CS 61B

Projects / Project 2B: Ngordnet (Wordnet)

Project 2B: Ngordnet (Wordnet)

FAQ

Each assignment will ha[illegible]t the top. You can also access it by adding "/faq" to the end of the URL. The FAQ for Project 2B is located here.

Checkpoint & Design Doc Due 03/15/2024

Coding Due 04/01/2024

In this project, you'll complete your implementation of the NGordnet tool.

As this is a quite new project, there may be occasional bugs or confusion with the spec. If you notice anything of this sort, please post on Ed.

> DANGER
>
> **IMPORTANT NOTE:** After you read the 2B spec, you may be tempted to start coding. Don't do this!
>
> Before implementing ANY code for 2B, please read the 2C spec, as your design may change depending on 2C. You can find it here.
>
> Then, complete the Project 2B/C: Checkpoint and Design Document before starting coding.

Design Notes

When designing your project, think about all of the requirements in advance. Planning ahead will ensure you don't need to rewrite all of your code when you get to certain points in the project.

If you find yourself copy-pasting or repeating a lot of the same code you've already written, there is probably an opportunity to reuse it directly, or slightly modify it so you don't have to repeat yourself as often.

Project Setup

DANGER

THE SETUP FOR THIS PROJECT IS DIFFERENT THAN THE OTHER LABS / PROJECTS. PLEASE DO NOT SKIP THIS STEP!

Skeleton Setup

1. Similar to other assig ss, run `git pull skeleton main` to get the skeleton code for this project.
2. Download the `data` ct using this link and move them into your `proj2b` folder on the same le

Once you are done, your `proj2b` directory should look like this:


```
proj2b
├── data
│   ├── ngrams
│   └── wordnet
├── src
├── static
├── tests
```

Copy

Getting Started

WARNING

IMPORTANT NOTE: You should *really* complete **Project 2B/C: Checkpoint** first before starting coding, or even designing your project. We think this would be helpful for your understanding of the project. We will also require you to submit a design document to Gradescope. More details about the design document can be found in Deliverables and Scoring..

TASK

Complete **Project 2B/C: Checkpoint**

After finishing the checkpoint, complete **Design Document**

This part of the project is designed for you to come up with efficient and correct design for your implementation. The design you come up with will be very important to handle these cases. Please read 2B & 2C spec carefully before starting your design document.

The course staff has created a couple of introductory videos to the project and the starter code available here. Bear in mind we have changed the structure of the project so some information might be outdated!

We've also created two wonderful tools that you can (and should!) use to explore the dataset, see how the staff solution behaves for specific inputs, and get expected outputs for your unit tests (see Testing Your [illegible]em here, as well as in other relevant parts of the spec.

- Wordnet Visualizer: [illegible] understanding how synsets and hyponyms work and testing different wor[illegible]r potential test case inputs. Click on the "?" bubbles to learn how to use the various features of this tool!
- Staff Solution Webpage: Useful for generating expected outputs for different test case inputs. Use this to write your unit tests!

TASK

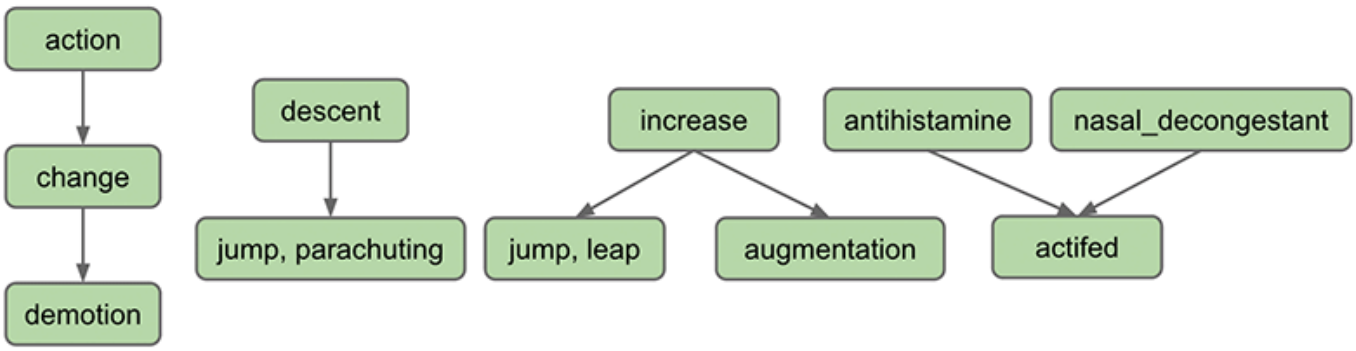
Read through entire 2B/C spec and complete **Project 2B/C: Checkpoint**

Read through entire 2B/C spec and complete **Design Document**

Using the WordNet Dataset

Before we can incorporate WordNet into our project, we first need to understand the WordNet dataset.

