WordNodeTest

- Testing a constructor given a word and its count. This is just a test to see if the program can properly create a wordNode with given values. In this case, the word alice is given with a count of 4. The JUnit tests test whether these values were correctly stored, returning true if the node has a count of 4 and a name of alice. This is the most important test for this class as it's the main method of inputting word nodes into a hashtable to calculate the count.
- Testing a constructor given just a word.
- Testing a constructor given nothing. Simply makes all values null or zero.

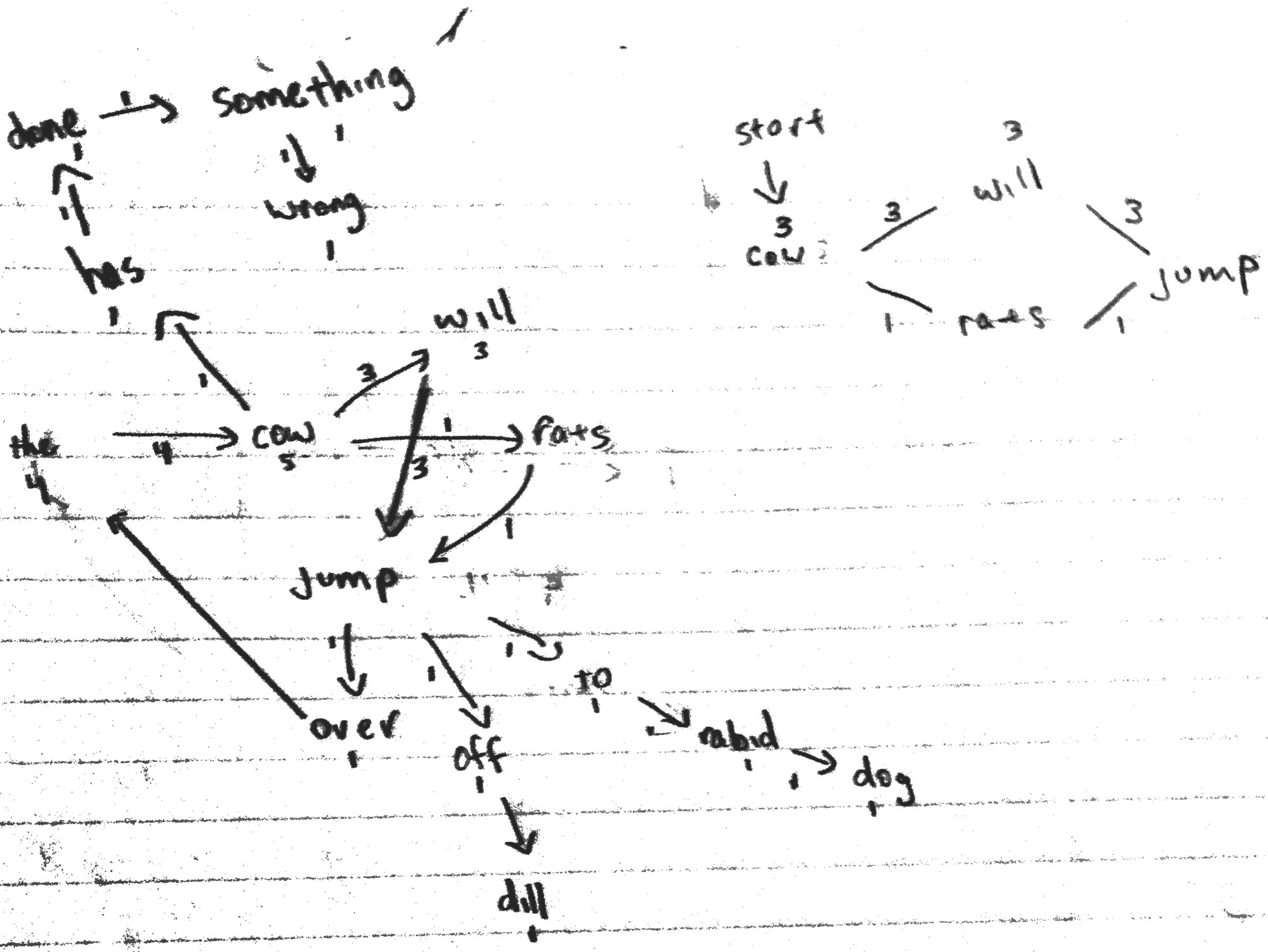
WordPairTest

- Testing a constructor given two words and the count. This tests if the word pair correctly stores given values.
- Testing with two words combined into one string. Constructor is able to differentiate which word is the base and which word is the target.
- Testing with no count given. Returns the name given, in this case: alice thought

WordGraphTest

- testLargeFileConstructor – tests the creation of a word graph from a large file. This is done to just see if the class can handle a large file
- testFileConstructor – tests a small file to see if the class is capable of creating a small word graph
- testConnectEdges – This tests the joining of the edges to nodes. Although there is no check for the method itself, it can be checked by looking at the neighbors stored in a node. When looking at text.txt, the word 'maid' should have only one edge, (will), and this is correct.
- testWordCount – tests to see if algorithm correctly counts the number of occurences for a word. When reading test2.txt, counting 'cow', returns 5, which is correct
- testInDegree andPrevDegree– tests if the count of how many unique words there are following a word is accurate. Testing the test.txt on the word "days" yields 1, which is correct as days only occurs once, so there can only be one occurrence. This also checks the words that precede the selected word. It checks the length of the returned array and what word it should be.
- testOutDegree – tests if counting the unique words following a word is correct. This test is the exact same as inDegreandPrevDegree, except that it's for words that follow the selected word. In this case 'day' and 'a'.
- testWordSeqCost – this test is a simple test to see if the function given has been put into the program properly. The test checks to see if the double returned is larger than zero, using the string array{alice,think,it,so}. It will also prevent any count from being calculated if not enough words were given. In these cases, the cost is 0. This is done since the path cost of a node to go to itself should be zero. This is the same method as pathCost, however, pathCost was easier for me to think about while programming.
- testGeneratePhrase – This creates a phrase from the test2.txt. It checks to see if the phrase is N or fewer words. In this case, the phrase being created must

start at cow and travel to jump. The graph of this text included below. Although cow would be able to travel to rats and reach jump that way, it correctly paths along will to reach jump, showing that it is functioning correctly. It also tests the extra credit, ensuring that no word shows up more than a number of times. There is no specification on how to deal with repeated words, so method used is to remove the words that are larger than allowed number of repeats. The test (the,over,5,1) successfully eliminates the repeats. This test also tests if the algorithm knows when to stop. If the endWord is too far away, the returned string will simply be null. This must be tested so it can be determined that N was implemented correctly. The test correctly returns an empty statement, so the program does recognize depth. It also compares two different phrases, to ensure the program selected the right path. In this case, (alice so) and (alice think it so) are compared, and alice so is determined to have the lower cost.


done
something
wrong
has
will
the
cow
pats
jump
over
off
to
rabid
dog
dull
start
cow
will
jump
pats