

eurolingo

Quantitative linguistic distance computation

Theodore G Tucker

27th June 2019

Submitted for the BSA Silver CREST Award

Contents

I. Pre-project work	2
1. Introduction	2
2. Aims and planning	3
II. Investigation	6
3. Research	6
4. Programming	9
III. Outcomes	19
5. Findings	19
6. Wider implications	20

Part I.

Pre-project work

0.1. Resources

Throughout this report, where references are made to files, they are relative to the `eurolingo` directory. All code files (contained within the `eurolingo` directory) may be downloaded from https://tti0.net/eurolingo/dist/eurolingo_latest.zip.

The *eurolingo* game can be played at <https://tti0.net/eurolingo/live/game/index.html>.

An electronic copy of this report is available at https://tti0.net/eurolingo/CREST_silver_report.pdf.

Appendix 1 of this report is available at https://tti0.net/eurolingo/CREST_silver_report_app1.pdf.

1. Introduction

1.1. Preface

This project grew out of the development of a game, *eurolingo*, in which the player is presented with a sentence, and must guess which language it is written in. The questions would be in multiple choice format, and, to increase the difficulty of the game, the incorrect languages (i.e. the ‘red herring’ answers) which appeared in the list would be linguistically ‘close’ to the correct answer.

All sentences included in the game would be in a language native to geographical ‘Europe’.

I decided that the computer might calculate linguistic distance, in order to determine which red herrings to show, by the creation of a ‘family tree’ of languages, using pre-existing data, in Python 3 objects. The algorithm could find the distance between a given node on the tree and all others, and hence, determine which languages used by the game are closest to the correct answer.

1.2. Inspiration

I was inspired, at least in part, to create the game after reading two *linguistic* books:

- **Lingo**¹ – Specifically chapter 25 (‘Pin the name on the language’), which describes the distinguishing features of the varied languages spoken on the European continent. The language used in this program (listed above) are (some of) those featured in this book.
- **Seven Languages in Seven Weeks**² – This book is about programming languages, not spoken languages, specifically chapter 2 (‘Ruby’), which discusses an example of Ruby’s class structure using a family tree. I thought it interesting to do something similar in Python and JSON.

1.3. Mentor

This project was mentored by Dr Sam Crawshaw, Head of Middle School and Teacher of Biology at Manchester Grammar School.

1.4. Licensing

This project (including its documentation) is licensed under the *MIT License*, but remains copyright Theodore Tucker, 2018-19. A copy of the license text can be found in the code files: `LICENSE.txt`.

¹Dorren, G. (2015). *Lingo*. London: Profile Books.

²Tate, B. (2010). *Seven Languages in Seven Weeks*. Dallas TX: Pragmatic Bookshelf.

2. Aims and planning

2.1. Aims

My main aims were as follows:

1. To complete the *eurolingo* game, as envisaged in Section 1
2. In doing so, compile tables containing the relative linguistic distances between all pairs of the following languages:

Language	ISO 639-2 code ³
Arabic (Modern Standard)	ara
Belarusian	bel
Breton	bre
Bulgarian	bul
Catalan	cat
Czech	ces
Cornish	cor
Welsh	cym
Danish	dan
German	deu
Greek (Modern)	ell
English	eng
Esperanto	epo
Estonian	est
Basque	eus
Faroese	fao
Finnish	fin
French	fra
Frisian (West)	fry
Scottish Gaelic	gla
Irish	gle
Galician	glg
Greek (Classical)	grc
Hebrew	heb
Croatian	hrv
Sorbian (Upper)	hsb
Hungarian	hun
Armenian (Eastern)	hye
Icelandic	isl
Italian	ita
Latin	lat
Lithuanian	lit

Language	ISO 639-2 code ³
Latvian (Standard)	lvs
Macedonian	mkd
Maltese	mlt
Dutch	nld
Norwegian (Bokmål)	nob
Ossetian	oss
Polish	pol
Portuguese	por
Romani	rom
Russian	rus
Slovak	slk
Slovenian	slv
Spanish	spa
Albanian	sqi
Serbian	srp
Swedish	swe
Turkish	tur
Ukrainian	ukr
Yiddish	yid

3. In doing so, observe the effectiveness of my approach for quantitatively calculating linguistic distance. Through this project, I intended to gain experience using:

1. Cloud computing technologies, to parse the dataset of sentences for the game;
2. Object-oriented programming (OOP) in Python 3;
3. L^AT_EX and L^AT_EX to typeset the report.

I already had extensive experience in frontend web development, using HTML5, CSS3, and JavaScript, with which I intended to create the game, before undertaking this project.

2.2. Stages

To achieve these aims and learning objectives, I split the project into the following stages:

1. **Pre-project work:** research on existing efforts in this field, and the theory behind linguistic distance
2. **Programming:** game interface, scoring, and logic
3. **Programming:** generating relationship tables for the game's multiple choice logic
4. **Evaluation**
5. **Post-project work:** write-up of this report, and completion of the CREST Student Profile Form

2.3. Time management

I completed this project over a 4-day period, from Monday 24th June 2019, to Thursday 27th June 2019: my school's 'Activities Week'. Working for 7.5 hours each day during this week would give a total work time of 30 hours, as recommended for the Silver Award.

Beforehand, I prepared the following objectives, to be completed by the end of each day:

³An internationally standardised set of short codes to represent languages; see https://loc.gov/standards/iso639-2/php/code_list.php

Monday: Development of game; write-up report (Sections 1, 2, 4.1, 4.2)
 Tuesday: Finish development of game; complete research for table generation; write-up report (Sections 3.1, 3.2, 4.3)
 Wednesday: Development of code for table generation; write-up report (Section 4.4)
 Thursday: Complete evaluation and post-project work; write-up remaining sections of report; complete CREST Student Profile Form

I intended then to hand in my project to my mentor for review in week commencing Monday 1st July 2019, and submit the entire project to CREST for assessment during the 2019 summer break.

I had started development of the game, and processed the dataset of sentences before Activities Week.

I worked alone on this project, hence, I completed all work myself.