WordNet is a "semantic lexicon for the English language" that is used extensively by computational linguists and cognitive scientists; for example, it was a key component in IBM's Watson. WordNet groups words into sets of synonyms called synsets and describes semantic relationships between them. One such relationship is the is-a relationship, which connects a **hypo**nym (more specific synset) to a **hyper**nym (more general synset). For example, "change" is a **hypernym** of "demotion", since "demotion" is-a (type of) "change". "change" is in turn a **hyponym** of "action", since "change" is-a (type of) "action". A visual depiction of some hyponym relationships in English is given below:



Each node in the graph above is a **synset**. Synsets consist of one or more words in English that all have the same meaning. For example, one synset is "jump, parachuting" , which represents the act of descending to the ground with a parachute. "jump, parachuting" is a hyponym of "descent", since "jump, parachuting" is-a "descent".

Words in English may belong to multiple synsets. This is just another way of saying words may have multiple meanings. [illegible] word "jump" also belongs to the synset "jump, leap" , which represents the n[illegible]on of jumping (e.g. a jump in attendance) rather the literal meaning of jump f[illegible]set (e.g. a jump over a puddle). The hypernym of the synset "jump, leap" is "i[illegible]mp, leap" is-an "increase". Of course, there are other ways to "increase" som[illegible]e, we can increase something through "augmentation," and thus it is no surprise that we have an arrow pointing downwards from "increase" to "augmentation" in the diagram above.

Synsets may include not just words, but also what are known as collocations. You can think of these as single words that occur next to each other so often that they are considered a single word, e.g. nasal_decongestant . To avoid ambiguity, we will represent the constituent words of collocations as being separated with an underscore _ instead of the usual convention in English of separating them with spaces. For simplicity, we will refer to collocations as simply "words" throughout this document.

A synset may be a hyponym of multiple synsets. For example, "actifed" is a hyponym of both "antihistamine" and " nasal_decongestant", since "actifed" is both of these things.

> **INFO**
> If you're curious, you can browse the Wordnet database by using the web interface , though this is not necessary for this project.

Hyponyms (Basic Case)

Setting up a HyponymsHandler

1. In your web browser, open the `ngordnet.html` file in the `static` folder. As a refresher, you can find how to do that here in bullet point 1. You'll see that there is a new button: "Hyponyms". Note that there is also a new input box called `k` .

2. Try clicking the Hyponyms button. You'll see nothing happens (and if you open the developer tools feature of your web browser, you'll see that your browser shows an error).

In Project 2B, your primary task is to implement this button, which will require reading in a different type of dataset and synthesizing the results with the dataset from Project 2A. Unlike 2A, it will be entirely up to you to decide what classes you need to support this task.

1. Edit the file called `HyponymsHandler` to simply return the word "Hello!" when the user clicks the Hyponyms button in the browser. You'll need to make the `HyponymsHandler` class extend the `NgordnetQueryHandler` class. See your other Handler classes for examples. Make sure when you register your handler that you use the string "hyponyms" as the first argument to the `register` method, and not "hyponym".
2. Start by opening you `Main.java` file.
3. Once you've modifie ur new handler is registered to handle hyponyms requests, start up `M` g the Hyponyms button in your web browser again. You should see text a Hello".

