RETURN s.name, COUNT(customer) as customers
ORDER BY customers DESC
```

✅ See how EndSupplier appears only in Query 2!

🔑 Key Takeaway

OPTIONAL MATCH = "Try to find this pattern, but don't fail if it doesn't exist"

Think of it like this:

- **MATCH:** "Give me all students WHO have grades"
- **OPTIONAL MATCH:** "Give me all students, AND their grades IF they have any"

In SQL terms: OPTIONAL MATCH ≈ LEFT OUTER JOIN

Find Upstream Dependencies

Question: What does the factory depend on?

```
MATCH (factory:Factory {name: "AutoMeca Plant"})
      <-[:SUPPLIES*]-(dependency)
RETURN DISTINCT labels(dependency)[0] as type,
                dependency.name as name,
                dependency.tier as tier
ORDER BY tier DESC
```

Shows: Complete dependency tree!

All suppliers the factory relies on, at all tiers

Algorithm 3

Centrality Analysis

⭐ 🎯 📈

Centrality: What Is It?

Measure how IMPORTANT a node is in the network

Degree Centrality

How many connections?

High → Hub node

Betweenness Centrality

How many paths pass through?

High → Bottleneck

Use cases: Find critical suppliers, bottleneck processes, key influencers

Degree Centrality (Simple)

Question: Which suppliers serve the most customers?

```
MATCH (s:Supplier)-[:SUPPLIES]->(customer)
WITH s, COUNT(customer) as connection_count
RETURN s.name as supplier,
       s.tier as tier,
       connection_count as connections
ORDER BY connection_count DESC
LIMIT 5
```

Shows: Hub suppliers with most connections

These are critical—their failure affects many!

Find Bottlenecks

> **Bottleneck:** A node that many paths pass through

```
MATCH (start:Supplier {tier: 3})
MATCH (end:Factory)
MATCH path = (start)-[:SUPPLIES*]->(middle)-[:SUPPLIES*]->(end)
WITH middle, COUNT(path) as paths_through
WHERE paths_through > 1
RETURN labels(middle)[0] as node_type,
       middle.name as bottleneck,
       paths_through
ORDER BY paths_through DESC
```

✅ Identifies single points of failure!

Algorithm 4

Community Detection

👥 🔗 🌐

Community Detection: What Is It?

Find clusters of tightly connected nodes

Engineering applications:

- **Supply chain:** Regional supplier clusters
- **Manufacturing:** Coupled subsystems
- **Organization:** Team structures
- **Products:** Modular components

Why useful? Simplify complex systems, optimize operations, reduce risk

🛠️ Hands-On: Build Collaboration Network

```
CREATE (alice:Engineer {name: "Alice", dept: "Mech"})
CREATE (bob:Engineer {name: "Bob", dept: "Mech"})
CREATE (carol:Engineer {name: "Carol", dept: "Mech"})
CREATE (david:Engineer {name: "David", dept: "Elec"})
CREATE (emma:Engineer {name: "Emma", dept: "Elec"})
CREATE (frank:Engineer {name: "Frank", dept: "Elec"})

CREATE (alice)-[:COLLABORATES]->(bob)
CREATE (bob)-[:COLLABORATES]->(carol)
CREATE (carol)-[:COLLABORATES]->(alice)
CREATE (david)-[:COLLABORATES]->(emma)
CREATE (emma)-[:COLLABORATES]->(frank)
CREATE (frank)-[:COLLABORATES]->(david)
CREATE (carol)-[:COLLABORATES]->(david)
```

✅ Two tight teams with one bridge connection!

Find Connected Groups

> **Question:** Who works closely together?

