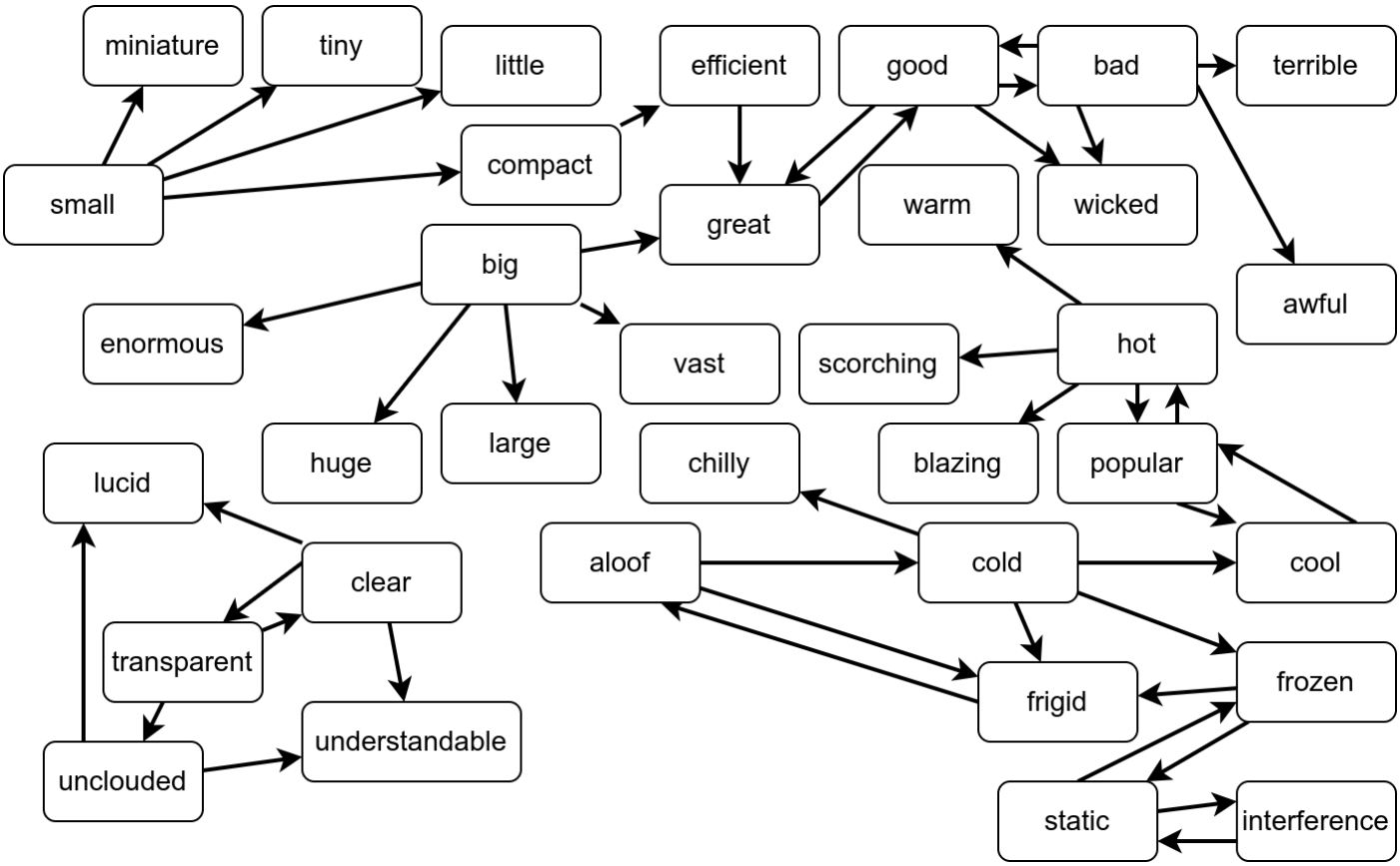
COSC 2P03 — Assignment 4 — Not really ‘mad’ libs, so much as ‘perturbed’

Premise: where’s a good thesaurus when you need one? We’ve all been there: you’ve prepared some document, but it lacks that *je ne sais quoi*. Obviously, the best way to make yourself sound smarter/worldly/more interesting/acrobatic is to start peppering in words from a larger lexicon, whether they really fit or not!

