

Graph Database Pokédex
Isaac Woods
B827729
Loughborough University

Contents

| | |
|---|-----------|
| Introduction and Technical Background..... | 3 |
| Design..... | 3 |
| Implementation..... | 4 |
| Setup..... | 4 |
| Importing Data..... | 5 |
| <i>Pokémon main CSV.....</i> | <i>5</i> |
| <i>Types CSV.....</i> | <i>6</i> |
| <i>Evolution families data.....</i> | <i>7</i> |
| Formatting and cleaning data..... | 9 |
| <i>Pokémon main CSV.....</i> | <i>9</i> |
| <i>Evolution families data.....</i> | <i>10</i> |
| Relationships..... | 11 |
| <i>Pokémon Types.....</i> | <i>11</i> |
| <i>Evolutionary Families.....</i> | <i>11</i> |
| Graphical User Interface..... | 12 |
| <i>Simple Search.....</i> | <i>14</i> |
| <i>Normal Pokédex.....</i> | <i>15</i> |
| <i>Result Formatting.....</i> | <i>18</i> |
| <i>Team Builder.....</i> | <i>20</i> |
| Testing..... | 22 |
| <i>Simple Search.....</i> | <i>22</i> |
| <i>Normal Pokédex.....</i> | <i>22</i> |
| <i>Cypher Pokédex.....</i> | <i>23</i> |
| <i>Team Builder.....</i> | <i>23</i> |
| Evaluation..... | 23 |
| <i>Future work.....</i> | <i>24</i> |
| <i>Hindsight.....</i> | <i>24</i> |
| Bibliography..... | 26 |

Introduction and Technical Background

What my project is about, the problem I am trying to solve and key knowledge and definitions.

This project is a great opportunity for me to combine my love of Pokémon and my keen interest in graph databases. It seemed like a unique way of exploring both the technical abilities of myself, and that of graph databases.

There is a lot of data about Pokémon on the internet that is hard to navigate, search and gain useful information from for the ordinary person. For people that may be playing the Pokémon games, it can be frustrating to try and find the information they need in order to further their progression. This is because it is hard to know what is relevant to the specific game they are playing, their current progress and their location.

A solution to this could be to collect the vast data and store it in a Graph Database structure. This structure is specifically suited to the situation because it prioritises relations between 'nodes' of data. With this, we can easier see relevant data as there is a strong relation to follow (for example all Pokémon from the same game would have a link to each other and so can be easily returned).

Therefore the goal is to create some software that provides easy access to the user to the information they need. The software will allow them to search the data efficiently, and provide extra features like a team builder.

As well as this, I hope to teach the user how to use Cypher Query Language, which is used to query the database. This will be done by having two main sections to search for information: One that is for simple natural language input, where the user's input will be translated into a Cypher query and executed, and one that is just for Cypher language input. Within the natural language section, the user's input will be translated to Cypher and will be displayed on the screen before being executed. This will hopefully allow them to learn the query language, so that they can then query for more specific data in the Cypher section of the software.

As mentioned, Graph Databases are a type of database that represents data entries as nodes, and relationships as arcs between nodes. This means that it prioritises relationships between data points, and allows for easy traversal of the network that is formed. This is suited to Pokémon data, as there are a lot of data connections and relationships to be formed. For example, a Pokémon can evolve into another, which can be represented by the two Pokémon's nodes being linked by an "evolves into" arc. Therefore I will be able to create a large network of nodes and arcs with the vast Pokémon data.

This project will allow me to explore just how beneficial this is as a way of representing data, and if this software would of actually benefitted from a different approach.

Design

Preliminary decisions I made on the structure or process of my project.

From my research into graph databases, I found a company called Neo4j ([1] Graph Database Platform | Graph Database Management System | Neo4j, 2021) that seem to be pioneering the graph database software. From here, I found that they have created a library for multiple coding languages that can be used to create graph databases ([2] Using Neo4j embedded in Java applications - Neo4j Java Reference, 2021). After looking into the supported languages, I decided to code my

project in Java, as I have a lot of experience using it and I think it will allow me the most flexibility.

I also downloaded Neo4j's desktop software that is used to view, access and query graph databases that are being hosted locally. This will later allow me to view my database as I add data, and to practice my own Cypher Query Language skills.

To get the data to populate the database, I will be using CSV files from the internet. During my research, I found a large CSV file containing all Pokémon and their data ([3] Armand, 2021).

However, I later found that this was missing the most recent generation of Pokémon, and so was incomplete. Therefore, I searched and found a more conclusive set of data to use ([4] Banik, 2021).

Please see the bibliography section for links to both of these datasets.

This change of data is later talked about in the Implementation section.

For the GUI, I decided that I wanted all of the fonts to represent the same style as the font used in other Pokémon media and content. Therefore, I did some research and found a good font to use that looks similar to the font used in other Pokémon content ([5] Sagas, 2021).

This font has a playful feel and will be nice for the user to see. As well as this, I decided to match the colour scheme to the colours commonly seen in Pokémon games and shows and used a simple red and white design.

For the structure of the graph database, I had to make a decision as to how I wanted some of the relationships to form. I decided to have each Pokémon type be represented by a node, and then have the type relationships be formed between these nodes. This was to reduce the overall number of relationships and hopefully improve the querying speeds across the database. This is explained in more detail in the Importing Data subsection within my Implementation section of this report.

Implementation

A step by step guide to how I implemented my software, and any problems I encountered.

Setup

To code this project, I decided to use IntelliJ IDE with Java as it is a familiar environment for me to use. To be able to use the Neo4j library, I had to add Neo4j as a dependency within my maven pom.xml file. This was new to me, so it took some trial and error to get this right, and to understand what it did. In the end my pom.xml file looked like the following:

```

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <source>8</source>
        <target>8</target>
      </configuration>
    </plugin>
  </plugins>
</build>

<dependencies>
  <dependency>
    <groupId>org.neo4j</groupId>
    <artifactId>neo4j</artifactId>
    <version>4.2.0</version>
  </dependency>
  <dependency>
    <groupId>org.neo4j</groupId>
    <artifactId>neo4j-bolt</artifactId>
    <version>4.2.0</version>
  </dependency>
</dependencies>

```

Figure 1. pom.xml

With this added, I could now import the Neo4j library to my IDE and begin learning how to build a database.

The Neo4j website includes a “Hello World” example that can be used to learn the basics of graph databases ([2] Using Neo4j embedded in Java applications - Neo4j Java Reference, 2021). I used this to learn how to create and access a database, then create nodes and relationships. At the same time, I learned that I could view this database using the Neo4j desktop software (that I mentioned in the Design section) and began teaching myself how to use Cypher query language to explore this new database.

Importing Data

Pokémon main CSV

With the basic structure of a graph database already built, I needed data to populate it. Therefore, I did some research and found a large CSV file ([3] Armand, 2021) of every Pokémon and their information, like what type they are and how much HP they have.

With the data sourced, I began writing code that can dynamically create a node for each Pokémon in the dataset. This took some trial and error to make sure the data was transferred properly and without any missing details, as each node had multiple properties that define the Pokémon. In the end, this was the code that worked nicely:

```

//importing the CSV file and creating a node for each pokemon
try{
  File CSV = new File( pathname: "src\\main\\resources\\pokemon.csv");
  Scanner myReader = new Scanner(CSV);

  String[] headers = new String[19];
  while (myReader.hasNextLine()){
    String lineRaw = myReader.nextLine();
    String[] line = lineRaw.split( regex: "," );
    //writes the first line to an array to use the headers later
    if (line[1].equals("name")) {
      headers = line;
    } else{
      try (Transaction tx = graphDb.beginTx()){
        Node node = tx.createNode(Label("Pokemon"));
        for (int i=0;i<19;i++){
          node.setProperty(headers[i], line[i]);
        }
        tx.commit();
      }
    }
  }
  myReader.close();
}

```

Figure 2. Importing Pokémon CSV file

To make sure the code doesn't crash, the file reading had to be done within a try and catch segment. This is so that if the file is missing, an appropriate error message is displayed. When the program is compiled, this will be irrelevant.