2.4. Materials needed

My main tool in the development of this project was my laptop computer, whose specifications are given below:

```
$ lshw -short # (truncated)
```

H/W path	Device	Class	Description
=====			
		system	Aspire E5-511 (Aspire E5-511_0905_V1.06)
/0/0		memory	64KiB BIOS
/0/4		processor	Intel(R) Celeron(R) CPU N2840 @ 2.16GHz
/0/4/8		memory	32KiB L1 cache
/0/4/9		memory	1MiB L2 cache
/0/7		memory	24KiB L1 cache
/0/f		memory	8GiB System Memory
/0/100/2		display	Atom Processor Z36xxx/Z37xxx Series Graphics & Display
/0/100/14/0/2/1		communication	Atheros AR3012 Bluetooth
/0/100/1b		multimedia	Atom Processor Z36xxx/Z37xxx Series High Definition Audio Controller
/0/100/1c/0.1	enp1s0f1	network	RTL8111/8168/8411 PCI Express Gigabit Ethernet Controller
/0/100/1c.1/0	wlp2s0	network	QCA9565 / AR9565 Wireless Network Adapter
/0/1/0.0.0/1	/dev/sda1	volume	931GiB EXT4 volume
/0/2/0.0.0	/dev/cdrom	disk	DVD-RAM UJ8E2Q

```
$ lsb_release --all
```

```
Distributor ID: elementary
Description: elementary OS 5.0 Juno
Release: 5.0
Codename: juno
```

```
$ uname -a
```

```
Linux ibex 4.15.0-52-generic #56-Ubuntu SMP Tue Jun 4 22:49:08 UTC 2019 x86_64 x86_64 x86_64 GNU/Linux
```

Code was written using the GitHub Atom editor⁴.

Throughout my work, I frequently referenced the following resources:

- Python 3 official documentation⁵
- PythonTutor code visualisation software⁶
- W3Schools reference guides (HTML, CSS, JavaScript, and jQuery)⁷
- Dasgupta, S., 2016. *Computer Science: A Very Short Introduction (Very Short Introductions)*. Oxford: Oxford University Press.

⁴<https://atom.io/>

⁵<https://docs.python.org/3/index.html>

⁶<http://pythontutor.com/>

⁷<https://www.w3schools.com/>

- LyX official documentation ⁸

All other resources used (websites, books, etc.), are referenced throughout the report.

2.5. Safety considerations

I did not anticipate any significant specific safety risks in the completion of this project.

However, I still employed best practice with long-term computer use, such as maintaining good posture (to mitigate the risk of contracting RSI), and taking regular screen breaks (to mitigate the risk of contracting Computer Vision Syndrome).

I backed up all work to a private Git server at the end of each day, to retain (some) version control, and to have a copy of the files if my computer's hard drive were to fail.

Part II.

Investigation

3. Research

3.1. Linguistics

'The formation of different languages and of distinct species and the proofs that have both been developed through a gradual process, are curiously parallel.' – Charles Darwin^a

^aDarwin, C. (1871). *The descent of man and selection in relation to sex*. London: John Murray.

A language consists of a vocabulary (words) and a grammar (the rules for amalgamating words to give a more complex meaning – i.e. in a sentence), which together allow a speaker to convey meaning to a listener.

Ethnologue estimate that over 7000 languages are spoken today, yet, one might notice similarities between them. For example, the French: *bonjour* and the Italian *buongiorno* both have the same meaning (*good day*), and are similar in spelling and pronunciation. Many more pairs like these, and historical evidence (showing similarity between Classical Latin and Modern Italian, both spoken in the same region, but during different time periods) has led most linguists to believe in the *stammbaum* theory: that languages are related to one another, and descended from common ancestors, and eventually, to a common ancestor - *Proto-Everything*: the first language spoken on Earth⁹.

Hence, based on analyses of vocabulary, grammar, historical artefacts, and numerous other factors well beyond the scope of this report, linguists have been able to classify languages into *families*. Those with which eurolingo is most concerned are *Indo-European*, *Uralic*, *Semitic*, and *Turkic*.

I intended, using a pre-existing family trees of languages for the *eurolingo* dataset, to find the relative distances between one language in *eurolingo* to every other on this tree – a somewhat crude (see Part III) measure of linguistic distance (how closely related one language is to another).

3.2. Computer Science

3.2.1. Object-oriented programming

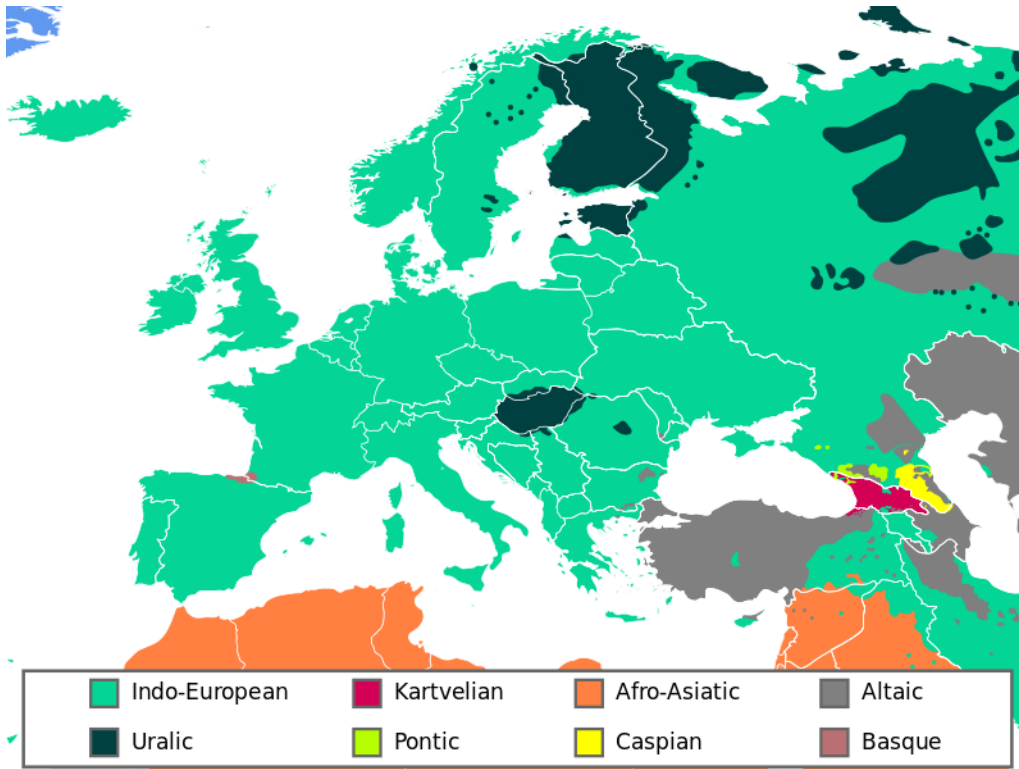
Before undertaking this project, I had no prior experience with OOP, but was aware that its use would be necessary in order to create a *binary tree* – the data structure most suitable for holding the *stammbaum* data.

Following my research, I understood that OOP involves *classes*: templates for *objects*. One defines a class once, and can then create objects based on this class. An object can have attributes (e.g. `fruitSpecies`),

⁸<https://wiki.lyx.org/>

⁹This *Proto-Everything* theory, whilst certainly plausible, is less universally accepted in the linguistic community, but convenient for *eurolingo*.

