Etymology Relationships

Description: This repository contains the second project of Artificial Intelligence course from Instituto Tecnológico de Costa Rica, imparted by the professor Juan Manuel Esquivel. The project consists on some queries to search relations on a word database named "Etymological Wordnet" (http://www1.icsi.berkeley.edu/~demelo/etymwn) based by en.wiktionary.org.

Content:

Installation:

Before using the project, first you have to install all the project dependencies.

- Python 3.5 or greater, and it has to be 64-bit.
- pyDatalog:
    - Install it with pip:
      `python -m pip install --user pyDatalog`

Then, you need to download the database file at Etymological Wordnet (http://www1.icsi.berkeley.edu/~demelo/etymwn), you have to save the file on the root folder of the project. You can save where you want, but you will have to remember where is it.

Now that all dependencies are installed. You can install the project using:

```
pip install -U tec.ic.ia.p2.g07.main
```

Or you can clone the repository by:

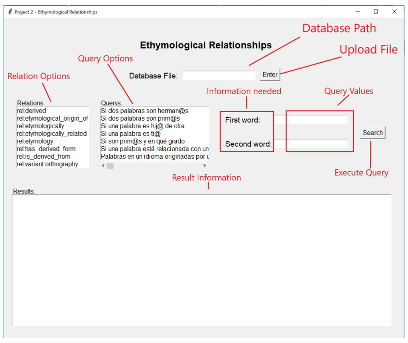
```
git clone https://github.com/mamemo/Etimology-Relationships.git
```

Usage:

When you have the directory with the repository, you need to search the repository on a console and execute the instruction:

```
python -m tec.ic.ia.p2.g07.main
```

This will display an user interface like this:



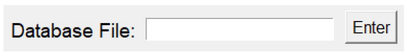
The process to execute a query will be the followin:

1. Upload a database file in **.tsv** format.
2. Select wich relations you want to execute the query on. (Multiple choice)
3. Select wich query you want to execute. (Single choice)
4. Fill the query values in the textboxes.
5. Press **Search** to execute the query.
6. Wait.
7. The results will be showing in the field below.

Now every section of the user interface will be explained:

Database

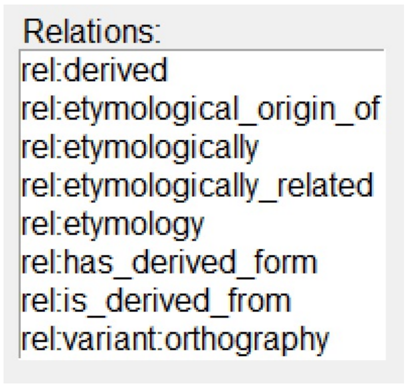
The Database objects are:



On the textbox you will write the database file path, then you will press **Enter** button to upload the file. The database file has to be on **.tsv** format. We recommend the file path at the root of the project, in that way you can write only the number of the file.

Relations

The Relation objects are:



On the list you will choose wich relations you want to consider in the query. You can choose multiple relations.
These relations have a generator word that generates another one. Given a relation, the generator and generated will change positions in the row of the database. The order will be shown in the following table:

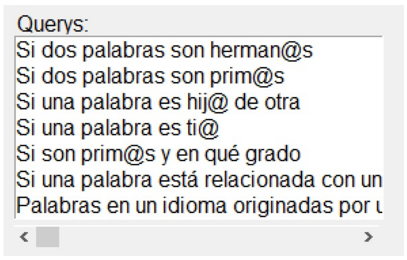
Relation	Order	Example
rel:derived	Generator -- Generated	lat: cerebellum rel:derived lat: cerebrum
rel:etymological_origin_of	Generator -- Generated	lat: formalis rel:etymological_origin_of crh: formal
rel:etymologically	Generator -- Generated	lat: guerra rel:etymologically frk: werra
rel:etymologically_related*****	Generated -- Generator	eng: aromatherapy rel:etymologically_related eng: aroma
rel:etymology	Generated -- Generator	eng: outbray rel:etymology eng: bray
rel:has_derived_form	Generator -- Generated	rom: vuneto rel:has_derived_form rom: vuneti
rel:is_derived_from	Generated -- Generator	spa: luchara rel:is_derived_from spa: luchar
rel:variant:orthography	Generated -- Generator	spa: mariguana rel:variant:orthography spa: marihuana

At the upload of the database those relations that don't follow Generated -- Generator order were switched to that order.

*****Some relations are symmetric but in the database file is already both rows, as: Generated -- Generator and Generator -- Generated. So, in these cases we upload both rows.