Back in the old days, we’d use something called a “thesaurus” to find words similar in meaning to other words (*synonyms*). Of course, two synonyms of a word aren’t necessarily synonyms of each other (e.g. two synonyms of ‘head’ might be ‘skull’ and ‘boss’, but very few people report to skeletons for their jobs). Similarly, one may or may not want to treat two words as being bidirectionally interchangeable (‘wet’ might be fine in place of ‘moist’, but nobody needs to *increase* the number of times ‘moist’ appears in writings).

One (incredibly truncated) mapping of words might look like this:



Notice how the meaning of words can quickly be lost if you try using synonyms of synonyms (of synonyms, of synonyms, etc.). Small→compact→efficient might actually make sense (because many miniaturized devices *are* more efficient), but big→great→good could be confusing, and cool→popular→hot is just goofy.

However, we’ll chalk that up to simply being “fun” (since we aren’t the ones who need to *read* what we produce).

The task:

You’ll be creating an application, and some tools to support that application.
Specifically, you’ll be loading a data file that contains lists of synonyms, and use its connections to populate a **graph**. Note that the graph itself will only maintain the connections themselves; the actual linguistic equivalents (i.e. the mappings to real words) will be handled by your **thesaurus**. That is, your thesaurus will *use* a graph to establish synonyms, and allow for looking them up.

Your application, after creating and populating the thesaurus, will read in a data file containing plain text.
Have the user enter a preferred level of separation *distance*, tokenize the plain text, and perform as many substitutions as possible on the terms according to that distance.

- In this context, distance is simply the number of ‘hops’ to *attempt* to make from a word
  - In the graph above, a synonym one ‘hop’ from *cool* is *popular*, while the only synonym two ‘hops’ away from it is *hot*
  - If a larger number is requested than possible, simply stop at the end. e.g. if a synonym is requested five ‘hops’ away from *small*, and you end up on *miniature*, then you simply get *miniature*
    - This also means that a word will be the synonym of itself iff it has no entries in the thesaurus
    - In the example above, *kwyjibo→kwyjibo*, since it has no actual entries in the Thesaurus
- When multiple synonyms are possible (as is often the case), *randomly* select one

Requirements:

In the interest of clarity at the end of the term, you're provided below with precisely what classes you *must* have, and how they *must* work. You can (and probably *should*) create any *additional* classes you might require.

- The `Graph` class
  - This will store an arbitrary graph as an **adjacency matrix**
  - A graph won't know in advance how many vertices or edges there will be, but can rely on the assumption that an *upper-bound* is knowable in advance (see below)
  - Graphs know vertices **solely** as *numerical* vertex ids. They have no notion of "words", "synonyms", etc.
    - To be absolutely clear: the graph portion of your submission *must* be purely about graphs; with **no** elements of the larger task
    - Further, the graphs portion is part of its own **`graphs`** package
  - 'Typical' graph functionality would be the ability to add edges, check if two vertices are neighbours, retrieve *some* arbitrary neighbour of a vertex (randomly-selected is fine), and retrieve *all* neighbours of a vertex
    - Adding edges is interesting, as you don't necessarily *need* to keep track of vertices themselves. Simply allocating for a maximum number of vertices and filling in edges as entries are encountered suffices
- The **`Thesaurus`** class, within your **`lexicon`** package, is a tool to be used. It is *not* "the program"
  - It will somehow be populated by the data file, and will coordinate the building-up of the graph
  - As the graph operates on numeric indices, and the Thesaurus deals with words, you'll need a *bidirectional mapping* from each to the other (more on this below)
  - The Thesaurus itself only *needs* two publicly-available functions:
    - A `getSynonym(String)` function to return some synonym of a provided term
    - A `getSynonym(String, int)` function to return the synonym, *of some distance*, of the term
    - Per above, if the target number is unreachable (or if a word has no synonyms), simply stop however far you got
- A **main program**, also within your `lexicon` package
- You'll likely want something to represent the Document to have its contents translated. This *can* be done as part of the main program
  - To clarify: the behaviour is mandatory; separating it into a separate class is simply advice to make it easier
  - If you do separate this into another class, put it into your `lexicon` package
- Some form of **Bidirectional Map** that translates numeric ids to words, and words to those same numeric ids
  - Put this into a **`storage`** package, which you'll also use for any other supplementary data structures you need
  - Likely required behaviours are:
    - Adding a term as a String (and probably receiving the numeric id of the entry)
    - Retrieving the numeric id of a String
    - Retrieving the String held at some numeric id
    - Possibly a count of how many terms are stored
  - You might find this better to define as an interface first, so you can start with a trivial implementation and replace it with something more efficient later
  - Advice: if a requested term isn't present, return an id of -1

