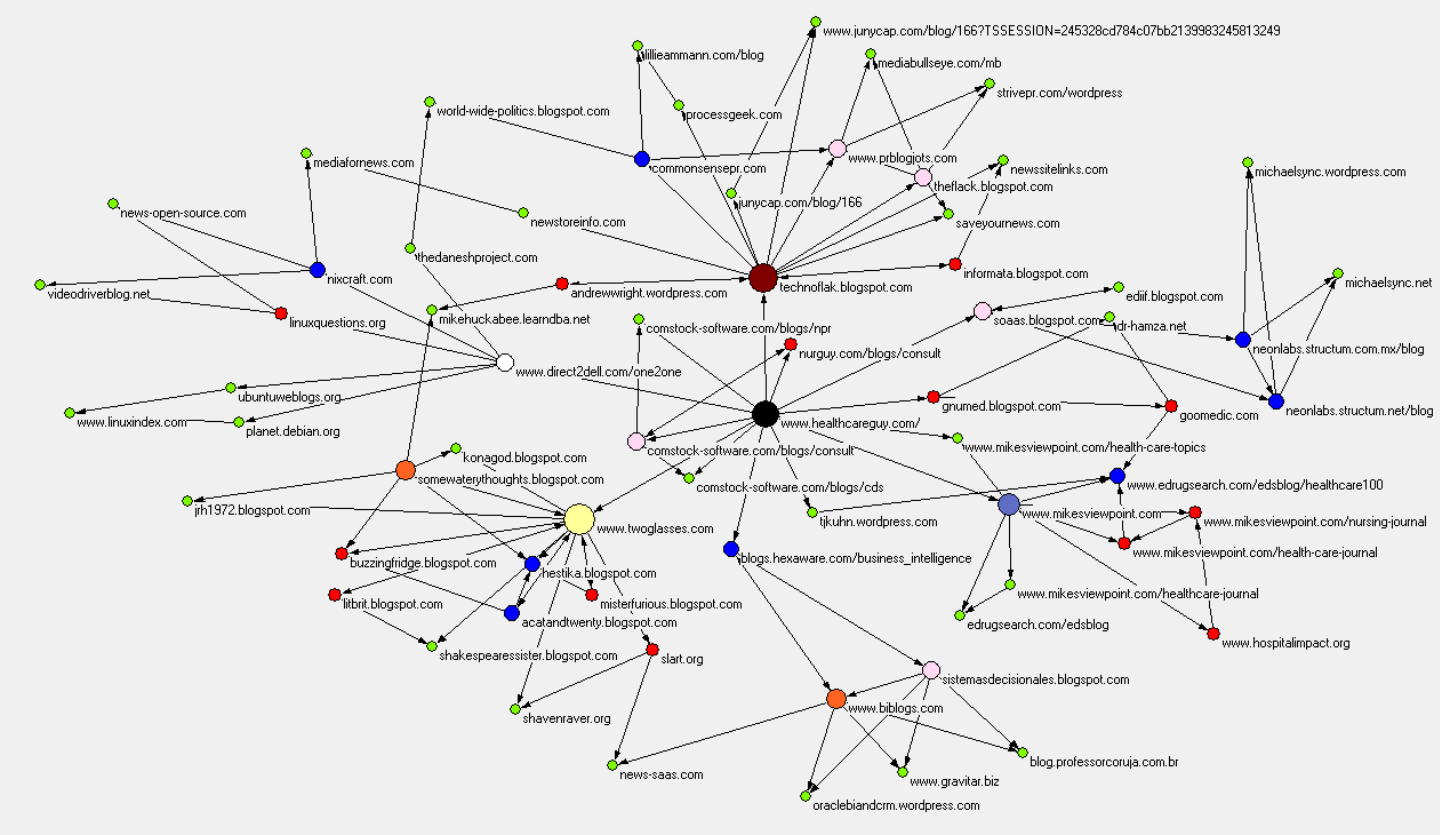
URLNET: A PYTHON CLASS LIBRARY FOR GENERATING NETWORKS FOR ANALYSIS

DRAFT—ALPHA

Revision 0.82—03 January, 2009

Dale Hunscher, MSI
University of Michigan Medical School



TECHNORATI COSMOS FOR HEALTHCAREGUY.COM

VERSION HISTORY

DATE	AUTHOR/EDITOR	REVISION	DESCRIPTION
24 APR 2008	DALE HUNSCHER	0.5	Initial alpha release
05 MAY 2008	DALE HUNSCHER	0.6	Added ability to turn property values into additional columns on GUESS networks.
30 MAY 2008	DALE HUNSCHER		Added example using search engine class and merge it with a "target" network.
23 JUN 2008	DALE HUNSCHER	0.7	Better diagram for site map example; minor touch-ups here and there.
07 OCT 2008	DALE HUNSCHER	0.71	Added note about spurious warning message when running technorati1.py.
26 NOV 2008	DALE HUNSCHER	0.81	In urlutils.py, changed lookup code for urlnet.cfg to look in current directory in addition to the directories listed in the PATH environment variable. Upper-cased PATH for compatibility with POSIX-compliant systems. In technoratiTree.py, removed single and double quotes (if any) around the passed Technorati API key, and added lookup in urlnet.cfg if None is passed for the key value.
03 JAN 2009	DALE HUNSCHER	0.82	Added save/load capability. Changes to urlutils.py and two new example programs.

TABLE OF CONTENTS

TABLE OF FIGURES

1 INTRODUCTION

When studying aspects of the World Wide Web using network analysis tools and techniques, building networks for analysis can be tedious, time-consuming, and error-prone. The UrlNet library, written in the Python scripting language, is intended to provide a powerful, flexible, easy to use mechanism for generating such networks.

1.1 OUTLINK HARVESTING AND WEB SERVICE APIS

Starting at a given page (the "root" page), we can collect, or "harvest", the outlinks on the page and follow them to other pages. We can harvest the outlinks on these pages, and follow them, et cetera, to the search depth we desire. This creates a directed graph which may loop back on itself, and is therefore not an acyclic graph. It is often useful to create such networks when studying phenomena on the Web, for example the findability of a given page from a known starting-point page.

While URLs are typically thought of as links to Web pages, Web Services based on the SOAP/WSDL standards and also lightweight APIs based on the REST paradigm can also be accessed via URLs. Social networking sites such as Technorati, Delicious, and Bibsonomy provide such APIs. The National Library of Medicine offers URL-based APIs that access its Medline database and its bioinformatics data repositories. There are many others too numerous to mention.

Because UrlNet is written as a class library, it is easy to specialize and/or extend its functionality by subclassing one or more of its classes. One example provided shows how to build networks using the Technorati Cosmos API by subclassing two classes and writing a total of four fairly short functions.

I currently use the library to gather result sets periodically from several popular search engines, for research I plan to do in assessing the quality of consumer health search. There are many other possible uses, waiting to be discovered.

1.2 TREES AND FORESTS

Some URL-based networks are by nature best thought of as upside-down tree structures, for example a site map. Such networks are actually directed graphs, since lower-level nodes may have links to siblings and ancestor nodes as well as their own child nodes. Nonetheless, each node can be said to exist at some level under the root node (e.g., the home page in a site map).

Other networks—often more interesting networks—can be depicted as "forests" rather than trees. A forest has multiple root nodes, each of which is a tree in its own right. The trees in the forest can have links to other trees, but there is no requirement that any given tree link to any other tree at all.

One example of a forest is the union of URL trees created from each of the URLs extracted from a search engine query result set. The root URLs of a forest could also come from a data set obtained from a search engine provider, such as the 2006 AOL data set (see http://gregsadetsky.com/aol-data/ for sources). One might, for example, investigate the "Vocabulary Problem" to see how the result sets of different search queries expressing the same concept overlap (or fail to overlap).

UrlNet supports building several variations on forest networks. The vanilla version uses a caller-provided list of URLs as the root nodes for the forest. Another variation allows you to supply a URL that will not be included in the network as the root node, but whose outlinks form the list of URLs used as the forest roots. We call the initial URL in this variation a *phantom root*. Other variations build on these routines, applying them, for example, to result sets from various search engines, including but not limited to Google, Yahoo! Search, Windows Live Search, and AOL Search.

There is even a way to create a forest from a set of URLs and to unify them into a tree structure by the addition of an arbitrary root node. We call the arbitrary root node a *placeholder root*.

1.3 GENERATING INPUT DATA FOR NETWORK VISUALIZATION SOFTWARE

UrlNet generates an abstract data structure in memory, from which any number of outputs could potentially be generated. Currently, the library supports output to three formats: a printable text hierarchy, Pajek project files, and GUESS network files. Other formats can be added easily.

I chose Pajek and GUESS because they are freely available programs, and because they are the software tools on which I learned everything I know about network visualization. You can download them from the Web at the addresses shown in the following table.

Pajek	Download from the Pajek Wiki at http://pajek.imfm.si/doku.php - the wiki has links to other useful software tools for working with Pajek, and tells how to sign up for the Pajek mailing list. There is a very insightful book written by W. de Nooy, A. Mrvar, V. Batagelj, entitled *Exploratory Social Network Analysis with Pajek*, published in 2005 by the Cambridge University Press. There is a downloadable version of Pajek that is much older and less functional than the most current version, but has the advantage of being fully compatible with the command set described in the book. UrlNet-generated Pajek projects are fully compatible with both the current version and the version that is compatible with the book. The older version of Pajek can be downloaded at http://vlado.fmf.uni-lj.si/pub/networks/data/esna/Pajek.be.exe.
GUESS	GUESS is the creation of Eytan Adar, and can be downloaded at the main GUESS website, http://graphexploration.cond.org/. UrlNet support for GUESS is presently limited to building simple networks for URL and domain networks. I have been using Pajek longer and more frequently than GUESS, so the GUESS facilities are more primitive than I would like them to be.

1.4 ACKNOWLEDGEMENTS

I owe a great debt to Drs. Lada Adamic and Suresh Bhavnani, my advisory team for my Master's thesis work at the University of Michigan School of Information. Both are much younger and wiser than I, and I have learned humility in their presence. Lada taught me almost everything I know about networking, and Suresh inspired me to go beyond my limits in exploring Pajek's features.

2 GETTING UP AND RUNNING

2.1 INSTALLING THE LIBRARY

At present there is no automated install, but the procedure is very simple. It assumes you have already installed Python 2.5 and downloaded the UrlNet zip file. Python 2.5 is required—my apologies to the die-hards.

2.1.1 UNZIP THE DISTRIBUTION

Unzip the distribution file to a location of your choice. It will create a tree with the following structure.

```
urlnet-v1.0
       urlnet
       examples
       doc
       conf
```

The version number in the name of the root directory may be different from what is shown here.

2.1.2 COPY THE LIBRARY DIRECTORY TO YOUR PYTHON INSTALLATION'S LIB/SITE-PACKAGES DIRECTORY

The directory referred to here is the urlnet subdirectory under the urlnet-v1.0 directory, which in turn is found under wherever you unzipped the distribution file. You can actually put it anywhere in the Python path (found in the PYTHONPATH environment variable). The Python installation's Lib/site-packages directory is the normal place to keep third-party modules, which is why I am recommending it here.

2.1.3 UPDATE URLNET.CFG AND COPY TO A LOCATION ON THE EXECUTION PATH

Edit the urlnet.cfg file in the conf subdirectory to, at minimum, set the workingDir vale to the default working directory of your choice. You can override this at will, but this eliminates the proliferation of calls to os.chdir.

Other variables you may want to set now are your Technorati key, if you have one, and your email address. The email address is an optional but polite parameter for the Web APIs of the National Center for Biomedical Informatics. Neither is needed until you actually work with a Technorati or NCBI example, but if you are editing, you might as well take care of it.

2.1.4 COPY THE EXAMPLE PROGRAMS TO A LOCATION OF YOUR CHOICE

The working directory you set in urlnet.cfg is a good starting point. If you copy the examples there, you won't have to modify any code to override the working directory.

2.1.5 TEST THE INSTALL

From your operating system shell's command line, enter the command

```
python urltree1.py
```

It should take a short while to run, after which you should find a small Pajek project file named urltree1.paj in the working directory.

2.2 THE EXAMPLE PROGRAMS

Numerous example programs are provided, illustrating many of the features of the library. Most are only a few lines long, making them easy to understand and also demonstrating the power of the library. Longer programs show more complicated scenarios, such as setting up a production batch program that runs periodically to retrieve the current state of a URL network.

Example program	Network build method	Comments
The Basics: Trees, forests, and site maps; Pajek, GUESS, and hierarchical formats; saving and reloading networks		
urltree1	UrlTree.BuildUrlTree	This is the "Hello World" of UrlNet, the simplest useful program possible. Constructs a 2-deep tree from the root URL of a simple website and generates a Pajek project.
urltree2	UrlTree.BuildUrlTree	Starting with the code from urltree1, this example overrides a constructor parameter to create a 3-deep tree from the root URL of a simple website.
urltree3	UrlTree.BuildUrlTree	Starting with the code from urltree1, this example uses page titles as node names (or the URL if a page does not have a title element).
urlforest1	UrlTree.BuildUrlForest	Builds a vanilla URL forest from a hard-coded list of URLs.
generateguessnets1	UrlTree.BuildUrlTree	Generates network files for GUESS, one for urls and one for domains
printhierarchy1	UrlTree.BuildUrlTree	Shows how to create a formatted printout of a tree-structured network.
printhierarchy2	UrlTree.BuildUrlForest	Demonstrates that the method for creating a formatted printout of a forest network works the same as for a tree-structured network.
changeworkingdir1	UrlTree.BuildUrlTree	Our "Hello World" program (urltree1) modified to show how to set the working directory to something other than the value in the configuration file.
sitemap1	UrlTree.BuildUrlTree	Shows how to create a site-map network.
phantomroot1	BuildUrlForestWithPhantomRoot	Retrieves the National Library of Medicine's MedlinePlus melanoma portal and creates a URL forest from its outlinks, then writes a Pajek project.

Example program	Network build method	Comments
placeholderroot1	UrlTree.BuildUrlTreeWithPlaceholderRoot	Shows how to create a forest from a set of URLs and unify it into a tree by creating a placeholder root.
savetreeloadtree1	urlutils.saveTree, urlutils.loadTree	Illustrates saving and reloading a small network.
savetreeloadtree2	urlutils.saveTree, urlutils.loadTree	Illustrates saving and reloading a much larger network.
Debugging and performance benchmarking facilities		
logging1	UrlTree.BuildUrlTree	Shows how to turn on logging.
logging2	UrlTree.BuildUrlTree	Shows how to tee logging output into a file of your choice.
logging3	UrlTree.BuildUrlTree	Exercises all features of the logging facility.
Advanced controls and operations		
ignorabletext1	UrlTree.BuildUrlTree	Shows how to leave URLs out of the network.
redirects1	BuildUrlForestWithPhantomRoot	Shows how to redirect the recursion from parameterized URLs to an embedded URL in each URL's parameter list.
ignorablesandredirects1	UrlTree.BuildUrlForestWithPhantomRoot	Combines the use of an ignorable text fragment list and a redirects list in one program.
partitions1	UrlTree.BuildUrlTree	Demonstrates two ways to create additional Pajek partitions from node properties.
netsmerged1	AOLTree. BuildUrlTreeWithPlaceholderRoot	Shows how to generate and merge multiple networks.
regexqueryurl1	UrlTree.BuildUrlTree	Harvest links from a top-level page using Python regular expressions.
Working with search engines		
searchengine1	GoogleTree.BuildUrlForestWithPhantomRoot	Use the GoogleTree class to create a forest from the result set of a Google query.
searchengine2	AOLTree. BuildUrlTreeWithPlaceholderRoot AOLTree.BuildUrlTree	Generate an AOL search engine result set network from the query `quit smoking`, and merge it with a network based on recommended links found in the MedlinePlus smoking cessation portal.
Web APIs 1: Technorati		
technorati1	TechnoratiTree.BuildUrlTree	Use the Technorati Web API to generate a network illustrating Technorati "cosmos" for a blog. Also shows how to use a SAX XML parser to process data. Requires a Technorati

Example program	Network build method	Comments
		API key and the 4suite XML library.
Web APIs 2: The National Center for Biomedical Informatics		
ncbiauthorcosmos1	NCBIAuthorCosmosTree. BuildUrlTree	Use NCBI PubMed APIs to generate a network illustrating the "cosmos" of an author (co-authors, publications, and MeSH headings associated with the pubs)
ncbicoauthortree1	NCBICoAuthorTree. BuildUrlTree	Use NCBI PubMed APIs to generate a network illustrating the co-author relationships for an author.
ncbilinkstree1	NCBILinksTree. BuildUrlForestWithPhantomRoot	Leverages the NCBI eLinks Web API to find connections between a gene and related proteins, nucleotides, and SNPs, and to construct a network therefrom.
Ncbilinkstree2	NCBILinksTree. BuildUrlForestWithPhantomRoot	Same program but with different query and databases.

3 THE BASICS, PART 1: TREES

3.1 INTRODUCTION

In this chapter, we'll look at how to generate trees and forests in their most basic form, by providing a URL (to create a tree) or a list of URLs (to create a forest).

3.2 GENERATING A TREE-STRUCTURED NETWORK FROM A URL

The simplest use of UrlNet is to generate a network by following the outlinks for a single URL:

```
from urlnet.urltree import UrlTree
net = UrlTree()
net.BuildUrlTree('http://www.southwindpress.com')
net.WritePajekFile('swp', 'swp')
```

This four-line program will generate a 2-deep network and write a Pajek project file in the work directory specified in urlnet.cfg. The WritePajekFile function takes two arguments: the first is the root of the generated network names, and the second is the file name, to which the .paj extension is appended.

Once this code executes (urltree1.py, by the way), a Pajek project file (swp.paj) should appear in the directory specified in the os.chdir call. It is not very interesting, because it only includes the URL passed to BuildUrlTree and its immediate outlinks. As you can see below, it contains four small networks (URLs and domains, directed and undirected) and two partitions (level for URLs and for domains). The value associated with each arc or edge is the frequency value associated with the arc or edge, i.e., the number of times this connection between URLs occurred.

The Pajek project file contains the following (it's the only one you'll see in the manual because they're all structured the same):

```
*Network urls_swp_directed
*Vertices 5
       1 "www.southwindpress.com"
       2 "www.southwindpress.com/catalog-scenplanning.html"
       3 "www.southwindpress.com/catalog-hci.html"
       4 "www.southwindpress.com/consulting.html"
       5 "s28.sitemeter.com/stats.asp?site=s28SouthWindPress"
*Arcs
       1       2 2
       1       3 2
       1       4 2
       1       5 1
*Edges

*Network urls_swp_undirected
*Vertices 5
       1 "www.southwindpress.com"
       2 "www.southwindpress.com/catalog-scenplanning.html"
       3 "www.southwindpress.com/catalog-hci.html"
       4 "www.southwindpress.com/consulting.html"
       5 "s28.sitemeter.com/stats.asp?site=s28SouthWindPress"
*Arcs
*Edges
```

```
       1       2 2
       1       3 2
       1       4 2
       1       5 1


*Partition URLLevels.clu
*Vertices 5
       0
       1
       1
       1
       1
*Network domains_swp_directed
*Vertices 2
       1 "southwindpress.com"
       2 "sitemeter.com"
*Arcs
       1       2 1
*Edges

*Network domains_swp_undirected
*Vertices 2
       1 "southwindpress.com"
       2 "sitemeter.com"
*Arcs
*Edges
       1       2 1


*Partition DomainLevels.clu
*Vertices 2
       0
       1
```

The following diagram shows Pajek's rendition of the directed Urls network. The blue node is the root or level 0 (zero) node, and the yellow nodes are level 1 nodes, representing the outlink anchors found in the HTML page obtained by following the root URL.

