Twitter Tag Cloud Documentation

Chryssy Joski – Capstone Seminar – Spring 2016

1. Hook-up and install

First, load the following scripts to an accessible website (I put mine on compsci02): `apicheck.php`, `TwitterAPIExchange.php`, and `upgrade.php`. The `upgrade.php` may or may not be necessary; I started this project before ITS updated compsci02. If you do not include `upgrade.php`, make sure to remove the requirement for it from the `apicheck.php` script. The Twitter API script requires a personal OAUTH access token, secret access token, consumer key, and consumer secret key from Twitter, so be sure to add your own. (See **Initialize** for more details.) Once the appropriate php files are uploaded and properly initialized, the php scripts can be run as described in **Use**. If you are only looking for the results of the script, feel free to run the existing script discussed in **Use**.

---

2. Compile and Link

There are no special requirements for compilation and linking. The Tag Cloud solution compiles as expected in Visual Studio, and the .php scripts compile at execution when the page is loaded.

---

3. Initialize

The part of the project that retrieves tweets from Twitter requires authorization keys. For my own security, I have removed my personal keys from the code online. In order to run the Twitter API access script, the user will need to get their own OAUTH access token, secret access token, consumer key, and consumer secret key. These can be requested from Twitter and are linked to the user's Twitter account.

---

4. Use (including API and sample how-to programs)

In short, these are the steps to execute the program:

1. Run the API access PHP script.
2. Download **`tweet_data.txt`** (known as the data file) from compsci02 and place in the debug folder for the Tag Cloud.
3. Run **`TagCloud.exe`** (This will take quite a while before anything shows up on the screen; it could take a whole day, depending on the size of the tag cloud and speed of your

computer. I run it in the debugger so I can pause it periodically and see how far the program is in its execution.)

Detailed explanations of each step:

1. Run the API access PHP script. To run the Twitter API script, go to the following link: http://compsci02.snc.edu/cs460/2016/joskch/twitter_api/apicheck.php. Loading this page will automatically run the PHP script. The script will write the applicable tweet data to the data file, which is in a .txt format. This script will append new data to the end of the existing data file (so I can make multiple runs before creating a tag cloud). To get a fresh set of data, delete the data from the existing `tweet_data.txt` file (which is found in the same folder as the Twitter API script (**`compsci02.snc.edu/cs460/2016/joskch/twitter_api/`**).

Twitter API testing

Format of data returned by Twitter API (JSON strings)


```
Array
(
    [0] => Array
        (
            [trends] => Array
                (
                    [0] => Array
                        (
                            [name] => #NationalMargaritaDay
                            [url] => http://twitter.com/search?q=%23NationalMargarita
                            [promoted_content] =>
                            [query] => %23NationalMargaritaDay
                            [tweet_volume] => 49400
                        )

                    [1] => Array
                        (
                            [name] => #WORKvideo
                            [url] => http://twitter.com/search?q=%23WORKvideo
                            [promoted_content] =>
                            [query] => %23WORKvideo
                            [tweet_volume] => 103959
                        )

                    [2] => Array
                        (
                            [name] => #mondaymotivation
                            [url] => http://twitter.com/search?q=%23mondaymotivation
                            [promoted_content] =>
                            [query] => %23mondaymotivation
                            [tweet_volume] => 131769
                        )
```


2. Download `tweet_data.txt`. Using Winscape, navigate to the previously mentioned folder and download the data file (`tweet_data.txt`). Place it in the debug folder for the Tag Cloud solution. Make sure that the file name is maintained and that it is in the debug folder, or else the program will not find it and it will not run. I have also included my sample data document (named `tweet_data_sample.txt`). To run with my sample data, either rename the file (and remove the other `tweet_data.txt` file) or rename the variable `wordfile` in `TagCloudV2.cs`.

Optimization note: the C# program will run the best if the data from the data file is sorted in order of most popular tweet to least. When the Twitter API writes to the file, it writes in order from highest volume to lowest volume. However, if you make multiple runs of the PHP script before downloading the .txt file, the data will be "out of order." The C# program will still run, but the tag cloud may not be as optimized as it could be.

3. Run TagCloud.exe. An explanation of the data structures, functions, and execution of TagCloudV2.

Data Structures:

The **TagData class** contains the following four pieces of information about each tag phrase.