As you can see, the program takes the headings of the CSV file and uses them as the property labels, then for each line it cycles through each heading and sets the property accordingly, then moves onto the next Pokémon.

After this, I soon realised that the dataset I was using did not include the latest generation of Pokémon, and therefore was not up to date. Therefore, I did some more research and found a more conclusive dataset ([4] Banik, 2021), which not only included the latest generation but also had more information for each Pokémon.

I simply swapped out this dataset and changed a few lines of code to allow for its different structure and extra headings.

Types CSV

I needed a way to represent the ways the different types of Pokémon interact with each other. For example, Fire Pokémon are weak to water Pokémon, and water Pokémon are weak to grass Pokémon, and so on. Therefore I needed some data that describes this, so I can add these interactions as relationships (shown in the Relationships section ahead). Therefore, I once again did some research and found another CSV file ([6] zonination/pokemon-chart, 2021) that represents the interactions between types in a matrix. The link to this dataset can be found in the bibliography section.

Using this data, I could now find what each Pokémon is weak to, weak against, strong to and strong against. As mentioned in the Design section, I had to make a decision as to how I wanted these type relationships to be formed in the database. There were two options, have each Pokémon point to every other Pokémon that it has a type relationship to, or have each Pokémon point to a node that represents their type, then have these nodes represent the type interactions within the matrix.

For example, the first option would have every fire type Pokémon have a relationship to every water type Pokémon saying "weak to". And the second option would have every fire type Pokémon point to a node called "Fire" and have every water type Pokémon point to a node called "Water", then there would be one relationship between these two type nodes.

I ended up deciding to go with the second option, because it seemed like it would reduce the overall number of relationships in the database and make it easier to navigate. The result of this decision means I have a network within my database that looks like the following:

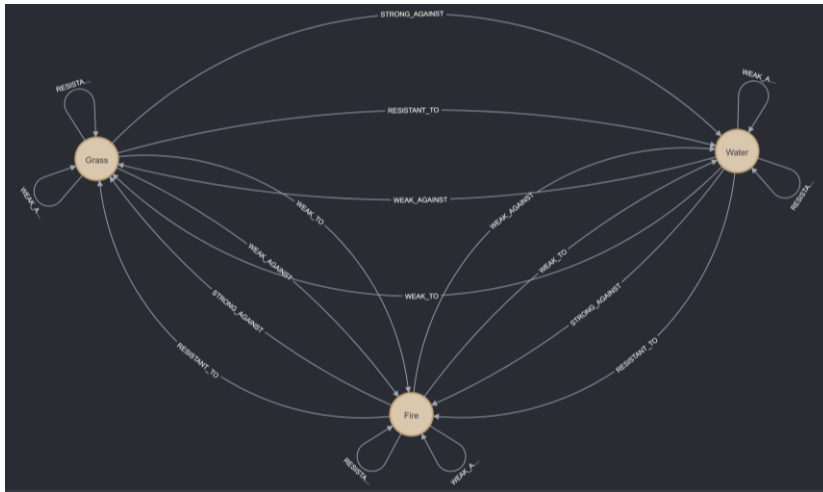


Figure 3. Showing just the relationships between fire, water and grass types

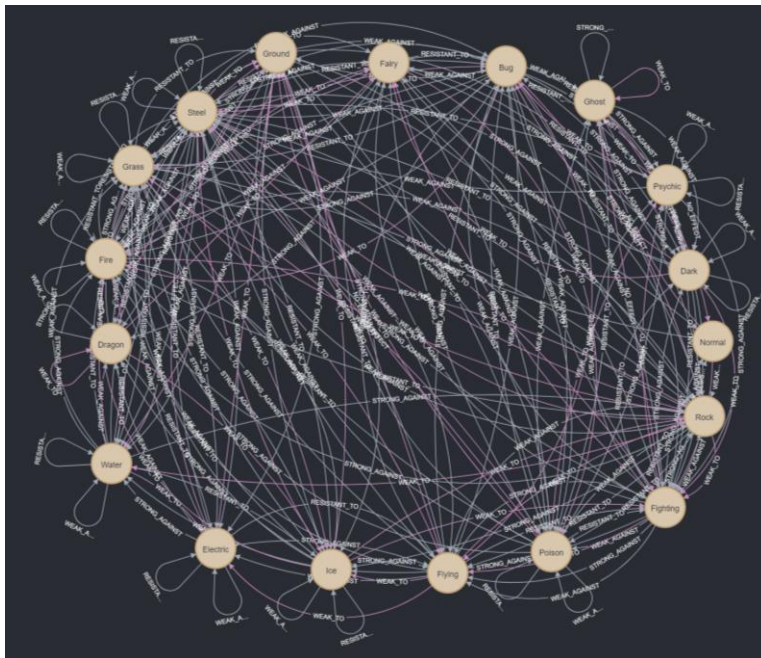


Figure 4. All types and their relationships with one another

To input the data into the database, I simply created the nodes for each Type and then applied the appropriate relationship (using the matrix CSV file) for every two types. For the relationships from each Pokémon to their type, I tried lots of options that could dynamically add the relationships with their properties. However I found that just executing a simple Cypher query provided me with the most 'power' in the least number of lines.

I go into a lot more detail about this in the Relationships section.

Evolution families data

Another set of data that was important to have was that describing every family of Pokémon. This is data showing what each Pokémon evolves into, and how it does so. This was tricky because not all Pokémon evolve, and not all Pokémon evolve in a linear manner ('A' evolves into 'B' evolves into 'C'). Therefore I had to find suitable data that can describe this, and that is easy enough for me to transfer into my database.

I found the data I needed on the Bulbapedia website ([7] List of Pokémon by evolution family - Bulbapedia, the community-driven Pokémon encyclopedia, 2021), where they have the source for each webpage freely available.

| Bulbasaur family | | | | | |
|---|------------|------------|---|------------|------------|
|  | Bulbasaur | Level 16 → |  | Ivysaur | Level 32 → |
| | | |  | Venusaur | |
| Charmander family | | | | | |
|  | Charmander | Level 16 → |  | Charmeleon | Level 36 → |
| | | |  | Charizard | |
| Squirtle family | | | | | |
|  | Squirtle | Level 16 → |  | Wartortle | Level 36 → |
| | | |  | Blastoise | |
| Caterpie family | | | | | |
|  | Caterpie | Level 7 → |  | Metapod | Level 10 → |
| | | |  | Butterfree | |
| Weedle family | | | | | |
|  | Weedle | Level 7 → |  | Kakuna | Level 10 → |
| | | |  | Beedrill | |
| Pidgey family | | | | | |
|  | Pidgey | Level 18 → |  | Pidgeotto | Level 36 → |
| | | |  | Pidgeot | |
| Rattata family | | | | | |
|  | Rattata | Level 20 → |  | Raticate | |

Figure 5. Visualisation of the data retrieved from Bulbapedia

After cleaning and formatting the data (see the next section for more details) I began to write the code that will input it into my database.

In order to input the data, I had to write code that can determine that size of the family it is dealing with, and other small details just by reading the string input.

Each Pokémon family may have a different number of family members. From analysing the data, I determined that the number of elements in each line is equal to this formula: $2 + (x * 3)$ where x is the number of Pokémon in the family. Rearranging this provides me with how many family members are in each family.

I also determined that each line will start with an identifier, with “evo” being for ordinary linear families, and variations on “evo-branch” to describe how a family branches. The following image describes the different ways they can branch:

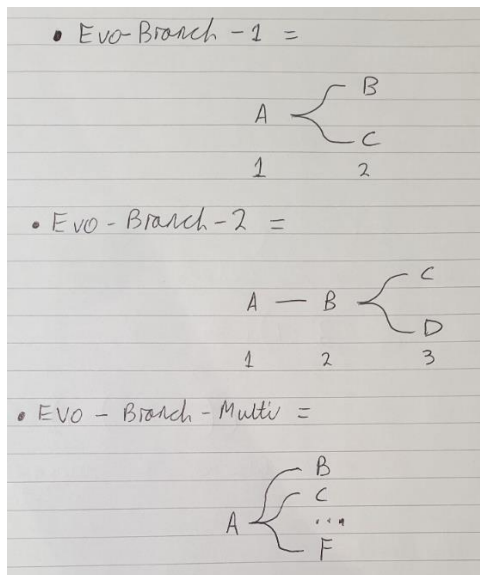


Figure 6. Diagram showing evolutionary branching – Where single capital letters represent a member of a Pokemon family

Lines starting with “evo-group” show the different forms of the same Pokémon, which I have decided to ignore for now as this is not necessary information for the user.