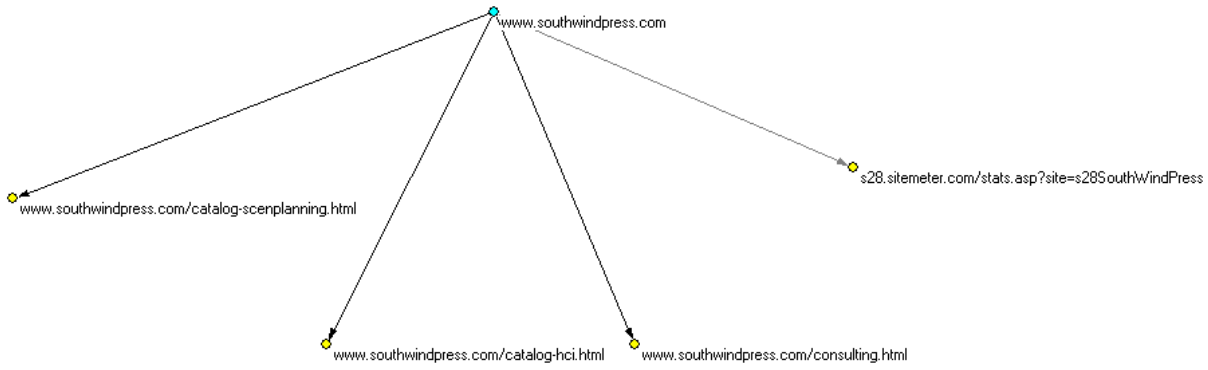


FIGURE 1 - A SIMPLE 2-LEVEL URL NETWORK

3.2.1 A LITTLE DEEPER, IF YOU PLEASE

The networks in the above project are most uninteresting, in part because it's a trivial website structure, but also partly because the UrlTree constructor defaults to only two levels. Let's change the UrlTree() call in the above example script to:

```
. . .
net = UrlTree( _maxLevel = 3 )
. . .
```

This code is found in urltree2.py. It produces a three-tier network with a number of additional nodes:

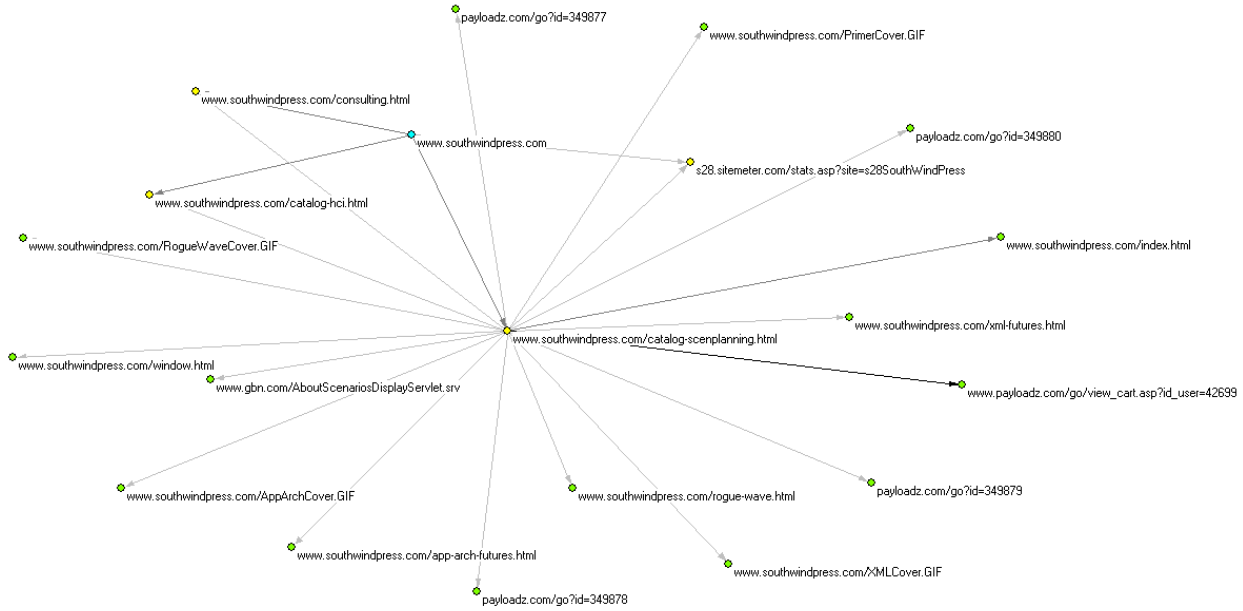


FIGURE 2 - SAME NETWORK INCREASED TO 3 LEVELS.

In addition to the blue root (level zero) node and the yellow level 1 nodes, there are now green nodes that are level 2. The _maxLevel parameter tells the UrlTree instance how many layers deep the network should be. The node numbering, which is used in generating a Pajek partition, is zero-based, hence the highest partition number will be _maxLevel – 1. The level value is included as an additional node attribute in GUESS networks.

This network is a little more interesting, but as I look at it, I realize I don't really care about payloadz.com and sitemeter.com pages, which are related to e-commerce and web analytics respectively. Fortunately, there is a way to exclude them from the network, as we shall see in chapter 8, Advanced controls and operations. For now, though, let's stick with the basics.

3.2.2 USING TITLES INSTEAD OF URLS AS NETWORK NODE NAMES

URLs are often long and cryptic. In fact, it can seem that their clarity is inversely proportional to their length! In urltree3.py, we take the code from urltree1.py and add a single parameter to the line that generates a Pajek project:

```
. . .
net.WritePajekFile('urltree3', 'urltree3',useTitles=True)
. . .
```

By setting the optional useTitles argument to WritePajekFile to True, we end up with a network that uses the page titles rather than their URLs wherever possible:

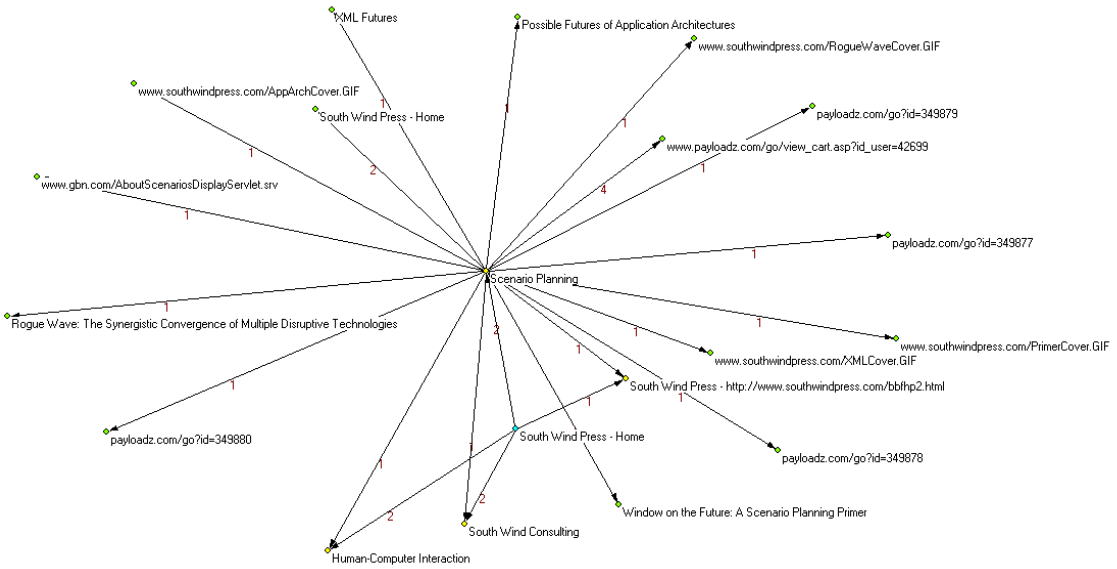


FIGURE 3 - 3-LEVEL NETWORK WITH PAGE TITLES FOR NODE NAMES INSTEAD OF URLS

That's a lot more readable! You can use this wherever you like, without further work. The routine that reads the page via HTTP also scans for a title element and if one is found, sets a property called 'title' on the object that represents the URL.

When we get to working with Web Service APIs, the HTTP GET will often return a document that has no title element, so in those cases you'll either need to do without this feature, or set the title property yourself to some meaningful datum from the retrieved document. We'll look at properties in more detail in chapter 7.

3.3 OVERRIDING THE WORKING DIRECTORY

We can override the working directory specified in urlnet.cfg by passing the working directory as another UrlTree constructor parameter. Here's how it's done, with modifications to urltree1.py:

```
from urlnet.urltree import UrlTree

# change this to a value that works for you...
workingDir = 'C:\\Users\\dalehuns\\Documents\\Python\\blogstuff'
```

```
net = UrlTree(_maxLevel=2, _workingDir=workingDir)
net.BuildUrlTree('http://www.southwindpress.com')
net.WritePajekFile('changeworkingdir1', 'changeworkingdir1')
```

The the new example is called changeworkingdir1.py. The output is the same as before. You'll need to modify the code in this example to reflect your local directory structure.

3.4 SITE MAPS

It is often useful to generate a site map, which may start at the root of a domain or at some node further down in the tree. Control of the construction of site maps is accomplished through two new arguments and one old argument to the UrlTree constructor.

The new arguments are _singleDomain and _showLinksToOtherDomains. Setting the _singleDomain argument to True limits network building to the domain of the root URL. Setting _showLinksToOtherDomains to True includes external links, i.e. links to URLs outside the domain; the outlinks from these will not be followed regardless of domain.

The old argument that is involved in site map construction is _maxLevel. With site maps, it is important to set the _maxLevel argument to a sufficiently large number, in order that it should include all levels of the site.

Here's the code to sitemap1.py, which shows how to build a site map network:

```
# sitemap1.py
from urlnet.urltree import UrlTree

# test building a site map network
net = UrlTree(_maxLevel=4,
              _singleDomain=True,
              _showLinksToOtherDomains=True
              )
net.SetProperty('getTitles',True)
net.BuildUrlTree('http://www.southwindpress.com/')
net.WritePajekFile('sitemap1', 'sitemap1',useTitles=True)
```

If you look at the generated Pajek project file (sitemap1.paj), you'll notice that there are some nodes, such as links to GIF and JPG format files, that do not normally appear on UrlNet-generated networks. That's because graphics, Java applets, and other objects that are not HTML are of no interest in most types of network generation, but of considerable interest in site maps.

Here's the resulting network visualized in Pajek:

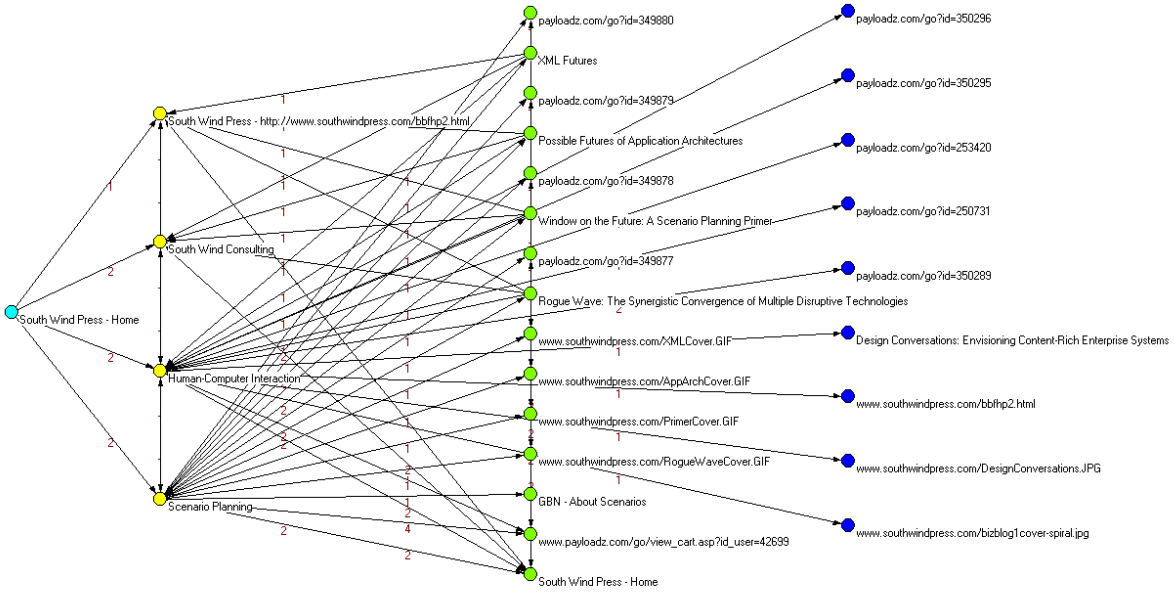


FIGURE 4 - A SITE MAP NETWORK

4 THE BASICS, PART 2: FORESTS

A forest is a collection of trees. In the networking sense, the trees in a forest may connect to each other, thereby sharing nodes. These interconnections are optional; any individual tree and any number of interconnected trees can form a component within a network. Analysis of forests often consists of looking for such components and attempting to identify the "glue" that holds a component together, the absence of which separates the components from each other.

UrlNet supports two types of forests, the selection of which is based on the author's research experience: forests derived from a list of URLs, and forests created from the outlinks of a page that is not itself included in the network (a "phantom root").

4.1 INTRODUCTION TO FORESTS

A forest is a network created by combining the outlink trees generated from a list of URLs with no root node. There are other ways to create a forest, but this simplest method is the basis for the others. We call it a "vanilla" forest. A simple example of this is a list of URLs from a search engine log data set. In this example, we'll look at a very few URLs from The MSN Search data set, and generate Pajek and GUESS networks. This is sample program urlforest1.py.

```
from urlnet.urltree import UrlTree

msn_melanoma_urls = (
'www.melanoma.com/site_map.html',
'www.skincancer.org/melanoma/index.php',
'www.melanoma.org/',
'www.mpip.org/',
'www.cancerresearch.org/melanomabook.html',
'www.nlm.nih.gov/medlineplus/melanoma.html',
)

net = UrlTree(_maxLevel=2)
success = net.BuildUrlForest(Urls=msn_melanoma_urls)
if success:
    net.WritePajekFile('urlforest1', 'urlforest1')
```

SIDEBAR: Checking status of high-level function calls

You'll notice that in the above example, we store the return value from net.BuildUrlForest in a variable called `success`. It will contain a Boolean value reflecting whether a fatal exception occurred. BuildUrlForest calls BuildUrlTree for each URL in the list, and any of these calls may fail due to ignorable text or a failure to match on redirects (more on these in section 8.2, A bit of rocket science: Handling redirection). These failures are not necessarily real failures, so the high-level function calls ignore them. Turn on logging to see messages that indicate problems at a lower level.

What do the merged outlink networks from this set of URLs show us? The domain network reveals something when we look for weak components:

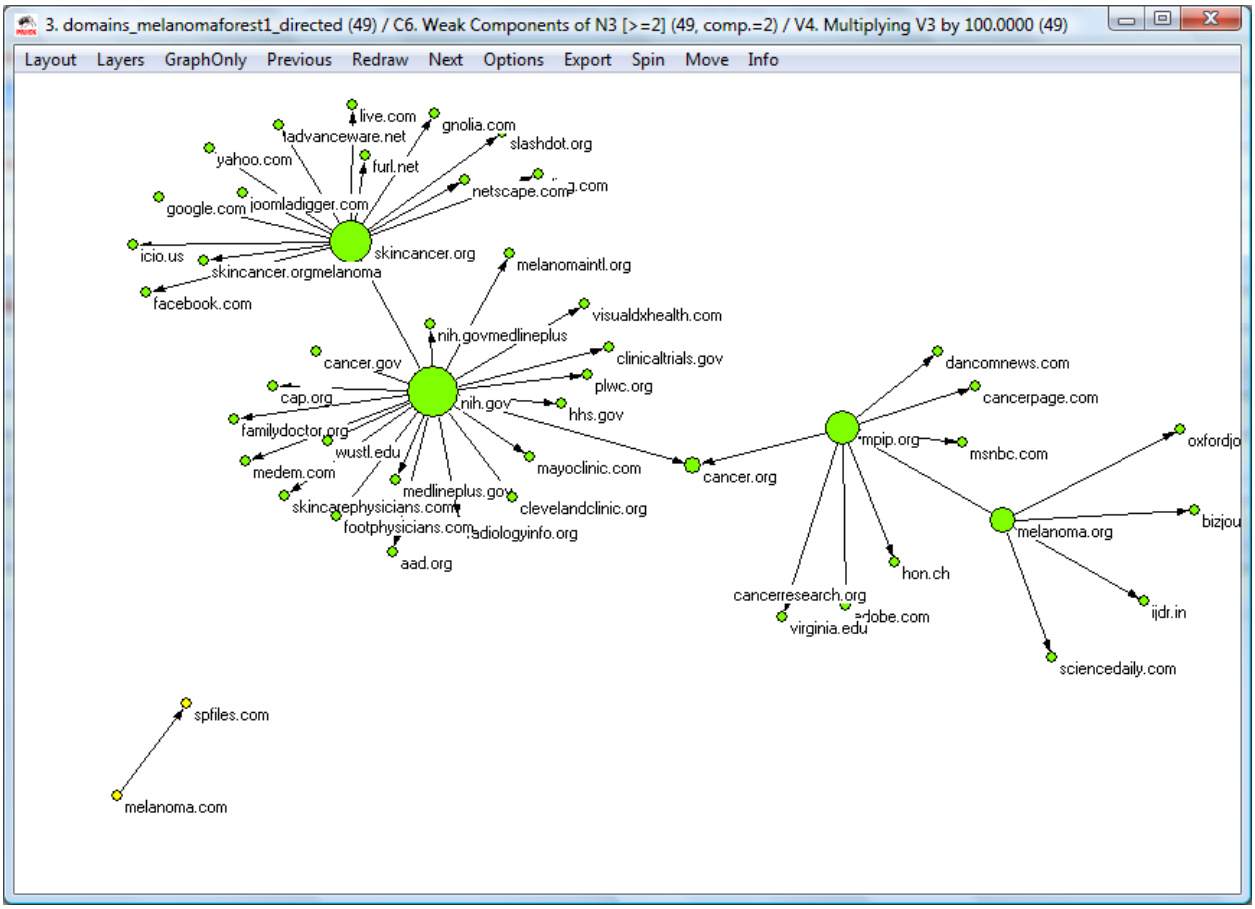


FIGURE 5 - A FOREST NETWORK

The URLs in the forest were links chosen by information seekers from the results of melanoma-related search queries. The non-profit organizations, including government agencies associated with the NIH and three foundations, are connected to each other. The tiny component in the lower left is melanoma.com, the site of a pharmaceutical drug, Intron A®, from the Schering Corporation, a therapy that is used in the treatment of melanoma. It neither references nor is referenced by any of the nodes in the non-profit component. The size of nodes indicates the number of inbound and outbound links.

This is a toy network generated from a few URLs, selected to illustrate the use of the UrlNet library rather than to uncover facts, so the inferences one can draw from this in terms of domain knowledge are limited. Nonetheless, you should be able to glean from this example some insight into the kinds of things you can learn through network analysis.

4.2 Phantom Roots: Generating a forest on-the-fly

Sometimes it is desirable to create a network from the outlinks of a page, without including the page itself in the network. For instance, suppose we want to study the pages referenced by the Medline Plus melanoma page, but want to treat them as a forest rather than a tree, to look for components (a tree will always be a single component).

In the example below, we gather further evidence regarding online search for information about melanoma by looking at the links recommended on the Medline Plus melanoma page. We use this page as a "phantom root", i.e., a root node that will go away once it spawns child nodes, thereby creating a forest without a single root node. We also use the ignorableText feature to exclude pages that are themselves part of the National Library of Medicine's Medline Plus site. You'll find a detailed explanation about this feature in section 8.1, Ignoring specific URLs and URL types in building the network.

```
from urlnet.urltree import UrlTree

net = UrlTree(_maxLevel=2)
ret =
net.BuildUrlForestWithPhantomRoot("http://www.nlm.nih.gov/medlineplus/melanoma.html")
if ret:
    net.WritePajekFile('phantomroot1', 'phantomroot1')
    net.WriteGuessFile('phantomroot1urls',doUrlNetwork=True)
    net.WriteGuessFile('phantomroot1domains',doUrlNetwork=False)
```

This is sample program phantomroot1.py. Here's an analysis of the network generated by the above code, rendered in Pajek with some labels added in a bitmap editing program:

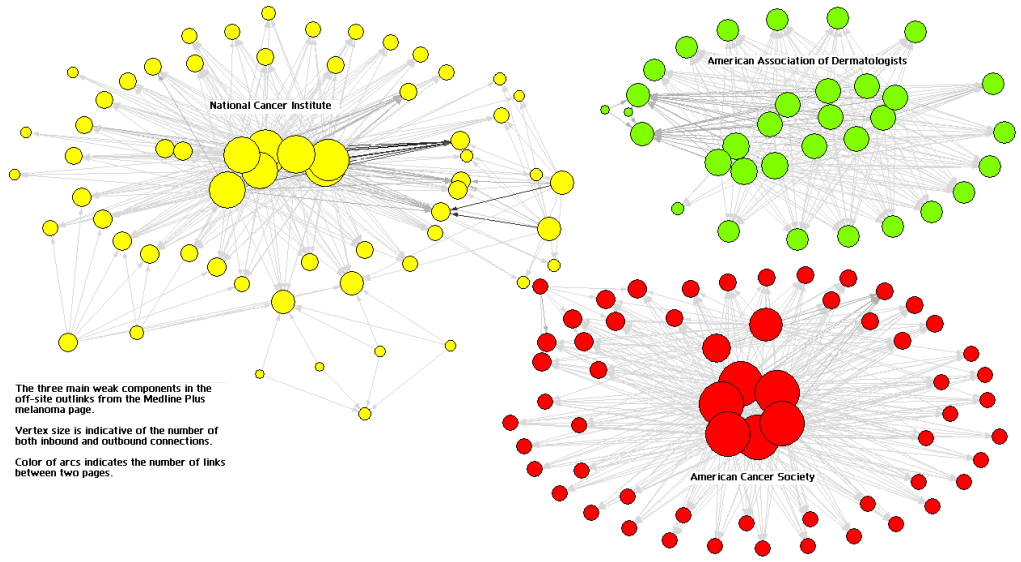


FIGURE 6 - FOREST NETWORK GENERATED FROM PHANTOM ROOT (CLEANED AND DRESSED UP A BIT)

The original directed URL network had 1331 nodes. Analysis started with the removal of all nodes with less than three links (inbound or outbound). To accomplish this, an all-degree partition was created, and a new 169-node network was extracted using partitions 3-*. A search for weak components of size > 10 was conducted, and another network was extracted that removed the twelve nodes not connected to a component. This resulted in the network shown above, with 157 vertices. An all-degree partition and normalized vector were created, and the vector multiplied by 200. This vector was used with the

components partition to create the diagram shown above, which revealed that there are three primary components. The three turn out to be non-profit cancer organization websites that are completely independent of each other for all practical purposes.