Querys

The Query choice objects are:


Querys:
Si dos palabras son herman@s
Si dos palabras son prim@s
Si una palabra es hij@ de otra
Si una palabra es ti@
Si son prim@s y en qué grado
Si una palabra está relacionada con un
Palabras en un idioma originadas por u

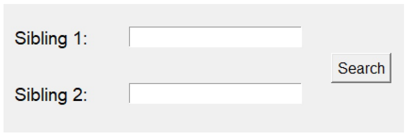
On the list you will choose wich query you want to execute. You can choose only one query at a time.
The possible queries are, respectively:

1. If two words are siblings.
2. If two words are cousins.
3. If a word is son of another one.
4. If a word is uncle of another one.
5. If two words are cousins and in which level (firs, second, etc.).

6. If a word is related to a language.
7. All the words generated by a given word in a specific language.
8. List all the languages related to a word.
9. Amount of words in common between two languages.
10. List of words in common between two languages.
11. Most relevant language for another one.
12. List of every relevatn language for another one.

Query Values

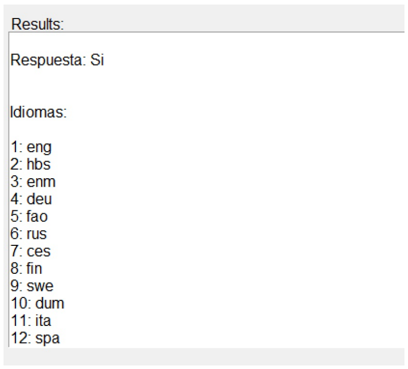
The Query values objects are:



On the textboxes you will have to fill the information asked to perform the query. When you are ready, press the **Search** button to execute the query. The Labels of the information asked will change according to the query.

Results

The Result objects are:


```
Results:

Respuesta: Si


Idiomas:

1: eng
2: hbs
3: enm
4: deu
5: fao
6: rus
7: ces
8: fin
9: swe
10: dum
11: ita
12: spa
```

On the Results field you will find the answer of the query. You may have to wait to the query to finish.

Queries' Report:

This section contains an interesting result and the implementation details for each query. All the queries can be found at Queries.py (../master/tec/ic/ia/p2/g07/Queries.py).

Siblings Query

For the siblings query this is an interesting result:

- The word *Lápiz* (pencil in spanish), gives birth to *lapicero* and *lápida* (pen and tombstone in spanish), so *lapicero* and *lápida* are siblings.

And for the implementation details:

- The query needs two words to search some siblings relation.
- To add the knowledge on pyDatalog were used an object at Relation (../master/tec/ic/ia/p2/g07/Relation.py) object.
- Using objects in the knowledge database, the upload of the database file becomes quicker.
- The last declaration is called in order to return an specific answer and don't do postprocessing on the result.
- A list of tuples are returned containing one sibling in X and one sibling in Y.
- First, we look up for every sibling, defining siblings as two words with the same parent and then, we look up if in all the siblings, one pair are the ones we are looking for.
- The query code is the following:

```
Siblings(X, Y) <= (Relation.parent[X] == Z) & (Relation.parent[Y] == Z) & (X != Y)
Siblings(X, Y, PX, PY) <= Siblings(X, Y) & (Relation.child[X] == PX) & (Relation.child[Y] == PY)
```

Cousins Query

For the cousins query this is an interesting result:

- The word *Luna* (moon in spanish), gives birth to *lunar* and *lunático* (mole and lunatic in spanish), the those words gives brith to their plurals *lunares* and *lunáticos*, so *lunares* and *lunáticos* are cousins.

And for the implementation details:

- The query needs two words to search some cousins relation.
- To add the knowledge on pyDatalog were used an object at [Relation (../master/tec/ic/ia/p2/g07/Relation.py)](../master/tec/ic/ia/p2/g07/Relation.py) object.
- Using objects in the knowledge database, the upload of the database file becomes quicker.
- The last declaration is called in order to return an specific answer and don't do postprocessing on the result.
- A list of tuples are returned containing one cousin in X, one cousin in Y and the level in Z.
- First, we define two words as cousins if their parents are siblings. Then, we try to find if the two words are direct cousins (level 1), if they are we returne the cousins with a Z of -1. If they aren't direct cousins, we search if they have an ancestor that is cousin with the another word, so at last we do a match between the ancestors of the word, all the cousins in the database and the words we are looking for. Finally, a rank is applied to the ancestors, so if the ancestor is higher on the tree, the level will grow up.
- The cousins level were obtained by logic, making the query more efficient and independent.
- The query code is the following:

```
  Son(X, PX, PY) <= (Relation.child[X] == PX) & (Relation.parent[X] == PY)
  Ancestor(X, Y) <= Son(X1, X, Y)
  Ancestor(X, Y) <= Son(X1, X, Z) & Ancestor(Z, Y)

  (Cousins2[X, Y] == rank_(group_by=X, order_by=Z)) <= Ancestor(X1, Y1) & (X == X1) & (Y == Y1) & (X == Z)

  Cousins(X, Y) <= Siblings(X1, Y1, Relation.parent[X], Relation.parent[Y])
  Cousins(X, Y, PX, PY, Z) <= Cousins(X, Y) & (Relation.child[X] == PX) & (Relation.child[Y] == PY) & (Z
== -1)
  Cousins(X, Y, PX, PY, Z) <= Ancestor(X1, Y1) & Cousins(X2, Y2) & (Relation.child[X] == X1)\
      & (Relation.child[X2] == Y1) & (Y == Y2) & (Relation.child[X] == PX) & (Relation.child[Y] == PY)\
      & (Cousins2[X3, Y3] == Z) & (X3 == X1) & (Y3 == Y1)
  Cousins(X, Y, PX, PY, Z) <= Cousins(X, Y, PY, PX, Z)
```

Son-Parent Query

For the son-parent query this is an interesting result:

- The word *Coca-cola*, gives birth to *Cocacolonization*.

And for the implementation details:

- The query needs two words to search some son-parent relation.
- To add the knowledge on pyDatalog were used an object at [Relation (../master/tec/ic/ia/p2/g07/Relation.py)](../master/tec/ic/ia/p2/g07/Relation.py) object.
- Using objects in the knowledge database, the upload of the database file becomes quicker.
- The declaration is implemented in order to return an specific answer and don't do postprocessing on the result.
- A list of tuples are returned containing the relation in X.
- We define a Son-Parent relation where in the object, the child word is the same as the first given and the parent is the same as the second one.
- The query code is the following:

```
Son(X, PX, PY) <= (Relation.child[X] == PX) & (Relation.parent[X] == PY)
```

Uncle-Nephew Query

For the uncle-nephew query this is an interesting result:

-

And for the implementation details:

- The query needs two words to search some uncle-nephew relation.
- To add the knowledge on pyDatalog were used an object at [Relation (../master/tec/ic/ia/p2/g07/Relation.py)](../master/tec/ic/ia/p2/g07/Relation.py) object.
- Using objects in the knowledge database, the upload of the database file becomes quicker.
- The last declaration is called in order to return an specific answer and don't do postprocessing on the result.
- A list of tuples are returned containing the uncle in X and the nephew in Y.
- First, we look up for every uncle-nephew, defining uncle-nephew as two words where the parent of one is sibling of the second one, then we look up if in all the uncle-nephew, one pair are the ones we are looking for.
- The query code is the following:

```
Uncle(X, Y) <= Siblings(X1, Y1, Relation.child[X], Relation.parent[Y])
Uncle(X, Y, PX, PY) <= Uncle(X, Y) & (Relation.child[X] == PX) & (Relation.child[Y] == PY)
```

Language related Query

For the language related query this is an interesting result:

- The word *rock*, is related to **russian** at the word *рок* (meaning rock).

And for the implementation details:

- The query needs two words to search some language-related relation.
- To add the knowledge on pyDatalog were used an object at Relation (../master/tec/ic/ia/p2/g07/Relation.py) object.
- Using objects in the knowledge database, the upload of the database file becomes quicker.
- A language is related with a word if at least one of the clauses below is accepted. Once a clause is resolved it doesn't going on.
- A list of tuples are returned containing the languages with word.
- We look up if the language and the word are in any relation.
- The query code is the following:

```
Lang_related(X, PX, PY) <= (Relation.child_lang[X] == PX) & (Relation.child[X] == PY)
Lang_related(X, PX, PY) <= (Relation.child_lang[X] == PX) & (Relation.parent[X] == PY)
Lang_related(X, PX, PY) <= (Relation.parent_lang[X] == PX) & (Relation.child[X] == PY)
Lang_related(X, PX, PY) <= (Relation.parent_lang[X] == PX) & (Relation.parent[X] == PY)
```

Language and origin Query

For the language and origin query this is an interesting result:

- The word *rock*, gives birth to *rockettara* in **italian**. (meaning someone who likes rock music).