After doing some data testing, I found that each type of family (linear or branching) has family sizes that it sticks to, with linear family’s being of size 2 or 3, and so on for the other types. This made the next part easier, as I knew I only had to account for very few changes and didn’t have to make code that is over generalised, making it more efficient.

The way I added the relationships was the same as with the Types data, by executing Cypher queries that were specific to the structure of the family. I go into detail as to how I actually created each relationship in the Relationships section.

Formatting and cleaning data

Pokémon main CSV

For the main Pokémon CSV file (specifically the second dataset ([4]Banik, 2021)) I had to remove and change some data in order for it to be suitable for the database.

First, I moved the columns containing the Pokédex number, name and types to the front to be the first four columns. This is because these are the most important and most used pieces of data for each Pokémon, and so will make selecting them a lot easier.

The data also held a complex set of columns stating how effective each Pokémon was against every type, similar data to that held in the Types Chart CSV. I decided that because this data was less specific than the Types CSV, it was redundant and removed.

As well as this, I also removed other redundant columns that would not benefit the user. These columns were as follows: classification, experience, Japanese name, base egg steps and base egg happiness.

Evolution families data

The original source data from Bulbapedia had a lot of formatting text for the webpage, including spacing and background colours. I removed all of this text, and anything else that wasn't directly needed. I also removed any additional curly brackets or punctuation to make each line easier to use later on.

An issue I ran into was that there are variations on some families for different generations. For example, the standard Rattata evolves into Raticate by reaching level 20. However, the Alolan Rattata does so by reaching level 20 only at night. The issue here is that the Alolan Rattata has the same Pokédex number, but with an "A" to mark it as the Alolan version, 19A. This is an issue because the main Pokémon CSV file treats these two versions as one Pokémon, and so does not contain the Pokédex number "19A". Therefore my solution was to remove the letter from the Pokédex number, and to add a tag to the evolve requirement stating it is for the Alolan version. This will make it so there is one occurrence of the Rattata node in the database, but it will have two different relationship edges for these differing versions:

```
lop/evo|Pidgey|Normal|016|Pidgey|Level 18|017|Pidgeotto|Level 36|018|Pidgeot
lop/evo|Rattata|Normal|019|Rattata|Level 20|020|Raticate
lop/evo|Alolan Rattata|Dark|019|Rattata|Level 20 at night (Alola)|020|Raticate
lop/evo|Spearow|Normal|021|Spearow|Level 20|022|Fearow
```

Figure 7. Two versions of the same family

I searched through the data and repeated this for every occurrence of an Alolan or Galarian version of a Pokémon. This took some time, but resulted in a better representation of each family. I also removed the evolution if the requirement was the same for the normal and Alolan or Galarian version. For example, Pikachu evolves into Raichu using the thunder stone, and it also evolves into the Alolan Raichu using the thunder stone. This is the same requirement, and the Alolan Raichu is so similar to the normal Raichu that I removed this redundant data.

After all this formatting and cleaning, the data now looks like this:

```
lop/evo|Bulbasaur|Grass|001|Bulbasaur|Level 16|002|Ivysaur|Level 32|003|Venusaur
lop/evo|Charmander|Fire|004|Charmander|Level 16|005|Charmeleon|Level 36|006|Charizard
lop/evo|Squirtle|Water|007|Squirtle|Level 16|008|Wartortle|Level 36|009|Blastoise
lop/evo|Caterpie|Bug|010|Caterpie|Level 7|011|Metapod|Level 10|012|Butterfree
lop/evo|Weedle|Bug|013|Weedle|Level 7|014|Kakuna|Level 10|015|Beedrill
lop/evo|Pidgey|Normal|016|Pidgey|Level 18|017|Pidgeotto|Level 36|018|Pidgeot
lop/evo|Rattata|Normal|019|Rattata|Level 20|020|Raticate
lop/evo|Alolan Rattata|Dark|019|Rattata|Level 20 at night (Alola)|020|Raticate
lop/evo|Spearow|Normal|021|Spearow|Level 20|022|Fearow
lop/evo|Ekans|Poison|023|Ekans|Level 22|024|Arbok
lop/evo|Pikachu|Electric|172|Pichu|Friendship|025|Pikachu|Thunder Stone|026|Raichu
lop/evo|Sandshrew|Ground|027|Sandshrew|Level 22|028|Sandslash
lop/evo|Alolan Sandshrew|Ice|027|Sandshrew|Ice Stone (Alola)|028|Sandslash
lop/evo|Nidoran
lop/evo|Clefairy|Fairy|173|Clefairy|Friendship|035|Clefairy|Moon Stone|036|Clefable
lop/evo|Vulpix|Fire|037|Vulpix|Fire Stone|038|Ninetales
lop/evo|Alolan Vulpix|Ice|037|Vulpix|Ice Stone (Alola)|038|Ninetales
lop/evo|Jigglypuff|Normal|174|Igglybuff|Friendship|039|Jigglypuff|Moon Stone|040|Wigglytuff
lop/evo|Zubat|Poison|041|Zubat|Level 22|042|Golbat|Friendship|169|Crobat
lop/evo-branch-2|Oddish|Grass|043|Oddish|Level 21|044|Gloom|Leaf Stone|045|Vileplume|Sun Stone|182|Bellc
lop/evo|Paras|Bug|046|Paras|Level 24|047|Parasect
lop/evo|Venonat|Bug|048|Venonat|Level 31|049|Venomoth
lop/evo|Diglett|Ground|050|Diglett|Level 26|051|Dugtrio
lop/evo|Alolan Diglett|Ground|050A|Diglett|Level 26|051A|Dugtrio
lop/evo|Meowth|Normal|052|Meowth|Level 28|053|Persian
lop/evo|Alolan Meowth|Dark|052|Meowth|Friendship|Level 28 (Alola)|053|Persian
lop/evo|Galarian Meowth|Steel|052|Meowth|Level 28 (Galar)|863|Perrserker
lop/evo|Psyduck|Water|054|Psyduck|Level 33|055|Golduck
lop/evo|Mankey|Fighting|056|Mankey|Level 28|057|Primeape
lop/evo|Growlithe|Fire|058|Growlithe|Fire Stone|059|Arcanine
lop/evo-branch-2|Poliwhirl|Water|060|Poliwhirl|Water Stone|062|Poliwrath|Trade holdir
lop/evo|Abra|Psychic|063|Abra|Level 16|064|Kadabra|Trade|065|Alakazam
lop/evo|Machop|Fighting|066|Machop|Level 28|067|Machoke|Trade|068|Machop
lop/evo|Bellsprout|Grass|069|Bellsprout|Level 21|070|Weepinbell|Leaf Stone|071|Victreebel
lop/evo|Tentacool|Water|072|Tentacool|Level 30|073|Tentacruel
lop/evo|Geodude|Rock|074|Geodude|Level 25|075|Graveler|Trade|076|Golem
```

Figure 8. Example of family data

This data now resembles that of a CSV file, but each element is separated with a “|” instead of a comma. This makes handling this data a lot easier in the future.

Relationships

Pokémon Types

To create the network of relationships between each type (as shown in figure 4) I simply made the program read the Types chart CSV and create nodes for each type, then for each line it would create a relationship according to the number value found at each element. If the number was a 1, then the damage is normal and so no relationship is required. If the number is 0 then the type has no effect, so a relationship stating “NO_EFFECT” is created. This is repeated for if the number is 2 and 0.5, resulting in “STRONG_AGAINST” and “WEAK_AGAINST” respectively.