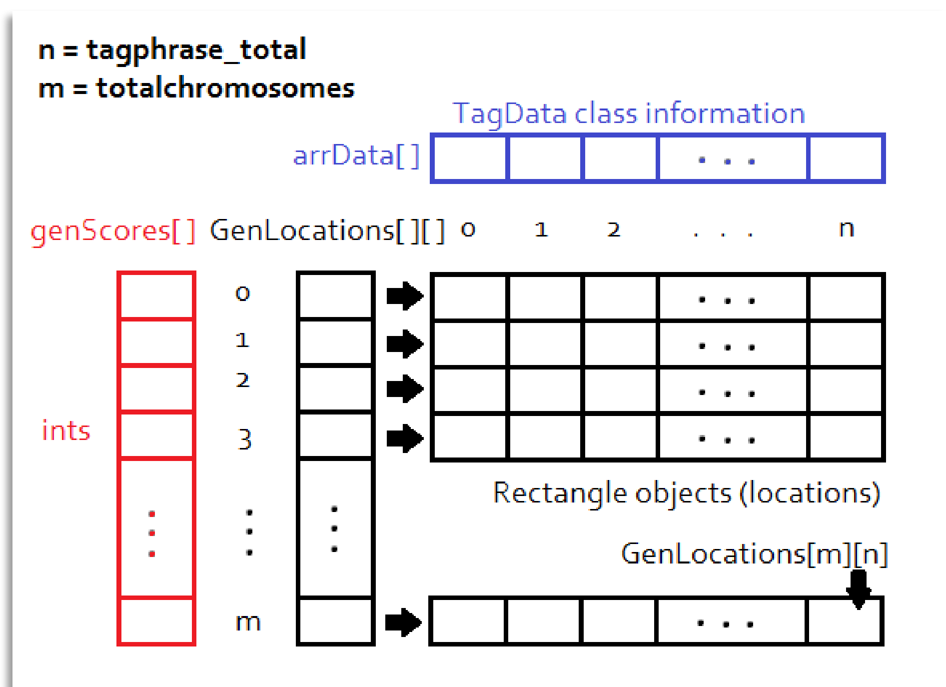
- **Volume:** this is the tweet_volume as reported by the Twitter API, and as processed in by reading the data file.
- **TagPhrase:** this is the string associated with each tag phrase. It can be a single word (computers), more than one word (Ada Lovelace) or a combination of words and special characters (#WomenInStem). In short, it is just a string.
- **Size:** the size is determined during pre-processing in the function **`DetermineFontSize()`**. It is then stored in the class.
- **PixelCount:** pixel count is determined during pre-processing in the function **`DetermineFontSize()`**, too. It is then stored in the class.

`arrData[tagphrase_total]`

This array contains elements of the TagData type. It tracks the aspects of each tag phrase and is to be used in parallel to the chromosomes that contain locations (see below).

`GenLocations[i][j]`:

Each chromosome is an array of rectangles containing just the locations of each tag phrase. It runs in parallel to the array that contains the tag data. The chromosomes are stored in a 2-dimensional array. For example, **`arrData[0]`** will have the tag phrase, volume, size and pixels for the item contained in **`GenLocations[i][0]`**, where **`i`** is the particular configuration (chromosome) being accessed at that time.



`genScores[i]`:

This array runs in parallel to the **`GenLocations`** array, but in a slightly different way. It tracks the genetic algorithm score for each chromosome. (See the discussion of functions **`CheckOverlap()`** and **`CheckProximity()`** below for more information on scoring.) This allows for easier sorting of good and bad chromosomes.

Functions:

I prefer to keep my code as modular as possible, so I wrote a number of functions for this program. I will describe in detail the active functions. The inactive (old) functions are included in the code at the end of the program and have a short explanation of their original intended purposes.

PRE-PROCESSING FUNCTION

`public void DetermineFontSize()`

This function is part of the pre-processing of the data. Once the data file is read in, this function iterates through **`arrData[]`** to assign font size and pixel counts.

It looks at the volume, and depending on the volume, it assigns a font size. I did not do a direct 1-1 correlation (such as dividing the volume by a certain number), because some tag phrases ended up enormous and others barely readable. Instead, of created ranges and assigned values.

Then, the function draws this word to the screen in that font size and counts the number of pixels it takes. Then it saves the pixel count to **`arrData[0]`** appropriately. It also keeps a running total of the number of pixels all of the tag phrases take.

RENDERING FUNCTIONS

`public void InitialScreenPlacementRender()`

This function creates the initial random locations that comprise the first generation of chromosomes. In short, it determines the size of rectangle required for each word and a random location for that rectangle. It draws the tag phrase on the screen using the location of that rectangle. Then, it stores that rectangle in a chromosome. It also tracks the rightmost, leftmost, topmost, and bottom most locations of each rectangle for proximity scoring.

