Website Genome Project

Using website trees and webpage attributes to analyze the “DNA” of the Internet
Source code: https://github.com/danielqu/WebGenomeProject

For Senior Thesis in Computer Science
Author: Daniel Qu
Advisor: Dr. Richard Yang
Date: May 1, 2013
CPSC 490 Senior Project
I. INTRODUCTION

Websites are becoming increasingly complex, and therefore harder to understand and design. In 2012, 52 million new websites were created and added to the Internet. As of today, there are an estimated 634 million websites publicly accessible online.¹ With such a wealth of information being added online every single day, the amount of information and data has far exceeded our ability to access them. The Web Genome Project (WGP) aims to better understand and design websites. WGP leverages an algorithm to dissect the Internet’s “DNA”, creating a framework to characterize and compare websites with one another. This is accomplished by crawling each website and assigning attributes and traits based on the structure and markup elements of its internal pages. Currently, 16 website traits are implemented and collected in detail, and the source code has been open sourced to accommodate future developers who may add their own website attribute functions.

The first step in the WGP is to build a tree for a given website. Specifically, WGP starts with an entry point website domain name. From there, the algorithm navigates the root node’s internal links (links belonging to the same website) to create the branches of the website tree. The algorithm uses a breadth-first approach to build the tree, prioritizing the depth of the link when building the tree, and avoiding any duplicates and loops. At each node, the algorithm opens the URL and scrapes the entire page markup for all 16 traits. Each node is populated with this data. The tree is completed when all internal links have been traversed and the website tree has been completely built. This process can be repeated for any number of websites.

The second step in WGP is to compare the website trees created by step one. To do this, the algorithm chooses an “anchor tree” as the means for comparison. It then traverses every node in the anchor tree. For each node, the algorithm tries to find an existing node in tree2 that is most identical, with priority given to the depth of the node and the values of its attributes data. After this is done for all nodes in the anchor tree, a cardinal value is returned, specifying the “distance” between the two tree structures.

The tree structure of each website, as well as their comparative distance from other tree structures have several useful applications. Among them include categorization of sites, elimination of duplicates sites, inclusion of natural language processing, and plagiarism detection. Early attempts at implementing these applications have been done, though much more development work is necessary before any useful commercialization of this concept.

To test WGP, the algorithm was run on an Amazon EC2 server overnight with a list of 9000 randomly generated URLs. After 40 hours of runtime, the algorithm was able to scrape 45 entire websites. This included scraping over 30,000 individual webpages and collecting over 500,000 individual points of data. The results can be viewed in human-readable format in attached files ResultsScraped.txt and ResultsCompare.txt on Github.

¹ Source: http://royal.pingdom.com/2013/01/16/internet-2012-in-numbers/
II. DATA STRUCTURE AND SCRAPER ARCHITECTURE

WGP uses a three dimensional list to store the entire tree structure of any given website. The dimensions of the array are specified as follows:

tree[x] contains the node x in the tree. x = 0 indicates that it is the root node.
tree[x][0] contains the url of the current webpage x

tree[x][1] contains the depth of x in the tree

tree[x][2][0…m] contains all of the attribute data points collected from x

tree[x][3][0…n] contains the array location of all internal links on page x

Example of a root node in tree:
["http://benhulse.com", 0, ["Data", 26, 0, 16, 0, 0, 0, 0, 0, 162, 50, 52, 0], ["Links", 0, 1, 2, 3, 4, 5, 6]]

For the root node, the algorithm collects data elements from the page markup and stores them in the Data subarray. The algorithm also detects all internal links and adds them to the Links subarray. It then creates nodes at positions 1, 2, 3, 4, 5, and 6 and creates new branch nodes at those locations.

Example:
["http://benhulse.com/filter/design", 1, ["Data", 19, 0, 14, 0, 0, 0, 0, 0, 121, 36, 38, 0], ["Links"]]
["http://benhulse.com/filter/photography", 1, ["Data", 7, 0, 9, 0, 0, 0, 0, 0, 49, 12, 14, 0], ["Links"]]
["http://benhulse.com/filter/music", 1, ["Data", 3, 0, 7, 0, 0, 0, 0, 0, 25, 4, 6, 0], ["Links"]]
["http://benhulse.com/filter/Art-Direction", 1, ["Data", 18, 0, 14, 0, 0, 0, 0, 0, 115, 34, 36, 0], ["Links"]]
["http://benhulse.com/filter/Design", 1, ["Data", 19, 0, 14, 0, 0, 0, 0, 0, 121, 36, 38, 0], ["Links"]]
["http://benhulse.com/filter/Photography", 1, ["Data", 7, 0, 9, 0, 0, 0, 0, 0, 49, 12, 14, 0], ["Links"]]
["http://benhulse.com/filter/Music", 1, ["Data", 3, 0, 7, 0, 0, 0, 0, 0, 25, 4, 6, 0], ["Links"]]

Each branch is then populated with it’s own information, and the process repeats itself. This example is a simple tree with only depth 1, so no new nodes are created beyond depth 1.