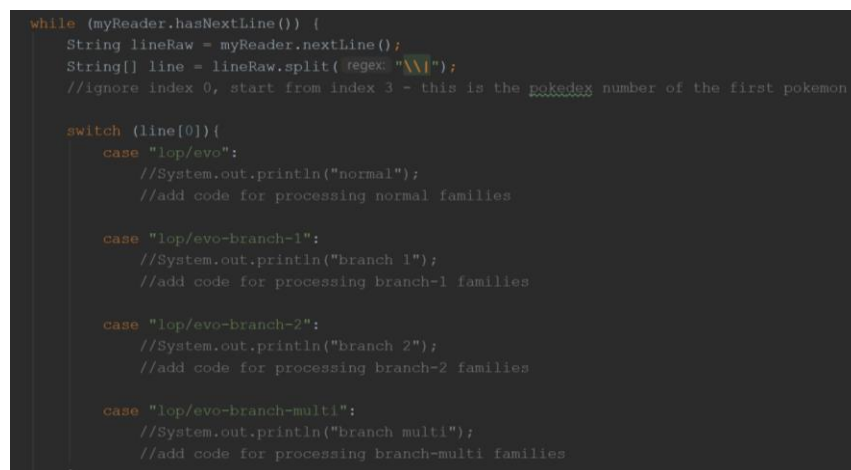
For the relationships between each Pokémon and their type, as previously mentioned, I executed two Cypher queries, one for each type a Pokémon can have. If a Pokémon only has one type, the second query simply doesn't create a relationship. These two queries are as follows:

```
MATCH (a:Pokemon), (b:Type) WHERE a.Type1 = b.Type CREATE (a)-[r:IS_TYPE]->(b) RETURN a,b
```

```
MATCH (a:Pokemon), (b:Type) WHERE a.Type2 = b.Type CREATE (a)-[r:IS_TYPE]->(b) RETURN a,b
```

Evolutionary Families

For the different types of families, I had a switch case segment to allow the program to efficiently apply the correct code to the right situation. The base structure of this code is the following:



```
while (myReader.hasNextLine()) {
    String lineRaw = myReader.nextLine();
    String[] line = lineRaw.split( regex: "X");
    //ignore index 0, start from index 3 - this is the pokodex number of the first pokemon

    switch (line[0]){
        case "lop/evo":
            //System.out.println("normal");
            //add code for processing normal families

        case "lop/evo-branch-1":
            //System.out.println("branch 1");
            //add code for processing branch-1 families

        case "lop/evo-branch-2":
            //System.out.println("branch 2");
            //add code for processing branch-2 families

        case "lop/evo-branch-multi":
            //System.out.println("branch multi");
            //add code for processing branch-multi families
    }
}
```

Figure 9. Switch case code for different family structures

This code splits each line by the special character, then compares the first element to the different cases and then proceeds. As mentioned before, testing showed that each type of family only has a small range of family sizes, meaning the next step was to add another selection segment for the different family sizes. Then I simply added a Cypher query for each case.

The code I ended up with used arrays of integers to hold all of the Pokédex numbers, and looked like the following:

```
switch (line[0]) {
    case "lop/evo":
        //all are size 2 and 3
        if (sizeOfFamily == 2){
            //create relationship for 2 family members
            try {
                ArrayList<Integer> pokdex_nums = new ArrayList<>();
                pokdex_nums.add(Integer.parseInt(line[3]));
                pokdex_nums.add(Integer.parseInt(line[6]));

                tx.execute( $ "MATCH (a:Pokemon), (b:Pokemon) WHERE a.pokdex_number = \"\" + pokdex_nums.get(0) + \"\" and b.pokdex_number = \"\" + pokdex_nums.get(1) + \"\"
            } catch (Exception e){
                //ignore if it cant turn the pokdex number into an int, just skip past it
            }

        } else if (sizeOfFamily == 3){
            //create relationship for 3 members
            try {
                ArrayList<Integer> pokdex_nums = new ArrayList<>();
                pokdex_nums.add(Integer.parseInt(line[3]));
                pokdex_nums.add(Integer.parseInt(line[6]));
                pokdex_nums.add(Integer.parseInt(line[9]));

                tx.execute( $ "MATCH (a:Pokemon), (b:Pokemon), (c:Pokemon) WHERE a.pokdex_number = \"\" + pokdex_nums.get(0) + \"\" and b.pokdex_number = \"\" + pokdex_nums.get(1) + \"\" and c.pokdex_number = \"\" + pokdex_nums.get(2) + \"\"
            } catch (Exception e){
                //ignore if it cant turn the pokdex number into an int, just skip past it
            }

        }
}
```

Figure 10. Switch case for family structures with working queries

However I quickly realised that this could be improved and made much more efficient by having the array built before the switch case segment. This resulted in much more efficient and cleaner looking code:

```
try {
    ArrayList<Integer> pokdex_nums = new ArrayList<>();
    for (int i = 3; i < line.length; i += 3) {
        pokdex_nums.add(Integer.parseInt(line[i]));
    }

    switch (line[0]) {
        case "lop/evo":
            //all are size 2 and 3
            if (sizeOfFamily == 2) {
                //create relationship for 2 family members
                tx.execute( $ "MATCH (a:Pokemon), (b:Pokemon) WHERE a.pokdex_number = \"\" + pokdex_nums.get(0) + \"\" and b.pokdex_number = \"\" + pokdex_nums.get(1) + \"\"
            } else if (sizeOfFamily == 3) {
                //create relationship for 3 members
                tx.execute( $ "MATCH (a:Pokemon), (b:Pokemon), (c:Pokemon) WHERE a.pokdex_number = \"\" + pokdex_nums.get(0) + \"\" and b.pokdex_number = \"\" + pokdex_nums.get(1) + \"\" and c.pokdex_number = \"\" + pokdex_nums.get(2) + \"\"
            }
            break;
        case "lop/evo-branch-1":
    }
}
```

Figure 11. Switch case for families improved

Graphical User Interface

With all of the data and relationships added to the database, I could now begin working on the GUI that the user will interact with.

First, I created a landing page that allows the user to choose between the various sections of my software. There are going to be sections for building a Pokémon team, for searching the database using natural language, and for querying the database using Cypher Query Language.

Once the landing page was created, I decided to make it look more pleasing to the user, so I imported a Pokemon themed font ([5] Sagas, 2021) and added a very simple colour scheme of red and white:

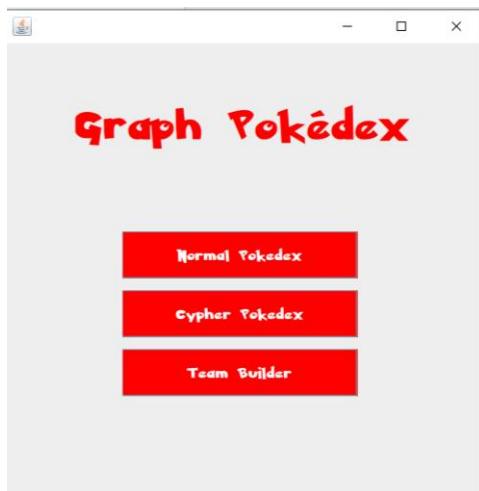


Figure 12. GUI landing page

Next, I created a simple window that is accessed by clicking the “Cypher Pokédex” button. This section will accept Cypher inputs and display what is returned by the database:

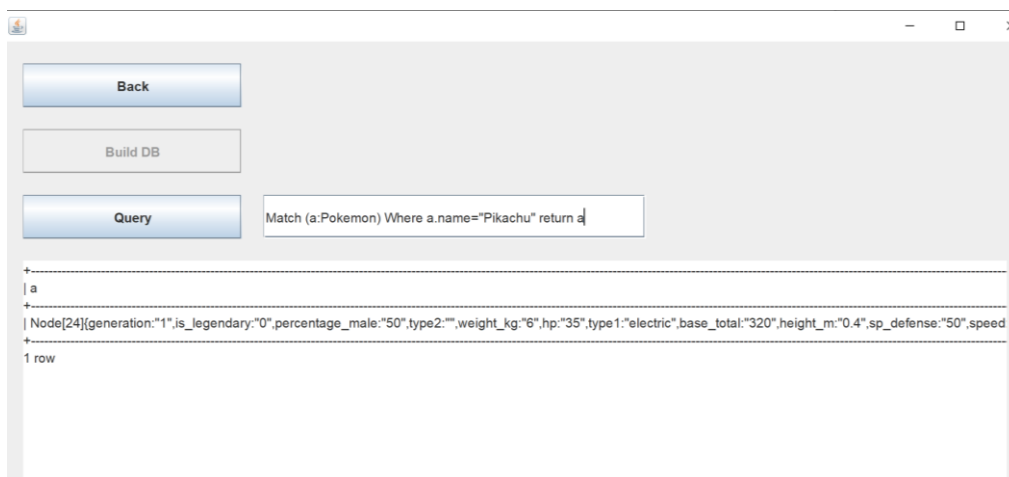


Figure 13. Cypher Pokédex section

As you can see, there is a button that builds the database. At the moment this needs to be pressed before querying the database. However I have decided that it is best to simply build the database on start-up to reduce waiting. Therefore this button was then removed and I began to match the styling of this window to the landing page by changing the colours and font.

