Graphical Representation of Semantic Distance Between Words

Daniel Vega-Myhre, Aidan Leuck, Carson Thompson, Zach Sherwood

November 30, 2020

Abstract

WordNet is a lexical database of the English language that was created to organize different types of words (nouns, verbs, adjectives, adverbs) into groups of cognitive synonyms called synsets. These synsets are then grouped and linked by their semantic and lexical relations into a network. This network of words is connected based upon concepts such as hyponymy, antonymy, cause-effect and several others that might connect several semantic relations to one another. For this project we designed and implemented an algorithm in Haskell, a purely functional programming language, to parse the hierarchical Wordnet ouput for the hyponym of a user specified word, into a data structure which represents an undirected graph of the semantic relationships between words in that hyponym. We then display this graph and visualize the semantic distance (shortest path) between two words in the hyponym specified by the user. For the purposes of this project, we have defined the semantic distance between two words to be the shortest path between the nodes in the semantic graph of the hyponym.

1 Introduction to Haskell

Our initial reason for choosing Haskell as the primary language for this project was due in part to its range in development options, the brevity of the code written needed to create and different attributes it carries like Purity, which can be defined as the concept of pure functions and immutable data. A pure function in this instance refers to a function that when given the correct input will always give the same output. This is highly desirable in any programming language because it makes the code easy to read, logically evaluate and easy to test. In accessing a database, one will have to insure that some of their functions are impure in order to accurately grab data from a dynamic data structure. Good development in Haskell will demand then that one might begin with a pure function and compose it to an impure function.

Haskell is a functional language in that instead of giving a program a series of imperative steps to perform and execute, a developer will outline what a function is and what it calculates. This requires one to develop by preferably building more complex functions by combining simple functions together. Haskell uses lazy evaluation in its code execution instead of the strict evaluation that is common in most programming languages. This means that if we create a function add x y = x + x and try to implement it with add 10 (57/0), instead of evaluating all of the arguments passed into the function Haskell will give the result of 20.Thus, only evaluating what operations the function had previously defined. Haskell is statically scoped meaning that for defining variables, the program will check local definitions before looking at the global definitions for an over-riding assignment.

Syntactically, Haskell has specific characteristics that differentiate itself from other programming languages. For example, Haskell has type inference which allows the user to write code without specifying type. One can specify a type for a function by giving the :: operator, 'z' :: Char translates to z having type char. Another aspect of Haskell that gives it unique attributes is shown by declaring functions. To declare, one must specify what type the function will take in and what it will return. For example, RemoveNonUppercase :: [Char] $\rightarrow$[Char] specifies a char type for both the parameter and result [1] . The last type written will always specify the return type. Thus, RemoveNonUppercase :: [Char] $\rightarrow$[Char] $\rightarrow$[Char] $\rightarrow$[Char] will require three char types as a parameter. Haskell also does not treat variables as mutable values but instead is seen as a definition of the variable. If one assigns $x = 5$, x will be defined as 5 and will be immutable for that program's scope.

In as far as analyzing the pit falls or weak points of Haskell, our group had significant issues in several different aspects of this project. The first was trying download a uniform environment for development for different operating systems. Using the IDE Intellij proved difficult to update and maintain libraries and dependencies with little help from troubleshooting documentation. For windows users, one would be better off installing a Linux based virtual machine for all Haskell development. The most difficult aspect for every member of our group was trying to translate an imperative procedural paradigm into a functional paradigm especially in the case of our algorithm's application. Haskell also does not seem to have an approachable continuity with the range of libraries they do offer and their respective quality. For example, trying to find a working library for a simple task like parsing a CSV was incredibly difficult.