Figure 1: A map showing language families in Europe



Source: [https://commons.wikimedia.org/wiki/File:Language_Families_in_Europe_\(en\).svg](https://commons.wikimedia.org/wiki/File:Language_Families_in_Europe_(en).svg)

and methods: operations one performs on an object (e.g. `eatFruit()`); attributes can be attached to the instance (the object), or the class (all objects of the class). This idea is known as *encapsulation* (widely described as the most fundamental concept in OOP), which may formally be described as bundling all data, and methods used to operate on that data, into a single entity: the object.

Subsequently, I enquired into another key OOP paradigm: *inheritance*. Simply, inheritance refers to the fact that it is possible to define a class as a sub-class of another, which, along with any instances, will inherit all attributes and methods from the parent. For example, we might have a class `fruit` with the method and attribute described above, but then define the sub-class `citrus`, with a new method (`peelFruit`) and attribute (`acidity`):

```
CLASS citrus:
    parent:
        CLASS fruit
    methods:
        eatFruit()
        peelFruit()
    attributes:
        fruitSpecies
        acidity
```

Finally, I applied my new understanding of theoretical OOP into Python 3, the language in which I would program the linguistic distance algorithm, by following examples set by *Malik, 2018*.

3.2.2. Trees

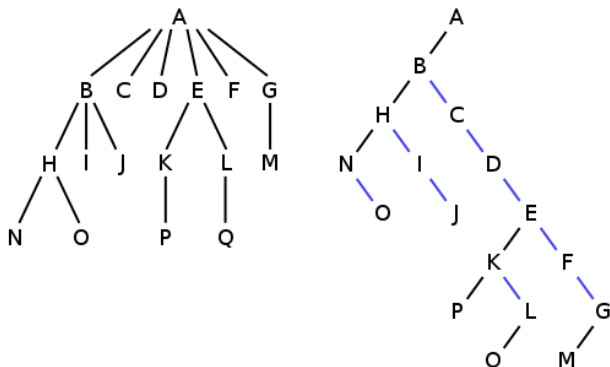
A binary tree is an example of a *graph*, which, in Computer Science, is an abstract data structure representing objects related in a hierarchy. The objects themselves are known as *nodes* (or *vertices*), and the links between them *edges* (or *links*).

For a binary tree, each parent node has only two children (usually referred to as *left* and *right*), and, as with all trees, there is only ever one possible path between two nodes.

At this point in my research, I believed that I had encountered a serious issue, as, in the language tree which I was attempting to encode, several parents had more than two children – the tree was n -ary; moreover, an axiom of my further research (on finding paths between nodes in binary trees) was that data would be in a binary tree. Therefore, I was required to research further into *left-child right-sibling binary trees*, and *Knuth transformations*, a method for the conversion of an n -ary tree (such as the language tree) into a binary tree. However, as my algorithm would find the distance between two nodes, the Knuth transformation does not preserve the significance of sibling relationships, as new edges must be created (shown in blue in Figure 2(b)). For example, in Figure 2(a), the distance from node J to Q is 5 (path $J > B > A > E > L > Q$), however, in Figure 2(b), the distance between the same nodes is 10 ($J > I > H > B > C > D > E > L > Q$).

Therefore, I had to investigate more fully how the algorithms which I intended to use later on might be applicable in the case of an n -ary tree, rather than the simpler binary tree.

Figure 2: Knuth transformation of an n -ary tree into a binary tree



Source: https://commons.wikimedia.org/wiki/File:N-ary_to_binary.svg

3.2.3. Distance between two nodes

Because there is only one path between any two nodes in a tree structure, any two nodes in a tree must share a single *lowest common ancestor* (LCA) – that is, the node furthest from the *root* (the initial node, on the highest layer of the tree) which is an ancestor of both of the two nodes. Once I found the LCA for all pairs of nodes, I could then easily find the distance between the nodes, completing the task.

I researched various algorithms for finding an LCA, all of which have efficiency $O(n)$, where n is the height (number of layers) on the tree. I decided to implement the following algorithm:

Algorithm 1 Algorithm to find the LCA of nodes x_1 and x_2 in a tree T

1. Find the path from node T_{root} to x_1 , using tree transversal, stored in an array, **path1**, as a series of instructions for the ‘journey’ (e.g. [**left**, **right**, **left**, **left**]).
 2. Find the path from node T_{root} to x_2 , using tree transversal, stored in a similar array, **path2**.
 3. Compare both arrays, and return the greatest **n** for which **path1[n] == path2[n]**. This is the LCA.
-

The distance between the two nodes may then be computed, using the following algorithm:

Algorithm 2 Algorithm to find the distance between nodes x_1 and x_2 in a tree T

The distance, d , between any arbitrary nodes x_1 and x_2 in a tree is given by:

$$d = r(x_1) + r(x_2) - 2(r(LCA))$$

Where $r(n)$ is the distance (in our case, number of layers) from T_{root} to node T_n , and LCA is as calculated earlier.

3.2.4. Sources

3.2.4.1. Linguistics

- Rowe, B. and Levine, D. (2018). *A Concise Introduction to Linguistics*. Abingdon: Routledge, pp. 340-341.
- Eifring, H. and Theil, R. (2005). *Linguistics for Students of Asian and African Languages*. Oslo: University of Oslo, Chapter 5. Available at: <https://www.uio.no/studier/emner/hf/ikos/EXFAC03-AAS/h05/larestoff/linguistics/>

3.2.4.2. Computer Science

- Djidjev, H., Pantziou, G. and Zaroliagis, C. (1991). Computing shortest paths and distances in planar graphs. *Automata, Languages and Programming*, pp. 327-338.
- Pfaltz, J. (1975). Representing Graphs by Knuth Trees. *Journal of the ACM*, 22(3), pp. 361-366.
- Malik, U. (2018). *Object Oriented Programming in Python*. Internet: Stack Abuse. Available at: <https://stackabuse.com/object-oriented-programming-in-python>
- Wentworth, P., Elkner, J., Downey, A. and Meyers, C. (2012). *Chapter 27. Trees*. Internet: How to Think Like a Computer Scientist: Learning with Python 3. Available at: <http://openbookproject.net/thinkcs/python/english3e/trees.html>
- TutorialsPoint. (n.d.). *Python - Binary Tree*. Internet. Available at: https://www.tutorialspoint.com/python/python_binary_tree.htm
- GeeksForGeeks. (2014). *Lowest Common Ancestor in a Binary Tree (1)*. Internet. Available at: <https://www.geeksforgeeks.org/lowest-common-ancestor-binary-tree-set-1>
- GeeksForGeeks. (2014). *Find distance between two nodes of a binary tree*. Internet. Available at: <https://www.geeksforgeeks.org/find-distance-between-two-nodes-of-a-binary-tree>

4. Programming

4.1. Parsing dataset

4.1.1. Tatoeba

The *eurolingo* game required a dataset of sentences, from which one would be randomly chosen to display to the user. After some brief research online, I encountered the Tatoeba¹⁰ project: a database containing over 7 million sentences in over 300 different languages, many of which are accompanied by translations.