With the software now building the database on start-up, it continues to do so in the background while the GUI is displayed, therefore the user is able to interact with the windows in the meantime. Below is what the GUI looks like while the database is still being built, and when it is finished building:



Figure 14. Cypher section while database is being built



Figure 15. Cypher section with the database built

Notice in figure 14 that the query button is disabled. This button is enabled only once the database is built, meaning the user cannot try to query if the database isn't finished yet.

At the moment, the software just displays the raw result from the query of the database. This is simply shown in the large Text Area below the entry box. In the future I will format the result to be easier to read, and to look nicer for the user.

Next I created windows for both the "Normal Pokédex" section - that will have natural language querying - and the "Team builder" section - that will have the ability to create a team of six Pokémon. The next aim is to start formatting the results from the cypher queries to make it look pleasing to the user, and to also start building the natural language translating section.

Simple Search

Before I begin the section for learning Cypher Query Language, I have decided to add a section for very simple searching. This would look very similar to the window shown in figure 15 above but will allow the user to simply enter a search for a single attribute of a Pokémon. For example, a Pokémon's name, or all Pokémon from generation 1. This will just include a drop-down box for the chosen attribute and an entry box for what they want to specify about the attribute.



Figure 16. Simple search window

As you can see from figure 16, I have implemented the simple search section and included the drop-down menu of attributes the user can choose from. I decided to not include some attributes like 'base_happiness' as this is a less important attribute to search by, and there are more complex searching sections that allow for this. Additionally, I decided to add scroll bars to the text area that holds the result of the query. This is to help display the data and has been added to all the other sections that have a text area.

One of the minor things I wanted to add to this simple search was the ability to use comparison operators like '>' or '<' for attributes that have numerical values. For example, I wanted the user to be able to search for Pokémon that have 'attack > 100'. However, when I populated the graph database, these numerical attributes were defaulted to be string or character data types. Therefore I had to add an extra cypher command to the part of my code where I populate the database that sets all of the selected attributes to be integers:

```
tx.execute("match (p:Pokemon) set p.hp=toInteger(p.hp),
p.attack=toInteger(p.attack), p.defense=toInteger(p.defense),
p.sp_attack=toInteger(p.sp_attack), p.sp_defense=toInteger(p.sp_defense),
p.speed=toInteger(p.speed), p.generation=toInteger(p.generation)");
```

Next, I want to start working on formatting the data that comes out of the database as a result of the query. For this section, the formatting will change for each attribute selected. For example, if a Pokémon is being searched by name or Pokédex number, it will return a page of details about that one Pokémon. But if more general attributes are selected like generation, then a list of Pokémon with basic details will be returned to the page. Because of this, I have already used an 'if-statement' to separate the different possible outcomes. I will return to this formatting task later on in the project, as it is a task that applies to the rest of the sections, and some that haven't been created yet at this stage.

Normal Pokédex

For this section, I decided that a visual learning approach would be best for the user to begin learning how to use the Cypher language. Therefore, after doing some research, I wanted to use Scratch as inspiration ([8] Scratch - Imagine, Program, Share, 2021) to create a query constructor that is easy to understand.

To start this, I had to implement an interactive window that can allow the user to click and drag a command to build a query. I began working with images and trying to make them draggable but found that it wasn't very efficient and had a lot of minor issues.

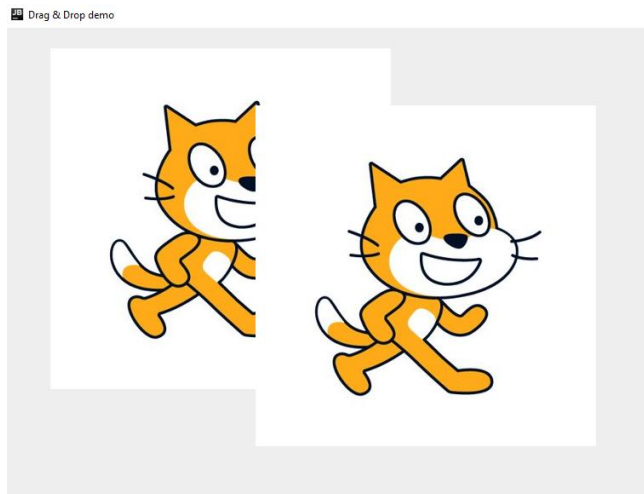


Figure 17. Drag and drop example

In the image above, you can see the result of me making two images able to be dragged and dropped. However, the code to do so was clunky and inefficient, and there were a few overlapping issues. What I needed was an object-oriented approach for this so I can create a new object every time the user wants to click and drag a new command.

Therefore, I decided that this drag and drop approach would not be best for this project, and I should start looking for another solution to create a query constructor.

To break down what the user needs to understand, I decided to create a flow chart that can show the flow of creating a query.

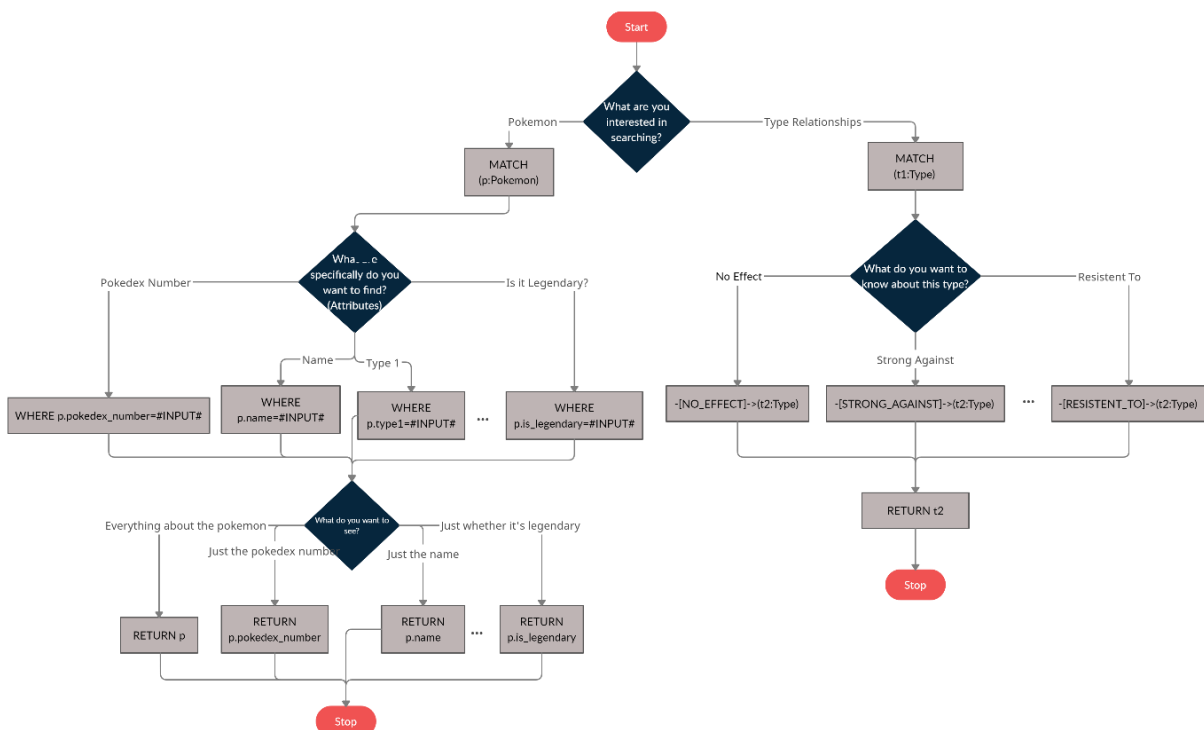


Figure 18. Query flowchart

From this flowchart, I can now go ahead and start constructing the query builder. I've decided that a simple way to do this could be to just have buttons and input text boxes appearing at the

different parts of the flowchart. At each stage of the flowchart, a box at the top of the window will update with each part of the query as the user goes through the steps.