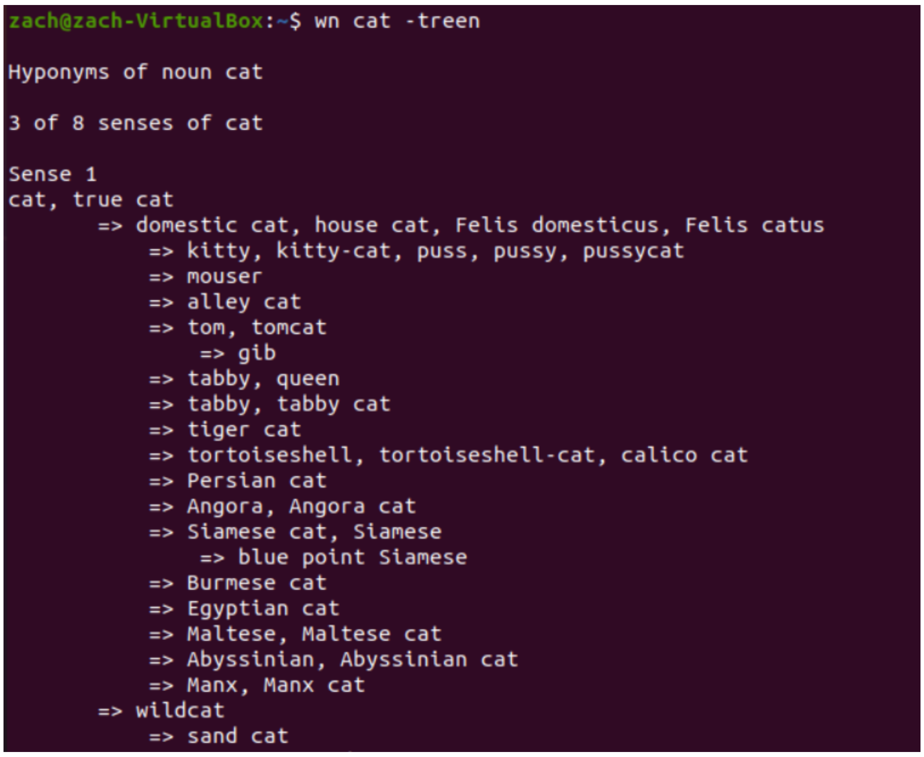
Despite the difficulties included in: actually being able to use Haskell in a development environment, learning how to appropriately code in a functional paradigm and learning the several syntactic differences present in the language Haskell does seem to lend itself to many modern uses. Facebook uses Haskell to reduce spam, malware [2], Microsoft has created a framework called Bond [3] that is used for data manipulation and several other companies have teams developing Haskell for various purposes including security, financial tech, hardware back end, block chain and big data manipulation. Working with this language helped develop each members ability to work in the functional language paradigm and was a powerful reminder of the number of ways you could programmatically solve an issue or provide a service. Through the experience of creating this project, our group found that while Haskell is a valid vehicle for programming, careful consideration would need to be applied if any member were to consider using it in the future.

2 Project Description

Our goal for this project was to accurately access the Wordnet database via the command line to grab the output of a words hyponym (the specific term used to designate a member of a class (i.e., clay is a type of soil)). This first word acts as a category for showing the semantic difference of the next two requested words required from the user. If the following words exist within the hyponym of the first category, a graph will be displayed showing the shortest semantic distance between the two words. In the following sections we will outline how we were able to build the data structure that held the queried output from Wordnet, how we determined the semantic difference and how we displayed that information for the user.

2.1 Algorithm

We decided to query Wordnet by specifically requesting the displayed hypnoym of the word and a recursive call that searches included hyponyms of each word. Each provided sense in the output is a determinant of a possible meaning of the word creating a synset. For this project we decided to only parse the first sense. In Figure 1 provided below, we see a queried example of the noun cat.



```
zach@zach-VirtualBox:~$ wn cat -treen

Hyponyms of noun cat

3 of 8 senses of cat

Sense 1
cat, true cat
       => domestic cat, house cat, Felis domesticus, Felis catus
           => kitty, kitty-cat, puss, pussy, pussycat
           => mouser
           => alley cat
           => tom, tomcat
               => gib
           => tabby, queen
           => tabby, tabby cat
           => tiger cat
           => tortoiseshell, tortoiseshell-cat, calico cat
           => Persian cat
           => Angora, Angora cat
           => Siamese cat, Siamese
               => blue point Siamese
           => Burmese cat
           => Egyptian cat
           => Maltese, Maltese cat
           => Abyssinian, Abyssinian cat
           => Manx, Manx cat
       => wildcat
           => sand cat
```

