Team 7 Members
Travis Kosarek
Christopher Weldon
Thomas Robbins
CPSC 315
04/09/2009

Code Checkpoint 1

Version Control System

Our group chose to use the Google Code Repository along with TortoiseSVN not only because they are very popular systems to use but also because we have all used them in our previous groups this semester.

TortoiseSVN is a simple program that allows us to keep our code up-to-date as we work on different parts individually. It notifies us if one of our teammates updates a different version of a file than the one we originally had been working on. It then provides us with the option to then download the updated file and therefore keeps all of code "up-to-date."

We chose to use the Google Code Repository because of its simplicity. We had all used it before so it was no hassle learning a new format. Google Code has the option to not only upload the code but also to view it directly on the web page, post issues/work to be done in the issues section, post our API in the Wiki section and also to post general comments in other Wiki pages that we create ourselves. It is overall a very useful source for communication between team members.

Team Organization

Because of the nature of our project, a lot of the code has to be done sequentially. This is unfortunate because this makes it hard to assign individual parts to each team member. Our plan is to first as a group work on the stemmer and have it completed by the end of the second week. This should not be a very difficult. After this is done we are able to work on the graphical aspects of our project and the actual sorting of documents. At this point I will split up the work as follows:

- I will work on how to sort the documents because obviously I am the most familiar with my design so this is easiest solution.
- I will assign the basic list interface to Chris because I believe he can accomplish this fairly quickly.
- I will assign the visualization basics to Thomas because I think it is important to have someone working on the entire design from the beginning. Of course he is not expected to complete the entire visualization and it is expected that whenever Chris completes the basic query window he will help Thomas with the visualization. (Our visualization is a basic "cloud cluster" system.

All of these assignments are general assignments and are by no means meant to restrict helping other team members. It is my hopes that we all complete this project efficiently together and have enough code stubs to work entirely in the lab time. If outside work is needed I believe that the team should be responsible for it and that everyone should actively be working on finding the solution.

Project Specification / Project API

When first designing this project we immediately recognized a few key parts during production that will take a little time to complete.

The first, most obvious source of difficulty when developing a search engine is that which hinders large widely used search engines like Google as well. This problem is of course the ability to accurately interpret the user's query and return documents ranked in order of value to the user. In order to combat this problem we had to think of a solution that would not only actively work to depict what information in a library of documents was useful but also to interpret what the user actually wanted. To do this we are implementing a contextual searching algorithm. This effectively stores words before and after a certain key words (non-stop words.) This greatly increases the precision of the documents returned but does not really effect the recall amount of documents returned.

If we were to do this strictly by word we would effectively create three times the size of the amount of documents we have just to search through them. This is very inefficient. In order to fix this problem we also need to implement a stemmer algorithm that will break a word down to its "root." If we are to do this, we also need to store what was "stemmed" off of the word (prefixes, suffixes, tenses, etc.) If we already storing this information we can be a less critical as to if a stem is correct or not. This decreases the precision of the search but greatly increases the recall of possible correct results.

Although these two algorithms are purely design decisions they greatly effect the outcome of the entire program. With this design our program is very efficient at retrieving information but is not as efficient at storing the information. Because of this flaw, we cannot hope to use this implementation at a higher level of use than for this project (as a commercial search engine.)

Our program is generally broken down as follows:

Main.java

- This file simply calls the SearchEngineGUI

SearchEngineGUI.java

- This file is purely the graphical part of the program (functions purely as a window to the data)
- This includes not only the basic search result list but also our visualization (a visual clustering of documents based on common terms shared between them.
- Upon creation asks the user for a directory to add to its Dictionary.

SearchEngine.java (implements WordStemmer, WordTokenizer)

This is where the meat of the searching algorithm is.

- Allows the user to query the Dictionary
- Contains the stemmer (takes in one word and finds its "root" value, this is not always correct to English language standards, but close is good enough.)
- Adds Words to the Dictionary and to Documents. This allows for contextual searches to take place.

- A searching algorithm that takes in a phrase (can be no words at all) and returns an ArrayList of Documents. This list is then converted into HTML web pages that are used in the SearchEngineGUI.
- Allows for ranking of Document ArrayList by frequency of word/phrase in Dictionary and Document.

Dictionary.java

This object contains all Words with context from the every Document in the directory.

- Allows the addition of a Word based on its root.
- Passes the actual word and the root to the Word object.
- Allows for fast searching through the root tree of Words.
- If a Word is found, its context is also checked across all Documents.

Document.java (extends Dictionary.java implements DocumentInfo)

This file is like a Dictionary for an individual document.

- Allows the addition of a Word based on its root.
- Passes the actual word and the root to the Word object.
- Allows for fast searching through the root tree of Words.
- If a Word is found, its context is also checked

Word.java (extends StemInfo)

This is an abstract idea of a word from a Document.

- A Word is only defined as a root.
- A Word stores possible other spellings of a word.
- A Word stores all possible stems of the root.
- A Word stores all words that could possibly precede it in a Document.
- A Word stores all words that could possibly come directly after it in a Document.
- A Word stores the frequency it is used in a certain Dictionary. (A Document is also a Dictionary.)

Additional code stubs of these files including javadoc files can be found at
http://code.google.com/p/cpsc315-project3/