Although this is a much more simple solution to the problem, I think that it is more important to have something simple that works properly instead of something that is overly complex that doesn't achieve it's goal.

To start, I created the window and added the first stage asking the user what they are interested in searching for, along with two buttons for the two choices. I also added the text bar along the top to hold the constantly updating query, and a text box to hold the result of the query.

The screenshot shows a window titled "Graph Pokémon" with a "Back" button in the top left. At the top, a text box displays the Cypher query: `MATCH (p:Pokemon) WHERE p.pokedex_number="100" RETURN p`. Below this, the flowchart consists of three steps:

- Step 1:** "What are you interested in searching for?" with two buttons: "Pokemon" and "Pokemon Types".
- Step 2:** "What do you specifically want to find?" with a dropdown menu showing "pokedex_number", an equals sign, a text input field containing "100", and a "Next" button.
- Step 3:** "What do you want to see displayed?" with a dropdown menu showing "Everything" and a "Query" button.

At the bottom, a text area displays the query result in JSON format:

```
{
  "p":
  [
    {
      "Node": "997",
      "generation": 1,
      "is_legendary": 0,
      "percentage_male": "",
      "type2": "",
      "weight_kg": 10.4,
      "hp": 40,
      "type1": "electric",
      "base_tor":
    }
  ]
}
```

Figure 19. All three steps of the Pokémon section

As you can see from figure 19, I next added all three steps from the Pokémon side of the flowchart, with action listeners updating the query at the top of the page. When the query button is pressed, the query at the top is sent to the database and the result is returned to the text area below.

This live updating query at the top really makes it clear what effect the user has when they change certain attributes, and hopefully helps them learn how to use the Cypher language. Although it isn't a translator from natural language to cypher language, it does translate the few steps a user needs to think about into cypher language and therefore helps them learn quickly.

Initially, I had issues with integer attributes not working with what the user was inputting. This was because I was automatically adding quote marks to the user's input, because the data was stored as a string. Therefore, I simply made the code check what data type the selected attribute is and only add quote marks to strings.

After this, I started creating the other side of the flowchart that searches for type relationships. This involved having all Pokémon section components disappear, and the Type section components take their place. Once all the flowchart steps were added, I was left with a very fluid form within a single window that can change to what the user desired, without resetting or wiping the window. This second layout can be seen in figure 20 below.

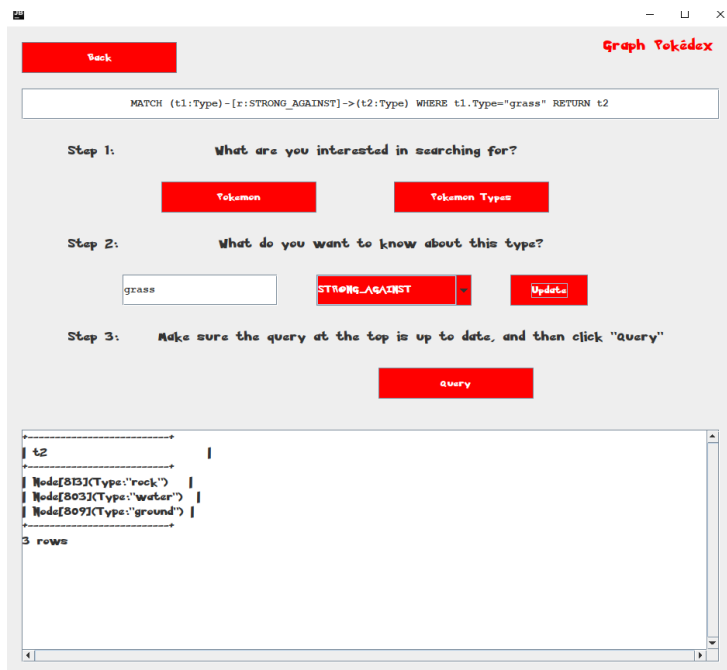


Figure 20. All three steps of the Types section

At this stage, both sections just simply passed the query from the text box at the top to the database, then displayed the result in the box at the bottom of the page. However, the result from the database wasn't very nice to look at and read, and the data was in a strange order. Figure 19 shows that things like the Pokémon's weight or type was being displayed before it's name, which is arguably the most important piece of data. Therefore, I began working on a way to format the database result so that it is a lot more user appropriate.

Result Formatting

As mentioned previously, the problem to solve is that the result from any query of the database is unordered and looks extremely busy and hard to read. I did some research on the Neo4j website ([9] Executing Cypher queries from Java - Neo4j Java Reference, 2021) to evaluate my options for formatting the result and found that there were actually very few options. Stepping through each row didn't seem to work, and accessing each column wasn't what I wanted either. Therefore, I settled on cleaning and formatting the string version of the result manually.

The first thing to do was to split the result into an ArrayList, so that each line was a new element in the array. This allowed me to easily remove the leading and trailing lines that held unimportant data or ASCII characters trying to simulate the border of a table. Next I had to do the same but to each line, as this too had leading and trailing unimportant data. This left me with just the lines of data that I was interested in.

However, these lines were still out of order. Therefore I came up with a method to sort all lines into a predefined ideal order: I first split each line into yet another array, with each element now being an attribute and value (e.g. name="Pikachu"). Then I created an array of integers that represented the order of indexes I wanted (e.g. [4,2,5] means I want index 4 at the front, 2 in the middle and 5 at the end, and so on). Then I simply compared these two arrays to populate a temporary array that now holds the sorted array. This is easier to understand by reading the code in the following image:

```

//reordering each line so that it is in the preferred order
String[] sortingArray = resultArray.get(i).split( regex: "\\|");
String[] temp = new String[sortingArray.length];
//The order of indexes that puts pokedex number and name at the front, and so on
int[] idealIndexOrder = {14, 17, 6, 3, 12, 16, 15, 18, 9, 10, 0, 5, 7, 4, 1, 11, 13, 2, 8};

//putting the line into order
for (int j = 0; j < sortingArray.length; j++) {
    temp[j] = sortingArray[idealIndexOrder[j]];
}

//putting the array of the line back into a single string
resultArray.set(i, String.join( delimiter: "|", temp));

```

Figure 21. Attribute reordering

Once these line arrays were put back into strings, they were simply returned by the function and displayed.

Figure 22. Formatted result on Pokémon section

Figure 22 shows that now the output is less confusing and is now ordered nicely.

This worked nicely for queries that returned all of a Pokémon's attributes, but would break when a query just wants a single attribute returned, like it's name. Therefore, I had to add a 'Mode' variable that is passed to the query function, and then split the query function to try either the 'Multi' mode or the 'Single' mode. The 'Multi' mode works exactly the same as above, but the single mode didn't require any reordering of attributes. However, the single mode faced some issues regarding the removal of unimportant data from the line, as different markers were being used to identify the important data. This was made even harder when faced with data that was an integer, because I couldn't use any quote marks to find the data. Therefore, I added a 'try catch' block that tries to isolate the data by quote marks, and if it doesn't have any quote marks then it searches until it finds a digit.