However, at the time of writing, the simple Tatoeba database¹¹, a CSV, formatted as follows: ID [tab] ISO [tab] Sentence, exceeded 110MB in size compressed – far too large for processing on my development machine, or a web browser on a user’s device.

Furthermore, many sentences in the database were unnecessary, as their language was not included in the scope of *eurolingo* (see table in Section 2.1). Hence, I resolved to select 150 sentences in each of the 51 languages, giving 7650 sentences total, and a significantly reduced file footprint.

This dataset would then be imported into the game, ideally in JSON format.

4.1.2. Approaches

I considered 3 different approaches in the reduction of the dataset:

1. Using a Python 3 script and the Requests II library¹², with which I had prior experience, to iterate through each language, and request 150 sentences for each, from some sort of API.
2. Downloading the entire dataset, and processing on my development machine with a Python 3 script.
3. Uploading the entire dataset to a cloud Platform as a Service (PaaS) provider, and perform processing there.

¹⁰<https://tatoeba.org/eng>

¹¹<http://downloads.tatoeba.org/exports/sentences.tar.bz2>

¹²<https://2.python-requests.org/en/master>

4.1.2.1. API Having been encouraged by the API Specification¹³ discovered on the GitHub Wiki of Tatoeba, I discovered that, whilst such an API had been developed in 2015¹⁴, it was incomplete, required cloning the entire Tatoeba site to my device, and its developers deemed it experimental. Moreover, exploration of Tatoeba's own documentation for end users¹⁵ confirmed that no stable API was currently available. I therefore decided that following Approach 1 would not be possible.

4.1.2.2. Local parsing After downloading and extracting the Tatoeba database, I converted the CSV file to a JSON file online¹⁶, with structure as described in Section 4.2.

I wrote a simple, but highly inefficient ($O(n^3)$), Python 3 script (which has now been lost), approximating to the following:

```
# sentences = JSON object above
newSentences = [];
possibleLangs = ["ara", "bel", "bre"] # etc. for other languages
FOR i IN possibleLangs:
    FOR j in RANGE [1, 150]:
        FOR k in sentences:
            if k.lang == i:
                newSentences.append(k)
```

Even if considering a more efficient algorithm than the one above, on my modest development machine, this code would have taken several months to years to execute over the 350MB+ extracted dataset.

Thus, I decided that following Approach 2, whilst theoretically possible, would be highly impractical.

4.1.2.3. PaaS Finally, I decided that, given the enormous dataset, I would use Google Cloud PaaS products, specifically *Cloud Storage*¹⁷ and *BigQuery*¹⁸. I had some prior experience with Google Cloud BigQuery, a PaaS product designed specifically for running operations on large datasets, and, which is free for my *relatively* small dataset.¹⁹

4.1.3. PaaS implementation

The Python 3 script `quotes/bigquery.py` runs a Google Cloud BigQuery request, using the BigQuery client library²⁰.

The list of languages for which queries are run is stored in `quotes/languages.json`.

`quotes/bigquery.py` exports to `quotes/quotes.csv`. However, the *eurolingo* game pulls its sentences from `quotes/quotes.json`, so I had to convert the CSV to JSON manually, using the same method described earlier.

I performed the steps outlined in my documentation: `quotes/README_DATASET.md`, executing the Python 3 script `quotes/bigquery.json` – this took approximately 45 seconds to complete.

4.2. JSON formatting

eurolingo involves three main datasets: the languages, the sentences, and the computed relationships between languages.

I decided to format all of these datasets using JavaScript Object Notation (JSON), due to the ease with which objects can be nested, necessary for relationships, my familiarity with the format, and out-of-the-box

¹³<https://github.com/Tatoeba/tatoeba-api/wiki/Tatoeba-API-specification-2>

¹⁴<https://github.com/Tatoeba/tatoeba-api>

¹⁵<https://en.wiki.tatoeba.org/articles/show/faq>

¹⁶<https://csvjson.com/csv2json>

¹⁷<https://cloud.google.com/storage/>

¹⁸<https://cloud.google.com/bigquery/>

¹⁹Google charge only when operations would exceed US\$300 in cost.

²⁰<https://cloud.google.com/bigquery/docs/reference/libraries>

compatibility with JavaScript and Python 3²¹. Furthermore, JSON has been standardised²², so anyone who wishes to adapt my code in the future, for example, to add another language, can refer to well-established documentation.

The datasets are stored in `.json` files, such that one file can be written by the Python generators, and read by the JavaScript game.

4.2.1. Challenges

I encountered the following challenges, specific to *eurolingo*, when handling, converting and parsing these datasets:

- Presence of non-ASCII characters in some sentences. Handled by configuration of programming languages to treat files as UTF-8 – modern web browsers should do this automatically.
- Presence of the ”(") character in some sentences involving direct speech, causing the JSON interpreter to think that a sentence had ended, and then throw a syntax error when the sentence continued after this point. Handled by the manual removal of sentences with containing the " character from the dataset, as this issue was discovered only after I had written most of the code for *eurolingo*.

4.2.2. quotes/languages.json

This is an array of JSON objects, each representing one of the 51 languages in which *eurolingo* sentences are written, of which an example is:

```
{
  "language": "Estonian",
  "native": "Eesti",
  "iso": "est"
}
```

The language’s ISO code (`iso`) is the primary key.

4.2.3. quotes/quotes.json

This is an array of JSON objects, each representing a sentence used by the *eurolingo* game, of which an example is:

```
{
  "id": 691393,
  "lang": "deu",
  "quote": "Herr Smith machte ihm einige Spielzeuge."
}
```

The sentence’s ID on Tatoeba (`id`) is the primary key.

4.2.4. Relationship tables

These are arrays of JSON objects, where each array (generated using a different method), is stored in its own `.json` file. These `.json` files are stored in `relationships/*`.`.json`.

Each object in each array represents one origin object (i.e. the correct language for a certain question. Each object also contains an array of destination objects. Each destination object has a distance score from it to the origin, relative to all other destinations, between 0 and 1, where 1 is the language itself.

²¹Technically, one must include the `json` library when working in Python, but this is distributed along with the Python interpreter.