And for the implementation details:

- The query needs two words to search some siblings relation.
- To add the knowledge on pyDatalog were used an object at Relation (../master/tec/ic/ia/p2/g07/Relation.py) object.
- Using objects in the knowledge database, the upload of the database file becomes quicker.
- A word is originated by another one, if the latter is an ancestor of the first.
- First, we look up if the language of the originated word is the one we are searching for. Then, we get all ancestors of the database and we look up if any word is generated by the given.
- The query code is the following:

```
Lang_and_origin(X, PX, PY) <= (Relation.child_lang[X] == PX) & Ancestor(\
  X1, Y1) & (Relation.child[X] == X1) & (Y1 == PY)
```

List related languages Query

Using the list related language query this is an interesting result:

- The word *rock* is related with at least these languages:
    - Faroese
    - Middle English
    - Italian
    - Middle Dutch
    - Czech
    - Russian
    - Swedish
    - Finnish
    - German
    - English
    - Spanish

And for the implementation details:

- The query needs one word to search the relation.
- To add the knowledge on pyDatalog were used an object at Relation (../master/tec/ic/ia/p2/g07/Relation.py) object.
- Using objects in the knowledge database, the upload of the database file becomes quicker.
- The language related query was used to reuse code.
- The languages of every relation are inside lists in the variable *answer*. Those lists stores the specific language at the second index.
- First, we asked for the language-related query to fill an empty variable *Y*. *Y* will be taken as any language, meaning to the query the following statement: "Search if ANY language is related to the word". Then, we iterate on the solution to get all the languages without repeated ones.
- The query code is the following:

```
answer = language_related_query(Y, word)
languages = set([y for x, y in answer])
```

Count common words Query

For the count-common-words query this is an interesting result:

- Between the *spanish* and *finnish* languages, at least 3 word in common were found:
    - canasta
    - tapas
    - ajo

And for the implementation details:

- The query needs two languages to search some common-words relation.
- To add the knowledge on pyDatalog were used an object at [Relation (../master/tec/ic/ia/p2/g07/Relation.py)](../master/tec/ic/ia/p2/g07/Relation.py) object.
- Using objects in the knowledge database, the upload of the database file becomes quicker.
- Words in common between two languages are the ones which are present in any relation with different languages.
- A list of tuples are returned containing in X the first relation, in Y the second relation and in Z the word.
- The same clauses were used between count and list common words queries.
- The query code is the following:

```
Common_words(X, Y, PX, PY, Z) <= (Relation.child[X] == Relation.child[Y]) & (X != Y) & \
  (Relation.child_lang[X] != Relation.child_lang[Y]) & (Relation.child_lang[X] == PX) & (\
  Relation.child_lang[Y] == PY) & (Relation.child[X] == Z)
Common_words(X, Y, PX, PY, Z) <= (Relation.parent[X] == Relation.parent[Y]) & (X != Y) & \
  (Relation.parent_lang[X] != Relation.parent_lang[Y]) & (Relation.parent_lang[X] == PX) & (\
  Relation.parent_lang[Y] == PY) & (Relation.parent[X] == Z)
```

```
answer = common_words_query(language1, language2)
words = len(set([z for x, y, z in answer]))
```

List common words Query

For the list-common-words query this is an interesting result:

- Between the *spanish* and *finnish* languages, at least 3 word in common were found:
    - canasta
    - tapas
    - ajo

And for the implementation details:

- The query needs two languages to search some common-words relation.
- To add the knowledge on pyDatalog were used an object at [Relation (../master/tec/ic/ia/p2/g07/Relation.py)](../master/tec/ic/ia/p2/g07/Relation.py) object.
- Using objects in the knowledge database, the upload of the database file becomes quicker.
- Words in common between two languages are the ones which are present in any relation with different languages.
- A list of tuples are returned containing in X the first relation, in Y the second relation and in Z the word.
- The same clauses were used between count and list common words queries.
- The query code is the following:

```
Common_words(X, Y, PX, PY, Z) <= (Relation.child[X] == Relation.child[Y]) & (X != Y) & \
  (Relation.child_lang[X] != Relation.child_lang[Y]) & (Relation.child_lang[X] == PX) & (\
  Relation.child_lang[Y] == PY) & (Relation.child[X] == Z)
Common_words(X, Y, PX, PY, Z) <= (Relation.parent[X] == Relation.parent[Y]) & (X != Y) & \
  (Relation.parent_lang[X] != Relation.parent_lang[Y]) & (Relation.parent_lang[X] == PX) & (\
  Relation.parent_lang[Y] == PY) & (Relation.parent[X] == Z)
```

```
answer = common_words_query(language1, language2)
words = set([z for x, y, z in answer])
```