`public void ScreenPlacementRender(int ind)`

This function does the same things as **`InitialScreenPlacementRender()`**, except that it doesn't determine the locations randomly; it uses locations in the chromosome. Also, it is sent an index for which chromosome is being rendered. That is done to simplify the loop that contains the call to this function.

`private void FinalRender()`

This function is called at the end to do the final rendering of the "winning" tag cloud. It does not do any storing. It displays a message box with the best score and the total number of generations it took to achieve that score (or indicates that it reached the maximum number of generations). It renders the tag phrases in **`GenLocations[0][]`** in different colors, cycling through black, red, and blue using modular math.

SCORING FUNCTIONS

`private int CheckOverlap()`

This function uses nested loops to count all of the colored pixels on the screen after an entire chromosome is rendered. If the total pixel count is less than the expected total pixel

count, it means there is overlap (some tag phrases are sharing pixels). I gave the function a 35 pixel buffer; sometimes the tag phrases would barely touch sharing just a couple pixels. I would rather have the corners of two words touch than throw out a generally good cloud.

If the tag phrases overlap, it returns a very negative score (-8000). If they don't overlap, it returns a sufficiently positive score (3000).

`private int CheckProximity()`

This function compares the difference between the topmost and bottom most rectangles and the leftmost and rightmost rectangles. Depending on how far apart they are, the function returns different values. The closer they are, the better the returned value.

The sum of **`CheckOverlap()`** and **`CheckProximity()`** determines the overall score that is stored in the scores array.

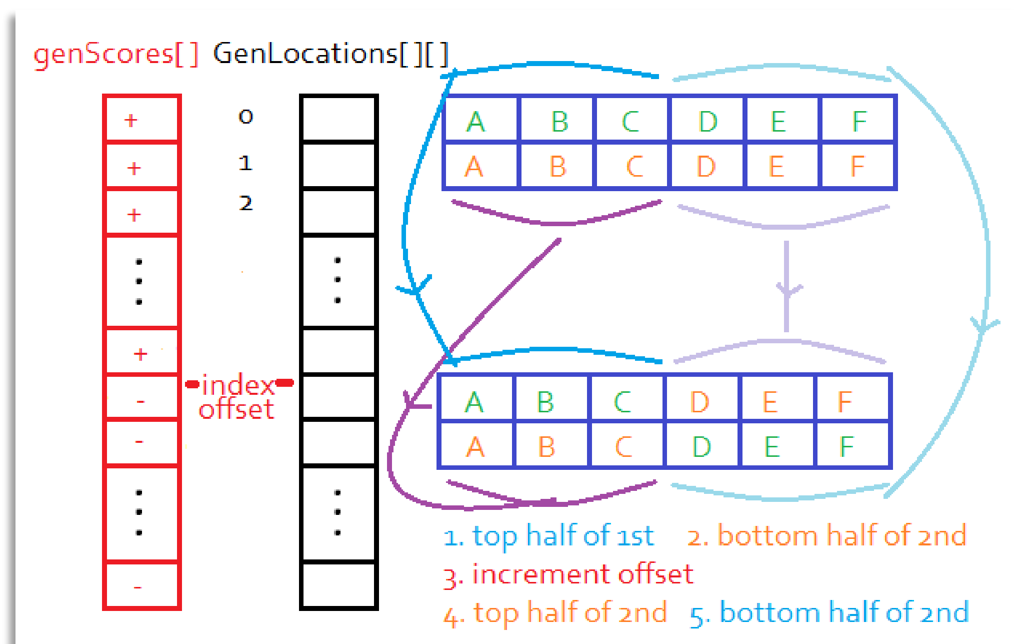
GENETIC ALGORITHM FUNCTIONS

`private void BubbleSort(int size)`

This is just a basic bubble sort. It sorts **`genScores`** and **`GenLocations`** in parallel based on the values in **`genScores`**.

`private int MateMe(int gi2)`

This function mates and mutates (as needed) the chromosomes. It receives the index of where the scores stop being at least 3000, so that it does not waste time mating negative scoring chromosomes. The index received also serves as an offset; the offset is where the children chromosomes start being written. That way, the function does not overwrite any good parent chromosomes during the mating process; all new ones are written at the end.



