The aim of the fourth project is to gain an understanding of the additional possibilities and conveniences that graph databases may offer.
The deadline of the project is **Friday May 17, 2024**. Submission will take place using the INGInious interface. Your submission will need to be individual.

Instructions

In this project, you are given a subset of the relations in the `mondial` database that was used in the first project, consisting only of countries, continents, and their links to each other. Using these links, a graph can be constructed in which country nodes are connected to each other if there is a border between them, and in which there is an edge between a country and a continent if the country is located in the continent.
You will need to create this graph in a Neo4J database, and subsequently run a number of Cypher queries on it.
To prepare your database, you need to follow these steps.

1. Go to the following website: `https://neo4j.com/deployment-center/` and download the **community server** version of the 'Graph Database Self-Managed' for your operating system. In the instructions below, we assume you have downloaded a `.tar` version of Neo4J 5.19.0 for either Linux or Mac.

2. Extract Neo4J in a folder of your choice. In the instructions below, we assume that this is the folder `~/neo4j`; if you store this folder in a different location, please use this alternative location in the instructions below.

3. Within this folder, you will find a folder called `bin`, within which an executable `neo4j` is contained. Open a console and go to the folder containing Neo4J using the command `cd ~/neo4j/bin/`. Start the executable using the command `./neo4j start`. If `neo4j` started successfully, you will see a line such as this in the terminal:

   ```
   Started neo4j (pid 6834). It is available at http://localhost:7474/
   ```

   Copy and paste the given URL (most likely `http://localhost:7474/`) in your webbrowser. If you receive an error, make sure that you have the correct version of Java installed, and that the server is not running already.

4. You will be presented a login screen. Use the string `neo4j` both as username and as password. At this moment, you should be presented with an interface in which you can enter Cypher queries.

5. To import the `mondial` database, download `mondial-csv.zip` from Moodle, and extract it to a location of your choice. This zip file contains `.csv` files that we will import into Neo4J.

6. In order to import these files into Neo4J, copy or move all `.csv` files into the `import` folder of Neo4J. If you installed Neo4J in the folder `~/neo4j`, this is the folder `~/neo4j/import`.

7. Execute the following Cypher queries one after the other to create the graph:

```
LOAD CSV WITH HEADERS FROM "file:/country.csv" AS row
CREATE (:Country { code: row.code, name: row.name });

LOAD CSV WITH HEADERS FROM "file:/borders.csv" AS row
MATCH (c1:Country { code: row.country1 }), (c2:Country { code: row.country2 })
CREATE (c1)-[:Borders]->(c2);

LOAD CSV WITH HEADERS FROM "file:/continent.csv" AS row
CREATE (:Continent { name: row.name });

LOAD CSV WITH HEADERS FROM "file:/encompasses.csv" AS row
MATCH (cn:Continent { name: row.continent} ), (cy:Country { code: row.country })
CREATE (cn)-[:Encompasses { percentage: row.percentage }]->(cy);
```

Consider running the following query after each step to see how the graph is constructed:

```
MATCH (p) RETURN p;
```

Once you have created this graph, you are required to write Cypher queries for the tasks described below.
Note that for some of these queries, you will require notation that was **not** discussed during the lectures! You will find some tips & tricks in the Appendix of this exercise. For more information on Cypher queries you may also consult the online documentation.
Finally, your answer to the queries below should also be correct for *other* versions of the world map, i.e., worlds in which the border relations would be different than in our world; it is hence not correct to write queries that would just return a static list of country names or nodes.

1. List the full names of countries that border Greece and are not 100% located in Europe.
2. List the pairs of countries that neighbor China, and that are also neighbors of each other. Provide the full names of both countries; only list the pairs in which the first name of the pair is strictly lower than the second name in the pair.
3. Provide the full names of all countries that are 100% located in Europe, but that border a country located (not, or not entirely located) in Europe.
4. Provide the full names of all countries in Asia that can be reached in at most two steps from Turkey, excluding Turkey itself. Make sure to include every country in the output only once.
5. List the number of neighboring countries for every country fully located in Europe (i.e., this is the *degree* of every country in Europe, when only considering the Borders relation). Provide for each country the full name, and its degree. Sort the output in decreasing order of degree.
6. Find the country with the largest number of neighbors; if there are multiple such countries, it is allowed to break ties arbitrarily; provide the full name and the number of neighbors.
7. Determine the shortest path from Belgium to China; provide the complete path, that is, all nodes and edges on this path, starting from Belgium and ending in China.

8. Find the country which has the longest shortest path to Belgium; provide the full name of this country.
9. Luxembourg is in a specific topographic situation: it has exactly three neighboring countries (Belgium, Germany, and France), each of which are pairwise neighbors of each other as well. As a result, Luxembourg is surrounded by exactly three countries. List the full names of all countries that are in a similar situation as Luxembourg.
10. List all countries in Europe for which there is no path of length in between 1 and 3 to a country (partially) located in Asia.

Tips & Tricks

1. If you wish to eliminate distinct nodes/tuples from the output of a Cypher query, Cypher also supports the keyword `DISTINCT`:

   ```
   MATCH (p) RETURN DISTINCT p.name;
   ```

2. If you wish to order the results of a query, the `ORDER BY` notation can be used:

   ```
   MATCH (p) RETURN p.name ORDER BY p.name DESC;
   ```

3. If you wish to limit the number of tuples in the output of a query, the `LIMIT` notation can be used:

   ```
   MATCH (p) RETURN p.name ORDER BY p.name DESC LIMIT 3;
   ```

4. To calculate the degree of a node, a `WITH` statement can be used:

   ```
   MATCH (c:Country {name:"Belgium"})
   WITH c, size((c)-[:Borders]->()) AS degree
   RETURN c.name, degree;
   ```

   In this notation, the output of the `MATCH` statement is processed by the `WITH` statement, yielding two new variables `c` and `degree`, which are used to return the final result. The query would not be correct if the "`c, `" would be removed from the `WITH` statement! After the `WITH` only the variables may be used that are mentioned within the `WITH`. The `size()` function calculates the number of results for the query passed to it in its parameter.

5. To calculate the length of a path, the `length()` function can be applied on a path. This function can also be used in a `WITH`.

6. Note that a Cypher query may include multiple `WHERE`s. After a `WITH` statement has created new variables, the `WHERE` can be applied again to constrain the results further:

   ```
   MATCH (c:Country)
   WHERE c.name <> "Russia"
   WITH c, size((c)-[:Borders]->()) AS degree
   WHERE degree > 10
   RETURN c.name, degree;
   ```