²²See IETF RFC #8259.

```

{
  "origin": "ara",
  "destinations": [
    {"dest": "ara", "score": 1},
    {"dest": "bel", "score": 0.39622641509433965},
    {"dest": "bre", "score": 0.3584905660377358},
    {"dest": "bul", "score": 0.6981132075471698},
    // etc. for all other languages
  ]
}

```

In the top level array, the object's origin ISO code (**origin**) is the primary key.

In the second level array of destinations, the object's origin ISO code (**dest**) is the primary key.

4.2.4.1. Test tables `relationships/test.py` is a Python 3 script which generates a random relationship table (`relationships/test.json`), in order that the JavaScript game could be tested before development of the actual linguistic distance code had been completed.

4.3. Game development

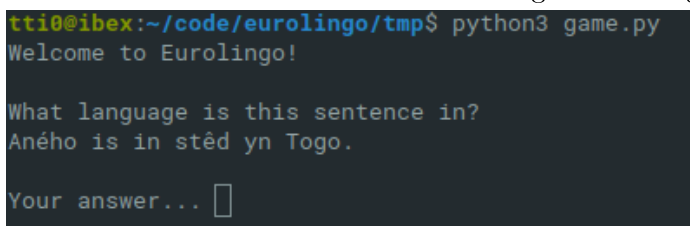
4.3.1. Approaches

I considered 3 different approaches to development of the game:

1. CLI game written in Python 3
2. Pygame game written in Python 3
3. Web-based game written in HTML5, CSS3, and JavaScript

4.3.1.1. CLI A CLI (Command Line Interface) game would be simple to implement, require minimal system resources, but necessitate that Python 3 be installed on the user's device. As I already intended to write large amounts of Python 3 code throughout the project, and also wished to make the game aesthetically pleasing, and easy to use for non-computer scientists, I decided that Approach 1 would not be appropriate.

Figure 3: CLI game mock-up



```

tti@ibex:~/code/eurolingo/tmp$ python3 game.py
Welcome to Eurolingo!

What language is this sentence in?
Aného is in stéd yn Togo.

Your answer... █

```

4.3.1.2. Pygame I had previously used the popular Pygame²³ library for Python 3, which allows for the easy creation of graphical games – giving a more pleasing GUI than a CLI game – however, found that its text-rendering mechanics are less-suited to text-heavy games, such as *eurolingo*. For this reason, and the fact that Python 3 would need to be installed on the user's device, along with the Pygame library, I chose not to follow Approach 2.

4.3.1.3. Web-based For a number of reasons, I decided that implementing Approach 3 would be most appropriate:

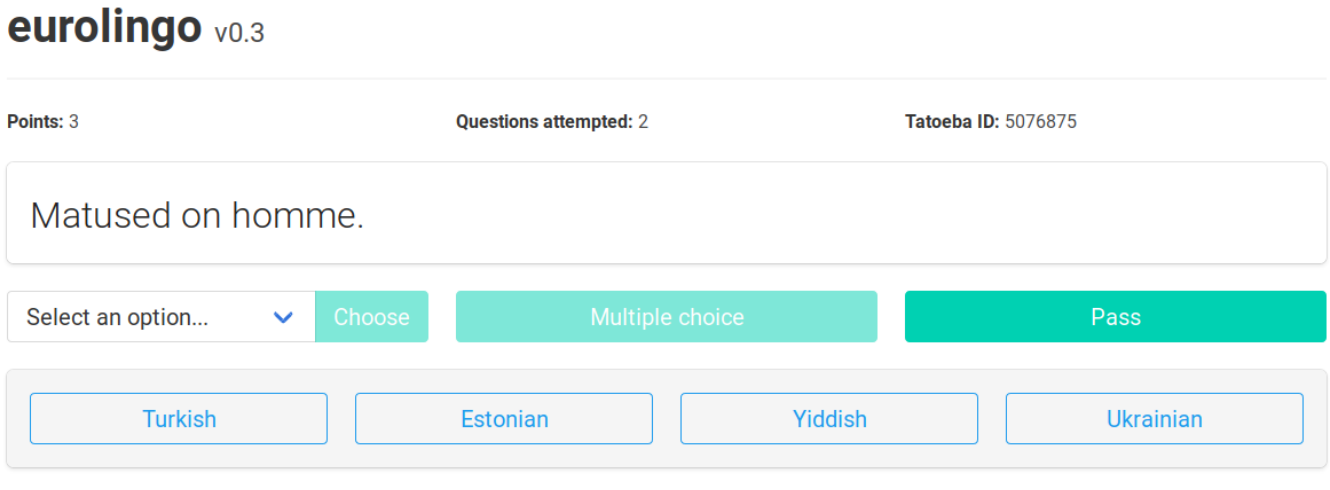
- My extensive prior experience in front-end web development
- User-friendly and well-designed GUI can easily be developed, using a CSS framework

²³<https://www.pygame.org>

- Code can run on any device with an Internet connection and modern web browser (including mobile devices), without need to install any additional software
- Game can easily be distributed

4.3.2. Implementation

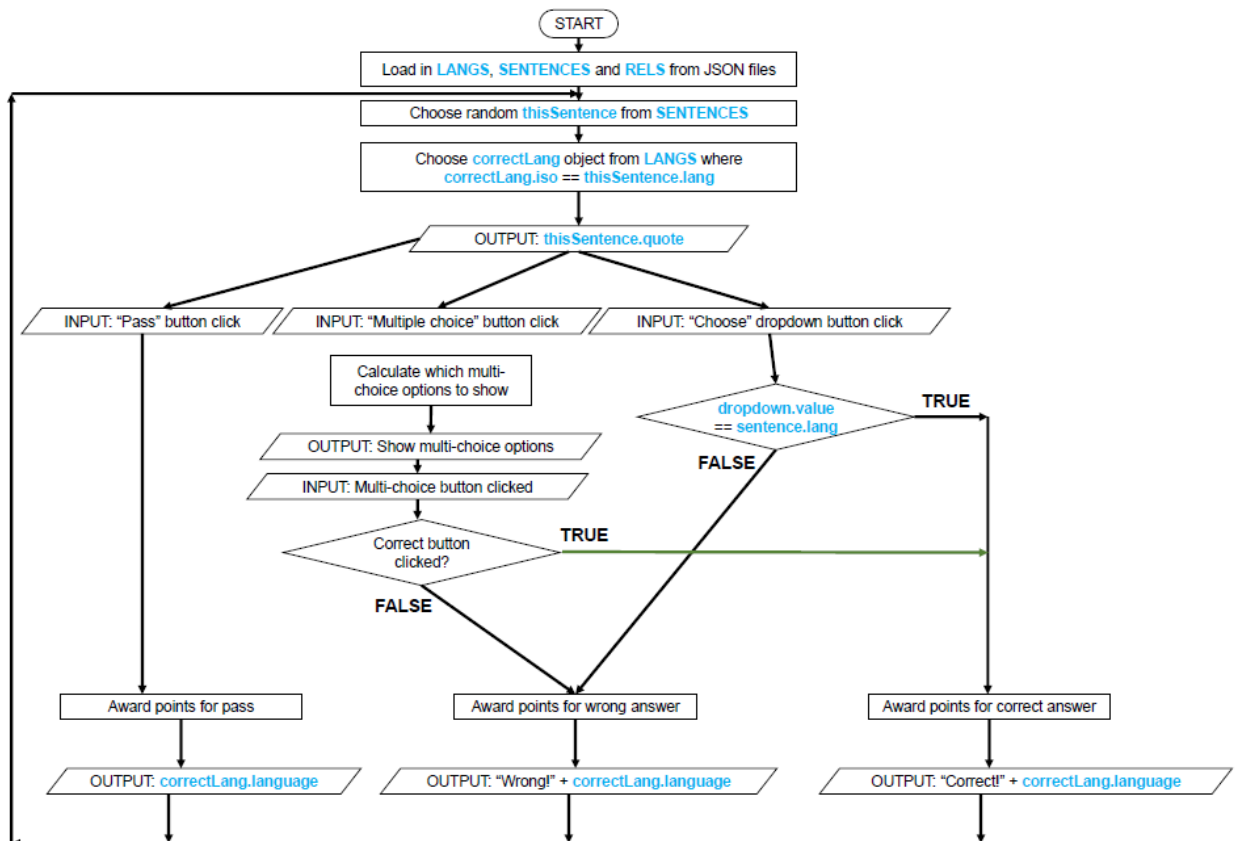
Figure 4: *eurolingo* web game interface