This worked for the Pokémon section, but would crash when I tried to query for type relationships. This was solved by simply checking what kind of data is wanted to be returned, and setting the mode accordingly, regardless of what variable is being used.

After making sure this new formatted output worked on all sections of the program, and with all inputs, I decided I can now move onto creating the Team Builder section of the code.

Team Builder

For this section, I decided that I wanted six boxes to hold each team member's information (six is the maximum size for a Pokémon team in the games) and another large text box at the bottom of the window to display any observations on the team data, or other information. I also decided that for each of the six text box "slots" I would need two buttons: one for adding a Pokémon to the box and one for removing it. This would mean that there would be at least six text boxes and twelve buttons on screen. Therefore, I quickly realised the most efficient way of creating these components was to have them stored in arrays, and then iterating through each element and initialising them that way. This involved using some slightly complicated maths to determine the X and Y coordinates of all of the components and making sure they are all lined up. However, once it was all sorted out, I was left with the following layout:

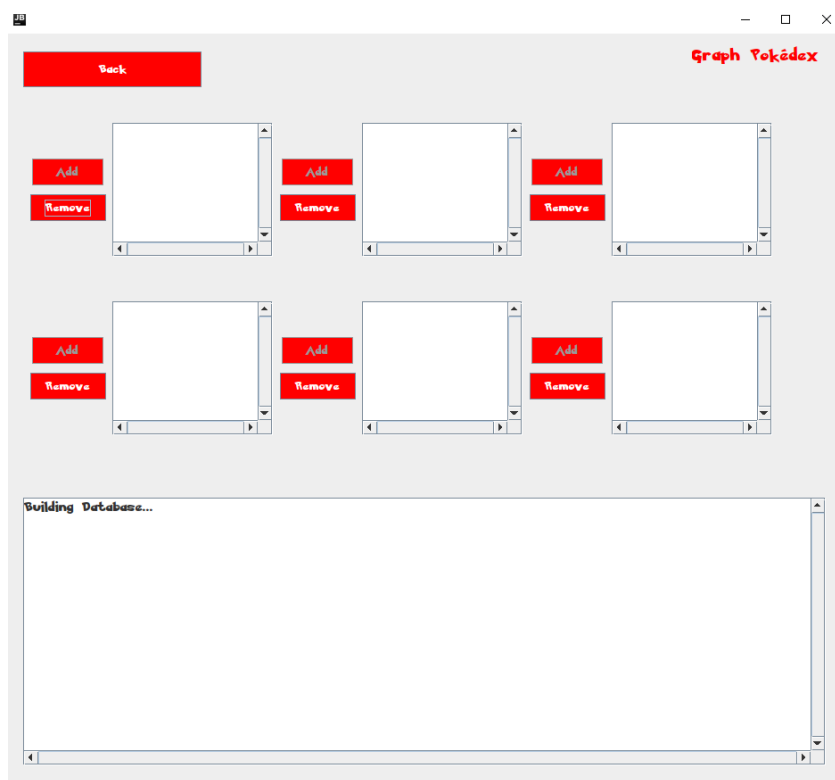


Figure 23. Team Builder section layout

Once this layout was completed, I had to decide how I was going to allow the user to add Pokémon to the slots. I settled on using pop-up dialog boxes that require the user to type in an input. These pop-up boxes request the name of a Pokémon that the user wishes to add to the team, and then returns the result of a query to the selected slot. If the input Pokémon name is a valid name, it is also added to an arraylist that holds the current team in it. If the "remove" button is pressed, then the selected slot is simply cleared, and the Pokémon is removed from the Pokémon team arraylist.

This arraylist is used to calculate all of the types that the team is vulnerable to. To calculate this, it is passed to a function called "calcVulnerabilities", where a cypher query is run to find the

weaknesses of each Pokémon. This single query is able to take just a Pokémon's name, and find this Pokémon's data, and then all of the relevant relationship information:

```
MATCH (p:Pokemon)-[r1:IS_TYPE]->(t1:Type)-[r2:WEAK_TO]->(t2:Type) WHERE
p.name=#INPUT# RETURN t2
```

This really shows just how powerful the cypher language is. This kind of data retrieval would have taken many more lines in other querying languages like SQL within a standard database.

Using just this query, I was able to create a set of unique types that each of the team members are vulnerable to. However, I wanted to take this a step further and remove any types that at least one team member is resistant to. This is so that the user can see what types of Pokémon their team would be vulnerable to as a *whole*.

To do this, I simply repeated the previous query but replaced the “WEAK_TO” with the relationship “RESISTANT_TO”. This then built a separate set of types the team is resistant to, which is then removed from the set of vulnerabilities.

I also decided to repeat this, but to show what types a team is ineffective against and would therefore struggle to deal damage to. Therefore, I created a very similar function that uses the same queries but uses relationships “WEAK_AGAINST” and “STRONG_AGAINST” respectively. This new function creates a set of types that the team is weak against but removes a type if at least one member is strong against the type.

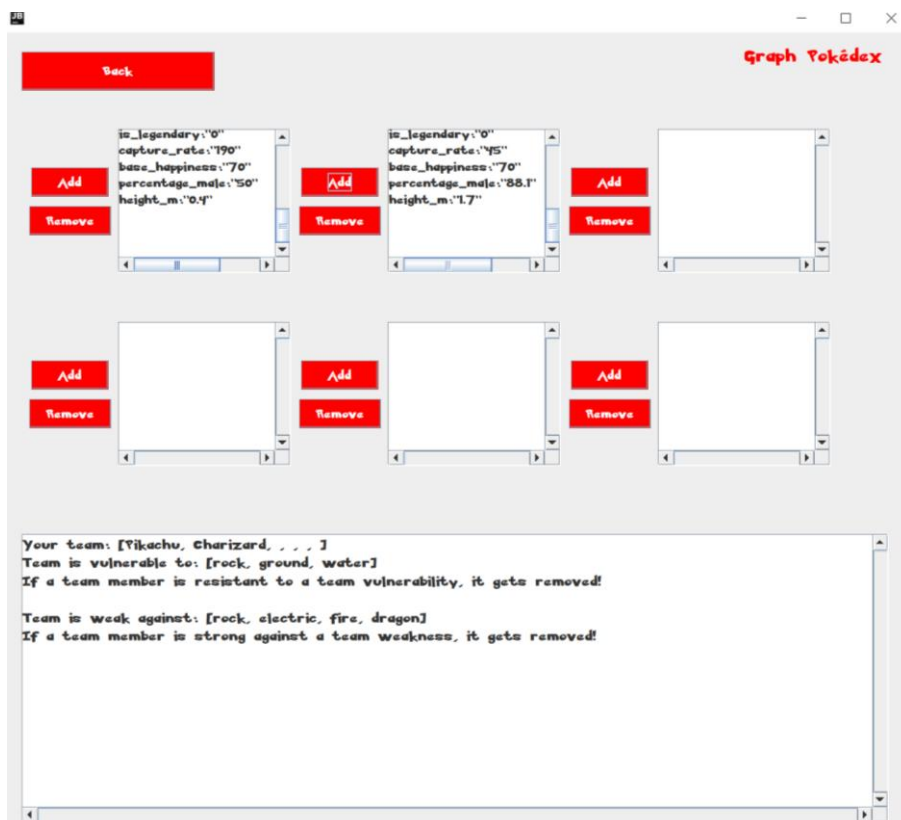


Figure 24. Functioning team builder with strengths and weaknesses showing

The figure above shows the team builder working and displaying the current list of strengths and weaknesses to the team. Each time a new member is added or removed, these lists are updated for the up to date team.

Testing

In this section I will be testing each part of my program to ensure that no errors occur unhandled, and that the software is easy for the user to interact with. I will summarise all successful tests into short paragraphs but go into more detail for each failed test – including screenshots if needed and my solution.