Data file formats:

The thesaurus data file is defined as:

- An arbitrary number of lines, with the first word in each line being the word defined to have synonyms
- A tab after that initial word
- Comma-delimited entries of synonyms of the word at the start of that line

Note that a synonym *could* be two words (could have a space inside it), but will never contain commas. Also each comma delimiter will *typically* have a space after it (so remember that Strings have a `.trim()` function).

The 'script' data file (the document to be 'translated') is simply text, of some length.

- It may or may not contain multiple lines
- It may contain any arbitrary number of punctuations

The potential awkwardness of identifying "The dog is blue." as four words and a period is part of why writing a separate class just to manage the document might make this part easier. You don't need to worry about hyper-precise formatting concerns (e.g. preserving cases of two spaces after periods, etc.), but you shouldn't be losing sentence structures.

Additional requirements:

First, to adequately convey this: this is "the graphs assignment". Your submission will follow the minimal requirements for (a) being a graph, (b) being the type of graph required, and (c) using graph-based algorithms, or it will **receive zero**. There will be no special considerations.

Similarly, you'll be implementing everything yourself. By all means, import File/Scanner (or BasicIO, though I wouldn't recommend it), but other than that? Everything's java.lang and you. While we're at it: other than your main method, no uses of the **`static`** keyword, at all.

Additionally: we've had some peculiar submissions for past assignments that were needlessly complicated, so for your actual implementation:

- Develop, from start to end, in IntelliJ. *Nothing* else
- Develop on a single computer, or include a .txt file (at the top level of your submission) explaining what computer(s) you used, and how you transferred it. No more of this 'multiple pieces of different things' schtick
- Create multiple packages for different components (e.g. something like a `storage` package for any stacks, queues, maps, etc. you might choose to create 'for holding stuff', a `graphs` package for graph stuff, etc.)

Submission:

This is roughly the same as for past assignments:

- Developed in IntelliJ, for JDK 11 (or higher)
- As mentioned above, the only library outside of java.lang you can use is for IO
- Submissions are via Brightspace:
    - A single .zip containing your complete submission
    - Sample output of your program, using the provided input and a distance of 2
    - Any instructions you think the marker might need to fully run your program

Sample data:

You're provided with a very trivial sample thesaurus and document (expect the markers to use something else!).

One such possible translation (with distance of 5) of this sample input:

> I once heard the tale of a great man, the Claws of Santa. They say the Claws of Santa lived in the cold, desolate wastelands north of the maple people. His frozen lair, standing in stark defiance of an angry world. In spite of the ceaseless noise of a raging populace, tearing away at the sanity of mere mortals like static evaporating the image of happy days long gone, his mind is clear; his resolve unshaken. Lo, he even finds time to bring justice to the most terrifying of creature known to humanity: children. The good, the bad, he judges them all. With the gift of mercy to the kind, and no such boon to the malevolent. And as he metes out his justice most efficient and foul, he laughs. His portly frame shaking like a big bowl of sweetened hoof distillate. By the way, I certainly hope nobody is actually bothering to read this nonsense; it is just a bunch of filler.

is:

> I once heard the tale of a good man, the Claws of Santa. They say the Claws of Santa lived in the interference, desolate wastelands north of the maple people. His cold lair, standing in stark defiance of an angry world. In spite of the ceaseless noise of a raging populace, tearing away at the sanity of mere mortals like aloof evaporating the image of happy days long gone, his mind is understandable; his resolve unshaken. Lo, he even finds time to bring justice to the most terrifying of creature known to humanity: children. The wicked, the wicked, he judges them all. With the gift of mercy to the kind, and no such boon to the malevolent. And as he metes out his justice most wicked and foul, he laughs. His portly frame shaking like a vast bowl of sweetened hoof distillate. By the way, I certainly hope nobody is actually bothering to read this nonsense; it is just a bunch of filler.