[eurolingo v0.3](#) by Theodore Tucker under the [GNU GPL 3.0](#).

The game consists of a single HTML file, which is located at `game/index.html`. All dependencies are located in `game/lib`, including the custom JavaScript file which drives the site: `game/lib/eurolingo – game.js`.

Figure 5: Gameplay logic flowchart



4.3.2.1. Logic commentary of `game/lib/eurolingo – game.js` This commentary excludes all UI changes.

- 1-7 On game load, initial global variables are set, and those which must be referenced later (`thisSentence` and `correctLang`) are initialised.
- 10-15 The JSON object `POINT_VALUES` stores the increments in which points are to be awarded, depending on how the user answers (correctly, with multiple choice, etc.).
- 17-22 **I used the jQuery function `$.getJSON` to load the three JSON arrays: languages, dataset of sentences, and the relationship table (see Section 4.2).**
These are nested callbacks: jQuery will ‘call back’ the code inside the function after it has made the synchronous request for the JSON file. Hence, most code for the game, needing access to all three JSON arrays, must be inside all of the callbacks. In the callbacks, the JSON returned from the request function is stored in the appropriate global variable: `LANGS`, `SENTENCES`, and `RELS`.
- 24 All code inside the jQuery `$(document).ready` method will run after the DOM has completed loading in the browser.
- 26-28 The drop-down menu is populated from the `LANGS` array, in alphabetical order of English name. The `innerHTML` of each HTML `option` is set to the English name of the language, and the `value` set to the corresponding ISO code.
- 30-35 **Definition²⁴ of code to run on `(“#btnPass”).click25`.**
Award correct points according to `POINT_VALUES`, increment `QUESTIONS` by 1, and call `ansRender`.
- 37-47 **Definition of code to run when the ‘Choose’ drop-down button is clicked.**
Retrieves current value of the drop-down, and compares to the `correctLang` for `thisSentence` (see lines 83-39).
- 49-68 **Definition of code to run when the ‘Multiple choice’ button is clicked.**
From the `RELS` array, find the first-level object whose `origin` attribute is equal to the correct answer.
Within this object, sort the possible `destinations` by highest score, and select the top four. Shuffle their order, and assign each to a HTML `button`, whose `innerHTML` is set to the English name of the language, and whose `value` is set to the corresponding ISO code.
- 71-80 **Definition of code to run when *any* of the four multiple choice option buttons are clicked.**
Get the `event.target.value` of the clicked button, so that we know which has been clicked, and compare this to the `correctLang`. Award points as appropriate.
- 83-89 **Definition of code to run when the ‘Next question’ button is clicked.**
Using UnderscoreJS’s `_.sample()` method, choose a random object from `SENTENCES`, and set as `thisSentence`.
Using UnderscoreJS’s `_.findWhere()` method, choose the object in `LANGS` whose ISO code is equal to that of `thisSentence`. Therefore, we can now use the full names of a language for display to the user, rather than ISO codes.
- 92 Simulate a click of the ‘Next question’ button once only, for the first question.
- 99-125 **Function definition: `guiQuestion(sentenceObject)`**
Initialises the GUI for a new question, based on the passed `sentenceObject`, which is displayed to the user.
- 127-151 **Function definition: `ansRender(state, correct)`**
Updates the GUI after a question has been answered, to show if the answer given was correct, incorrect, or a pass (`state`), and the correct answer (`correct`).

4.3.2.2. Libraries I used the following external libraries to save time when coding, as the aims of this project were not primarily related to web development or JSON object manipulation in JavaScript:

Bulma CSS v0.7.5²⁶ Minimalist and responsive CSS framework.

jQuery v3.4.1²⁷ JavaScript library to greatly simplify manipulation of DOM elements, removing necessity for code like:

²⁴The code inside this function will run whenever the event it is attached to fires. Hence, it only has to be defined once, and we do not have to continually listen for a button press, or repeat the code for the next question.

²⁵i.e. when the HTML element with the ID `btnPass` is clicked

```
var x = document.getElementById("btnX");
```

UnderscoreJS v1.9.1²⁸ JavaScript library containing over 100 helper functions; for example, a simple function to sort an array of JSON objects.

4.4. Relationship table generation

4.4.1. Tree object structure

For use as my pre-existing stammbaum of languages, I used resources published at MultiTree²⁹ (new version), a free website, operated by the University of Indiana, which produces diagrams of language trees based on academic studies. I utilised the following studies, which encompassed all languages in *eurolingo*:

Indo-European family Ethnologue, 2009.

Uralic family Ethnologue, 2013.

Afro-Asiatic family (inc. Semitic) Ethnologue, 2005.