Figure 1: Wordnet output

First, we parse the Wordnet output line by line and build an ordered list of tuples, which contains the first word on that line and the number of leading spaces on that line (in cases where there is more than one word in a single line, we chose to just use the first word in order to simplify the problem). For example, in Figure 1 above, the list of tuples would begin [(”cat”,0), (”domestic”,7), (”kitty”, 11), (”mouser”,11) ...]

Next, we use this data to build an adjacency list which represents an undirected graph of semantic relationships between words. We implemented our adjacency list as a hashmap where each key is a single word and its value is a list of its neighbor nodes/words. To accomplish this, we wrote a recursive algorithm which compares the number of leading spaces for the current word with that of the previous word, and based on the results of that comparison has branching logic which determines how to add the current word into the adjacency list. We also use a stack to push/pop nodes to/from as we traverse into deeper/shallower levels of the Wordnet output hierarchy; the goal of this is to construct the stack such that it has an invariant property that the element at the top of the stack is always the node that we are currently adding neighbors to in the adjacency list.

The algorithm can be summarized in the following steps:

1. Base case: If list of remaining words to be added to the adjacency list is empty, return the adjacency list.

2. Remove next element from list of words to be added. This element is a tuple containing the word and the associated number of leading spaces. The number of leading for the current word is compared to that of the previous word, and logical branching occurs based on the results of this comparison, which can result in 3 possible scenarios:

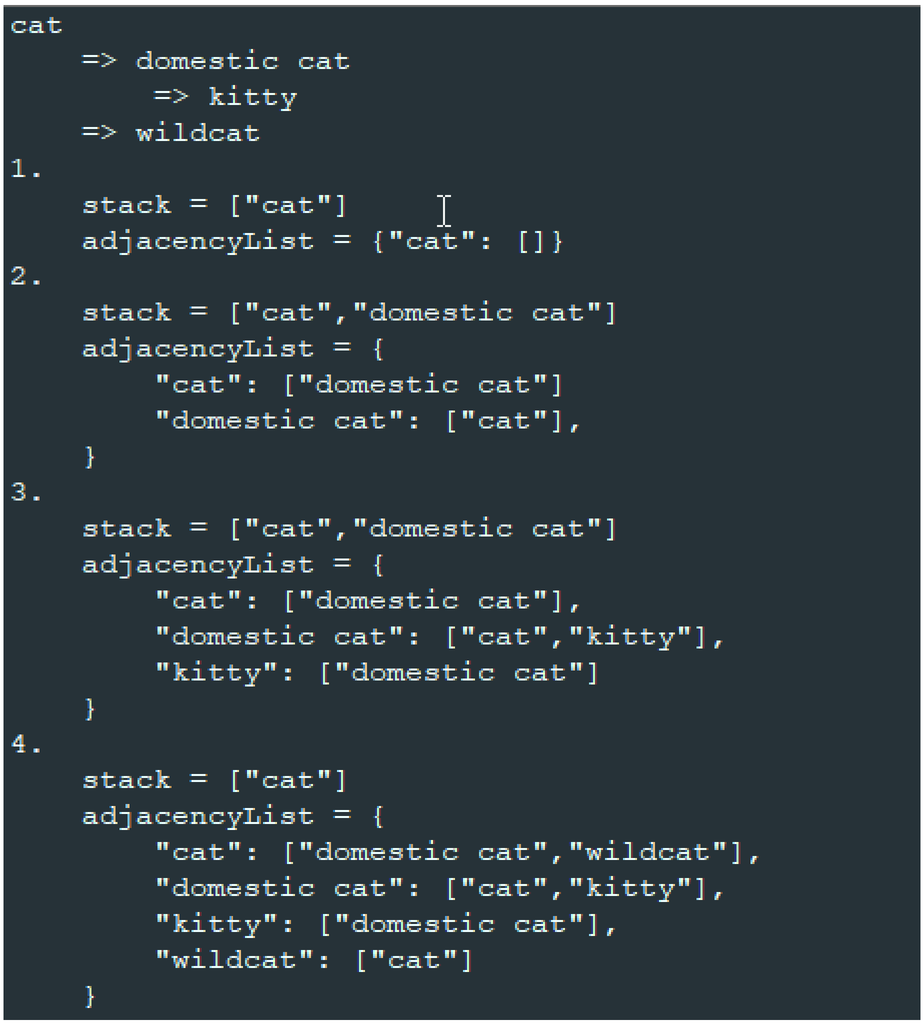
3. Case 1: the current line's number of leading spaces is greater than that of the previous line (i.e. we are going down a level in the Wordnet output hierarchy). In this case, the "parent" node will be the element at the top of the stack, and we add the current word as neighbor to the parent node in the adjacency list. Finally, we push the current word to the stack, as it will potentially be the new parent for subsequent words.