4.3 Placeholder Roots: Turning a Forest into a Tree

Sometimes it is useful to create a root node "above" a forest to unify it, for example when you have a list of URLs that are derived from a search engine data set such as the AOL data set (as previously noted, you can visit http://gregsadetsky.com/aol-data/ for sources). This root node is a "placeholder root". The following code (placeholderroot1.py) accomplishes this:

```
from urlnet.urltree import UrlTree

some_msn_melanoma_urls = (
'www.melanoma.com/site_map.html',
'www.skincancer.org/melanoma/index.php',
'www.melanoma.org/',
'www.mpip.org/',
)

net = UrlTree(_maxLevel=2)
success = net.BuildUrlTreeWithPlaceholderRoot(\
    rootPlaceholder="http://search.msn.com/",\
    Urls=some_msn_melanoma_urls)
if success:
    net.WritePajekFile('placeholderroot1', 'placeholderroot1')
```

To give some imaginary context to this example, the list of URLs might be the anchor list from an earlier version of the page, no longer directly accessible on the Web. We would add a placeholder root to show the provenance of the URLs in what would otherwise be a forest.

Here's the domain network resulting from a run of the above script:

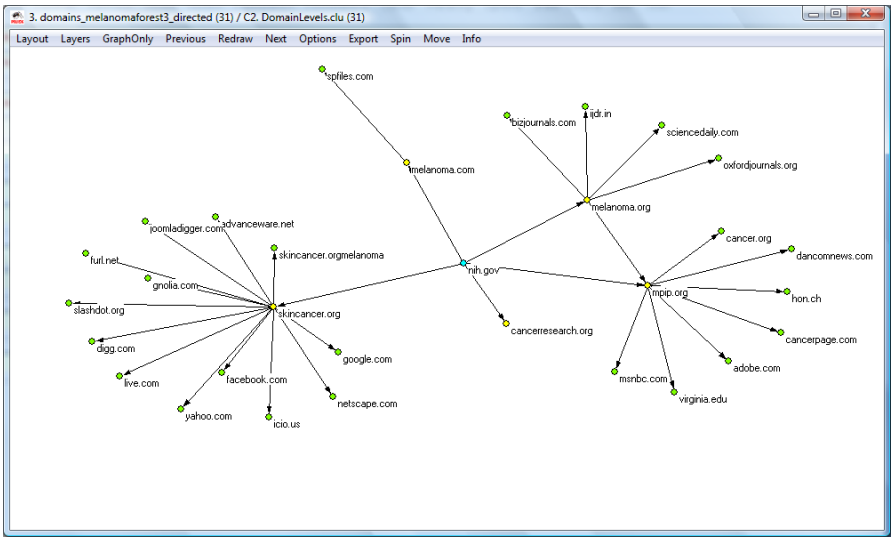


FIGURE 7 - NETWORK WITH PLACEHOLDER ROOT NODE

5 GENERATING OUTPUT

5.1 OVERVIEW

The UrlTree network creation routines produce a data structure in memory. This is quite a feat in and of itself, but it is of little practical use unless you generate some sort of output from the in-memory structure. Currently the UrlNet library supports three output methods, two destined for network analysis programs, and one for the printer: Pajek projects, GUESS networks for URLs and domains, and a printable hierarchy.

5.2 GENERATING OUTPUT THAT IS INPUT FOR NETWORK ANALYSIS PROGRAMS

The library is capable of building network files for both Pajek and GUESS. Because the generation process creates both URL and domain networks, the network file building routines build both types.

5.2.1 PAJEK

In the examples above, we saw that the Pajek network generator creates both directed and undirected networks, hence four networks in total in a Pajek project file; it also creates a levels partition for the URLs and for the domains. The same partition works for both directed and undirected networks with respect to both URLs and domains.

5.2.2 GUESS

For GUESS (the Graph Exploration System), we can only write one network at a time, as its current incarnation does not have the concept of a project. For this reason, there are two ways to execute the GUESS network generation function. The function takes two arguments: the first is the root name for the file, to which the extension urls.gdf (for URL networks) or domains.gdf (for domain networks) will be appended. The second argument is a Boolean flag. When the Boolean flag is set to True (the default), a URL network is generated; when its value is False, a domain network is generated. In the code below, from generateguessnets1.py, we replace the WritePajekFile call with two calls to WriteGuessFile:

```
. . .
# generate GUESS URL network
net.WriteGuessFile('generateguessnets1urls',doUrlNetwork=True)

# generate GUESS domain network
net.WriteGuessFile('generateguessnets1domains',doUrlNetwork=False)
```

Although we didn't use it here, WriteGuessFile also takes an optional useTitles argument, which affects what appears in the url attribute of GUESS nodes. The generated GUESS URL network looks like this:

```
nodedef>name VARCHAR, url VARCHAR,domain VARCHAR
southwindpress_com1,www_southwindpress_com,southwindpress_com
southwindpress_com2,www_southwindpress_com_catalog_scenplanning_html,southwindpress_com
southwindpress_com3,www_southwindpress_com_index_html,southwindpress_com
southwindpress_com4,www_southwindpress_com_catalog_hci_html,southwindpress_com
southwindpress_com5,www_southwindpress_com_consulting_html,southwindpress_com
gbn_com6,www_gbn_com_AboutScenariosDisplayServlet_srv,gbn_com
southwindpress_com7,www_southwindpress_com_RogueWaveCover_GIF,southwindpress_com
southwindpress_com8,www_southwindpress_com_PrimerCover_GIF,southwindpress_com
```

```
southwindpress_com9,www_southwindpress_com_AppArchCover_GIF,southwindpress_com
southwindpress_com10,www_southwindpress_com_XMLCover_GIF,southwindpress_com
southwindpress_com11,www_southwindpress_com_rogue_wave_html,southwindpress_com
southwindpress_com12,www_southwindpress_com_window_html,southwindpress_com
southwindpress_com13,www_southwindpress_com_app_arch_futures_html,southwindpress_com
southwindpress_com14,www_southwindpress_com_xml_futures_html,southwindpress_com
edgedef>node1 VARCHAR,node2 VARCHAR,frequency INT
southwindpress_com1,southwindpress_com2,1
southwindpress_com1,southwindpress_com4,1
southwindpress_com1,southwindpress_com5,1
southwindpress_com2,southwindpress_com3,1
southwindpress_com2,southwindpress_com4,1
southwindpress_com2,southwindpress_com5,1
southwindpress_com2,gbn_com6,1
southwindpress_com2,southwindpress_com7,1
southwindpress_com2,southwindpress_com8,1
southwindpress_com2,southwindpress_com9,1
southwindpress_com2,southwindpress_com10,1
southwindpress_com2,southwindpress_com11,1
southwindpress_com2,southwindpress_com12,1
southwindpress_com2,southwindpress_com13,1
southwindpress_com2,southwindpress_com14,1
```

The visualization in GUESS looks like this, with a radial layout and a teeny bit of manual manipulation:

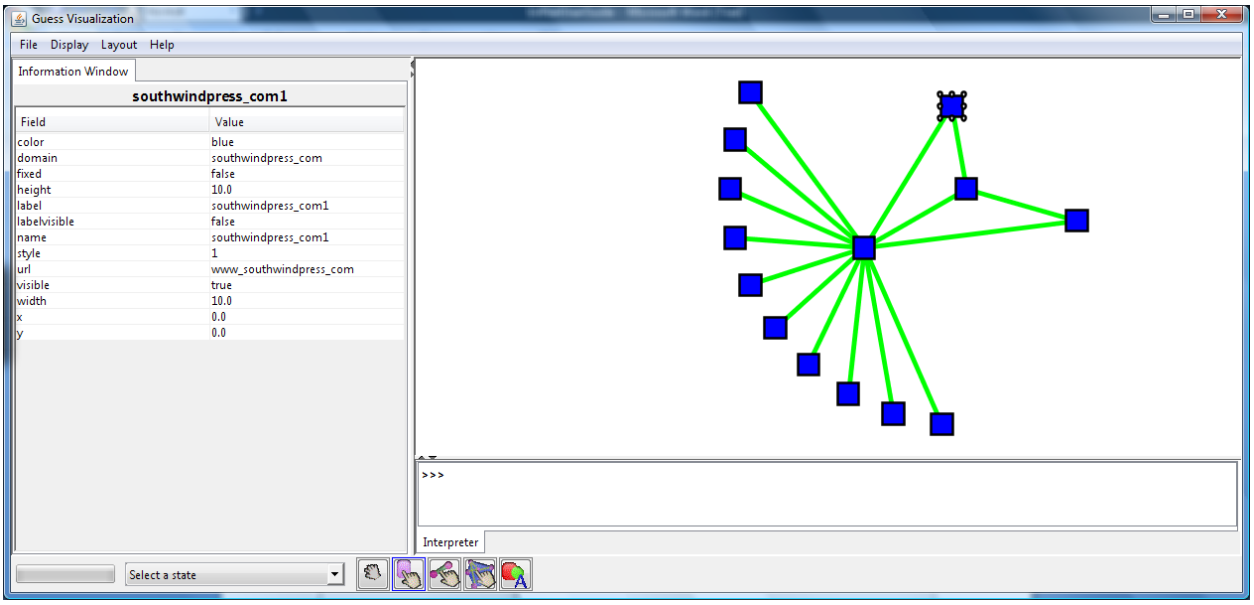


FIGURE 8 - 2-LEVEL NETWORK IN GUESS

As with generated Pajek networks, the arcs have the frequency value associated with them, i.e., the number of times this connection between URLs occurred. To comply with GUESS requirements for node names, alphanumeric characters in the URLs have been replaced with underscores.

The example script also generated a GUESS network of domains, but it's not worth illustrating here because it contains only two nodes.

A great deal more can be done with GUESS than is shown in this manual. GUESS's data structure is easily extensible. I haven't been working much with GUESS lately, so I don't have a good example to show; that will be coming soon. See the GUESS home page (http://graphexploration.cond.org/), the GUESS Wiki (http://guess.wikispot.org/), and Dr. Lada Adamic's GUESS demo site (http://www-personal.umich.edu/~ladamic/GUESS/index.html) for more information.

5.2.3 When vertex names are too long…

Sometimes the generation process produces vertex names that are unwieldy. You can set a limit on the name length by setting a property called 'nodeLengthLimit'. We'll learn more about properties in section 7.1.1.1, The property list.

5.3 Generating a printable hierarchy

The example programs printhierarchy1.py and printhierarchy2.py generate a tree-structured and a forest network respectively, then print the network to a file in tabbed hierarchical format.

Here's the code for printhierarchy1.py.

```
# printhierarchy1.py
from urlnet.urltree import UrlTree
from urlnet.urlutils import PrintHierarchy

net = UrlTree(_maxLevel=3)
success = net.BuildUrlTree('http://www.southwindpress.com')
if success:
    try:
        net.WriteUrlHierarchyFile('printhierarchyurls1.txt')
        net.WriteDomainHierarchyFile('printhierarchydomains1.txt')
    except Exception, e:
        print str(e)
```

The code for printhierarchy2.py is different only in how it builds the network, using BuildUrlForest instead of BuildUrlTree and passing a list of URLs instead of a root URL. Writing the printable hierarchy to a file is the same for both, though in printhierarchy2.py, we add an additional argument, useTitles, to the WriteUrlHierarchyFile() call. This argument defaults to False, indicating that the URL itself should be used to denote the node. By setting it to True, we cause the node to be represented by its title (if any). If no title element was found, the URL is used instead.

```
# printhierarchy2.py
from urlnet.urltree import UrlTree

msn_melanoma_urls = (
'http://www.melanoma.com/site_map.html',
'http://www.skincancer.org/melanoma/index.php',
'http://www.melanoma.org/',
'http://www.mpip.org/',
'http://www.cancerresearch.org/melanomabook.html',
'http://www.nlm.nih.gov/medlineplus/melanoma.html',
)
```

```
net = UrlTree(_maxLevel=1)
success = net.BuildUrlForest(Urls=msn_melanoma_urls)
if success:
    try:
        net.WriteUrlHierarchyFile('printhierarchyurls2.txt',useTitles=True)
        net.WriteDomainHierarchyFile('printhierarchydomains2.txt')
    except Exception, e:
        print str(e)
```

Here's an example of the output, showing the domains of a forest network:

```
********** Domain Network ***************

melanoma.com
       spfiles.com
skincancer.org
       advanceware.net
       digg.com
       del.icio.us
       google.com
       live.com
       facebook.com
       slashdot.org
       netscape.com
       furl.net
       yahoo.com
       gnolia.com
       joomladigger.com
melanoma.org
mpip.org
       medscimonit.com
       oxfordjournals.org
       feedburner.com
       medicalnewstoday.com
       adobe.com
       virginia.edu
       cancer.org
       msnbc.com
       dancomnews.com
       cancerpage.com
       www.hon.ch
cancerresearch.org
nih.gov
       medlineplus.gov
       familydoctor.org
       cancer.gov
       aad.org
       skincarephysicians.com
       radiologyinfo.org
       cap.org
       mayoclinic.com
       footphysicians.com
       visualdxhealth.com
       wustl.edu
       clinicaltrials.gov
       cancer.net
       medem.com
       melanomaintl.org
```

```
        clevelandclinic.org
        hhs.gov
```

The URLTree class member routines WriteUrlHierarchyFile and WriteDomainHierarchyFile accept an argument called writeHeaders that defaults to True. Pass False for this argument, and the header line (e.g., '********** Domain Network **********' in the above example) will not be printed.

5.4 WRITING TO A STREAM INSTEAD OF A FILE

Each of the UrlTree class member output routines (e.g., WritePajekFile()) has a corresponding member function that accepts a stream descriptor instead of a filename. The arguments and defaults are the same otherwise. This will be useful, for example, when using the GUESS applet and generating a network dynamically for inclusion in the applet's HTML code. I'm working on an example of this.

6 PERSISTENT NETWORKS: SAVE NOW, LOAD LATER

Because the process of generating networks by spidering Web links is time-intensive, the ability to save a network and reload it later is a genuine productivity aid. UrlNet provides easy-to-use facilities for exactly that purpose.

The following statement imports the necessary functions for saving and reloading into your Python script:

```
from urlnet.urlutils import saveTree, loadTree
```

6.1 SAVING A NETWORK

To save a network, you call the saveTree function, passing the network object instance and the name (or complete path) of the file in which to save the network. The file will be overwritten during the saveTree call.

```
saveTree(net,'savetreeloadtree1.pckl')
```

That's all there is to it.

.pckl?

The extension '.pckl' is not a requirement, but in the Python world this extension or '.pickle' are in common use. The reason is that the module in which the Python object save and load code resides is the Pickle module, and the processes of saving and loading are referred to as Pickling and Unpickling respectively. See http://www.python.org/doc/2.5.2/lib/module-pickle.html for more details about the Pickle module.

The object is saved in a cryptic format—some would say incomprehensible—but it is perfectly safe to regard the saved file as a black box. In engineer-speak, a black box is something whose contents are not visible but whose behavior is dependable.

6.2 RELOADING A PREVIOUSLY SAVED NETWORK

Reloading the object is as simple as assigning the result of a call to loadTree to a variable of your choice:

```
savedNet = loadTree(net,'savetreeloadtree1.pckl')
```

Once you have successfully loaded the object, the new network will be the exact same object (from a behavioral perspective) as the one you saved. It will be the same class and all its data will be in the same state as when saved.

6.3 CAVEATS AND EASTER EGG[1]

6.3.1 CAVEATS

- I have just begun playing with this feature, so it may be buggy, but it hasn't bit me so far.
- I haven't done any "test to destruction" scenarios, so I don't know what the performance degradation curve is like as the network being saved and reloaded gets larger and larger.
- Don't forget to import the appropriate module defining the network you are reloading. This is another scenario I haven't tested, but common sense suggests that the Python interpreter may not be able to find the module on its own.

6.3.2 EASTER EGG

- These functions are in no way specific to UrlNet, so common sense also suggests that you can safely use them to save and reload any kind of Python object, subject to the restrictions of the Pickle module itself. See the Python documentation for more details: you can find it at http://www.python.org/doc/2.5.2/lib/module-pickle.html. This is yet another scenario I haven't tested extensively, so there are no guarantees.

[1] In American and some European cultures, an Easter Egg is a decorated chicken egg (hard-boiled or emptied out through a pin-hole) that is one of the accoutrements of the Easter holiday season (http://en.wikipedia.org/wiki/Easter). Easter Eggs are often hidden, usually along with candy, and children are sent out on Easter Egg hunts on Easter morning, hence the term connotes any type of hidden treasure. In keeping with this meaning of the term, the term is also used to describe hidden and/or undocumented features or messages in a book, movie, musical work, or software program (http://en.wikipedia.org/wiki/Easter_egg_(media)).

7 LOOKING UNDER THE HOOD

7.1 TOURING THE CLASS HIERARCHY

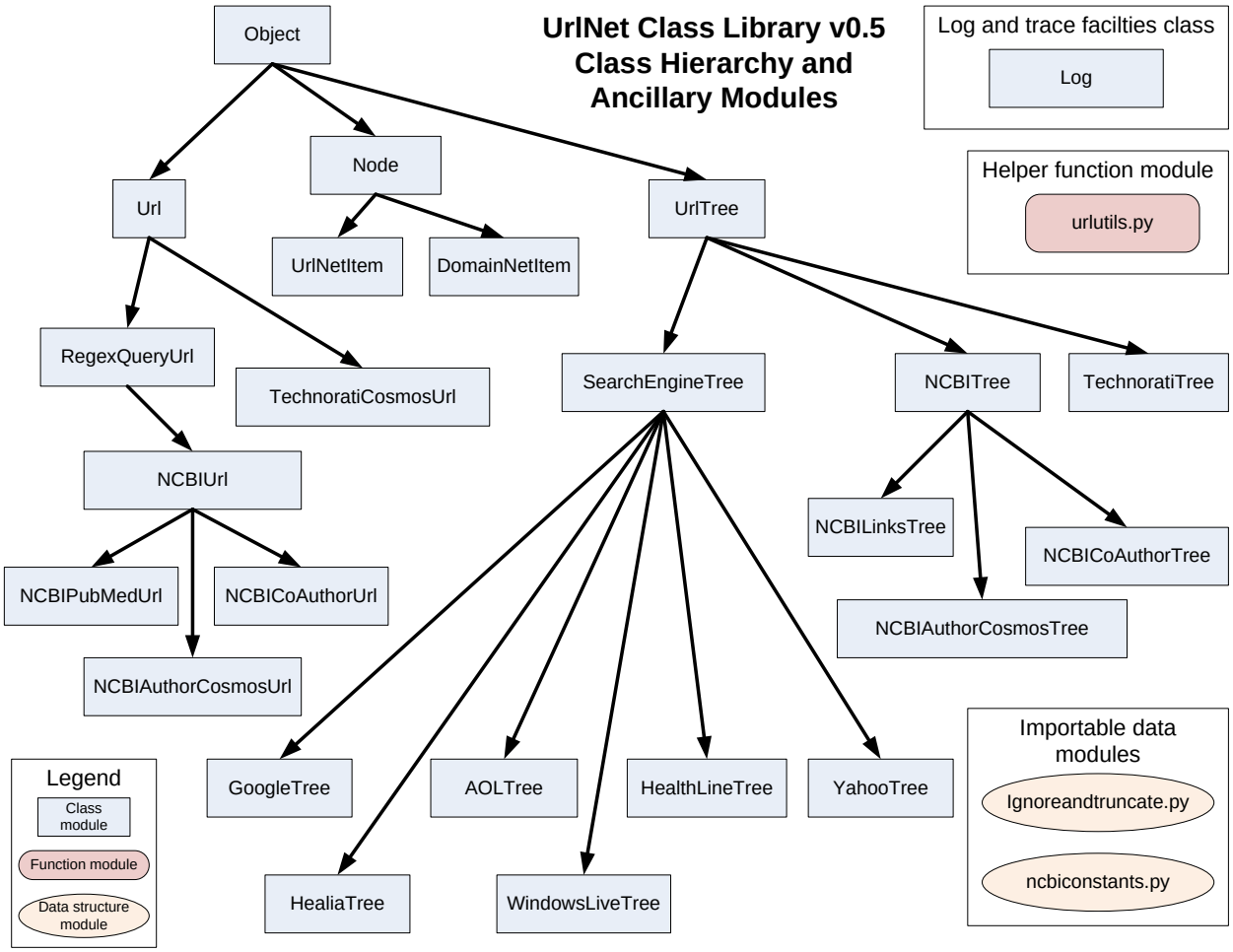


FIGURE 9 - URLNET CLASS AND MODULE HIERARCHY

There are five fundamental classes you should get to know. Others derive from or delegate functions to them.

7.1.1 OBJECT

The class Object is used as the root class for all UrlNet objects except for the Log class. Object handles error recording, and delegates to a Log instance the task of persisting error messages. The Log class handles function call tracing, timing for performance analysis, and logging of messages, to sys.stderr and optionally to an alternate file output stream.