Before I tested each section, I tried executing queries before the database had finished building. However, this was not possible because all query or search buttons are disabled until the database is built. With this tested, I proceeded to test each section:

Simple Search

- For each attribute within the dropdown, I tested correct data, incorrectly spelt data, blank inputs, extreme data and different data types like integers. For most attributes, all of these inputs either returned a result or displayed an according message – either there were “no results found” or “the query was not valid”.
- However, the abilities attribute did not work - no result was returned regardless of input. This is due to the way the list of abilities is stored for each Pokémon. To fix this issue, I will just remove the ability to search for this attribute. I decided to do this because these abilities are not immediately important to know but are useful additional information that can be displayed after searching. If I had longer to work on this project, I would have reformatted these lists of abilities to make searching a lot easier. I removed this ability searching from all other occurrences in the software.



Figure 25. Ability searching not working

- When searching by hp, it works perfectly for any integer input, but when using a string it displays a different message: “It looks like that query isn’t valid, change it and try again”. This is because the query has failed to execute due to the change of data type, therefore this message shows the error has been caught successfully
- Same as above for attack, defense, sp_attack, sp_defense, speed, generation and is_legendary attributes.

Normal Pokédex

- Before I tested each subsection, I tested the ability for the window to change between them. The window was happily able to hide and show the relevant components to me when I needed, and even saved the previously entered data if it was an input.

- Pokémon section
 - All methods of searching within this section worked perfectly. I tested correct data, boundary data, extreme data and different data types. All errors were handled properly.
- Types section
 - Once again, this section performed as expected with all inputs. I input correct data, incorrectly spelt data and then different data types. For each input, any errors were handled properly

Cypher Pokédex

- This section has the least room for errors to occur, as it is simply passing the input query to the database to be executed. If the query is incorrect or invalid, it simply displays a message saying “that query is invalid, change it and try again”
- I tested basic queries and then queries that included complex relationship searching. All inputs were either successful, or returned a suitable message if I had typed the input incorrectly

Team Builder

- Similarly to the previous section, this part has very little that can go wrong due to the limited input the user has.
- I tried filling the team with correct Pokémon, and then removing them in different orders. This worked perfectly and the lists of strengths and weaknesses updated accordingly. I then tried inputting incorrect data in place of the Pokémon names, and received the correct message stating “no results found”. This was the same when I entered in incorrect data types.

The results of these tests show that the program is very robust and is difficult to break. This is likely because of my style of programming, which involved lots of small simple tests every time I added a new feature. This meant that as I created each section, I made sure that it would work robustly before moving onto the next one.

Another factor for this robustness could be that I have limited the amount of user input at various stages, thereby reducing the number of ways the user could explore and break the software.

Evaluation

For the sake of this project, I used data from the Pokémon franchise to populate my graph database. This data was used because of the variety and quantity available, and the number of relationships between the various Pokémon and their types. This allowed me to show off the different nuances of graph databases and explore all of its uses, but only in a fictional setting.

Graph databases have many useful applications in the real world, where we could replace this Pokémon data with that of hospital patients or, more specifically, people with diseases and viruses. Using hospital data, we can create and observe the relationships between various patients and their symptoms. We can use these relationships to spot patterns and links to other issues, like future problems that could occur for a patient, or to predict the way something could be spreading or changing.

It is important to note that storing data in a graph database does not necessarily generate more data or create more answers, it simply sets out and stores the existing data in a way that allows

for easier data analytics and manipulation. Because of this, a graph database is not perfect for all kinds of data – there is no real benefit to storing simple data in a graph database if there are no relationships to be created and explored.

This project has allowed me to really explore graph databases and has helped me to realise that it has many excellent aspects, but can sometimes feel like overkill for simple tasks. Although the code I have created utilises a lot of what is available from graph databases, even going as far as teaching the user Cypher Query Language, the majority of the tasks it performs are simple data searching and fetching jobs. These simple tasks can be executed extremely efficiently and compactly thanks to the power of the Cypher Query Language, but as mentioned before, it seems quite excessive to structure the whole database around relationships, just for them to be used rarely.

The true benefits of using a graph database would become more apparent with software that focuses primarily on these relationships and what they mean. As mentioned previously, my code uses these relationships to show extra links and data between Pokémon types, but does not use these relationships as a primary focus, thereby not fully realising the benefit of the graph database.

I *do* believe that my program benefits greatly from having its data stored in a graph database, however I still think there is so much more benefit that could be had if a few changes were made.

Future work

If I were to continue the development of this project past the submission date, I would want to add several more features and designs.

Firstly, I would explore the data further and find more ways I can include and create relationships to link the nodes and provide the user with more ways to explore them. This would utilise more of the benefits available from using graph databases, as I personally think this project has only just scratched the surface in what is possible.

As well as this, I wanted to see if it is possible to visualise the nodes and edges to the user in a similar way that Neo4j ([1] Graph Database Platform | Graph Database Management System | Neo4j, 2021) does with their software. This would have involved either simply embedding their software into a window and using it that way, or coding it from scratch. I currently do not know if this is even possible, but doing so would make it very clear and easy for the user to understand how the data is structured and could even help them identify patterns.

Finally, a nice smaller addition would be to add the little Pokémon images to the returned data. These sprites would be a nice touch to the Pokédex and make it feel like something you would see in the games or television shows.

Hindsight

If I were to go back and start the project again, there are a few things that I would have done differently.

Firstly, in the early days of my development I learnt how to build a graph database using the Java language and Neo4j's libraries ([2] Using Neo4j embedded in Java applications - Neo4j Java Reference, 2021). I found that in an older version of Neo4j's libraries, it was very easy to create a database and then have it stored locally in a folder on the computer. This was made a lot harder in more recent versions of the libraries, and so I decided to simply create the database each time the software was run. I understood that this is a very inefficient way of having the software run, but at the time it was the best solution I could find, because this local storing of the database became almost impossible to do with the version I was running. If I could go back, I would make sure to have this problem fixed to reduce the wasted building time and make the program a lot more efficient.

The other major thing I would do differently is focus a lot more on the relationships of the data, and how I can explore them more, instead of simple data searching and fetching. Throughout development I greatly enjoyed learning and using the Cypher Query Language, and I only realised its immense power and capability when the project was halfway through development. If I could go back, I would have focused on further utilising this Cypher power with the data's relationships.

Bibliography

- 1) Neo4j Graph Database Platform. 2021. Graph Database Platform | Graph Database Management System | Neo4j. [online] Available at: <<https://neo4j.com/>> [Accessed 10 November 2020].
- 2) Neo4j Graph Database Platform. 2021. Using Neo4j embedded in Java applications - Neo4j Java Reference. [online] Available at: <<https://neo4j.com/docs/java-reference/current/java-embedded/>> [Accessed 10 November 2020].
- 3) Armand, G., 2021. pokemon.csv. [online] Gist. Available at: <<https://gist.github.com/armgilles/194bcff35001e7eb53a2a8b441e8b2c6>> [Accessed 10 November 2020].
- 4) Banik, R., 2021. The Complete Pokémon Dataset. [online] Kaggle.com. Available at: <<https://www.kaggle.com/rounakbanik/pokemon>> [Accessed 14 November 2020].
- 5) Sagas, P., 2021. Ketchum Font | dafont.com. [online] Dafont.com. Available at: <<https://www.dafont.com/ketchum.font?psize=l&text=Graph+Pok%E9dex>> [Accessed 15 November 2020].
- 6) GitHub. 2021. zonination/pokemon-chart. [online] Available at: <<https://github.com/zonination/pokemon-chart/blob/master/chart.csv>> [Accessed 15 November 2020].
- 7) Bulbapedia.bulbagarden.net. 2021. List of Pokémon by evolution family - Bulbapedia, the community-driven Pokémon encyclopedia. [online] Available at: <https://bulbapedia.bulbagarden.net/w/index.php?title=List_of_Pok%C3%A9mon_by_evolution_family&action=edit> [Accessed 20 November 2020].
- 8) Scratch.mit.edu. 2021. Scratch - Imagine, Program, Share. [online] Available at: <<https://scratch.mit.edu/>> [Accessed 4 February 2021].
- 9) Neo4j Graph Database Platform. 2021. Executing Cypher queries from Java - Neo4j Java Reference. [online] Available at: <<https://neo4j.com/docs/java-reference/current/java-embedded/cypher-java/>> [Accessed 13 March 2021].