List Relavant Languages Query

For the list-relevant-language query this is an interesting result:

- The relevant languages for *Old English* are:
    - Latin: 0.9273%.
    - Old Norse: 0.1291%
    - Old French: 0.1174%
    - Germanic: 0.047%
    - Ancient Greek: 0.0235%
    - English: 0.0117%
    - Gml: 0.0117%
    - Osx: 0.0117%
    - Gmw: 0.01173%

And for the implementation details:

- The query needs a language to search the relevant-languages relation.
- To add the knowledge on pyDatalog were used an object at [Relation (../master/tec/ic/ia/p2/g07/Relation.py)](../master/tec/ic/ia/p2/g07/Relation.py) object.
- Using objects in the knowledge database, the upload of the database file becomes quicker.
- A language is relevant to another one if the first one generates a word in the second one.
- A list of tuples are returned containing in X the relation, in Y the language and in Z the number of appearances.
- We use the *rank_* function given by pyDatalog to create a rank meaning the number of appearances.
- We called the Relevat_lang query and then process the answer, we iterate over the answer to look up for the first mention of the language containing the total number of appearances. Then we calculate the percentages for every languages and sort them by its value.
- The query code is the following:

```
(Relevant_lang[X, Y, PX] == rank_(group_by=Y, order_by=Z)) <= (Relation.child_lang[X] == PX) & (\
     Relation.parent_lang[X] == Y) & (Relation.parent_lang[X] == Z)
```

```
answer = relevant_language_query(language)
amount = dict()
for i in answer:
    if i[1] not in amount.keys():
        amount[i[1]] = i[2]+1
total = sum(amount.values())
percentages = [[k, v*100/total] for k, v in amount.items()]
percentages = sorted(percentages, key=lambda x: x[1])[::-1]
```

Most Relavant Language Query

For the most-relevant-language query this is an interesting result:

- The most relevant language for *Old English* is **latin** with a **0.9273%** of words generated in the latter language.

And for the implementation details:

- The query needs a language to search the relevant-languages relation.
- To add the knowledge on pyDatalog were used an object at [Relation (../master/tec/ic/ia/p2/g07/Relation.py)](../master/tec/ic/ia/p2/g07/Relation.py) object.
- Using objects in the knowledge database, the upload of the database file becomes quicker.
- A language is relevant to another one if the first one generates a word in the second one.
- A list of tuples are returned containing in X the relation, in Y the language and in Z the number of appearances.
- We use the *rank_* function given by pyDatalog to create a rank meaning the number of appearances.
- We called the Relevat_lang query and then process the answer, we search for the most relevant located at the beginning, if the most relevant is the same as the language prompted, we look for the second most relevant language.
- The query code is the following:

```
(Relevant_lang[X, Y, PX] == rank_(group_by=Y, order_by=Z)) <= (Relation.child_lang[X] == PX) & (\
     Relation.parent_lang[X] == Y) & (Relation.parent_lang[X] == Z)
```

```
percentages, answer = percentages_relevant_language_query(language)
perc = percentages[0]
all_perc = percentages
if perc[0] == language:
    percentages.remove(perc)
    perc = percentages[0]
```

Work Distribution

For these project 3 collaborators were present:

- Valeria Bolaños (V)
- Gloriana Flores (G)
- Mauro Méndez (M)

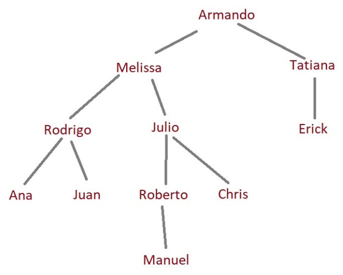
The distribution of the work was the following:

- User Interface: G
- Queries:
    - 1. M
        - 2. M
        - 3. M

        - 4. M
        - 5. M
        - 6. G
        - 7. G
        - 8. G
        - 9. V
        - 10. V
        - 11. V
        - 12. V
- Database upload: V
- Report: Everyone

Unit Testing

For this project we created unit testing for every query. All the tests can be located at [test_queries.py (../master/tec/ic/ia/p2/g07/tests/test_queries.py)](../master/tec/ic/ia/p2/g07/tests/test_queries.py). For the first 12 relations the following tree were used:


Armando
Melissa
Tatiana
Rodrigo
Julio
Erick
Ana
Juan
Roberto
Chris
Manuel