The affordances of the Object class are always available through any instance of any of UrlNet classes except Log. Log is independent, to facilitate logging and trace in application programs and classes outside of the UrlNet class hierarchy, for example applications and classes that employ UrlNet objects.

7.1.1.1 THE PROPERTY LIST

Any instance of a descendant class of Object a dictionary of name-value pairs called the *property list*. Functions anywhere in the same instance, or on instances of other classes that have access to the instance, can retrieve property values. This is useful for setting session variables for use in classes where work is to be delegated, or for avoiding the need to create a derived class simply to add an additional data member.

The TechnoratiTree and TechnoratiCosmosUrl classes illustrate the use of this mechanism, wherein the TechnoratiTree instance stores a property whose value is the Technorati API developer key, required for all Technorati API calls. The RegexQueryUrl class expects its regular expression (or list thereof) to be set in a property on the encompassing UrlTree or descendant instance. The SearchEngineTree class and all classes derived from it use this mechanism, but the same mechanism would work for any network in which RegexQueryUrl is employed.

Another use of the property system, not illustrated yet in any of the examples, would be to store additional information to be used in network generation, e.g., additional fields in GUESS vertex and edge records, or additional types of partitions in Pajek network project generation.

7.1.1.2 RECORDING THE LAST ERROR

The Object class also has a facility for recording and retrieving the last error made. This is useful when you wish to have functions return an integer status that may carry additional information in case of errors. Python allows functions to return multiple values, but there are cases where you may want to return extra information only under certain circumstances. The functions SetLastError(), GetLastError(), and ResetLastError() are used to manage the last-error record stored in each instance of the Object class.

7.1.2 NODE

The Node class represents a node in the network. It is aware of its “parents” (incoming links from other nodes) and its “children” (outgoing links to other nodes). UrlNetItem and DomainNetItem are derived from Node and inherit its functionality.

7.1.3 URL

The Url class encapsulates a URL. Note the difference in case. The URL is a text string representing an address and additional specifying information needed to retrieve a content item (usually called a page) from the Internet. The Url class encapsulates the URL and the knowledge of how to use it to retrieve content and get the list of anchors (outlinks) in the content obtained.

7.1.4 URLTREE

The UrlTree class encapsulates a network. The name is a bit misleading, because UrlTree can actually encapsulate a forest-type network (i.e. one with multiple root nodes) as well as trees. This class has

methods for building trees and forests, and for generating input for the popular network analysis programs Pajek and GUESS.

7.1.5 LOG

The Log class is independent of the object hierarchy. It is used to generate a list of events, most often used for debugging, and to record performance information (i.e. how long a function takes between entry and exit).

7.2 TREE-BUILDING SCHEMATIC

The following diagram shows the operations and data structures involved in building the network in memory. There are a number of variations from this schematic in classes derived from UrlTree, e.g., some derived classes do not build domain networks.

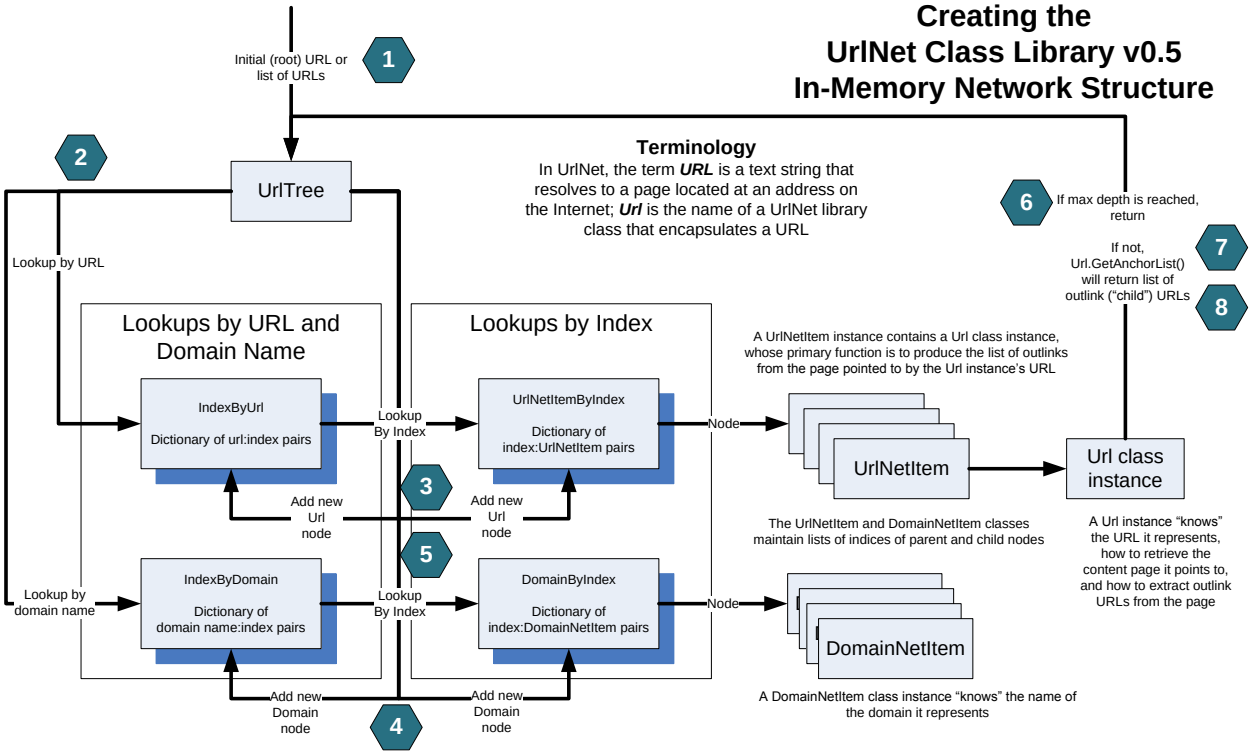


FIGURE 10 - SCHEMATIC OF THE NETWORK GENERATION ALGORITHM

7.2.1 ALGORITHM WALK-THROUGH

The tree is built using a depth-first search algorithm. The numbered step descriptions below correspond to the numbers in hexagonal blocks in the diagram above.

1. The root URL, or each of a set of root URLs in the case of a forest, is submitted to the tree using one of the top-level network-building functions.

2. The URL is looked up in the IndexByUrl dictionary.

3. If not found, a new UrlNetItem (node) instance is created, which in its constructor creates a new Url instance from the passed URL. The next available index number is used as the key and the new node object is used as the value of a new UrlnetItemByIndex dictionary entry. IndexByUrl is updated with the index of the new node as value and the URL as key.

4. The domain of the URL is looked up in IndexByDomain. If it is not found, a DomainNetItem node is created using the next available index value, then added to DomainByIndex. The index value is added to IndexByDomain as the value, with the domain name as key.

5. In both the UrlNetItem and the DomainNetItem, the parent index is added to this node's parent list, and the current node's index is added to the parent node's child list. If the index to be added is already present in either case, a count variable for that relationship is incremented.

6. If the maximum depth specified in the tree has been reached, return without retrieving the page and harvesting its outlinks. This ends recursion.

7. The node's outlinks are obtained by calling the GetUrlAnchors()function of the Url instance contained in the UrlNetItem instance, which returns a generator object that in turn will call the Url instance's GetAnchorList() function. Internally, GetAnchorList() calls GetPage(), which in turn calls RetrieveUrlContent() to access the Internet to retrieve the page content. Each of the three lower-level functions just mentioned (GetAnchorList, GetPage, and RetrieveUrlContent) plays a particular role, each of which is useful to understand if you are going to create subclasses of Url.

   GetAnchorList() is responsible for composing the URL actually used to retrieve content, and for parsing the returned page to harvest outlink URLs. GetAnchorList() may make multiple GetPage() calls in cases where the URL anchors must be aggregated from multiple page retrievals. GetPage() is responsible for assembling chunks into a composite page in cases where multiple RetrieveUrlContent() calls are necessary.

   Finally, RetrieveUrlContent() is responsible for the atomic HTTP open, get, and close operations; it also sleeps if a waiting time is defined in the UrlTree's sleeptime property. As an added bonus, it captures the content of the <title> element, if it exists, and sets a 'title' property on the Url object.

8. If the maximum depth has not already been reached, repeat steps 2-6 for each URL in the outlinks of the page by recursing down into GetUrlForest().

7.2.2 CHERCHEZ LA URL

You'll notice that in the description above, the step describing the Url class instance's responsibilities is the longest step by far. There's good reason for this, and it's reflected in the code. More than half of the library's 8,000+ lines of code is in the Url class or its derivatives.

Much of the variation in how different Web-derived networks are constructed has to do with the URL and its consequences—how it is composed, how the retrieved page is processed, and what data structure is derived from the page. This is normally a list of outlinks, but in some cases it will not, depending on the functionality of the Url class encapsulating the URL. The Node-derived class "owning" the Url-derived instance or the all-encompassing UrlTree-derived class must know how to handle the data structure returned by variant GetAnchorList() calls.

7.3 LOGGING AND TRACE FACILITIES

There are two utility classes that perform cross-cutting "grunt work" tasks: Object and Log. We've already discussed Object, whose facilities are available to all others by inheritance. The one exception is the Log class, which has no base class. Log provides facilities for programmers to write information useful for debugging when things go wrong, especially when it is necessary to write to something other than the standard output and error file descriptors, for example programs that run in the background without a user interface. Log also automatically captures function entry and exit, allowing you to follow the sequence of events leading to an exception, for example. Because it is capturing both entry and exit, it can also time functions so you can find performance bottlenecks.

7.3.1 LOG

By instantiating an instance of the Log class at the beginning of a function, its entry and exit will be recorded in logging output, along with the elapsed time (assuming the value of log.trace is set to True, its default value in the distribution code). If the elapsed time exceeds the limit defined in the log module, the log output will flag the function as a potential performance issue. The limit defaults to 10 seconds, but can be set dynamically as follows:

```
# ex6.py
from urlnet.urltree import UrlTree
import urlnet.log

urlnet.log.logging = True
urlnet.log.trace = True # expect things to slow down if this is set to True!
urlnet.log.limit = 5 # number of seconds beyond which a function is deemed slow

mylog = urlnet.log.Log('main')

. . .
```

Here's an example of the log output for a performance issue:

```
at 17:54:24 in foo (no args): exiting, 75.676000 secs **** > 10.000000 sec limit ****
```

To turn off the trace (which generates voluminous output and slows things down noticeably, by the way), use the following:

```
. . .
from urlnet.urltree import UrlTree
import urlnet.log
```

```
urlnet.log.logging = True
urlnet.log.trace = False
mylog = urlnet.log.Log('main')

. . .
```

Fortunately, you can turn any of the logging controls—logging, trace, altfd, and file_only—on and off at any time. It is possible, for example to create a trace file that is limited to the call tree underneath a particular function.

When logging is turned on, all errors recorded in SetLastError() are output to sys.stderr and to the optional additional output stream if one is set up. The following code is used to turn on logging:

```
# ex6.py
from urlnet.urltree import UrlTree
import urlnet.log

urlnet.log.logging = True
mylog = urlnet.log.Log('main')

. . .
```

The Log class is usable either inside or outside the primary UrlNet object type hierarchy. Descendants of Object delegate to the Log class most responsibility for logging and trace activities.

8 ADVANCED CONTROLS AND OPERATIONS

8.1 IGNORING SPECIFIC URLS AND URL TYPES IN BUILDING THE NETWORK

We can provide the UrlTree constructor with a sequence or list of text strings to look for in child URLs, and when it finds any one of these text strings, it will exclude the URL from the generated network. We do this by passing the list of ignorable text strings to the UrlTree constructor. We do this with a modification of the code we used in section 3.2.1.

```
# ignorabletext.py
from urlnet.urltree import UrlTree

ignorableText = [‘payloadz’,’sitemeter’,]

net = UrlTree(_maxLevel=3, _ignorableText=ignorableText)
net.BuildUrlTree('http://www.southwindpress.com')
net.WritePajekFile(' ignorabletext', ' ignorabletext')
```

This produces the same network as before, minus three URLs:

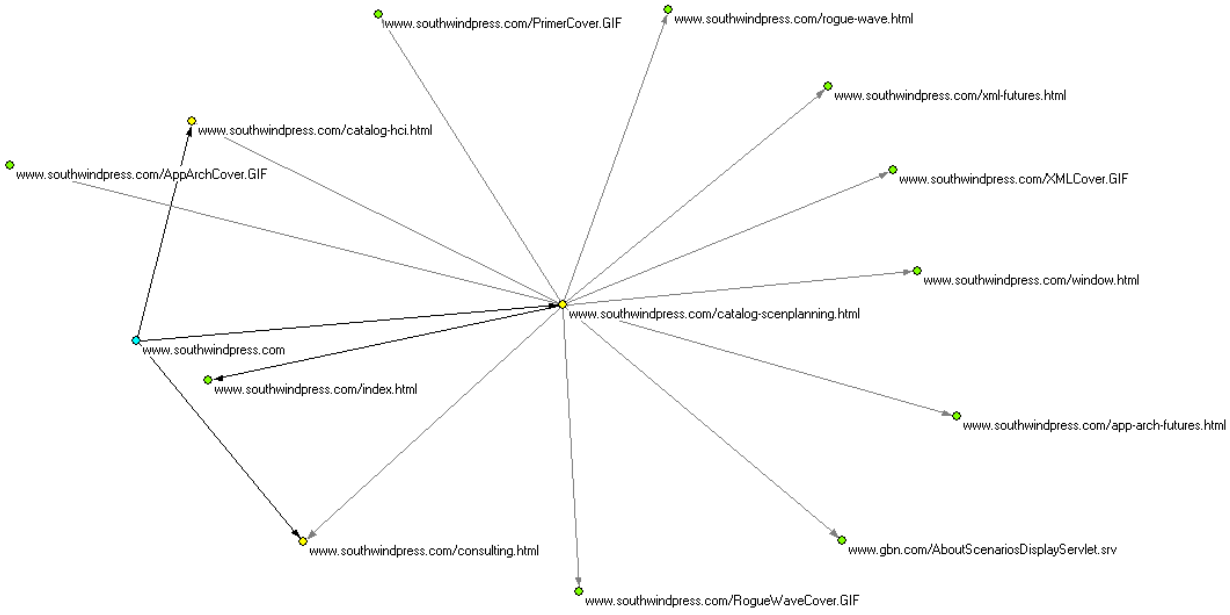


FIGURE 11 - IGNORING SOME URLS

You’ll notice that we still have the page from gbn.com, but the sitemeter and payloadz URLs are now gone.

This mechanism is never applied to the root node when building the network. If it were applied, you could inadvertently truncate the network building process before it even begins! This makes it possible

to eliminate internal references within the domain or host, allowing you to look at interactions between the domain or host and external entities.

8.2 A BIT OF ROCKET SCIENCE: HANDLING REDIRECTION

Sometimes outlinks are embedded in URLs in a page, and it is the embedded URL you wish to consider as the outlink rather than the URL you first encounter. The URL first encountered will normally be structured in the standard MIME type `application/x-www-form-urlencoded` format, and will contain the embedded URL as one of the name-value pair parameters. Because I encountered such a situation in one of my investigations, I added code to the library that handles this situation in a generalized way. It hasn't been tested extensively, but in the few situations where I've had to use it, the feature worked like magic.

SIDEBAR: AN IMPORTANT LIMITATION

The algorithm assumes that if you are looking for redirects at a given level, you are looking *only* for redirects at that level, which has been the case thus far in my limited experience. If it were necessary to consider all URLs but look for and use redirects where they apply, the library will need to be modified. Let me know if you encounter this situation and I will try to deal with it. If you choose to extend the library yourself so it handles this situation, please let me know that as well, and also donate the modified code if you are willing.

8.2.1 THE DATA STRUCTURE FOR REDIRECTS

The redirect data structure is a tuple of tuples, with the inner tuple having three objects inside it: a text fragment that we will use to recognize URLs we want to process; a name to look for in the URL's name-value pair list; and a level to which we will limit our use of this pattern.

```
redirect_list =
    (
        ( <text-string>, <parameter-name>, <level> ),
        ( <text-string>, <parameter-name>, <level> ),
        . . .
    )
```

For example, suppose we are looking for URLs this:

```
http://www.hitsphere.com/mc/mc.php?link=http://healthnex.typepad.com/web_log/
```

Also suppose we are looking for such URLs at level 0 (zero). Our redirect structure will look like this:

```
testRedirects = (('/mc/mc.php?', 'link',0),
                 )
```

In this case the URL parameter is usable exactly as shown; in other cases the parameterized URL will be URL-encoded, with percent signs followed by two hexadecimal digits substituted for certain characters. The parameterized URL might end up looking something like this:

```
http://www.hitsphere.com/mc/mc.php?link=http%3A%2F%2Fhealthnex.typepad.com%2Fweb_log%2F
```

If this is the case, don't worry—the library will correctly handle such situations by decoding the parameterized URL.

Here are the steps in the algorithm for handling redirects.

1. Because a redirects structure exists, we'll cycle through any tuples in the redirects list. For each such tuple:
    a. If we are at the level indicated by the third item in the tuple, in this case the top-level (level zero), take note of the fact that we have at least one tuple for this level, and then search the URL for the text-string that is the first item in the tuple.
        i. If this is present, we'll parse the current URL looking for a list of parameters, and see if the parameter given as the second element in the tuple is present.
            1. If so, we substitute the value associated with that parameter for the current URL as we continue processing.
            2. Else, try the next tuple.
        ii. Else, try the next tuple.
    b. Else, try the next tuple.
2. If we are not at the level indicated in any of the tuples, keep the current URL and continue processing.
3. Else, i.e. if we found at least one tuple for this level, ignore the current URL.

8.2.2 EXAMPLE

Let's look at Shahid Shah's HITSphere blog aggregator (http://www.hitsphere.com/). We'll look at his take on the healthcare information technology (IT) Blogosphere. We are interested in seeing if it is one giant conversation or a collection of smaller dialogues, so we'll be looking for weak components, which means we don't want a root node. The way we can do this is to use a phantom root.

Because we are using a phantom root, the URLs found in the phantom root page's anchorlist will be at level 0 (zero), which is where we'll want to do our checking for the redirect pattern. In this case we are looking for just one pattern, but we could look for additional patterns if we needed to do so. Here's the code from redirects1.py:

```
from urlnet.urltree import UrlTree

# use a standard list of ignorables provided by the library.
from urlnet.ignoreandtruncate import textToIgnore

# our tuple of tuples; the inner tuple provides the information we need
# to recognize and process a URL of the kind we seek.
testRedirects = (('/mc/mc.php?','link',1),
```

```
                    )

net = UrlTree(_maxLevel=2,_workingDir=workingDir,_redirects=testRedirects)

# we could have passed this as an argument to the constructor;
# this is another way to set ignorable text
net.SetIgnorableText(textToIgnore)

ret = net.BuildUrlForestWithPhantomRoot('http://www.hitsphere.com/')
if ret:
    net.WritePajekFile(' redirects1',' redirects1')
```

We are building a forest network based on the blogs featured in HITSphere.com, a directory of health information technology-focused blogs. The phantom root will give us a page with tons of URLs, but only the ones that match the redirect pattern matter to us. Each of the level zero URLs included in the network will be derived from the link parameter of a URL that matches the pattern found in our redirect data structure.

This results in a network like the following:

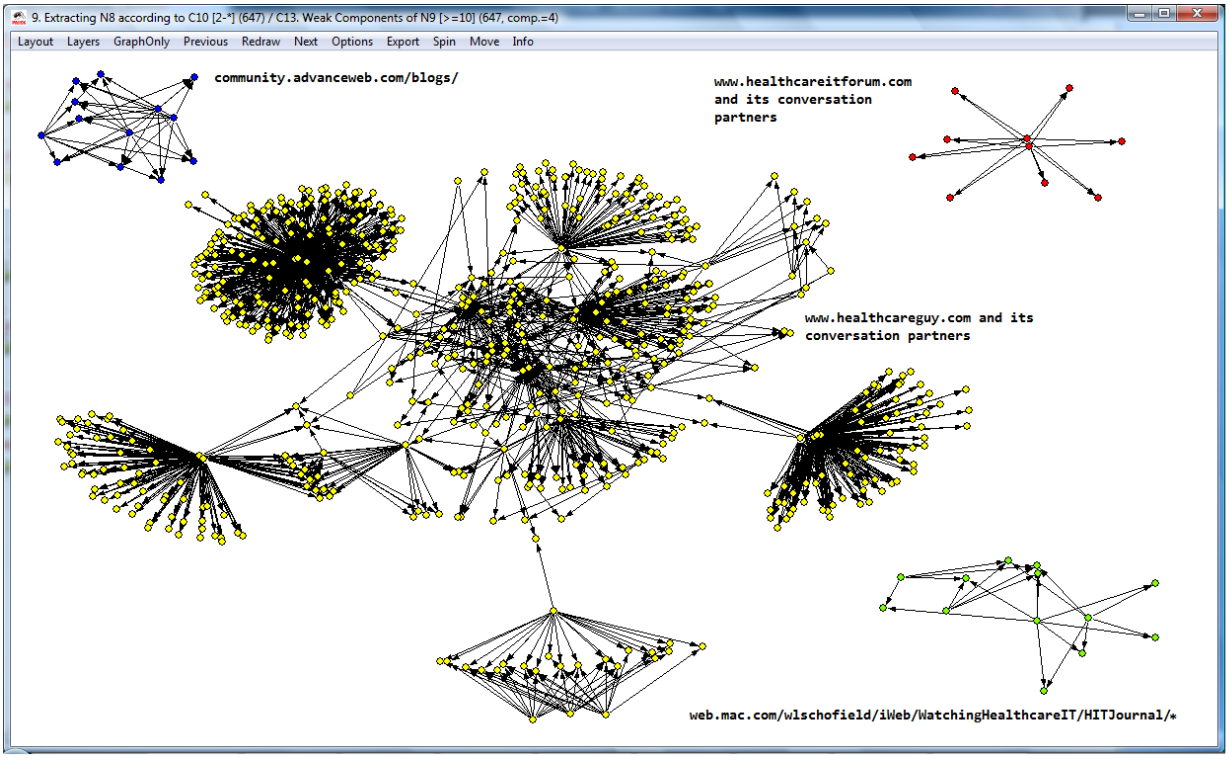


FIGURE 12 - A NETWORK GENERATED USING REDIRECTS

From this we can see there is a giant component, which centers on healthcareguy.com. There are four smaller components (size >= 10), each of which centers on a different healthcare IT blog. Shahid Shah, who runs HITSphere and is the founder and chief author of the healthcareguy.com blog, is clearly at the center of the maelstrom. His HITSphere aggregator has been instrumental in getting conversations going

in the healthcare IT Blogosphere. There are, in fact, other conversations going on in the realm of healthcare IT, but healthcareguy.com and HITSphere are clearly good starting points for getting into the conversation.