It mates the chromosomes two at a time. It takes the top half of the first chromosome and writes it to the location indicated by the offset. Then, it takes the bottom half of the second chromosome and writes it to that same location. As it writes each element (rectangle) of the chromosome, it checks to see if that

element is an "outlier." An outlier is a word that is out on the edges of the tag cloud; i.e., it is not creating a close tag cloud. If the element is an outlier, it mutates the element by giving it a new random location before writing.

Next, it does the same thing with the top half of the second chromosome and the bottom half of the first chromosome.

NOTE ON MUTATION: If I had the processing power, I would also use the function **`SnugUp()`** to mutate. That function slowly moved outliers towards the center instead of generating new random locations. After calling this function, the element would be written to the spot of offset+1. The program takes way too long with both running, and since the random location was encouraged by Dr. McVey, I went with that option. That is why the offset increments by two. That is an error I just noticed and forgot to fix.

`private void RandomizePopulation(int gInd)`

This function randomizes the decent scoring chromosomes. It receives an index (the point where the scores stop being at least 3000). From there, it uses a random number (between the current chromosome and the index) to swap the current chromosome with a randomly located chromosome. Then, it does a simple swap.

FINALIZING FUNCTIONS

`private void onPaint()`

This is the event handler for the paint event. It tells the program to draw the current image to the screen.

`private void ShowVolume()`

This function is the event handler for the mouse click event. When the mouse clicks on a tag phrase in the final rendering, this function determines which rectangle was clicked on and displays a message box containing the tag phrase and the actual tweet volume of that tag phrase.

Execution:

- The overall execution of this program is
- Read data file in
- Preprocessing function (**`DetermineFontSize()`**)
- Initialize chromosomes (**`InitialScreenPlacementRender()`**)
- Sort (**`BubbleSort(totalchromosomes)`**)
- Repeat the following steps until maximum generations reached or good score reached:
    - Find index
    - Mate & Mutate chromosomes above index (**`MateMe(index)`**)
    - Determine scores (**`CheckOverlap() + CheckProximity()`**)
    - Randomize chromosomes (**`RandomizePopulation()`**)
    - Sort (**`BubbleSort(totalchromosomes)`**)
- Render the tag cloud in the top position (**`FinalRender()`**)

I am a heavy commenter; the code is loaded with line comments. Please refer to the code for detailed explanations of how the functions work, line by line.

5. Detailed Exceptions

I haven't found any flat-out exceptions that make the program crash; as long as the data file is in the right format with the correct delimiters between the tag phrase and the tweet volume, the program will run and eventually complete. However, there are certain problems with the solution to this problem.

First, if there are no positive scoring chromosomes in the initial batch of chromosomes, there is no code to make bad ones better. The test file I had been using was small and always produced a few good ones; the more tag phrases, the harder it is to randomly produce a chromosome where there is no overlap. Unfortunately, I didn't realize this until way too late to write and test new functions.

There is no script to automatically pull the data file from compsci02 and place it in the debug folder for the Tag Cloud program. This is something to consider for future extensions of the program. The data file must be manually downloaded.

A tag cloud generating random locations for tag phrases is not the best way to create a tag cloud. It is a lot of guessing, a lot of randomness, and a lot of hoping it eventually figures it out. A better way to do this would be to use a type of breadth-first search. Start by placing the most popular word (the largest word) in the center, then methodically move out from center, placing the other tag phrases where there will be no overlap. This is much more efficient and methodical and will run much more quickly.

Finally, this is a consideration: as mentioned in Step 2 of **Use**, the program runs best when the data file is in order from the most popular to the least popular tweet. This is because of the way the genetic algorithm runs. When it mates the two chromosomes, it mixes the top half of one with the bottom half of another. The top half has the tag phrases that are the hardest to place; these are the largest tag phrases and the most likely to cause overlap. If these phrases manage not to overlap, it is considered highly successful. The bottom half of the chromosome has the smaller phrases; these are easier to place because they take up less screen space. If you end up mixing large with large, you are more likely to cause overlap and less likely to have an optimal tag cloud.

6. Hints on Operation

THIS WILL TAKE A LONG TIME TO RUN MULTIPLE GENERATIONS as long as there are chromosomes that produce positive scores. To shorten execution time, decrease the number of generations it runs. That is in the while statement on line 154 of the C# code. To visualize the execution (and to be able to check its progress), I tend to run it in debug mode. It takes a little longer, but at least I can pause it and check what generation I'm on. (I set a watch on the variable **`total_gens`**).

I made some tweaks to the code since this screen shot, and the scores do slowly improve (this was before I was randomizing), but it still takes a very long time to run sufficient generations to provide the possibility of an optimized tag cloud.