While traversing the tree, there are two main problems

1. Encountering loops
It is natural for a website to have many loops in it’s natural link structure. This problem has been solved. The algorithm collects a list of previously traveled links. When reviewing a new link to add to the tree, the algorithm first makes sure that it hasn’t been traveled before. Only then, will it get added to the tree, preventing any possible looping.

2. Depth First Tree Traversal
Initial implementations of the website tree created a depth first tree. This inherently skews the tree by creating one large branch with extreme depth, and then creating small offshoot branches from the large branch. This is not a proper representation of a website. The solution was carefully implementing a breadth-first approach where each level of the tree was traversed first, before moving on to a deeper level of links.
III. SITE ATTRIBUTES

Aside from creating a node from each new traversed link, it was necessary to populate each node with specific data from the node’s markup. For example, techcrunch.com/page1 and techcrunch.com/page2 are both nodes of techcrunch.com. Each of these nodes will have their own data array containing details collected from their html markup.

Below are the traits to be collected:
1. Links to external pages (out of site)
2. Links to internal pages
3. Images
4. Videos embedded
5. Headings -> Important to delineate structure
6. Heading types (h1, h2, h3, h4, h5, h6)
7. Words displayed on site
8. Paragraphs denoted p
9. Total characters in markup
10. Average character length of paragraph
11. Site Orientation: Left, Right, Center
12. Presence of Logo / Site Name
13. Header Elements (Number, type)
14. Footer Elements (Number, type)
15. Color Palette -> CSS RGB colors
16. HTML structure

Modularization and Extensibility:
In anticipation of future development efforts, each site attribute has been created into it’s own module in wgModules.py. New attributes/modules can easily be added by creating a new file containing the module, and then simply adding the module name to the existing array of modules defined in wgModules.py. When running the scraping algorithm, the new data will automatically be appended to the end of the Data list in each node of the tree.

Additional Tags include Sign In / Sign Up, Contact Form, Buttons (similar to links, different CSS treatment), Payment Gateways: Paypal, Credit Card, Payza, Skrill, Search Engine Optimization, and typography.

Additional Ideas:
1. Use an element hierarchy: defined by tags embedded within each other. This leads to groupings, which are important to determine structure and presentation. After a relevant attribute is scraped, the code should count how deep it is embedded in the page (how many tags it is contained within).

2. Natural Language Processing:
A python library is readily available at http://nltk.org/. Integrating natural language processing
will potentially give much more information about a specific web page. It would give information on the actual human-readable text on a page. This would allow easy categorization of webpages depending on certain keywords, and other potential applications to make the DNA much richer.

IV. WEIGHTING AND COMPARISONS

After the Algorithm scrapes each website and builds a tree, it stores each tree in machine-readable format. These trees are permanently stored, and are the main input into the different applications for WGP. Most importantly, the trees are compared with one another to determine their differences, or distance in structure and data from one another. The algorithm returns a single cardinal number to declare the differences between two sites.

First, the comparison algorithm chooses an anchor tree (this can also be an empty tree). This will serve as a baseline for comparison. Afterwards, all other trees that have been previously stored are compared one by one with the anchor. The algorithm traverses through every single node in the anchor tree. For each node, it looks into the second tree (tree2) for a node of the same depth that best matches it’s data attributes. The data points between those nodes are then collected through the distance formula \( \sqrt{(x_1 - x_2)^2 + \ldots} \). Once the distance has been calculated, it is multiplied by a weight given the depth of the node in the tree. Nodes closer to the root have an order of 10 greater weights than the set of nodes below it.

After two nodes are compared and their comparative weights are stored, the two nodes are then removed from future consideration. The algorithm repeats this loop until all nodes in the anchor tree have been compared. If there is no similar node in tree2 at any point, the node in the anchor tree is compared with an empty node. After all nodes in the anchor tree are traversed and if there are still nodes remaining in tree2, those nodes are all assigned weights and subtracted from the overall distance collected to that point.

As an end result of the weighting and tree comparison portion of WGP, all trees stored by the algorithm will have been given a cardinal distance from all other trees. Theoretically, these distances can be used to create a graph with all trees as points on the graph. These weights are useful because they give us a singular means to characterize an entire website in comparison to another.