8.3 Combinations of Forests and Trees

In my thesis research, it proved useful to combine a forest—a network created from multiple root URLs—with a tree generated from a single root. For example, I needed to build networks that combined a forest derived from a list of historical URLs from a search engine data set with another forest built using the results of a real-time query against the same search engine with the same query, to analyze connections between the search engine data set and the query result set. I used placeholder root nodes with both the historical data set forest and the on-the-fly-generated search results forest, so I could drive networks that showed just the outlinks reachable from either of the starting-point URL sets.

It is also sometimes useful to compare two URL data sets to identify components within and between the data sets, for example when analyzing the similarities and differences between the results of two conceptually identical but differently worded queries (the so-called “Vocabulary Problem”).

8.3.1 A simple combinatory example

Here’s an example of the latter use, analysis of a Vocabulary Problem instance. It shows the creation of a network combining three forests generated from queries submitted to the AOL search engine. We’ll look at classes that handle search engine result sets when we get to chapter 10, but for now, we’ll use the tried and true method you’ve no doubt been employing for several hours by now. Here’s the code:

```
From urlnet.urltree import UrlTree
From urlnet.ignoreandtruncate import textToIgnore, textToTruncate

# combine forests from 'quit smoking', 'stop smoking', and 'smoking cessation'
# query result trees, based on queries submitted to AOL Search
net = UrlTree(_maxLevel=2,
               _resultLimit=10)
net.SetIgnorableText(textToIgnore)
net.SetTruncatableText(textToTruncate)
for query in ('quit smoking','stop smoking','smoking cessation'):
    root='http://www.aol.com/search?q='+re.sub(' ','+',query)
    net.BuildUrlTreeWithPlaceholderRoot(root,query)
```

8.4 Generating Pajek partitions

By default, a Pajek partition is created reflecting the level at which a URL is found, with the root node being level zero (0). In the case of forests, there are multiple root nodes. This partition will always be present.

There are two additional ways to create partitions using library facilities. Both use properties on the UrlNetItem-derived node class as the determinant of the partition number to be used for a given vertex. Both are methods on the UrlTree class, and are to be invoked after the initial Pajek project is written to the output stream.

The first method, the simplest, assumes the property set on each node has an integer value that is to be used as the partition number. The function is called WritePajekPartitionFromPropertyValueLookup, and the normal way to invoke it would be as follows (the UrlTree-derived class instance is assumed to be called 'net'):

```
fd = open(myPajekProjectFilename, 'w')
net.WritePajekStream(networkName,fd)
net. WritePajekPartitionFromPropertyValueLookup(
            fd,myPartitionName,thePropertyName)
fd.close()
```

In this method, it is wise to include a decoding of the partition integer values into something meaningful in the partition name you pass in. For example, if the property is gender and the values are 0=male and 1=female, you might want to pass the partition name as something like 'Gender (0=male, 1=female)'.

The second method is a bit more complicated, but not much. In addition to the arguments to the above-described method, a dictionary is passed, in which the property value is to be used as a key, and the partition number will be the value looked up in the dictionary using the property value as the key. This member function is called WritePajekPartitionFromPropertyDict:

```
fd = open(myPajekProjectFilename, 'w')
net.WritePajekStream(networkName,fd)
myDictionary = { 'male': 0, 'female' : 1, }
net. WritePajekPartitionFromPropertyDict(
            fd,myPartitionName,thePropertyName)
fd.close()
```

This method is useful when the property value is something other than an integer, e.g., a string or a float. In this case, you do not need to include the decoding of partition numbers to meaningful strings in the partition name; the function uses the dictionary key-value pairs to do this for you.

Both functions have two optional arguments. doDomains defaults to False; if you set it to True, the domain network is used instead of the URL network as the basis for the partition. The defaultPartitionNumber argument defaults to None in the signature, but None is converted to the default Pajek null partition value, 9999998 in the function body.

The example program partition1.py shows how to use these.

8.5 TRANSLATING NODE PROPERTIES INTO ADDITIONAL GUESS NETWORK ATTRIBUTES

8.5.1 URL NETWORKS

GUESS does not have a direct equivalent to the Pajek partition and vector concepts. Instead, the network builder can add additional attributes[2] to the node definitions (and to the edge definitions too, but we don't support that yet). These additional attributes are in many ways a more direct approach to what the Pajek partition accomplishes, the tradeoff being that GUESS attributes are more tightly coupled to the network data structure, and tight coupling is not always a desirable characteristic of a software engineering design.

UrlNet provides a reasonably straightforward way to translate node properties into GUESS network node attributes. The UrlTree class already incorporates the level property on every node into a GUESS network node attribute, but it does this in a hard-coded fashion, because the level attribute will always be present.

The SearchEngineTree class, which we will encounter in chapter 10, Generating trees and forests from search engine query result sets, uses our more generic facility for translating node properties into GUESS network node attributes. It sets up the necessary data structure to process a node property called 'pos_prob' (meaning 'positional probability', the likelihood of a search engine result set outlink being chosen by the information seeker based on its ordinal position in the result set). The relevant code looks like this, from the __init__() function of the SearchEngineTree class:

```
. . .
                self.SetProperty('additionalUrlAttrs',[('pos_prob','DOUBLE',),])
. . .
```

If present, the additionalUrlAttrs property value should be a list of two-item tuples containing name/type pairs. The name should be a legal Python variable name, and the type should be one of the strings 'VARCHAR', 'DATE', 'TIME', 'DATETIME', 'BOOLEAN', 'INT', 'FLOAT', 'DOUBLE', 'BIGINT', or 'TINYINT'. The name should be a property on each URlNetItem instance in the network. As shown in this example, since the network's additionalUrlAttrs property contains [('pos_prob','DOUBLE',),] each item should have a property named 'pos_prob' whose value is a double float. If the property is not present (or its value is deliberately set to the Python constant None), a default value will be used: integer zero for integer types, 0.0 for floating-point types, and the empty string '' for everything else.

The resulting GUESS network will look like this:

```
nodedef>name VARCHAR, url VARCHAR,domain VARCHAR, theLevel INT, pos_prob DOUBLE
smokefree_gov1,"Smokefree_gov","smokefree_gov",0,0.3861
smokefree_gov2,"Smokefree_gov","smokefree_gov",1,0.0551571428571
smokefree_gov3,"Smokefree_gov","smokefree_gov",1,0.0509142857143
```

[2] Note that the GUESS manual uses the term 'property' where I am using the term 'attribute'. I am using a different term here to avoid confusion between our concept of 'property' and the GUESS equivalent. If this creates more confusion than it alleviates, I apologize; let me know and I will correct the manual.

```
smokefree_gov4,"Smokefree_gov","smokefree_gov",1,0.0466714285714
. . .
```

This network was generated using the example program searchengine1.py.

8.5.2 DOMAIN NETWORKS

An equivalent mechanism exists for GUESS domain networks, though no example is provided at this time. The constructor code would differ as follows:

```
. . .

self.SetProperty('additional<b>Domain</b>Attrs',[('property_name','DOUBLE',),])
. . .
```

8.5.3 USE IN DERIVED CLASSES

The above example code that supposedly came from the SearchEngineTree class constructor was actually an oversimplified version for educational purposes. In derived classes, for example those derived from SearchEngineTree, care must be taken to ensure that the existing value of the additionalUrlAttrs and additionalDomainAttrs properties, if any, is not overwritten when this feature is used. The actual code from the SearchEngineTree shows how this would be done:

```
proplist = self.GetProperty('additionalUrlAttrs')
if proplist == None:
    proplist = []
proplist.append(\
    ('pos_prob','DOUBLE',)\
    )
self.SetProperty('additionalUrlAttrs',proplist)
```

The existing list, if any is obtained; if none exists, an empty list is provided. The tuple to be added is then appended to the list,a nd the property value reset.

8.5.4 OVERRIDING DEFAULT VALUES IN GUESS PRE-DEFINED COLUMNS

The above-described mechanism can be used to override the default values for any of the GUESS pre-defined attributes, described in the current GUESS manual in the section entitled “The Guess .gdf Format”. For example, by populating a property called ‘style’ with an appropriate value on each node, then addition the tuple (‘style’,’INT’) to the additionalUrlAttrs property, the node’s style can be specified, overriding the GUESS default rectangle node style. The GUESS manual currently specifies the following values for the style attribute:

> Currently GUESS maps: rectangle = 1, ellipse = 2, rounded rectangle = 3, text inside a rectangle = 4, text inside an ellipse = 5, text inside a rounded rectangle = 6.

The other

9 USING PYTHON REGULAR EXPRESSIONS TO FIND URL ANCHORS

9.1 OVERVIEW

Python's urllib and urllib2 modules provide default ways of finding URL anchors (outlinks) in an HTML page. Sometimes the default isn't so useful, because it finds all of the URLs in the page, and you may be interested in a specific subset, e.g., only those where the <a> element has a class attribute of a certain value. UrlNet provides facilities to handle this situation.

9.2 THE REGEXQUERYURL CLASS: PARSING ANCHORS FROM RETRIEVED DOCUMENTS

RegexQueryUrl is a descendant of the Url class, designed to parse retrieved documents using Python regular expressions. Four properties control the behavior of the RegexQueryUrl class.

1. The regular expression(s) to be used in scanning the retrieved document for URL anchors is passed in the regexPattern property. This is the only required property. The value of this property can be either a list of regular expression strings, or a single regular expression string.
2. The optional findall_args property, if present, is used as the flags argument in the call to re.findall().
3. The optional nextUrlClass property, if present, provides the class to use for instantiation of Url-derived instances to follow in the network generation process, and if not present, the base Url class itself will be used.
4. The optional SEQueryFileName property, if present, is used in two ways, both helpful for debugging. First, the page retrieved by the Python urllib library calls is written to a local file with a name composed of the SEQueryFileName property value plus '_page.html'. This gives you the raw material from which the anchorlist will be derived using your regular expression(s). Second, SEQueryFileName is used as the name of a file that will contain the anchors derived from the retrieved page using your regular expression(s).

Your curiosity is probably piqued, and three questions are on your mind. First, *Why are regex-related parameters passed as properties rather than constructor arguments?* Second, *What's all this about nextUrlClass?* And third, *Why allow multiple regular expressions?*

9.2.1 PROPERTIES VERSUS ARGUMENTS

Why are regex-related parameters passed as properties rather than constructor arguments? It is optimized for use with search engines. When building trees and forests from search engine query result sets, the top-level search engine URL is parsed to get the result set URLs using this class, and the result URLs are treated as garden-variety URLs by default.

9.2.2 CHANGING URL-DERIVED CLASSES IN MIDSTREAM

What's this about nextUrlClass? Regular expressions will usually be very specific to a particular page or type of page. When following outlink networks, the pages pointed to by the outlinks are likely to be formatted very differently than the initial page; most likely you will want to rely on the generic

mechanism for finding outlinks, which is built into Python's standard libraries, and is the method used by the base Url class. By *not* setting the nextUrlClass property, the class used for all URLs after the initial URL will be the base Url class, and this will work fine for most purposes.

If the network is based on a Web Service API and the outlinks' pages will be formatted identically to the initial page, setting the nextUrlClass property to RegexQueryUrl will create a network in which all URLs are represented by instances of RegexQueryUrl rather than Url.

9.2.3 ONE REGULAR EXPRESSION OR A LIST?

If a single regular expression is used, it is expected to yield a list of URLs, which will constitute the anchorlist for the URL encapsulated by the RegexQueryUrl instance. If a list of regular expressions is used, only the *last* regular expression in the list is expected to yield the list of URLs that will act as the anchorlist. Here's why you may want this behavior.

It is often difficult to target a single regular expression to the target text item(s) you are expecting it to match. The approach usually taken in such instances is to use a regular expression to subset the text, creating a list of one or more text strings that can be searched successfully for the desired text item(s). In practice, I have found it necessary at times to use the subsetting approach iteratively before the final regular expression can match the correct set of text items (in our case URLs). This leads to the following algorithm:

1. Put the text for the page to be searched for anchors in a list called items, containing just the page.
2. Declare the variable i = 0.
3. For i in the range 0 to the length of the list of regular expressions:
    a. Declare a new list called results.
    b. For each item in items:
        i. Add to the results list the result of a re.findall operation, with the the i'th item in the list of regular expressions as the regex pattern, the item in hand as the text to search, and the value of the findall_args property (if present) as the flags argument.
        ii. Next item.
    c. Replace the items list with the results list.
    d. Next i.
4. The items list contents is the page's anchorlist.

9.2.4 AN EXAMPLE

This class is used in the search engine result set processors discussed in chapter 10, but it has wide applicability. Here's a somewhat-simple example, regexqueryurl1.py, which builds a tree from only the recommended external outlinks from the MedlinePlus melanoma portal page. The code is found in regexqueryurl1.py.

```
from urlnet.urltree import UrlTree
from urlnet.regexqueryurl import RegexQueryUrl
```

```
import re

net=UrlTree(_maxLevel=2,\
            _urlclass=RegexQueryUrl)
regexPats = [
    '<ul id="subcatlist">.*</ul>',
    '<span class="categoryname"><a name=".*?</ul>',
    '<a href="([^/#].*?)"',
    ]

net.SetProperty('regexPattern',regexPats)
net.SetProperty('findall_args',re.S)
net.SetProperty('SEQueryFileName','regexqueryurl1_out')
success = net.BuildUrlTree('http://www.nlm.nih.gov/medlineplus/melanoma.html')
if success:
    net.WritePajekFile('regexqueryurl1','regexqueryurl1')
```

This example uses a list of three regular expressions to narrow down the search for the correct set of URL anchors, which I have defined as all the URLs inside the unordered list (ul) element whose id attribute value is 'subcatlist', excluding those that are internal references within this page. Once this list of URLs is obtained, RegexQueryUrl sets the URLTree instance's urlclass member to a reference to the Url class, which is used to encapsulate each URL in the anchorlist, and all other outlink anchors processed during program execution. It leaves the root page in regexqueryurl1_out.html, and the list of URLs derived using the regular expressions in regexqueryurl1_out.txt.

Below is the Pajek domain network from an execution of regexqueryurl1.py.

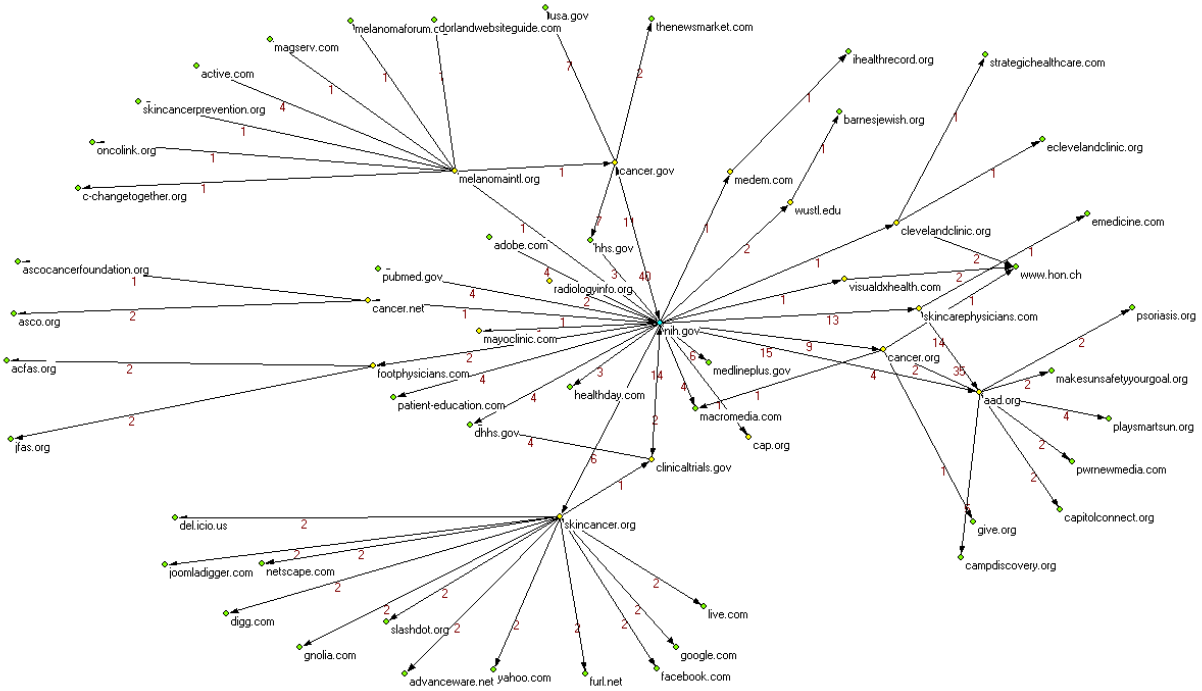


FIGURE 13 - REGEXQUERYURL IN ACTION: A SEARCH ENGINE RESULT SET DOMAIN NETWORK

9.3 WORKING WITH REGULAR EXPRESSIONS

Regular expressions are admittedly cryptic, but they are arguably an essential tool in any domain where the searching and manipulation of electronic text is important. Fortunately there are some tools you can use to bring yourself up to speed.

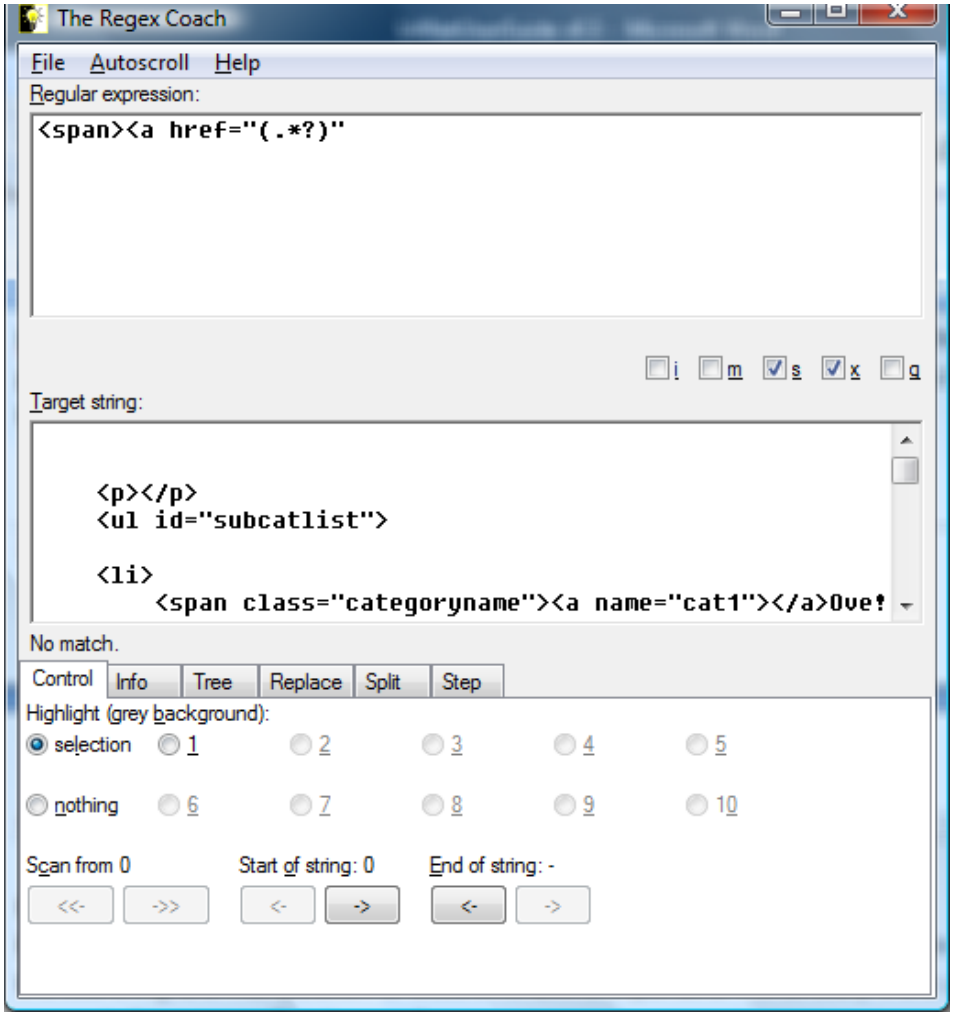


FIGURE 14 - THE REGEX COACH

I use a program called the Regex Coach to figure out the regular expressions I use. You can find it at http://weitz.de/regex-coach/, a page that contains links to download the program and reasonably extensive program documentation.