```
MATCH (e:Engineer)
OPTIONAL MATCH (e)-[:COLLABORATES*1..2]-(colleague:Engineer)
WITH e, COLLECT(DISTINCT colleague.name) as team
RETURN e.name as engineer,
       e.dept as department,
       SIZE(team) as team_size,
       team
ORDER BY team_size DESC
```

✅ Shows collaboration network for each engineer!

Find Bridge Connections

Bridge: A person connecting two otherwise separate groups

```
MATCH (e:Engineer)-[:COLLABORATES]-(colleague1:Engineer)
MATCH (e)-[:COLLABORATES]-(colleague2:Engineer)
WHERE colleague1.dept <> colleague2.dept
  AND colleague1 <> colleague2
RETURN DISTINCT e.name as bridge_person,
                e.dept as department,
                COUNT(DISTINCT colleague1) +
                COUNT(DISTINCT colleague2) as connections
ORDER BY connections DESC
```

✅ These people are critical for cross-team communication!

Putting It All Together

Complete Supply Chain Analysis

🎯 🔥 💪

🏆 Final Challenge

Crisis Simulation: Complete supply chain risk analysis

Your mission:

1. Build a realistic supply chain network
2. Find shortest delivery paths
3. Identify critical suppliers (centrality)
4. Analyze failure impact
5. Find alternative routes

Use ALL algorithms we learned!

Step 1: Build Complete Network

```
// Suppliers
CREATE (s1:Supplier {name: "Steel Inc", tier: 3, location: "Germany"})
CREATE (s2:Supplier {name: "Plastics Co", tier: 3, location: "Italy"})
CREATE (s3:Supplier {name: "Parts GmbH", tier: 2, location: "France"})
CREATE (s4:Supplier {name: "Components Ltd", tier: 2, location: "Spain"})
CREATE (s5:Supplier {name: "Assembly Pro", tier: 1, location: "France"})

// Factories
CREATE (f1:Factory {name: "Factory Paris", location: "France"})
CREATE (f2:Factory {name: "Factory Munich", location: "Germany"})

// Supply relationships with lead times
CREATE (s1)-[:SUPPLIES {leadTime: 5, cost: 1000}]->(s3)
CREATE (s1)-[:SUPPLIES {leadTime: 7, cost: 1200}]->(s4)
CREATE (s2)-[:SUPPLIES {leadTime: 4, cost: 800}]->(s3)
CREATE (s2)-[:SUPPLIES {leadTime: 6, cost: 900}]->(s4)
CREATE (s3)-[:SUPPLIES {leadTime: 3, cost: 2000}]->(s5)
CREATE (s4)-[:SUPPLIES {leadTime: 3, cost: 1800}]->(s5)
CREATE (s5)-[:SUPPLIES {leadTime: 2, cost: 5000}]->(f1)
CREATE (s5)-[:SUPPLIES {leadTime: 4, cost: 5500}]->(f2)
```

Analysis 1: Fastest Supply Route

Question: What's the fastest path from Steel Inc to Factory Paris?

```
MATCH path = shortestPath(
  (start:Supplier {name: "Steel Inc"})
  -[:SUPPLIES*]->
  (end:Factory {name: "Factory Paris"})
)
WITH path,
     reduce(time = 0, rel in relationships(path) |
            time + rel.leadTime) as total_time
RETURN [n in nodes(path) | n.name] as route,
       total_time as days
ORDER BY total_time
LIMIT 1
```

Result: Steel Inc → Parts GmbH → Assembly Pro → Factory Paris (10 days)

Analysis 2: Find Critical Suppliers

Question: Which suppliers have highest impact?

```
MATCH (s:Supplier)
OPTIONAL MATCH (s)-[:SUPPLIES*]->(downstream)
WITH s,
     COUNT(DISTINCT downstream) as impact,
     s.tier as tier
RETURN s.name as supplier,
       tier,
       s.location as location,
       impact as downstream_entities
ORDER BY impact DESC, tier DESC
```

✅ Shows which suppliers affect the most downstream entities!

Analysis 3: Simulate Supplier Failure

Simulation: What if Steel Inc fails?

```
MATCH (failed:Supplier {name: "Steel Inc"})
      -[:SUPPLIES*]->(affected)
WITH DISTINCT labels(affected)[0] as entity_type,
              affected.name as name,
              affected.location as location
RETURN entity_type,
       name,
       location
ORDER BY entity_type, name
```

Impact Assessment:

- 2 Tier-2 suppliers affected
- 1 Tier-1 supplier affected
- 2 Factories at risk

Analysis 4: Find Alternative Routes

> **Question:** If Parts GmbH fails, what are backup routes?