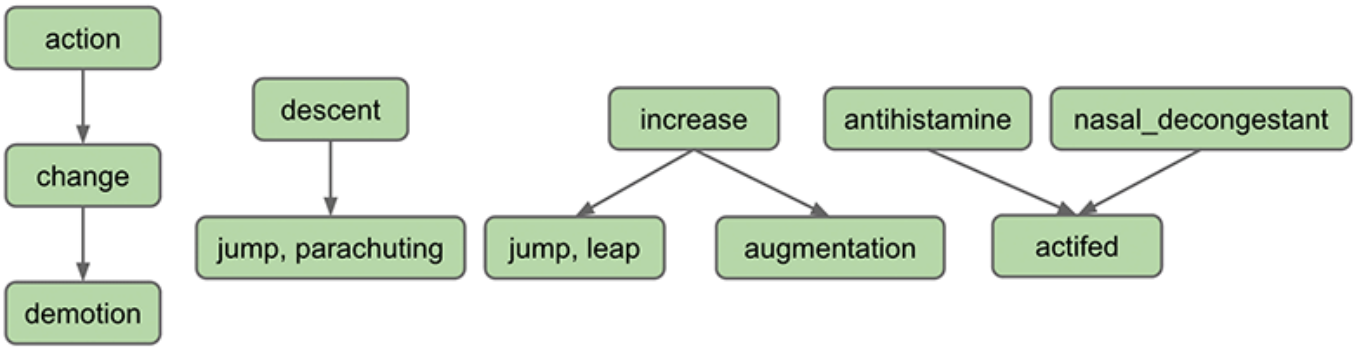
> **INFO**
> If you see some error like "Could not load file `some_file_here.txt` ", it probably means that your project is not set up correctly. Be sure that you have the same structure as stated in the Project Setup section.

Hyponyms Handler (Basic Case)

Next, you'll create a partial implementation of the Hyponyms button. For now, this button should:

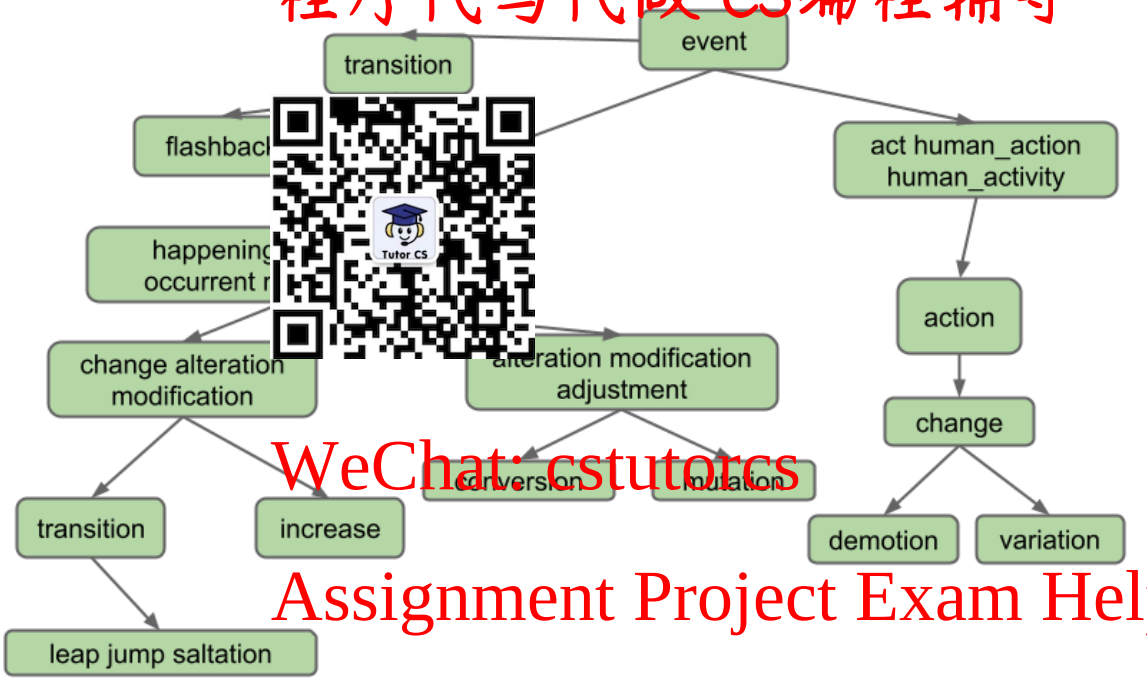
- Assume that the "words" entered is only a single word.
- Ignore startYear, endYear, and k.
- Return a string representation of a list of the hyponyms of the single word, including the word itself. The list should be in **alphabetical order**, with **no repeated words**.