There are a number of tutorials on regular expressions on the Web. The simplest way to find them is to use the query `regular expressions tutorials` in the search engine of your choice. I have looked at several, and have found none that is significantly better than any other from my perspective. You should look at two or three (or more) and see which one works best for you.

Python's own documentation on regular expressions and the re module can be found starting at http://docs.python.org/lib/module-re.html.

10 GENERATING TREES AND FORESTS FROM SEARCH ENGINE QUERY RESULT SETS

10.1 OVERVIEW

In our examples so far, we start in one of two ways. First, we can begin with one root URL and create a tree structure by following outlinks. The tree often turns into a network as outlinks lead to pages already encountered, but this is not necessarily the case. In the second method, we start with a list of URLs and create a tree from each—a structure we have called a *forest*. This structure may contain individual trees for each URL in the list, or links within and across the trees in the forest create larger interconnected components within the network. In both cases, we treated all URLs in the network using the same algorithm: gather all the outlinks and follow them until we reach the maximum depth.

With search engine query result sets, if we are investigating the efficacy of the search engine algorithm, we normally want to restrict our tree to a specific set of outlinks, the so-called organic search results. Virtually all search engines default to ten such outlinks per page; some provide the means to ask for larger result set chunks, and others restrict you to ten results unconditionally.

There are often many other outlinks on the result set page, for sponsored results, suggestions for alternative queries, search engine portal pages such as terms of use and privacy policies, and so on. You'll need to decide whether these are part of the analysis; if not, you'll do as I did, and look at the organic search results.

The forest generated from the top 10 search results represents, in most cases, the pages that have at least some degree of findability from the result set. A tree of depth 2 (you'll recall that means a three-layer deep forest) contains the pages that are within two clicks of the original results.

The library offers specialized features for search engine results, features that make it easy to apply the library's affordances to new search engines. You've already encountered the RegexQueryUrl class in chapter 9; you'll see in the various search engine handler modules that they use the RegexQueryUrl class in just the same way it's used in that chapter.

10.1.1 ESTIMATING CLICK-THROUGH PROBABILITIES

One dimension of interest in examining information seeking behavior using search engines is the likelihood that any given result item is going to be selected. The Windows Live Labs data from MSN Search showed us that there is a relatively consistent probability distribution of click-throughs by position in result sets. For instance, here' s the distribution for almost all 12 million click-throughs by position:

```
probabilityByPositionAllClicks = (
    0.4960,
    0.1363,
    0.0924,
    0.0629,
    0.0499,
    0.0395,
    0.0343,
```

```
    0.0295,
    0.0279,
    0.0275,
    )
```

I say “almost all” because the above numbers add up to only 99.62%. Very similar distributions appeared for subjects as diverse as cancer, automobiles, politics, recipes, and pop stars.

Facilities are provided to calculate probabilities by position for use, for example, in a Pajek vector. The difficult and chancy aspect is how to divide up the probabilities as you go deeper into the network. If the first item in the result set gets 50% of the clicks, and its page content has 50 outlinks, do you assume each outlink has an equal share of the parent page’s probability? If so, each outlink has a 1% chance of being followed. Or do you assume that outlinks are not equal, and that, for example, outlinks higher up in the page are more likely to be clicked than those lower on the page?

We provide probability calculators for both approaches, but we advocate neither over the other. Neither is empirically defensible at this point, though the latter approach fits the existing evidence better than the equal-distribution approach. Evidence from eye-tracking studies has shown that the user’s attention is variable dependent on the page content type, and the actual positions of outlinks on the page are difficult to detect from the raw HTML. In each study we have seen, though, it is clear that more attention is paid to the upper part of the portion of the page that is visible on the user’s monitor than to parts of the page that are lower down. For this reason, our example code favors the use of the probability generator we provide that distributes probabilities for lower-level click-throughs using a straight-line trend where the allocated share of the parent page’s probability is inversely proportional to the link’s position in the order of appearance in raw page text.

A network node’s positional probability is stored in its ‘pos_prob’ property. For nodes that are encounterd more than once in network generation, this property stores the highest probability value. If this is a top-level node, its probability comes from the initial probability vector; otherwise, the node’s positional probability is calculated by the probability vector calculator in use, and represents the node’s share of the node’s parent’s probability.

10.1.2 The SearchEngineTree Class

The SearchEngineTree class is responsible for setting up the positional probabilities functionality, if it is being used. Two functions specific to search engine trees, GetAnchorList() and FormatQueryURL(), are provided, which are responsible for formatting the search engine query and retrieving the list of result set URLs from the search engine. Three of the network-generating functions are overridden to reflect the use of GetAnchorList() and FormatQueryURL() in retrieving the URLs in the search engine query result set.

The module also contains the two probability distribution calculator functions we have provided. The example (searchengine1.py) shows how these are used.

10.1.3 SETTING PROPERTIES IN THE CONSTRUCTOR

To handle a search engine, we derive a class from SearchEngineTree. There are a number of such classes in the UrlNet library, including but not limited to GoogleTree, AOLTree, and YahooTree. We need to set some properties in the constructor. numSearchEngineResults tells how many results we want to process; most search engines offer some flexibility here, and you'll need to do your research to find out what options are supported. This property is used in FormatQueryURL() only, unless the search engine does not support more than ten results at a time (Healia, for example), in which case GetAnchorList() will also use the property to determine how many calls to the search engine are needed to fetch the desired number of results.

Here's part of the constructor code in the GoogleTree class, which is derived from SearchEngineTree:

```
            # set necessary properties for use in Url-derived class
            # number of results to fetch - must be 10,20,30,50, or 100
            # default is 10 results.
            self.SetProperty('numSearchEngineResults',_resultLimit)

            # Url-derived class to use for all but the Google query Url.
            # If the property is not set, the default Url class will be used.
            self.SetProperty('nextUrlClass',Url)

            # Google-specific regex pattern
            self.SetProperty('regexPattern','<h2 class=r><a href=\"(.*?)\"')
```

10.1.4 OVERRIDING URLTREE.FORMATQUERYURL()

Google overrides UrlTree.FormatQueryURL(), but not SearchEngineTree.GetAnchorList(); the parent class's version of GetAnchorList() works just fine for Google.

Here's the code for GoogleTree.FormatQueryURL():

```
    def FormatQueryURL(self,freeTextQuery):
        """
        This function is Google-specific.
        """
        log = Log('GoogleTree.FormatQueryURL',freeTextQuery)
        numResults = self.GetProperty('numSearchEngineResults')
        if (not numResults):
            numResults = 10
        if str(numResults) not in ('10','20','30','40','50','100'):
            raise Exception, \
                  'Exception in GoogleTree.FormatQueryURL: ' \
                  + "numSearchEngineResults property must be one of " +
                      "'10','20','30','40','50', or '100'"
        prefix = 'http://www.google.com/search?hl=en&'
        query=urlencode({'num' : numResults, 'q': freeTextQuery, })
        query = prefix + query + '&btnG=Search'

        # create a name we can use for writing a file with the result set URLs later
        name = ''
        for c in freeTextQuery.lower():
            if c in 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789-_':
                name = name + c
```

```
            else:
                name = name + '_'
        if len(name) == 0:
            name = 'googletree_query'
        timestamp = self.GetProperty('timestamp')
        if not timestamp:
            timestamp = ''
        self.SetProperty('SEQueryFileName', timestamp + '-' + name + '.txt')

        return query
```

It uses urllib.urlencode() to create the parameter list, around which it sandwiches a prefix and suffix. urllib.urlencode() takes a dictionary as its argument, and in this case we are encoding the query and the number of results desired.

GoogleTree.FormatQueryURL() also creates a filename from the query by substituting underscores for any character that is not alphanumeric, dash, or underscore. It sets this as the value of the SEQueryFileName property, which as you'll recall is employed by the RegexQueryUrl class to write the retrieved page and the list of derived URL outlinks to local disk.

10.1.5 Overriding SearchEngineTree.GetAnchorList()

SearchEngineTree.GetAnchorList() accepts a query token (which may or may not be a URL), calls self.FormatQueryUrl() to transform it into a URL destined for processing; uses this URL to construct an instance of the Url-derived class indicated in the UrlTree.urlclass attribute if one was not passed in; calls the GetAnchorList() member function on that Url-derived instance to get the list of child tokens (which may or may not be URLs); and finally, if so directed by the putRoot parameter, calls the UrlTree.PutRootUrl() function to establish the constructed Url-derived instance as the root of the tree. It returns a sequence containing the transformed URL, the Url-derived instance, and the list of child tokens.

SIDEBAR: Refactoring Alert

The original use of this function was to retrieve search engine result sets, but it has proven valuable in other contexts, because it allows network nodes to be defined by a token other than a URL. For example, in the analysis of NCBI author cosmos networks, nodes can represent authors, publications, and MeSH topics, as described in chapter 12, Working with Web Service APIs, Part Two: The NCBI APIs. In fact, this capability is useful in most social networking API contexts, including but not limited to del.icio.us, Technorati, and Facebook. The next refactoring of the library will see the introduction of a new UrlTree-derived class that encapsulates API-related functionality independent of the search engine context.

Here's some example code from HealiaTree, which shows you one practical reason why you might want to override SearchEngineTree.GetAnchorList() in the analysis of data constructed from search engine

result sets. Unlike all the other search engines I've worked with, Healia has no parameter option to support getting more than ten results at a time. SearchEngineTree.GetAnchorList() and UrlTree.FormatQueryURL() cooperate here to retrieve an anchor list containing up to 100 result URLs (always multiples of ten). Similar logic would be needed in other contexts where an arbitrarily large number of anchors is possible, but the API imposes a limit on the number of results returned at one time.

```
def GetAnchorList(self,query,putRoot=False):
    """
    TODO: This needs to be modified to iterate calls to Healia when the desired
    number of results is > 10.
    """
    try:
        numResults = self.GetProperty('numSearchEngineResults')
        if (not numResults):
            numResults = 10
        if str(numResults) not in ('10','20','30','40','50','60','70','80','90','100'):
            raise Exception, \
              "Exception in HealiaTree.GetAnchorList: " \
              + "numSearchEngineResults property must be one of "\
              + "'10','20','30','40','50','60','70','80','90', or '100'"
        numResults = int(numResults)
        Urls = []

        """
        Because the Url class GetAnchorList() function will set this
        instance's urlclass member to the Url-derived class for
        the lower-level nodes, we must save the RegexQueryUrl-derived
        class that was passed to us in the constructor, and use it
        for what may be multiple calls to GetAnchorList().
        """

        healiaURLClass = self.urlclass
        for num in range(10,numResults+10,10):
            queryURL = self.FormatQueryURL(query,args=num)
            url = healiaURLClass(_inboundUrl=queryURL,_network=self)
            Urls = Urls + url.GetAnchorList()
        if putRoot:
            self.PutRootUrl(queryURL)
        self.topLevelUrls = Urls
        return (queryURL,url,Urls)
    except Exception, e:
        raise Exception, 'in HealiaTree.GetAnchorList: ' + str(e)


    def FormatQueryURL(self,freeTextQuery,args=None):
    """
    This function is Healia-specific.
    """
    log = Log('HealiaTree.FormatQueryURL',freeTextQuery)
    if args:
        numResults = int(args)
    else:
        numResults = self.GetProperty('numSearchEngineResults')
    if (not numResults):
        numResults = 10
    if str(numResults) not in ('10','20','30','40','50'):
```

```
        raise Exception, \
              'Exception in HealiaTree.FormatQueryURL: ' \
              + "numSearchEngineResults property must be one of " + \
                  "'10','20','30','40', or '50'"
    """
    Healia supports filtering results to a basic reading level,
    The better to support consumer health search
    """
    if self.GetProperty('filterToBasicReadingLevel'):
        prefix = 'http://www.healia.com/healia/search.jsp?'
    else:
        prefix = 'http://www.healia.com/healia/search.do?'
    query=urlencode({'start' : numResults-10, 'query': freeTextQuery, })
    if self.GetProperty('filterToBasicReadingLevel'):
        query = query + '&hauto=easy'
    query = prefix + query

    # create a name we can use for writing a file with the result set URLs later,
    # and set the 'SEQueryFileName' property with the generated filename
    self.SetFilenameFromQuery(freeTextQuery)

    return query
```

10.2 SOME SPECIFIC SEARCH ENGINE NETWORK GENERATORS

Examples are provided for six search engines, including the four most popular general-purpose search engines and two healthcare-specific “boutique” search engines. All use the RegexQueryUrl for the root URL instance, and switch to the Url class for all other nodes. In each case, the sample code shows the use of a probability distribution generator. Some have additional interesting features. For example, Healia never returns more than 10 results, so if the number of results specified is greater than 10, iterative HTTP GETs are needed to get successive pages and the list of outlinks is built up incrementally.

The search engines for which classes are provided include:

Search Engine	UrlNet class	Module	Notes
Google	GoogleTree	googletree	A relatively simple and straightforward example. This was done first, and is the code template on which the other search engine tree classes are based.
America Online	AOLTree	aoltree	The FormatQueryURL() function is complicated by the fact that AOL only allows you to specify a result set size other than 10 in the Advanced Search. As a result, the parameter list is longer and more complicated than any of the others.
Healia	HealiaTree	healiatree	As you’ve already seen, this “Boutique” healthcare search engine is distinguished by a limitation to returning only ten results at a time.

Search Engine	UrlNet class	Module	Notes
Healthline	HealthLineTree	healthlinetree	Boutique" healthcare search engine #2. Healthline had a weird set of numbers for the number of results to return, so I normalized it by making multiple calls to get successive pages, using the code from HealiaTree as a base.
Windows Live	WindowsLiveTree	windowslivetree	Windows Live also had a weird set of numbers for the number of results to return, so I normalized it as with HealiaTree and HealthLineTree. This begs for refactoring to become the normal way to handle result set sizes other than 10.
Yahoo!	YahooTree	yahootree	Another relatively simple and straightforward example.

10.3 SEARCH ENGINE EXAMPLE

The first search engine example program is searchengine1.py. This is a fairly complicated program that does things more like a production link harvesting program. It captures logging output to a separate file; uses a probability vector generator and some seed data calculated from the MSN Search data set; uses the GoogleTree class to create a network from Google search results; and finally, includes some under-the-hood code you can uncomment to see how the class's primary functions operate.

10.4 A COMPLEX EXAMPLE: COMBINING A SEARCH ENGINE QUERY RESULT SET NETWORK WITH A DESIRED TARGET NETWORK

If you looked at the Table of Contents, you may be aware the UrlNet library contains classes that are specific to some popular search engines, including the America Online search engine. The next example, searchengine2.py, employs the AOLTree class in a variation on the first example. This program closely emulates the data collection algorithm I employed in my Master's thesis work at the University of Michigan School of Information.[3]

The example program first builds a network based on the recommendation links in the National Library of Medicine's MedlinePlus smoking cessation portal. The links in this first network represent the "gold standard" in information about smoking cessation, as assessed by experts in the field. The program then generates and merges a network derived from the result set of a query to the AOL search engine. This involves a bit of tricky coding, because we generate the MedlinePlus network first.

[3] I didn't develop the UrlNet library in time to use this code for my thesis data collection effort; however, it was during that period that I developed the core algorithms, and some of the code I wrote back then has ended up in the class library in greatly augmented form.

The merged network shows us how well-connected the search engine result set with the recommended links. This gives us a concrete and visualizable metric for the findability of the "gold standard" information is through the AOL search engine based on a the most common smoking cessation query.

```
from urlnet.log import Log, logging, altfd
from urlnet.aoltree import AOLTree
from urlnet.urlutils import PrintHierarchy
import sys
from time import strftime, localtime
import os
from urlnet.searchenginetree import computeDescendingStraightLineProbabilityVector,\
                                    computeEqualProbabilityVector
from urlnet.ignoreandtruncate import textToIgnore, textToTruncate
from urlnet.clickprobabilities import probabilityByPositionStopSmokingClicks \
                                      as probability_by_position

from urlnet.regexqueryurl import RegexQueryUrl
import re
medlineplusRegexPats = [
    '<ul id="subcatlist">.*</ul>',
    '<span class="categoryname"><a name=".*?</ul>',
    '<a href="([^/#].*?)"',
    ]

def main():
    import urlnet.log
    from urlnet.urlutils import GetConfigValue
    from os.path import join
    """
    We are going to make a subdirectory under
    the working directory that will be different each run.
    """

    baseDir = GetConfigValue('workingDir')
    # dir to write to
    timestamp = strftime('%Y-%m-%d--%H-%M-%S',localtime())
    workingDir = join(baseDir,'stopsmoking',timestamp)

    oldDir = os.getcwd()

    # uncomment one of the vectorGenerator assignments below

    # vectorGenerator = computeEqualProbabilityVector
    vectorGenerator = computeDescendingStraightLineProbabilityVector
    myLog = Log('main')
    urlnet.log.logging=True
    #urlnet.log.trace=True
    urlnet.log.altfd=open('aoltree.log','w')
    try:
        try:
            os.mkdir(workingDir)
        except Exception, e:
            pass #TODO: make sure it's because the dir already exists
        os.chdir(workingDir)
        urlnet.log.logging=True
        #urlnet.log.trace=True
        urlnet.log.altfd=open('aoltree.log','w')
    except Exception,e:
        myLog.Write(str(e)+'\n')
        return
```

```
    try:
        # build initial network based on query 'quit smoking'

        net = AOLTree(_maxLevel=2,
                      _workingDir=workingDir,
                      _resultLimit=10,
                      _probabilityVector = probability_by_position,
                      _probabilityVectorGenerator = vectorGenerator)
        #
        # tell the algorithm to ignore the MedlinePlus smoking cessation portal
        # for now; we'll process it in the second network.

        textToIgnoreThisTime = textToIgnore
        textToIgnoreThisTime.append('medlineplus/smokingcess')
        net.SetIgnorableText(textToIgnoreThisTime)
        #
        # Don't pursue links in Amazon, YouTube, etc.
        net.SetTruncatableText(textToTruncate)
        net.BuildUrlForestWithPhantomRoot('quit smoking')

        # We now have a network consisting of the first- and second-level
        # links generated from the result set of the query to the AOL engine.
        # Now we want to merge in a network generated from the Medline Plus
        # smoking cessation portal's recommendations. We need to reset some class
        # properties in order to use the RegexQueryUrl class; the MedlinePlus
        # portal page has a lot of generic links we don't want to include in
        # the network.
        #
        # First, make sure we haven't already seen it.

        mlp_smokingcessation_portal = \
            'http://www.nlm.nih.gov/medlineplus/smokingcessation.html'
        item = net.GetUrlNetItemByUrl(mlp_smokingcessation_portal)
        if net.UrlExists(mlp_smokingcessation_portal):
            net.ForceNodeToLevel(level=0,url=mlp_smokingcessation_portal)
        else:
            # First we change the filename that will be used to output top-level URLs

            net.SetFilenameFromQuery('medlineplus_smokingcessation')

            # Next, we set the regular expressions to be used in parsing the
            # MedlinePlus smoking cessation portal page. This is a list of three
            # regular expressions that are used in sequence. It's declared above.

            net.SetProperty('regexPattern',medlineplusRegexPats)

            # Make sure the regex parser treats newlines as just more whitespace.

            net.SetProperty('findall_args',re.S)

            # Finally, reset the Url-derived class to use for the root URL.

            net.urlclass = RegexQueryUrl

            # The AOLTree.BuildUrlTree function will add to the network already present
            # in the AOLTree instance.
            net.BuildUrlTree( \
                mlp_smokingcessation_portal)

        net.WritePajekFile('aoltree-quitsmoking','aoltree-quitsmoking')
        net.WriteGuessFile('aoltree-quitsmoking_urls')
```

```
        net.WriteGuessFile('aoltree-quitsmoking_domains',False)

    except Exception,e:
        myLog.Write(str(e)+'\n on quit smoking query\n')

if __name__ == '__main__':
    main()
    sys.exit(0)
```

11 Working with Web Service APIs, Part one: Technorati

11.1 Introduction

Because the World Wide Web is a venue for applications in addition to Web sites, the Hyper Text Transfer Protocol (HTTP) is used as a transport layer for application programming interfaces (APIs). The UrlNet library allows you to easily extend its capabilities to capture data from networks that can be generated using such APIs. Web Service[4] APIs are provided by many social networking sites, such as Technorati and Delicious. In addition, public resources like the National Library of Medicine have made available APIs you can use to access many of their online databases programmatically.