```
MATCH path = (start:Supplier {tier: 3})
             -[:SUPPLIES*]->
             (end:Factory {name: "Factory Paris"})
WHERE NONE(node in nodes(path)
           WHERE node.name = "Parts GmbH")
WITH path,
     reduce(time = 0, rel in relationships(path) |
            time + rel.leadTime) as total_time
RETURN [n in nodes(path) | n.name] as alternative_route,
       total_time as days
ORDER BY total_time
LIMIT 3
```

✅ Shows backup routes avoiding the failed supplier!

Analysis 5: Cost vs Time Trade-off

Question: Compare routes by cost AND time

```
MATCH path = (start:Supplier {name: "Plastics Co"})
             -[:SUPPLIES*]->
             (end:Factory {name: "Factory Munich"})
WITH path,
     reduce(time = 0, rel in relationships(path) |
            time + rel.leadTime) as total_time,
     reduce(cost = 0, rel in relationships(path) |
            cost + rel.cost) as total_cost
RETURN [n in nodes(path) | n.name] as route,
       total_time as days,
       total_cost as euros,
       round(total_cost / total_time) as cost_per_day
ORDER BY total_time, total_cost
LIMIT 5
```

✅ Helps make informed business decisions!

Real-World Success Story

Company: Large automotive manufacturer

Challenge: COVID-19 chip shortage (2021)

Solution: Graph algorithms for supply chain analysis

Results:

- Identified alternative suppliers in 2 hours (vs 2 weeks)
- Found optimal rerouting saving €2M
- Prevented 3 factory shutdowns

Graph algorithms saved their business!

Summary: 4 Key Algorithms

1. Shortest Path (Dijkstra): Find optimal routes

Use: Logistics, navigation, optimization

2. Impact Analysis: Find downstream effects

Use: Risk assessment, failure analysis

3. Centrality: Identify critical nodes

Use: Find hubs, bottlenecks, key players

4. Community Detection: Find clusters

Use: Simplify systems, optimize organization

Key Takeaways

✅ **Algorithms turn graphs into INSIGHTS!**

Remember:

- Use **shortest path** for optimization
- Use **impact analysis** for risk
- Use **centrality** for importance
- Use **community detection** for structure

Pro tip: Combine multiple algorithms for deeper insights!

Quick Decision Guide

I need to...

- Find best route → **Shortest Path**
- Assess failure risk → **Impact Analysis**
- Identify critical nodes → **Centrality**
- Group related items → **Community Detection**
- Find all dependencies → **Traversal (*)**
- Find alternative paths → **Multiple Shortest Paths**
- Detect bottlenecks → **Betweenness Centrality**
- Find connection strength → **Degree Centrality**

Advanced Topics (Beyond This Course)

Graph Data Science Library (Neo4j GDS)

- PageRank (Google's algorithm)
- Louvain (community detection)
- Node similarity
- Link prediction
- Graph embeddings (ML)

These require additional installation but are VERY powerful!

Cypher Algorithm Cheat Sheet

Shortest Path:

```
MATCH path = shortestPath((a)-[:REL*]->(b))
RETURN path
```

Impact (downstream):

```
MATCH (n)-[:REL*]->(affected) RETURN affected
```

Dependencies (upstream):

```
MATCH (n)<-[:REL*]-(dependency) RETURN dependency
```

Degree Centrality:

```
MATCH (n)-[r]->() RETURN n, COUNT(r) as degree
```

All Paths:

```
MATCH path = (a)-[:REL*1..5]->(b) RETURN path
```

Practice Exercises

Try these on your own:

1. Build your own supply chain and find critical paths
2. Model your university course prerequisites
3. Create a project dependency graph
4. Map your team's collaboration network
5. Design a manufacturing process flow

Goal: Apply algorithms to YOUR domain!

Learning Resources

Free Resources:

- **Neo4j GraphAcademy:** Free courses on algorithms
- **Neo4j Sandbox:** Try algorithms online
- **Graph Data Science Playground:** Interactive tutorials

Documentation:

neo4j.com/docs/cypher-manual → Path Functions

neo4j.com/docs/graph-data-science → Algorithms

Questions?

❓ 💬 🙋

Thank you!

joseph.azar@utbm.fr

Now go analyze some graphs! 🚀