For example, suppose the WordNet dataset looks like the diagram below (given to you as the input files `synsets11.txt` and `hyponyms11.txt`). Suppose that the user enters "descent" and clicks on the Hyponyms button.

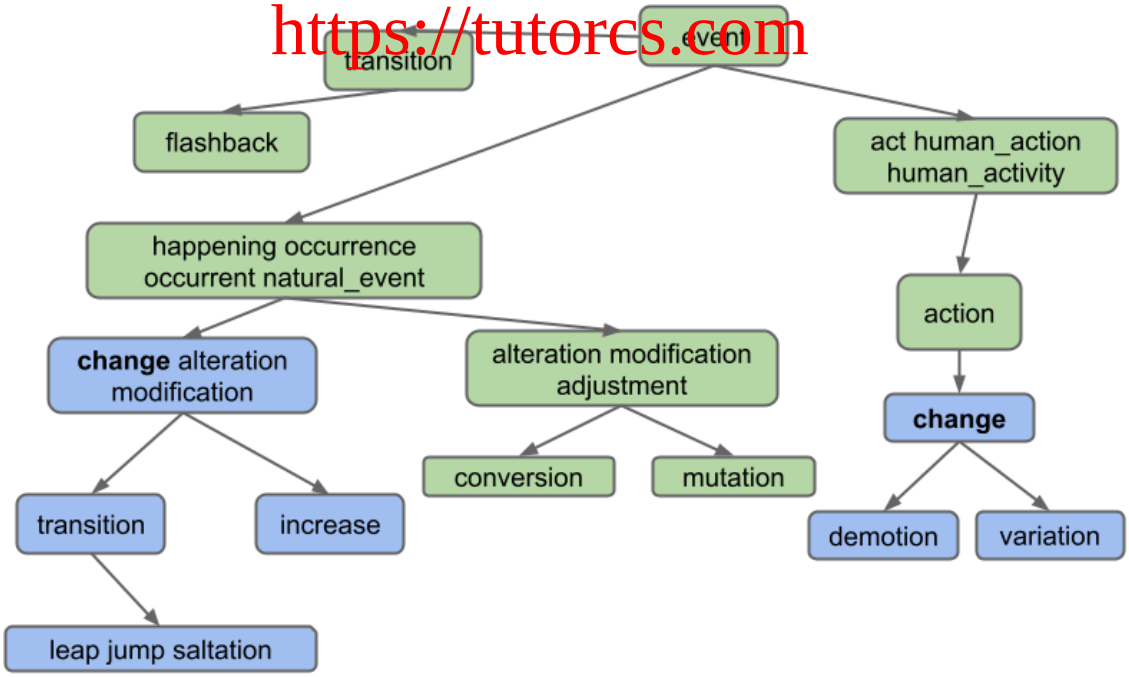


In this case, the output of your handler should be the string representation of a list containing "descent", "jump" and " parachuting", i.e `[descent, jump, parachuting]` . Note that the words are in alphabetical order.

As another example, suppose we're using a bigger dataset such as the one below (given to you as the input files `synsets16.txt` and `hyponyms16.txt`):



Suppose the user enters "change" and clicks on the Hyponyms button. In this case, the hyponyms are all the words in the blue nodes in the diagram below:



That is the output is `[alteration, change, demotion, increase, jump, leap, modification, saltation, transition, variation]` . Note that even though "change" belongs to two different synsets, it only appears once.

Note: Try not to overthink this. Specifically, observe that the output **does not** include:

- Synonyms of synonyms (e.g. does not include `"adjustment"`)
- Hyponyms of synonyms (e.g. does not include `"convert"`)
- Hyponyms of other definitions of hyponyms (e.g. does not include `"flashback"` , which is a hyponym of another definition of `"transition"`)

TASK

Implement `HyponymsHandler` . Do not use any helper classes.

Note: Please read the warning below, you shouldn't be writing all of your code in this class.

WARNING

To complete this task, you'll need to decide what classes you need to create to support the `HyponymsHandler` . **DO NOT DO ALL THE WORK IN HYPONYMSHANDLER.** Instead, you should have helper classes. For example, in 2A, to handle the "History" button, we created an `NGramMap` class. You'll want to do something similar in 2B, with your own classes.