11.2 An XML API example: Technorati's Cosmos API

The Technorati Cosmos API is used in this manual as an example of how to extend the library to adapt to such an API. The network diagram shown on the cover is an example of such a network. As you will see, two new classes are all that is required to extend the library in this manner. It depends on the 4Suite XML Library (http://4suite.org/?xslt=downloads.xslt).

11.2.1 First, a word about your manners

When invoking the APIs of others, it's good to be polite; this implies we should not be hammering them with HTTP requests. The UrlTree constructor has an argument, _sleeptime, that defaults to zero, and is the number of seconds the RetrieveUrlContent() function will wait between requests. We'll be using that here, when we get to the example code. In cases where the API we access specifies a minimum wait time, that constraint can be enforced by the UrlTree subclass, as you can see in the next example.

11.2.2 Subclassing the UrlTree class

The first thing we need to do is to subclass UrlTree, using code like this from technoratitree.py:

```
. . .
class TechnoratiTree(UrlTree):
    """
    Class representing a tree of Technorati URIs
    """

    def __init__(self,
                 _technoratiKey,
                 _technoratiApi = TECHNORATI_COSMOS_API,
                 _maxLevel = 2,
                 _singleDomain=False,
                 _showLinksToOtherDomains=False,
                 _workingDir=None,
                 _redirects = None,
                 _ignorableText = None,
                 _default_socket_timeout = 15,
```

[4] I use the term "Web Service" loosely here, referring to any structured API that is accessed via a URI, as distinct to restricting the meaning of the term to services that conform to the World Wide Web Consortium (W3C) Web Service activity specifications. You can find more information on this activity at http://www.w3.org/2002/ws/.

```
                _sleeptime = 2,
                _userAgent=None):
        try:
            log = Log('TechnoratiTree ctor')
            #
            # Most of the work is done by the parent class. All we do
            # here is specify what class to use for constructing Url
            # instances. We need to use a descendant of the Url class
            # for this purpose.
            #
            if _technoratiApi == TECHNORATI_COSMOS_API:
                UrlTree.__init__(self,
                                 _maxLevel,
                                 TechnoratiCosmosUrl,
                                 _singleDomain=False,
                                 _showLinksToOtherDomains=False,
                                 _workingDir=_workingDir,
                                 _redirects = _redirects,
                                 _ignorableText = _ignorableText,
                                 _regexExtractorPatterns = _regexExtractorPatterns,
                                 _default_socket_timeout = _default_socket_timeout,
                                 _sleeptime=_sleeptime,
                                 _userAgent=_userAgent)
            . . .

            else:
                raise Exception, 'no Technorati API specified'

            # We need to make our technorati developer key available...
            self.SetProperty('technoratiKey',_technoratiKey)

        except Exception, e:
            self.SetLastError('in __init__: ' + str(e))
```

As you can see, really the only things we do in the constructor, in addition to instantiating the superclass, are to 1) override the UrlTree constructor's default URL class argument so the TechnoratiCosmosUrl will be used instead of the Url class; and 2) set a property so instances of the TechnoratiCosmosUrl class can find our Technorati developer key for use in constructing the API call URL.

We have set up the class so it can eventually handle additional Technorati APIs, but at this point we only support the Cosmos API. The constant TECHNORATI_COSMOS_API is defined in the same file, above this code fragment.

11.2.3 Subclassing the Url Class

Next we need to create the TechnoratiCosmosUrl class, which will be a subclass of Url. All its constructor will do is to call the superclass's __init__() function, and set a few member variables. The interesting code is found in our override of the Url class's GetAnchorList() function (see technoraticosmosurl.py for all the code; what follows are just fragments), and in a new function called GetTapiTree().

GetTapiTree constructs a URL, then calls the superclass's GetPage function to do the actual work of invoking the API.

```
# first get the data
prefix = 'http://api.technorati.com/cosmos?'
args = {'key' : self.key, 'url' : self.root, 'limit' : str(self.limit),
        'start' : str(self.start), 'type' : 'link', 'format' : 'xml',}
suffix = urlencode(args)
self.url = prefix + suffix
page = self.GetPage()
```

We use a SAX XML parser, whose event processor class is nested inside the TechnoratiCosmosUrl class's GetTapiTree() function.

```
# next parse using SAX
try:
    import xml
    self.ResetLastError()
    factory = InputSource.DefaultFactory
    isrc = factory.fromString(page)
    parser = Sax.CreateParser()
    handler = tapi_processor()
    parser.setFeature(xml.sax.handler.feature_validation, False)
    parser.setContentHandler(handler)
    parser.parse(isrc)
    return handler.tapiTree
except Exception, e:
    self.SetLastError( e )
    return None
```

If successful, GetTapiTree() returns a dictionary with one entry (in this case), whose key is the URL we were given in the constructor (which is not the same as the URL passed to the Technorati API; that URL did include the original URL, but also the Technorati API key and other parameters). A successful lookup in the GetTapiTree() call's dictionary gives us a list of "anchors", which in this case are NOT the links embedded in the page returned by the URL we were given at construction time; instead, they are the URLs of blog entries that reference our target blog URL. By finding blog entries that reference these URLs, we construct a network that is a tree with our passed URL as the root. All we need to do to make this recursive net-building happen is to provide the correct anchorlist for this URL, the anchorlist we retrieved using the Technorati Cosmos API, when GetAnchorList is called on this TechnoratiUrl instance.

```
def GetAnchorList(self):
    """
    Overriding the same function in the Url class.
    """
    log = Log('GetAnchorList')
    #print "00"
    if self.url == None:
        #print "01"
        return []
    elif self.anchors != None:
        #print "02"
        return self.anchors
```

```
    else:
        #print "03"
        try:
            # parse to get the href
            #
            tree = self.GetTapiTree()
            lookupUrl = self.methodPrefix + '://' + self.root

            # check for errors
            error = None
            if u'error' in tree.keys():
                raise Exception, tree[u'error']

            # no error, so proceed
            treelist = tree[lookupUrl]
            self.anchors = treelist
            return self.anchors

        except Exception, inst:
            self.SetLastError( 'GetAnchorList' + ": " + str(type(inst)) + '\n' +
self.url )
            return []
```

SIDEBAR: A heretical revelation regarding XML

XML is the technology du jour for many purposes, and no one doubts its usefulness. However, parsing XML is computationally expensive, so it is useful to consider whether you can identify the data points of interest via the use of regular expressions, in which case you can use the RegexQueryUrl class, as we saw in chapter 9. If that approach can work for you, you'll write less code and it will run faster.

11.2.4 Using the Technorati Cosmos API classes

The code using these classes will look very familiar. This program generates a network representing the Technorati Cosmos of Paul Courant, the University of Michigan Provost and Über-Librarian.

```
# technorati1.py

# build networks using the Technorati API

import sys

from urlnet.technoratitree import TechnoratiTree, TECHNORATI_COSMOS_API
from urlnet.urlutils import PrintHierarchy
import urlnet.log
from urlnet.urlutils import GetConfigValue
from os.path import join

workingDir = GetConfigValue('workingDir')

urlnet.log.logging = True
# write the log output to a file...
urlnet.log.altfd = open(join( workingDir, "log-technorati1.txt"),'w')
# ...and only to the file, not to sys.stderr.
urlnet.log.file_only = True
```

```
mylog = urlnet.log.Log('main')

myTechnoratiKey=GetConfigValue("technoratiKey")
if myTechnoratiKey == None:
    raise Exception, 'You must provide a technorati key in urlnet.cfg'

net = TechnoratiTree(_maxLevel=2,
                  _technoratiApi=TECHNORATI_COSMOS_API,
                  _workingDir=workingDir,
                  _technoratiKey=myTechnoratiKey,
                  _sleeptime=1)
ret = net.BuildUrlTree('http://paulcourant.net/')

if ret:
    net.WritePajekFile('technorati1_cosmos_courant','technorati1_cosmos_courant')
    net.WriteGuessFile('technorati1_cosmos_courant_urls')            # url network
    net.WriteGuessFile('technorati1_cosmos_courant_domains',False)      #domain network

urlnet.log.altfd.close()
urlnet.log.altfd=None
```

You will see a warning message from the XML library, which you can safely ignore:

```
>>> FtWarning: Creation of InputSource without a URI
```

You notice we set _sleeptime to 1, overriding the TechnoratiTree constructor's default of 2 seconds; any delay less than one second may result in your getting blocked by the Technorati server's watchdog daemons. Whether the block is by IP address or API key, and how long the block lasts, I don't know, and I hope I never need to know.

An example of a Technorati cosmos was shown on the cover, and is repeated here; it was created from the output of the program shown above, except the starting URL was http://www.healthcareguy.com.

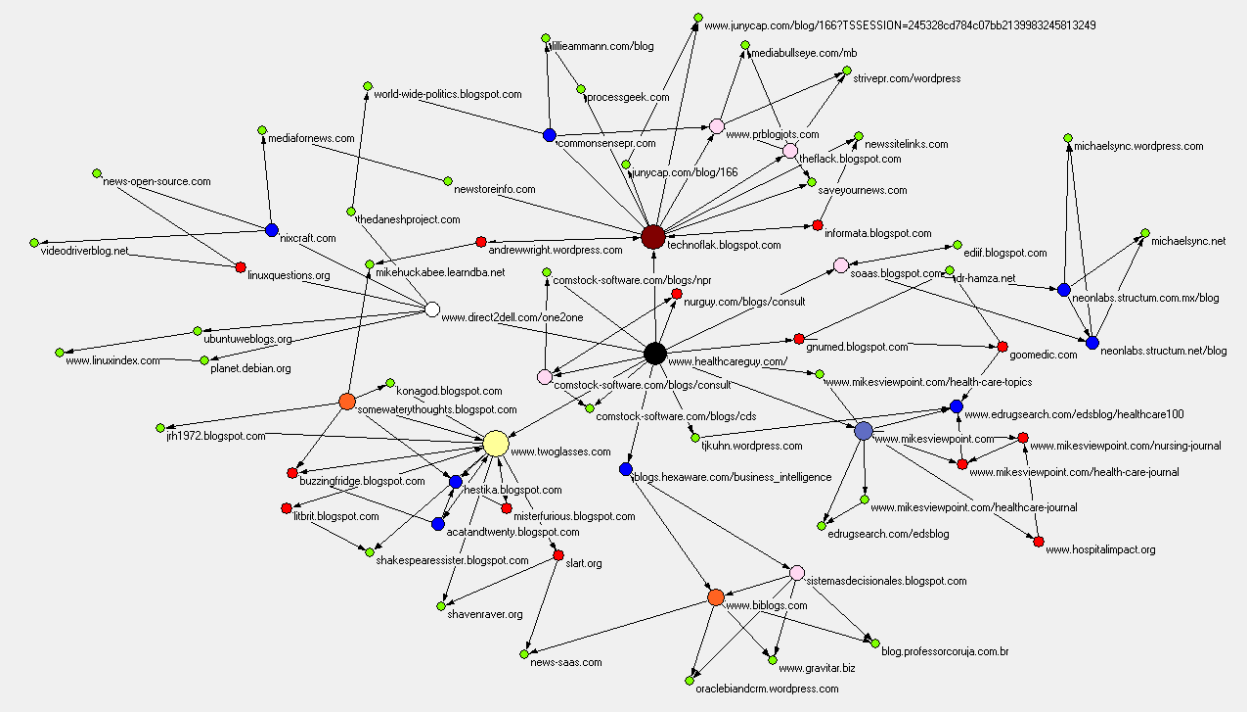


FIGURE 15 – A TECHNORATI COSMOS NETWORK

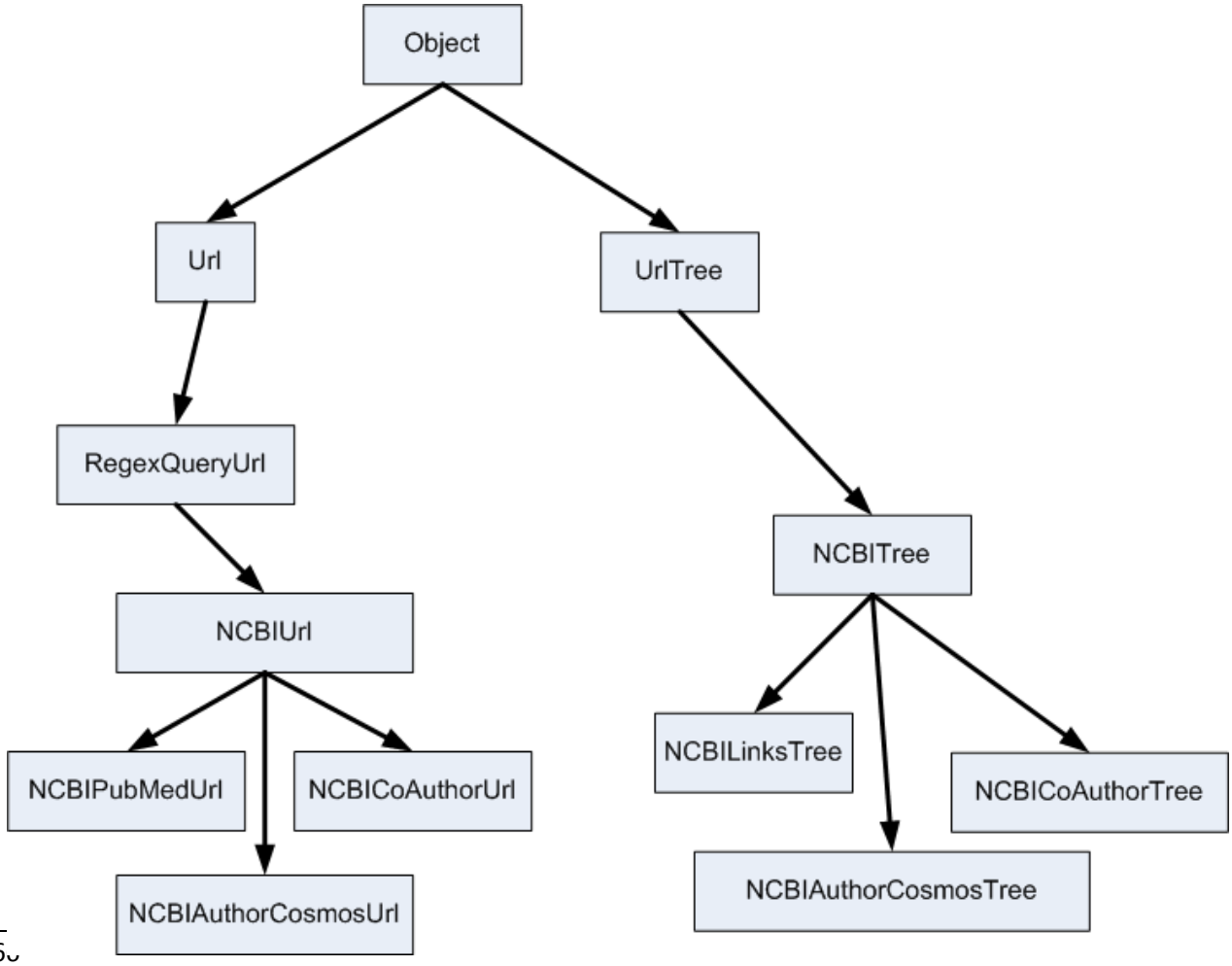
12 Working with Web Service APIs, Part Two: The NCBI APIs

12.1 Overview

The National Center for Biological Information (NCBI) is a unit of the National Library of Medicine (NLM), which in turn is one of the National Institutes of Health (NIH). NCBI has released a set of Web Service APIs that provide access to the NLM's many electronic databases. Their example programs are in Perl, and utilize a Perl module that contains an extensive set of functions. I've begun adding functionality to UrlNet to support calls to their Web APIs, and have some interesting features in place. However, the UrlNet classes and example programs I've added only scratched the surface of what's possible—and what needs to be done to achieve the level of support the NCBI Perl APIs provide. Like the NCBI Perl APIs, the UrlNet NCBI* classes work via HTTP GETs (or POSTs); the scripting code provides a convenient way to make these calls.

12.2 Modules

We have three Python classes that build networks using the NCBI API calls to look at here: co-author networks (NCBICoAuthorTree), author cosmos networks (NCBIAuthorCosmosTree), and linkage networks (NCBILinksTree). These three Python classes have a common parent, the NCBITree class,



whose parent is URLTree. These classes delegate URL-related work to corresponding classes (NCBICoAuthorUrl, NCBIAuthorCosmosUrl, and NCBILinksUrl), all of which share a common parent class, NCBIUrl, whose parent is RegexQueryUrl, which in turn has the Url class as its parent. For each of the NCBI* classes, there is a corresponding module named after the class it contains, with the class name converted to lower-case. There is also a module that contains only constants (ncbiconstants.py).

This document isn't going to talk much about what's going on under the hood in these classes, because the code is in serious flux. Walking through the code is the way to find out what's going on at this point.

12.2.1 Co-Author Networks

The NCBICoAuthorTree class finds the co-authors of an author; finds the co-authors of the co-authors; etc., to the depth specific for the instance. The example program ncbicoauthortree1.py shows how it is invoked:

```
# ncbicoauthortree1.py
import sys

from urlnet.ncbicoauthortree import NCBICoAuthorTree
# ncbicoauthortree1.py
import sys

from urlnet.ncbicoauthortree import NCBICoAuthorTree
import urlnet.log
from urlnet.urlutils import GetConfigValue
from os.path import join

workingDir = GetConfigValue('workingDir')

urlnet.log.logging = True
# write the log output to a file...
urlnet.log.altfd = open(join( workingDir, "log-ncbicoauthortree1.txt"),'w')

mylog = urlnet.log.Log('main')

net = NCBICoAuthorTree(_maxLevel=2)
ret = net.BuildUrlTree('Hunscher DA')

if ret:
    net.WritePajekFile('ncbicoauthortree1','ncbicoauthortree1')
    net.WriteGuessFile('ncbicoauthortree1_urls')            # url network

urlnet.log.altfd.close()
urlnet.log.altfd=None
```

This program will build a co-author network three levels deep for the author (PubMed name 'Hunscher DA'). It looks like this:

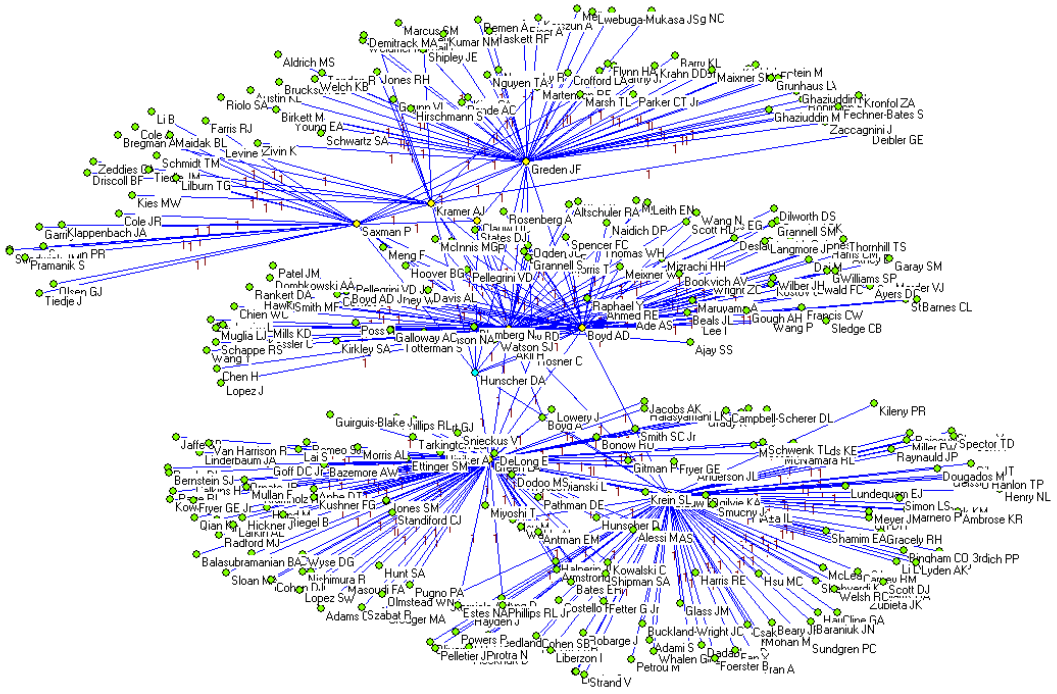


FIGURE 16 - NCBI-CO-AUTHOR NETWORK

As you can see, instead of URLs, we're looking at author names in typical PubMed format: Last name, space, and one or two initials for first and middle name. One of the quirks of the NCBI networks is that the URLs involved in the Web Service invocations are hidden. Actually they are more properly termed evanescent, because they come into existence solely for the HTTP call, and are not stored. Anchor lists consist (in this case) of author names, and Web Service URLs for each of these will be composed as needed.

12.2.2 AUTHOR COSMOS NETWORKS

The NCBIAuthorCosmosTree class generates a network that contains an author's papers, co-authors, and MeSH topics (NLM's Medical Subject Headings) interlinked. Three full-page graphics on the following pages show the NCBIAuthorCosmosTree class in action.

Figure 17 - NCBI Author cosmos on page 69 was created in Pajek, using the cosmos of Dr. Victor Strecher, a University of Michigan investigator.