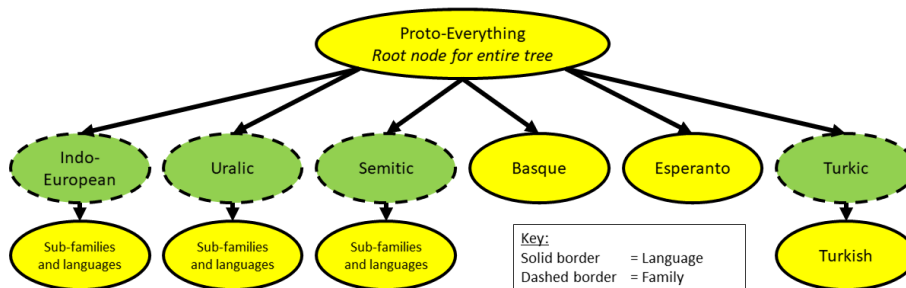
Basque Not part of any above family.

Esperanto Not part of any above family.

Turkish Not part of any above family. Only member of Turkic family in *eurolingo*.

Having identified the *eurolingo* languages present in each of the above trees, I then prepared the following meta-structure, within which I would construct the tree object:

Figure 6: *eurolingo* main tree meta-structure



The full trees underneath *Indo-European*, *Uralic*, and *Semitic* in the diagram above, are included in Appendix 1³⁰ to this report.

Based on my research, I wrote a Python 3 script to generate the relationship tables: `relationships/oop.py`.

4.4.1.1. Logic commentary of `relationships/oop.py`

4-18 **Definition of Tree object class.**

Objects of this class have two attributes: `cargo` (datatype = string) and `children` (datatype = array), and one method (`add_children(array)`).

In order to fully replicate the stammbaum structure proposed by linguists, the members of `children` can themselves be instances of `Tree` (representing the next layer of the tree, with the `cargo` as the name of the sub-family, as suggested by MultiTree (e.g. `Finnic`)), or, a languages themselves, stored as a string, containing the ISO code for the language.

71-329 **Population of languages.**

73 I initialised the variable `languages`, as an instance of `Tree`, with a `cargo` as the name of the root of the tree – the only node on Level 0: `Proto – Everything`.

²⁶<https://bulma.io/>

²⁷<https://jquery.com/>

²⁸<https://underscorejs.org/>

²⁹<http://new.multitree.org>

³⁰https://tti0.net/eurolingo/CREST_silver_report_app1.pdf

- 76-83 Following this, I was able to populate Level 1 of the array, by calling `languages'` `add_children` method, with an array of `Tree` instances and strings, identical to those shown on Level 1 of the meta-structure above. However, as I would later be referencing these children by their index number in `languages.children`, the preservation of their order, as siblings, whilst not usually significant in a tree structure, was obligatory. Therefore, for Levels 0 and 1, the sibling order remained as in the diagram above, and for Levels 2 to 10, the sibling order remains as in Appendix 1.
- 86-96 I performed the `add_children` method on `languages.children[0]`, which represents the `Tree` object with cargo *Indo-European* – the 0th sub-family.
- 97-100 Similarly, I performed the `add_children` method on `languages.children[1]`, which represents the `Tree` object with cargo *Uralic* – the 1st sub-family. Thus, the significance of sibling order is shown.
- 101-329 I repeated this process for the rest of the relevant languages and sub-families in the MultiTree datasets, although, by Level 10, the index referencing had become rather unwieldy:

```
languages.children[0].children[0].children[0].children[0].children
  [0].children[0].children[0].children[0].children[0].add_children
  ([])
```

4.4.1.2. Challenges involving the Tree object class As this project was my first major undertaking in OOP, I attempted to keep the object class as simple as possible, in order to minimise the risk of making a serious logic error. Although I was successful to this end (2 attributes and 1 method), if I were to attempt this project again, I would certainly include more attributes, and a method to more accurately delve deeper into the tree, avoiding the unintelligible references by Level 10, and decreasing the likelihood of making a transcription error. For example:

```
languages.children["Indo-European", "Germanic", "etc"].add_children([])
```

In this way, my code might be more easily understood and adapted, and fully embrace the OOP principle of encapsulation, as here, in order to assign nodes to the correct parents, I wrote out index numbers on an A3-sized printout of the MultiTree export; this took handling of objects not only out of the class, but the computer itself.

4.4.1.3. Challenges involving the MultiTree data Moreover, the data exported from MultiTree, as seen in Appendix 1, was in SVG format, and I had not planned time to reverse-engineer their front-end JavaScript, in order to obtain the raw relationship data. This would have avoided the time-consuming and fastidious task of transcription, and perhaps provided an alternative object class, more refined and suitable than my own.

4.4.2. LCA and distance computation

4.4.2.1. Logic commentary (cont.) of relationships/oop.py

20-24 The function `lenJourney(array)` allows for 1 to be returned, where an array has no items – necessary later, as such an array of nodes would imply identity between those nodes.

26-54 **Definition of `findPath(tree, destination)` function.**

This function returns an array of node `cargoes`, through which one would travel in an instance of `Tree`, to make a journey from the root to a node with `cargo= destination` (whether that be a sub-family or language). This is an example of a tree traversal algorithm, specifically, a pre-order traversal.

28 Initialisation of empty array, `path`.

29-46 **Definition of recursive `tracePath(tree, destination)` function, inside `findPath`.**

As we must return `path`, the recursive nature of `tracePath` means that the array would be constantly overwritten, as we iterate through nodes on the tree. Therefore, we retain the integrity of `path`, despite the unusual nature of the nested functions.

- 30 We append `str(tree)`, the `cargo` of the current node, to `path`. This index can be popped later, if we do not find our `destination`.
- 32 For each child of the current node...
- 33-36 If `child == destination` (thus implying that it is a string), append the child to `path`. Then raise an exception, to halt execution of the `tracePath`, and all its recursive calls, leaving `path` in its current, desirable, state.
- 37-40 As above, but for instances where the child is an instance of `Tree`, checking if `str(child)` (i.e. its `self.cargo`) is a match, where `findPath` has been called to locate a sub-family (likely as a LCA). This code must be repeated, as Python will throw an exception if we attempted to check `child.cargo`, where `child` is a string.
- 41-42 Otherwise, if the child is an instance of `Tree`, and we have not already found our `destination`, recursively call `tracePath` on the child.
- 43-46 If, after iterating through all children of the current node, including recursive calls, we have not found our `destination`, pop the `cargo` of the node string from `path`, as it is not a part of our correct journey.
- 47-50 Attempt to call `tracePath` on the `tree` and `destination` in the original function call. The `try...except...` block is necessary to allow the recursive `tracePath` to halt upon discovery, without causing the entire Python script to crash.
- 51-54 This code will run after `tracePath` has halted, either due to a discovery, and the exception, or due to exhaustion of all possible routes through the tree, without a match.
If `path` is empty, the root of the tree must have been popped by `tracePath`, hence, there is no match anywhere in the tree, so `None` is returned.
Otherwise, a match has been found, and `path` is returned.
- 56-61 **Definition of `LCA(path1, path2)` function.**
This returns a string, which is the LCA of `path1` and `path2`.³¹
I implemented Algorithm 1, to find the LCA, in Python.
- 63-68 **Definition of `calcDistance(node1, node2)` function.**
This returns an integer, which is the journey length from `node1` to `node2`.
I implemented Algorithm 2, to find the distance between the two nodes, based on their LCA, in Python.
- 331-356 **Calculation of distance for all possible language pairs.**
I opened `quotes/languages.json`, and appended the ISO code of each language to an array – `langs`. Then, for each `i` in `langs`, for each `j` in `langs`, I calculated the distance between `i` and `j` in `languages`, using the previously defined `calcDistance` function, and wrote this to `relationships/oop.json`.