You'll also need to understand the input format of the WordNet dataset. This description is given in the section below.

DANGER

For this part, you may NOT import any existing graph library into your code. That is you can't import, for example, the graph implementations from the optional Princeton algorithms textbook. Instead, you should build your own graph class or classes.

Tips

- Just like 2A's NGramMap, you'll want your helper classes to only parse the input files once, in the constructor. **DO NOT CREATE METHODS WHICH HAVE TO READ THE ENTIRE INPUT FILE EVERY TIME THEY ARE CALLED.** This will be too slow!
- We strongly recommend creating at least two classes for this part of the project as follows: One which implements the idea of a directed graph. One which reads in the WordNet dataset and constructs an instance of the directed graph class. This second class should also be able to take a word and return its hyponyms. You may also want additional helper classes that represent the idea of a traversal but this is not required - you can implement your traversal within your graph class as well.
- Don't worry about writing Truth tests yet, we'll talk about how to do that later in the spec. Simply use the web front end to check the two input examples ("descent" and "change") from the diagrams above for `synsets16.txt` and `hyponyms16.txt` .

- While you can (and should) write unit tests for the helper classes/methods that you create for this project, another good way to test and see what's going on with your code is to simply run `Main.java`, open `ngordnet.html`, enter some inputs into the boxes, and click the "Hyponyms" button. You may find visual debugging can lead to some useful discoveries in this project.

WordNet File Format

We now describe the two files that store the WordNet dataset. These files are in comma separated format. Each line contains a sequence of fields, separated by commas.

- File Type #1: List of noun synsets. The file `synsets.txt` (and other smaller files with `synset` in the name) lists all the synsets in WordNet. The first field is the synset id (an integer), the second field is the synonym set (or synset), and the third field is its dictionary definition. For example, the line


```
6829,Goofy,a cartoon character created by Walt Disney
```

Copy

means that the synset `{ Goofy }` has an id number of 6829, and its definition is "a cartoon character created by Walt Disney". The individual nouns that comprise a synset are separated by spaces (and a synset element is not permitted to contain a space). The S synset ids are numbered 0 through S – 1; the id numbers will appear consecutively in the synset file. The id numbers are useful because they also appear in the hyponym files, described as file type #2.

- File Type #2: List of hyponyms. The file `hyponyms.txt` (and other smaller files with hyponym in the name) contains the hyponym relationships: The first field is a synset id; subsequent fields are the id numbers of the synset's direct hyponyms. For example, the following line

```
79537,38611,9007
```

Copy

means that the synset 79537 ("viceroy vicereine") has two hyponyms: 38611 ("exarch") and 9007 ("Khedive"), representing that exarchs and Khedives are both types of viceroys (or vicereine). The synsets are obtained from the corresponding lines in the file `synsets.txt`:

```
79537,viceroy vicereine,governor of a country or province who rules...
38611,exarch,a viceroy who governed a large province in the Roman Empire
9007,Khedive,one of the Turkish viceroys who ruled Egypt between...
```

Copy

There may be more than one line that starts with the same synset ID. For example, in `hyponyms16.txt`, we have


```
11,12
11,13
```

Copy

This indicates that b 13 are direct hyponyms of synset 11. These two could also have beer ne line, i.e. the line below would have the exact same meaning, nam and 13 are direct hyponyms of synset 11.

```
11,12,13
```

Copy

You might ask why there are two ways of specifying the same thing. Real world data is often messy, and we have to deal with it.

Suggested Steps to Take

To get the "Hyponyms" button working you'll need to:

- Develop a **graph class**. If you aren't familiar with this data structure, take a look at lectures 21 and 22. You should test this with operations that are independent of the given data files. For example, our tests evaluated that our `createNode` and `addEdge` functions yielded appropriate graphs by using our graph classes's `getNodes` and `neighbors` functions. For inspiration, you can check out Lecture 22 and 23.

- Write code that **converts the WordNet dataset files into a graph**. This could be part of your graph class, or it could be a class that uses your graph class.
- Write code that takes a word, and uses a **graph traversal** to find all hyponyms of that word in the given graph.