In Figure 18 on page 70, he NodeTypes partition was used to extract a subnet containing only the co-authors and MeSH topics. These are the co-authors of Dr. Strecher on his last twenty papers, and the topics under which the papers were classified.

Figure 19 on page 71 was also done in Pajek after extracting a network using the generated NodeTypes partition. I used the Pajek option to scale font size according to the third partition, which in this case was the all-degree partition. It gives, in effect, a "tag cloud" for the author who is the subject of the network.

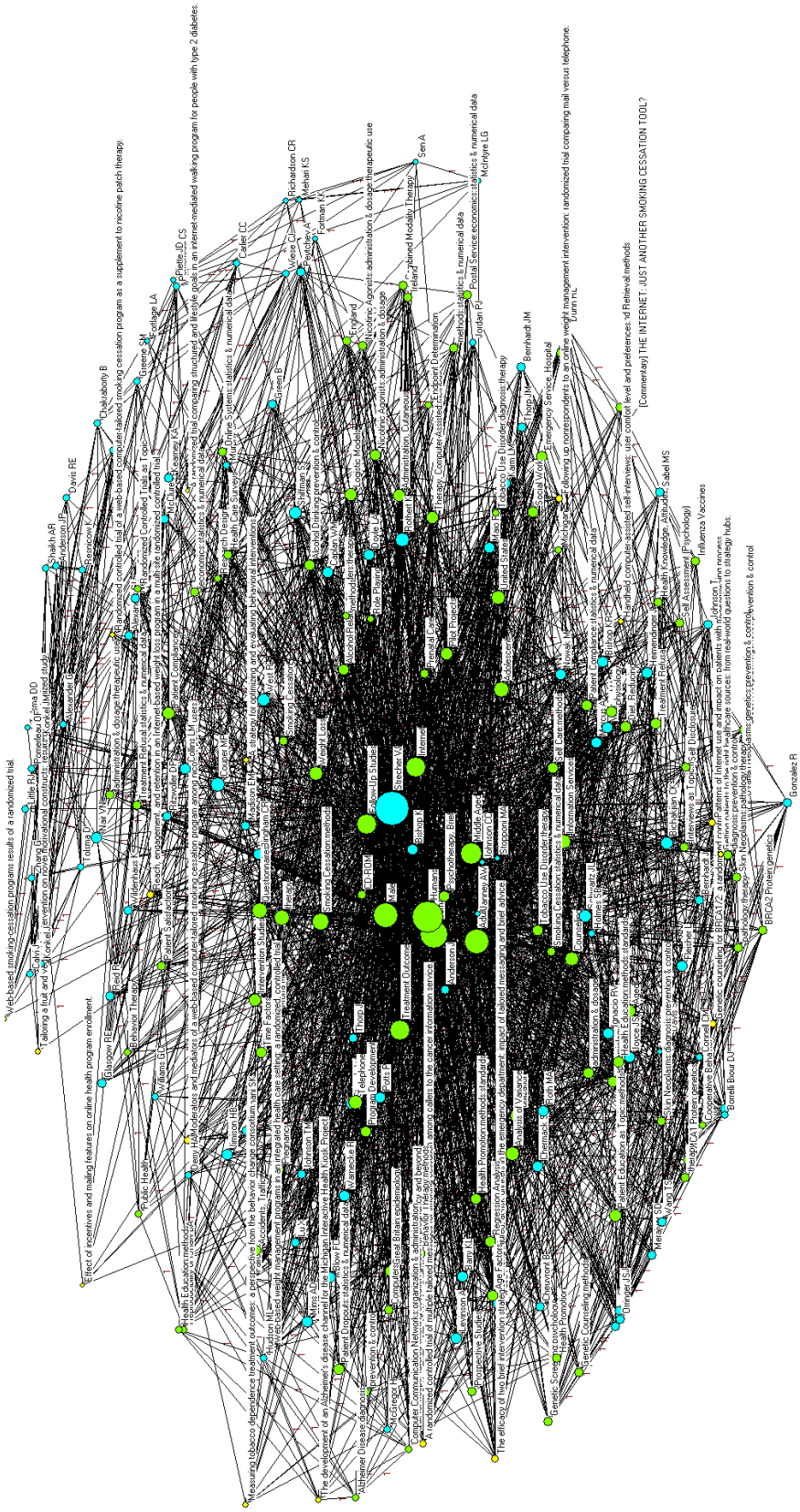


FIGURE 17 - NCBI AUTHOR COSMOS

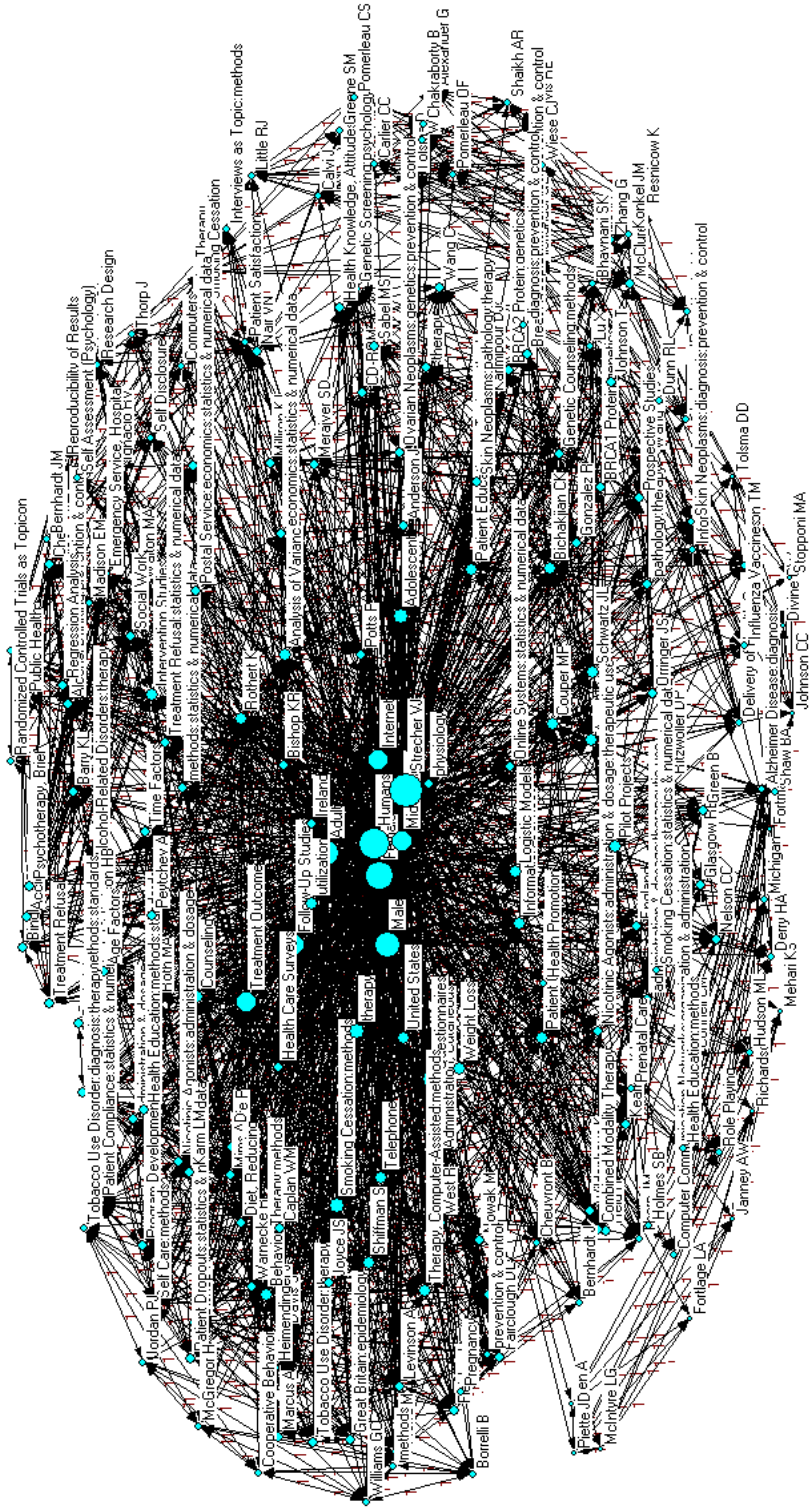


FIGURE 18 – AUTHORS AND MESH TOPICS

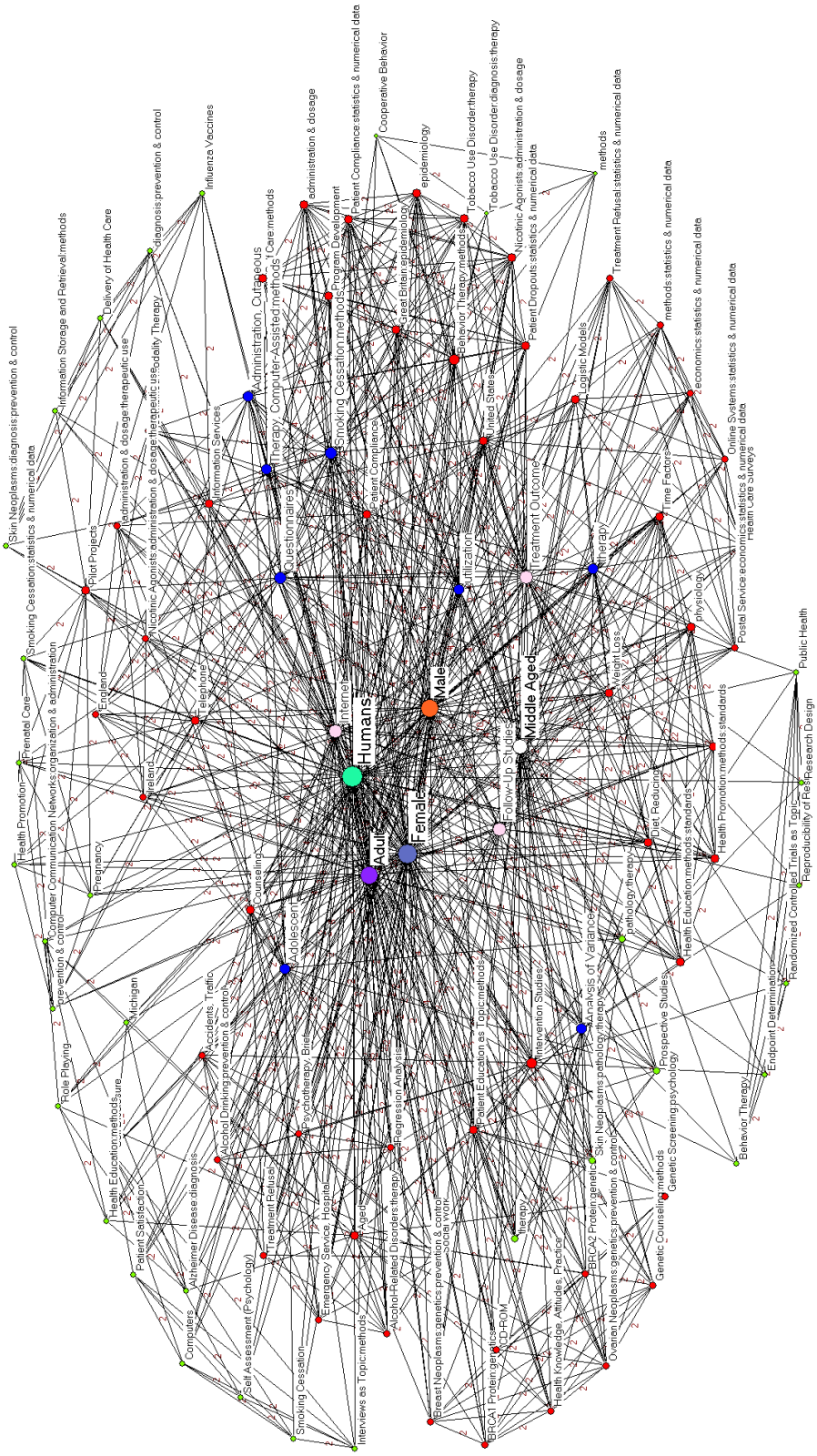


FIGURE 19 - MESH TOPICS NETWORK

12.2.3 LINKAGE NETWORKS

The third type of network you can currently generate from NCBI Web Services draws on the eLinks API, which allows you to extract linkages between databases that contain different data about the same topics. For example, you may want to see what entries exist for a particular gene in the NCBI Gene database along with the linkages to Protein, Nucleotide, and Single Nucleotide Polymorphism databases. NCBILinksTree routines will build a network that shows the linkages between these three databases with respect to the gene.

Here's the network:

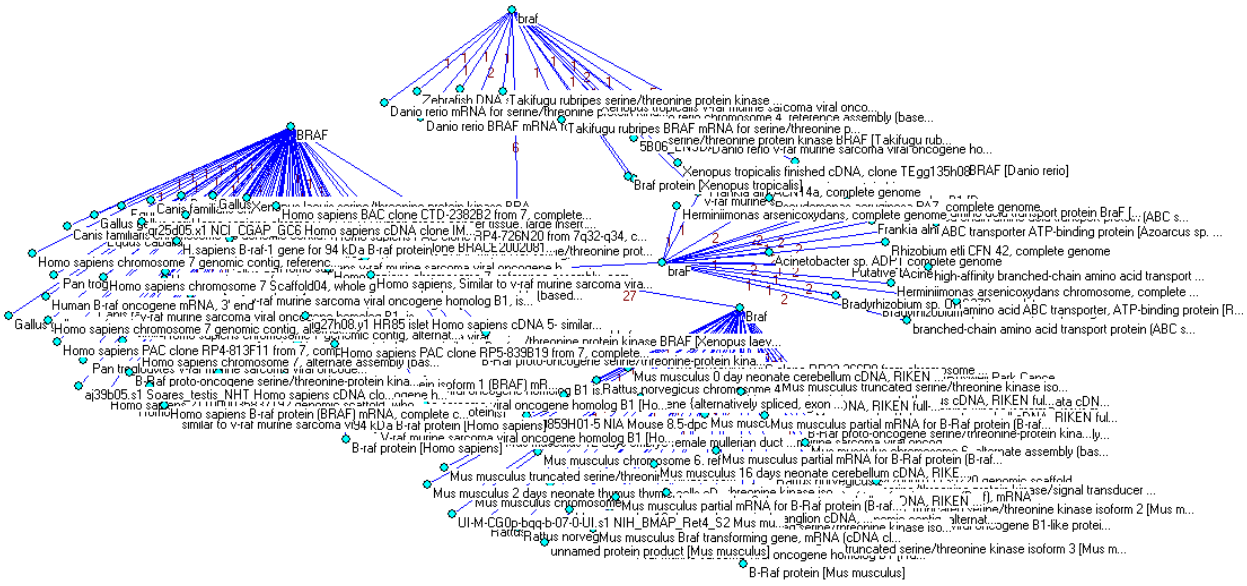


FIGURE 20 - LINKS BETWEEN GENE, PROTEIN, NUCLEOTIDE, AND SNP DATABASES REGARDING GENE 'BRAF'

What's interesting about this is that the four nodes with the most outlinks are all named BRAF in various combinations of upper- and lower-case letters. This suggests that while the search engine is case-insensitive, the database indices are case-sensitive; 'BRAF' matches all four spellings in case-insensitive mode, but the four spellings denote separate entries in one or another of the databases.

Here's the code from example program ncbilinkstree1.py, showing how this is done:

```
. . .
from urlnet.ncbilinkstree import NCBILinksTree
import urlnet.ncbiconstants
. . .
        try:
            net = NCBILinksTree(_maxLevel=1)
            net.SetProperty('nodeLengthLimit',50)
            # build the cosmos network of proteins, nucleotides, and SNPs around a gene
            dbs = urlnet.ncbiconstants.ConcatDBNames( (urlnet.ncbiconstants.PROTEIN,
                                                       urlnet.ncbiconstants.NUCLEOTIDE,
                                                       urlnet.ncbiconstants.SNP) )
```

```
        qry = 'BRAF[GENE]'
        net.BuildUrlForestWithPhantomRoot(qry,
                   DbSrcOfIds=urlnet.ncbiconstants.GENE,
                   DbsToLink=dbs)
        net.WritePajekFile(qry,qry)
    except Exception, e:
        myLog.Write( str(e) )
. . .
```

We see here some new and notable features related to the underlying eLinks API. The first is the use of constants from the ncbiconstants.py module. Most of these are simple upper-case-named symbols for eponymous lower-cased string values. I'm not sure there's a rational reason why I did this; it's really because I just don't feel comfortable with "magic numbers" and "magic tokens" in my code.

Another feature of the ncbiconstants module is the ConcatDBNames function, which correctly formats a list of databases for use in the eLinks API call. Always use this to concatenate the list of constants representing the databases involved in the eLink call. These are the 'to' databases referenced in the eLinks documentation, and are passed to NCBILinksTree.BuildUrlForestWithPhantomRoot as the DbsToLink argument. The 'from' database constant is passed as the DbSrcOfIds argument.

We set the nodeLengthLimit property, which limits Pajek and GUESS vertex names to the length you pass plus a 3-character ellipsis (…). This is by no means specific to the NCBI classes, but is useful here because there are some tediously long document, protein, gene, and SNP names. This might need to be adjusted to make sure enough context is provided to recognized the vertex by its truncated name.

We set the _maxLevel to 1, but with the current NCBILinksTree implementation, it doesn't matter; it gets what it gets and stops there.

Another example: You might wish to examine linkages in NCBI's Gene, Protein, and PubMed databases regarding a specific protein you have heard of, dUTPase, that is known to be related somehow to HIV. NCBILinksTree routines will build a network that shows the linkages between these three databases with respect to the protein. The code for this (ncbilinkstree2.py) differs only in the query and the databases involved; here are the modified lines:

```
. . .
        # build the cosmos network of genes, nucleotides, and SNPs around a protein
        # (dUTPase) related to HIV; throw in the related documents as well.
        dbs = urlnet.ncbiconstants.ConcatDBNames( (urlnet.ncbiconstants.GENE,
                                                   urlnet.ncbiconstants.NUCLEOTIDE,
                                                   urlnet.ncbiconstants.SNP,
                                                   urlnet.ncbiconstants.PUBMED) )
        qry = 'dUTPase HIV'
        net.BuildUrlForestWithPhantomRoot(qry,
                                  DbSrcOfIds=urlnet.ncbiconstants.PROTEIN,
                                  DbsToLink=dbs)
        net.WritePajekFile(qry,qry)
. . .
```

Here's the network resulting from this investigation:

FIGURE 21 - LINKS BETWEEN PROTEIN, GENE, AND PUBMED WITH RESPECT TO THE PROTEIN DUTPASE

What does this mean? I'm still trying to figure it out.

12.3 CAVEATS

The NCBI classes are a work in progress. I am progressing slowly. Expect bugs and anomalies, and please provide feedback and suggestions.

13 CONCLUSION

The library is pretty stable, but it's not commercial quality. Here are some of its flaws at present:

- **Code documentation.** I have been meaning to look into PyDoc, but have had more fun things to do for some time. The code is usually readable and sometimes commented; the comments are often still applicable. This manual should take care of out-of-the-box uses, but if you want to muck with the code, your best bet is to follow it in the debugger and then study it. Don't be surprised if you find the occasional kludge or dust-bunny. Also, most of the external-API classes (i.e. the ones you instantiate yourself, which are mostly the classes descended from UrlTree) have a wealth of defaulted arguments that allow you to do many cool things that are not documented. Similarly, there are some cool but undocumented properties that pop up in various places.
- **Defensive coding.** Python is not a language that does type-checking, preferring instead to go belly-up (i.e. throw an exception) when a datum's type is inappropriate for the occasion. I should be checking inputs more carefully; or maybe I need to learn to relax and enjoy the freedom.
- **Eat its own dog food.** The library should be able to read the output streams it writes. It doesn't. It bothers me, but not enough to do anything about it yet.
- **Better support for GUESS.** It is an abomination that I know so little about GUESS. I think it really annoys Lada and I value her opinion highly, so I may do something about this sooner than some of the others. **5/4/08 update: I have added new features for adding user-defined properties to GUESS networks, which is a step in the right direction.**
- **Support for other network analysis and visualization programs and packages.** I'd like to be able to read and write other formats, including but not limited to UCINET and NetworkX.
- **Consistency.** You'll notice that some functions have arguments that begin with an underscore, and others don't, with no apparent pattern to the deviations. You'll notice that sometimes I use properties when an argument would have been more appropriate. I find this sort of thing really annoying in other people's code, but when I do it myself, it feels strangely and irrationally OK. Sorry about that.
- **Un-Pythonic.** I suspect my code is un-Pythonic. It seems like every time I learn a new language, the code looks strangely like C and C++, the languages of my halcyon days. Educate me.
- **Unit-test code in the modules doesn't always work.** I used to put unit-test code in a main block under each class module, but moved a few months ago to creating the example programs. These started out the same as the test code in the class modules in many cases, but have evolved considerably over time, and the test code has in many cases become deprecated (i.e., it will blow up when you try to run it). I will fix or remove the in-module test code sometime in the future; meanwhile, focus on the examples as a learning tool.

I plan to keep working on it in my spare time, and appreciate any feedback you can provide.