It is worth noting that there are multiple ways to compare website trees and assign weights. The current codebase is flexible and allows for developers to easily implement their own comparison and weighting algorithms. To do this, a developer would just swap out wgCompareMass.py with their own algorithm. All website trees have been stored in machine-readable format through the python “pickle” library and stored in folder wgTrees/. The developer’s code would merely need to “unpickle” the trees and then compare the trees with their own algorithm.
V. RUNNING THE ALGORITHM

The algorithm was run on randomly generated websites on an Amazon EC2 server. The algorithm was given a list of 9000 randomly generated URLs. Over 40 hours of runtime, the algorithm was able to complete 50 out of 9000 websites. When traversing the 50 websites, the algorithm scraped more than 30,000 individual web pages and collected over 500,000 unique data points. Human-readable versions of these trees were automatically written to ResultsScraped.txt. Afterwards, the tree structures of the 50 websites were compared with one another, and results were automatically recorded to ResultsCompare.txt files.

Studying the final structure of website distances, it can be seen that similar websites do indeed have smaller distances with each other than non-similar websites. However, the current code base is still a work in progress, and the problem is much bigger than to be tackled in this single project. The limitations are discussed below.

Limitations

WGP doesn't current work with web applications where user data entry is necessary, such as logging in, searching, etc. Additionally, the program currently abstracts the appearance of the site given by data collected from the html document. This can be very different from the information that human vision can pick up about a given website.

Overall, what this means is that the project has many elements that need to be implemented that are beyond the scope of the current efforts. It would be necessary to gather many more attributes on each website. Realistically, several hundred different attributes should be collected before a truly representative view of a single website, or even webpage can be given.

Additionally, the current code takes a long time to run with enormous trees (which are characteristic of many websites on the internet). This indicates that the code can use a lot of optimization improvements.

Some possible solutions involve:
- Greater computing power. The code was run on Amazon EC2’s micro servers (free tier). If the algorithm was run on better hardware, the runtime could see greater improvements.
- Implement the code in C, which would be a thousand times faster. However, the code itself will be a lot more difficult, since it will no longer have the easy workings of the python language and it’s parsing libraries.
- The code currently uses some brute force approaches to solve problems. This involves sorting, searching, etc. All of these processes can be simplified with more efficient algorithms.
VI. APPLICATION 1: PLAGIARISM DETECTION

Plagiarism detection is the process of discovering whether a document or piece of work is plagiarized from another source. Plagiarism in computer source code is also frequent, and requires different tools than those found in textual document plagiarism. Significant research has been dedicated to academic source-code plagiarism. The main difference between our approach and traditional plagiarism detection for essays and plaintext is that there may be no common purpose between one website and a copied website. That is, an online shop may be completely copied, but it will most likely used for a different brand. Thus, all of the actual content on the page will be changed. Therefore, we care most about the structure and design of the site.

Plagiarism detection programs for source code include Plague, the YAP programs (YAP, YAP2, and YAP3), JPlag, and MOSS.

One main reason these detection programs are highly implemented these days is that code that one programmer will write for one company (under contract) may suddenly appear at another company.

The main application here is whether the code can detect if somebody stole or duplicated the source code for a website. This can be simplified to: Suppose someone takes two distinct templates (very different), and created two web pages. Do they come from the same template or not?

VII. APPLICATION 2: SITE CATEGORIZATION

One significant application is to group websites into categories. This translates into determining whether the collected attributes allow us to place a website into one or multiple categories. Using the characteristics to evaluate a website, can the algorithm distinguish between a blog, business website, academic website, online shop, online newspaper, web application (ie Gmail, Facebook, etc).

Additionally, this may help discover whether a website span multiple categories, or even help discover new categories or subcategories to group websites into.

Examples:
1. Landing Page - Livelyfe.com
   Trait: Small depth website, simple features, fewer characters.

   Trait: Blog Headlines and Descriptions
3. Online Shop - Crosscloth.com, Dodocase.com
   Trait: item catalogue
   Trait: shopping cart

4. Online Newspaper – NY Times.com, Techcrunch.com
   Trait: Story Headlines and Descriptions

5. Academic Websites - Yale.edu, Stanford.edu, tphs.net.
   Trait: Highly informational

VIII. RELATED WORKS

1. The Music Genome Project
   The Music Genome Project was an effort to "capture the essence of music at the fundamental level" using almost 400 attributes to describe songs and a complex mathematical algorithm to organize them. Five Genomes (Pop/Rock, Hip-Hop/Electronica, Jazz, World Music, and Classical) were codified.
   Each song is represented by a vector (a list of attributes) containing approximately 400 "genes". Each gene corresponds to a characteristic of the music, for example, gender of lead vocalist, level of distortion on the electric guitar, type of background vocals, etc. The system depends on a sufficient number of genes to render useful results. Each gene is assigned a number between 0 and 5, in half-integer increments. Given the vector of one or more songs, a list of other similar songs is constructed using a distance function. Each song is analyzed by a musician in a process that takes 20 to 30 minutes per song.
   The company was renamed Pandora Media and launched Pandora Radio on July 21, 2005.
   The Music Genome Project is one of core technologies currently used by Pandora to play music for Internet users based on their preferences.