We strongly recommended writing tests that evaluate queries on the examples above (for example, you might look at the hyponyms of "descent" in synsets11/hypernyms11, or the hyponyms of "change" in synsets16/hypernyms16).

Tests should be written at a level of abstraction appropriate to what they're evaluating. For example, we have a class called `TestGraph` that evaluates various aspects of our `Graph` class.

Or as another example, our code has a class called `TestWordNet` containing the function below.

```
@Test
public void testHyponymsSimple(){
```

```
    WordNet wn=new WordNet("./data/wordnet/synsets11.txt","./data/wordnet/hyponyms1
    assertThat(wn.hyponyms("antihistamine")).isEqualTo(Set.of("antihistamine","act
}
```


Note that your WordNet class may not have the same functions as ours so the test shown will probably not work verba[illegible]e. Note that our test does NOT use an `NGramMap` anywhere, nor is it using [illegible]er , nor is it directly invoking an object of type `Graph` . It is specifically [illegible] the `WordNet` class. **Relying on only browser tests will be incredibly frust[illegible]. Use your JUnit skills to build confidence in the foundational abstracti[illegible]l (e.g. Graph, WordNet, etc.).**

Design Tips

This project involves having to do all sorts of different lookups, graph operations, and data processing operations. There is no one right way to do this.

Some example lookups that you might need to perform:

- Given a word (e.g. “change”), what nodes contain that word?
  - Example in synsets16.txt: change is in synsets 2 and 8
- Given an integer, what node goes with that index?
  - Necessary for processing hyponyms.txt. For example in hyponyms16.txt, we know that the node with synset 8 points at synsets 9 and 10, so we need to be able to find node 8 to get its neighbors.
- Given a node, what words are in that node?
  - Example in synsets16.txt: synset 11 contains alteration, modification, and adjustment

Some example graph operations you might need to perform:

- Creating a node, e.g. each line of synsets16.txt contains the information for a node.
- Adding an edge to a node, e.g. each line of hyponyms16.txt contains one or more edges that should be added to the corresponding node.
- Finding reachable vertices, e.g. the vertices reachable from vertex #7 in hyponyms16.txt are 7, 8, 9, 10.

Your life will be a lot easier if you select instance variables and/or data structures for your classes that naturally help solve all six of the problems above.

Some example data processing operations:

- Given a collection of things, how do you find all non-duplicate items? (Hint: There is a data structure that makes this very easy and efficient). Don’t be afraid to also Google

documentation for the data structure that you choose (e.g. if you choose to use a TreeMap for whatever reason, feel free to look up "TreeMap methods java", "Map methods java", or "Collection methods java", etc).

- Given a collection of things, how do you sort them? (Hint: Google how to sort the collection that you're using)

WARNING

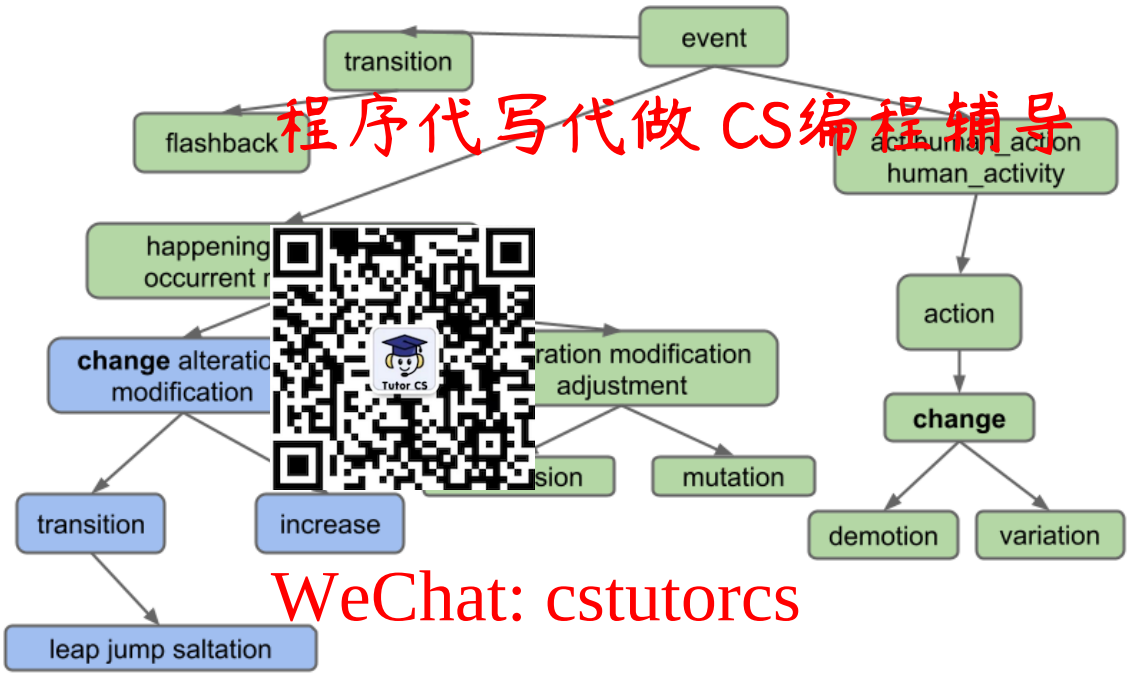
Also, a reminder from [illegible] sted generics are a warning sign that you are doing something too compli[illegible] a simpler way or create a helper class to help manage the complexi[illegible] you find yourself trying to use something like Map<Set<Set<..., you have started a walk down an unnecessarily difficult path.