4. Case 2: the current line's number of leading spaces is less than that of the previous line (i.e. we are going up a level in the Wordnet output hierarchy). In this case, we first pop the top element from the stack and discard it. Now the new top element of the stack contains the correct element we should add this new word as a neighbor to in the adjacency list. Finally, we push the current word to the stack, as it will potentially be the new parent for subsequent words.

5. Case 3: the current line's number of leading spaces is equal to that of the previous line (i.e. we are still at the same level in the Wordnet output hierarchy). In this case, we first pop the top element from the stack. We had pushed this element to the stack previously in case the next line hits Case 1 (going down a level in the Wordnet hierarchy, which would use this element as the parent to add a word as a neighbor to in the adjacency list). However, since we are still at the same level, we pop this element and discard it, and now the new top element of the stack is the correct node we will be adding a neighbor to in the adjacency list. Finally, we push the current word to the stack, as it will potentially be the new parent for subsequent words.

6. All possible cases end with a recursive call with an updated set of parameters after completing their operations (i.e. tail recursion).

Below we have traced an example of the data structures through the course of the algorithms execution while parsing the following Wordnet output:



```
cat
    => domestic cat
        => kitty
    => wildcat
1.
    stack = ["cat"]
    adjacencyList = {"cat": []}
2.
    stack = ["cat","domestic cat"]
    adjacencyList = {
        "cat": ["domestic cat"]
        "domestic cat": ["cat"],
    }
3.
    stack = ["cat","domestic cat"]
    adjacencyList = {
        "cat": ["domestic cat"],
        "domestic cat": ["cat","kitty"],
        "kitty": ["domestic cat"]
    }
4.
    stack = ["cat"]
    adjacencyList = {
        "cat": ["domestic cat","wildcat"],
        "domestic cat": ["cat","kitty"],
        "kitty": ["domestic cat"],
        "wildcat": ["cat"]
    }
```

Figure 2: Data Structures throughout algorithm execution

2.2 Semantic Distance

In order to understand the graphical output of our project one must understand the idea of semantic distance and its role in our project. Semantic distance is generally defined as how similar two words are to each other [4]. Similarity of a word is calculated based off of the meaning of the two words. The distance is a metric to measure the similarity of these two words. The lower the semantic distance, the more similar the words are to one another.

To calculate semantic distance for our project, WordNet was used in order to do the actual calculations for determining the semantic distance. WordNet's output output the information that was needed in order to create an adjacency list as stated above. Once the adjacency list was built we could calculate semantic distance between the two by counting the number of nodes between one word and the word we were comparing.