2. The Document Genome Project
   Docracy.com is a startup with a goal of creating a database of open legal documents. After raising 225k from investors, they have spearheaded the Document Genome Project, built based on the insight that “Many lawyers and others use a standard template for documents, and then lift certain phrasing from other contracts to mix and match.” The algorithm characterizes documents based on attributes, and allows users to search for similarities with existing patents. This technology will be commercialized to help with finding patents and filing patent applications.

3. iOS Interface Builder
   The iOS Interface Builder allows developers to simply dragging the object onto the window will add it to the view. The library is divided into 3 sections.
Controllers: These are the objects that control a set of associated views in a particular context of content. An example would be a Messages Controller showing messages that the user has available.

4. Pagerank
Pagerank is an algorithm named after Google’s Larry Page, and is a key component of Google Search. It assigns a cardinal weight to each website scraped by the engine based on the web of links from each site to one another. It would be interesting to apply this function to the individual internal web pages within a given website.

IX. TECHNOLOGIES USED

1. Software and Code Libraries

Python – Programming Language
Beautiful Soup – Scraping Functionality
“Beautiful Soup is a Python library for pulling data out of HTML and XML files. It works with your favorite parser to provide idiomatic ways of navigating, searching, and modifying the parse tree. It commonly saves programmers hours or days of work.”
Website: http://www.crummy.com/software/BeautifulSoup/

LXML Parser – parse html for Beautiful Soup library
Python’s own parse has some problems
http://www.crummy.com/software/BeautifulSoup/bs4/doc/#installing-a-parser

Urllib2 – Opening Web Documents
Documentation: http://docs.python.org/2/library/urllib2.html

2. Amazon Elastic-Compute 2 and Ubuntu Shell

Because Scraping services require high computational power, the code was set up on Amazon’s EC2 and executed in the cloud. The following steps describe how this process can be replicated:

Steps
1. Create an account on aws.amazon.com

2. Launch EC2 instance – Ubuntu Server 12.04.1 LTS 32 Bit
Make sure to install your key pair .pem file on a secure location on your local device. Also, enable port 22 security group to allow SSH into the instance.

3. SSH into the instance, update instance, check python version
ssh -i NewHaven.pem ubuntu@ec2-50-19-43-105.compute-1.amazonaws.com
sudo apt-get update
python --version (check for at least python 2.6, required by BeautifulSoup library)

4. Install Beautiful Soup and LXML parser libraries
sudo apt-get install python-lxml python-beautifulsoup
Credit for this easy install goes to https://news.ycombinator.com/user?id=lamby and http://wesleyzhao.com/python-web-page-scraping-installing-libxml

5. Copy relevant zipped files to EC2 Instance, and uncompress them
scp -i NewHaven.pem somefile.zip ubuntu@ec2-50-19-43-105.compute-1.amazonaws.com:~/scraper
sudo apt-get install zip / sudo apt-get install unzip

6. To move files back on to local machine

7. Unzip the file
unzip zipfile.zip

8. Run the code with nohup (nohangup), close terminal, and run it overnight.
sudo nohup python wgMain.py scrape compare &
Take note of the pid.
ps aux - to see all processes
sudo kill <pid> - to end the process

3. Research for building web applications (Yii Framework)

Yii is short for “Yes-it-is!” It is an open source, object-oriented, component-based PHP application framework. It derives its namesake for its versatility, able to adapt impressively for both rapid prototyping, and large projects that need security, speed, and scalability. It is predicted to soon be one of the highest performing frameworks available to developers. The framework may be leveraged to create the documentation for WGP.
X. SOURCE CODE

All Source Code can be downloaded here:
https://github.com/danielqu/WebGenomeProject

README

Welcome to the Web Genome Project

wgMain.py

Call with:
python wgMain.py [scrape] [compare] [emptycompare] [applications]

wgMain Manual:
python wgMain.py [scrape] [compare] [emptycompare] [applications]
1. scrape will build trees from the file Websites.txt
2. compare will compare all trees and assign cardinal differences
3. emptycompare will compare all trees with an Empty Tree
4. applications will allow you to run applications with existing trees

Websites.txt: Websites to scrape in directory
ResultsScraped.txt: Contains results of scraped trees
ResultsCompare.txt: Contains results of tree comparisons

INCLUDED FILES

wgMain.py #Main access point to source code
wgAlgorithm.py #Scraps websites and builds breadth-first tree
wgModules.py #Contains all attributes collected by each node
wgCompare.py #Manually call to compare two trees (deprecated)
wgCompareMass.py #Compares Anchor tree with all trees in wgTrees
wgApplications.py #Contains all applications to use on wgTrees directory
ResultsScraped.txt #Contains all trees in human-readable form
ResultsCompare.txt #Contains all comparison distances in h-r form
Websites.txt #List of 9000 randomly generated URLs