WARNING

As usual, if you have a design that is painful and with which you cannot make progress, don't be afraid to delete your existing instance variables and try again. The hard part of this project is the design, not the programming. You can always use git to recover your old design if you decide you actually liked it.

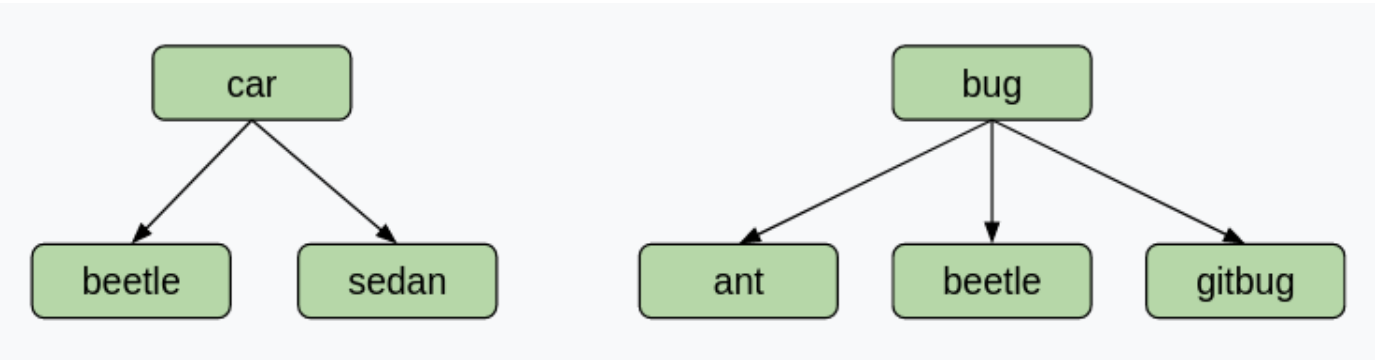
Handling Lists of Words

Your next task is to handle lists of words. As an example, if the user enters "change, occurrence" for the diagram below, we should only return common hyponyms of each word, i.e. `[alteration, change, increase, jump, leap, modification, saltation, transition]` . "Demotion" and "variation" are not included because they are not hyponyms of both words; specifically, they are not hyponyms of "occurrence".



As you can see, we only want to return words which are hyponyms of ALL words in the list. Furthermore, note that the list of words provided by the user can include more than just 2 words, even though our examples in this spec do not.

Note that it is possible for two words to share hyponyms without necessarily sharing nodes. Take a look at this example. If the user enters "car, bug" for the diagram below, we should get `[beetle]`, not `[]` (empty list)! This example shows that we are getting the intersection of **words**, not **nodes**.



For some more examples which demonstrate the usefulness of this feature, let's say we are using the full `synsets.txt` and `hyponyms.txt`.

- Entering "video, recording" in the words box and clicking "Hyponyms" should display `[video, video_recording, videocassette, videotape]`, as these are all the words which are hyponyms of "video" and "recording".
- Entering "pastry, tart" in the words box and then clicking "Hyponyms" should display `[apple_tart, lobster_tart, quiche, quiche_Lorraine, tart, tartlet]`.