4.4.2.2. Efficiency Where the depth of the tree is n , an execution of `findPath(n,*)` has efficiency $O(n)$. However, when the list of languages is increased by p , p^2 operations must be completed, giving the entire program an efficiency of $O(n + p^2)$.

I could have made this program more efficient if the relationship tables were stored truly as tables, rather than nested JSON objects. With my present system, journey length from two discrete points is stored twice, for example:

```
ara.destinations[bel] = bel.destinations[ara]
```

³¹This assumes that all node `cargoes` throughout the tree are unique. Otherwise, when we call `findPath` on the LCA (necessary to calculate distance), it will return the path to the left-most matching node in the tree, ignoring the second (which may be desired). However, for `eurolingo`, all `cargoes` were unique.

I formatted the tables in this way for ease of programming the game, avoiding multi-dimensional arrays; representation of the distance data in the style of a *mileage chart* (see below), whilst more difficult to program in JavaScript, would reduce possibility of error and somewhat simplify my code in Python.

Figure 7: A chart showing the distance, in miles, between six English cities

HULL					
60	LEEDS				
47	72	LINCOLN			
97	44	85	MANCHESTER		
66	36	47	39	SHEFFIELD	
38	24	80	71	57	YORK

Source: https://www.cimt.org.uk/projects/mepres/book8/bk8i1/bk8_1i1.htm

4.4.3. Computation of *relative* distances

Having computed distances between all language pairs, I iterated through all distances, in order to find the largest, which I discovered to be 16. This represented the pair of languages the furthest from each other, therefore, the pair which ought to have the lowest similarity score possible: 0. Again, a pair of identical languages (e.g. [eng, eng]) ought to have a similarity score of 1, and would have distance 1 returned from the tree.

In order to find the equation to use to modify my raw data to fit the similarity score model, I plotted the points on a graph of raw distance against theoretical similarity score using the computer software *Autograph v4.0.11*³², producing the following linear graph:

Figure 8: A graph showing raw distance against similarity score



Autograph computed the equation of this line to be $y = -0.0667x + 1.067$, where y is the similarity score, and x the distance. I was therefore able to modify `relationships/oop.py` to write the adjusted score to `relationships/oop.json`, instead of the raw distance.

4.4.4. Implementation into game

Finally, I adjusted `game/lib/eurolingo - game.js` to source language relationship data from `relationships/oop.json` which had been generated in the appropriate format (see Section 4.2.4) by `relationships/oop.py`.

No errors were encountered, and the OOP-generated relationship tables performed pleasingly, producing 4 reasonable multiple choice answers for all 51 origin languages.

³²<http://www.autograph-maths.com>

Part III.

Outcomes

5. Findings

5.1. Conclusion

I conclude that the linguistic distance tables I have generated are effective, and, having tested the game (with their inclusion) on numerous occasions, I do not believe that any serious error has been made. For example, the following set of multiple choice options, generated from my data, are sufficiently challenging, and would discourage guessing, given the sentence:

Figure 9: Example of appropriate multiple-choice options generated by *eurolingo* for a sentence

The screenshot shows a user interface for a language game. At the top, a text box contains the sentence "Yth eson ni ow tiberth." Below this, there is a row of four buttons: "Select an option..." (with a dropdown arrow), "Choose", "Multiple choice", and "Pass". The "Multiple choice" button is highlighted in a light green color. Below this row, there is another row of four buttons labeled "Welsh", "Irish", "Cornish", and "Breton", all in a light blue color.

Therefore, I believe that I can rather assuredly state that, where respected academic proposals regarding a hierarchy exist, quantitative computation of the relationship between two nodes leads to sensible, and potentially useful, insights.

I can also conclude that, when handling datasets as large as that provided by Tatoeba, the use of cloud computing services, of which I was highly sceptical before undertaking this project, is not only beneficial, but also essential. This is representative of the trend in computing back to ‘thin-client’-style devices, where mainframes perform mathematical operations to reveal patterns in data, only that these patterns are more precise, and hence more noteworthy, than those of sixty years ago.

5.2. Evaluation: Potential flaws

- A greater number of steps on a journey between nodes does not necessarily show disparity between those nodes. For example, (see Appendix 1a), Croatian has a distance of 3 from Serbian, however, this is only as Croatian is more diverse, and has more dialects. Hence, whilst the *dialect* itself is indeed more disparate from Serbian, the general Croatian and Serbian languages as a whole, are not.
- A journey to a ‘higher’ level of the tree represents a greater abstraction; for example, the jump from Indo-European to Proto-Everything is far more significant than Portuguese-Galician to West Iberian, yet this significance was not reflected in journey length. My **languages** object assumed that each edge on the graph had a weight of 1, whereas, in reality, this is not the case. As no data was available to more appropriately determine edge weights, I was not able to consider them, although, their inclusion would have significantly changed my algorithm, and required deeper research, beyond what has currently been published. Only after hours of research did I discover that the problem, in its simplest (edge weight = 1) form, was asked of prospective Senior Engineers for Google!
- The algorithm I used to find the LCA of two nodes, and hence calculate distance (proposed by Djidjev, et. al. (1991)), whilst the easiest for me to understand, was not the most efficient. Had I implemented *Tarjan’s off-line lowest common ancestors algorithm*, my program would have been $O(n)$.

6. Wider implications

N.B. I have considered wider implications of my work further in the Student Profile Form.

Whilst the techniques which I have developed in the development of *eurolingo* are not suitable when comparing similarity of data *ab initio*, they could be useful when comparing widely held scientific belief (e.g. predicted data) to radical new evidence. If for example, taxonomical rank would suggest that species A is the biological third-cousin twice-removed of species B, we could calculate an expected similarity score for these species, which could be compared to genetic evidence. These predictions, possible where historical evidence exists (such as a genealogical tree), or contemporary scientific understanding is close to definite, may help inform avenues of further research, guiding scientists to those where major breakthroughs may lie, rather than providing the breakthroughs themselves.