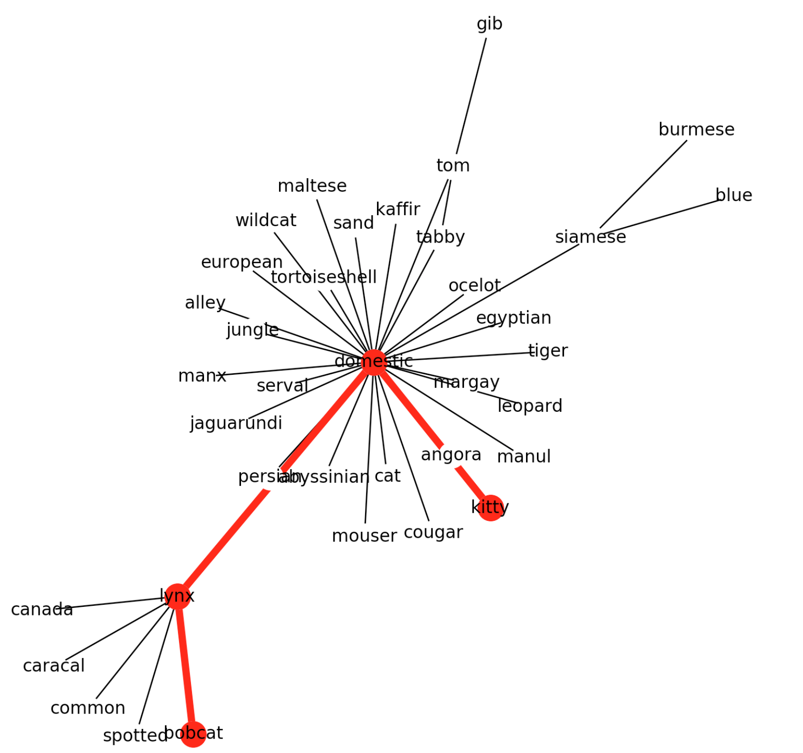
2.3 Visualizing

The method that was used to output data from our adjacency list was the graphing tool NetworkX [5], a wrapper around the industry standard data visualization library Matplotlib. This is a Python library as opposed to a Haskell library. Discussion on why NetworkX was used can be seen below, but we were able to use Haskell's system process library in order to shell out to a Python script and display the graph.

Python was used as opposed to Haskell to display the results because of its ease of implementation, without NetworkX we would have had to write a complex graphing algorithm using linear algebra and springs that was beyond the scope of the project. For that reason we decided to not use Haskell to create our visual output.

In order to display results, the adjacency list was exported to a text file using Haskell and then read into a python file. Although NetworkX is normally used for Network analysis it had a specific feature that allowed for a more understandable output. The feature in question is an implementation of springs, which allows us to achieve the clustering effect seen in our graph output. Clusters represent words that are more closely related to another visually, where closer distances represents semantic similarity.

Below we can see the resulting graph, Displaying the hyponym of cat and the semantic distance between bobcat and kitty.



3 Evaluation

To verify that our program is functioning correctly we first would analyze the output given from Wordnet in relation to the category. After counting the number of spaces and analyzing which words are neighbors and children of the category (Figure 1). We then verify that graphical output matches the adjacency list created by Haskell parsing the Wordnet output (Figure 2).

References

[1] Miran Lipovača. Learn you a haskell for great good! `http://www.http://learnyouahaskell.com/`.

[2] Simon Marlow. Fighting spam with haskell. `https://engineering.fb.com/2015/06/26/security/fighting-spam-with-haskell/?ref=hackernoon.com/`, 2015.

[3] Microsoft. Bond. `https://github.com/microsoft/bond?ref=hackernoon.com/`, 2020.

[4] Graeme Hirst. Semantic distance. `http://www.cs.toronto.edu/~gh/research-pages/research-semantic-distance.html`.

[5] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux,

Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.