TASK

Modify your `HyponymsHandler` and the rest of your implementation to deal with the List of Words case.

WARNING

To test this part of yo[illegible]mend manually constructing examples using `synsets16.txt` and [illegible]nd using the provided front end to evaluate correctness.

Deliverables and Scoring

For Project 2B, the only required deliverable is the `HyponymsHandler.java` file, in addition to any helper classes. However, we will not be directly grading these classes, since they can vary from student to student.

- Project 2B/C: Checkpoint: 5 points - **Due March 15th**
- Project 2B Coding: 50 points - **Due April 1st**
  - `HyponymsHandler` single word case: 50%, k = 0
  - `HyponymsHandler` multi-word case: 30%, k = 0
  - `HyponymsHandler` eecs-one-multi-word case: 20%, k = 0 (Tests for one and multiple words case but strictly uses `frequency-EECS.csv`, `hyponyms-EECS.txt`, `synonyms-EECS.txt`. You can find more information about EECS class list in 2C.)

In addition to Project 2B, you will also have to turn in your design document. This will be worth 5 points and it is due March 15th. The design document's main purpose is to serve as a foundation for your project. It is important to think and ideate before coding. What we are looking for in the design document:

- Identify the data structures we have learned in the class that you will be using in your implementation.
- Pseudocode / general overiview of your algorigthm for your implementation.

Your design document should be around 1 - 2 pages long. Design document will be mainly graded on effort, thought and completion.

Please make a copy of this template and submit to Gradescope.

Don't worry if you decide to change your design document after. You are free to do so! We want you to think about the implementation before coding therefore we require you to submit your design as the part of the project.

The token limiting policy for this project will be as follows: You will start with 8 tokens, each of which has a 24-hour refresh time.

Testing Your Code

We've provided you with two short unit test files for this project in the `proj2b/tests` directory:

- `TestOneWordK0Hypon`
- `TestMultiWordK0Hyp`

The two provided test files correspond to the first two cases that you solved in this project, that is:

- Finding hyponyms of a single word where k = 0.
- Finding hyponyms of multiple words where k = 0 (e.g. `gallery, bowl`).

You will need to complete `AutograderBuddy.java` to test your code. See the Submitting Your Code section for more details.

These test files are not comprehensive; in fact, they each only contain one sanity check test. You should fill each file with more unit tests, and also use them as a template to create two new test files for the respective cases.

If you need help figuring out what the expected outputs of your tests should be, you should use the two tools that we linked in the Getting Started section.

Debugging Tips

- Use the small files while testing! This decreases the startup time to run `Main.java` and makes it easier to reason about the code. If you're running `Main.java`, these files are set in the first few lines of the `main` method. For unit tests, the file names are passed into the `getHyponymsHandler` method.
- You can run `Main.java` with the debugger to debug different inputs quickly. After clicking the "Hyponyms" button, your code will execute with the debugger - breakpoints will be triggered, you can use the variables window, etc.
- There are a lot of moving parts to this project. Don't start by debugging line-by-line. Instead, narrow down which function/region of your code is not working correctly then search more closely in those lines.
- Check the FAQ for common issues and questions.

Submitting Your Code

Throughout this assignment, we've had you use your front end to test your code. Our grader is not sophisticated enough to pretend to be a web browser and call your code. Instead, we'll need you to provide a method in the `proj2b.src.main.AutograderBuddy` class that provides a handler that can deal with hyponyms requests.

When you ran `git pull skeleton main` at the start of this spec, you should have received a file called `AutograderBu`

Open `AutograderBuddy` `e getHyponymsHandler` method such that it returns a `HyponymsHandler` tha en files. Your code here should be quite similar to your code in `Main.java`

Now that you've created `proj2b.src.main.AutograderBuddy`, you can submit to the autograder. If you fail any tests, you should be able to replicate them locally as JUnit tests by building on the test files above. If any additional datafiles are needed, they will be added to this section as links.

Acknowledgements

The WordNet part of this assignment is loosely adapted from Alina Ene and Kevin Wayne's Wordnet assignment at Princeton University.