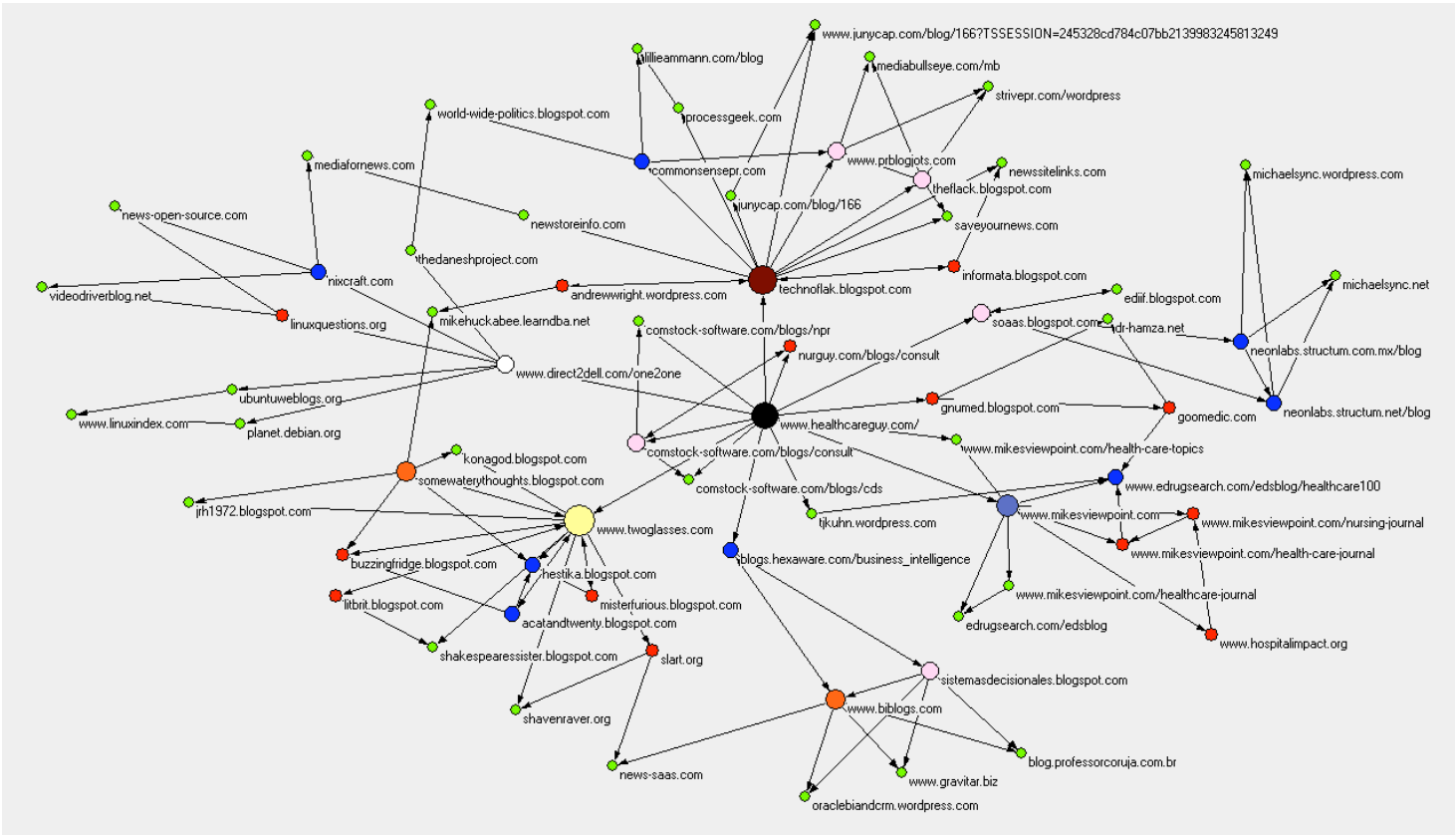
URLNET PYTHON LIBRARY:

COLLECTING DATA FOR NETWORK ANALYSIS BY SPIDERING THE WEB AND ACCESSING WEB SERVICES

DRAFT—BETA

Revision 0.90.05 RC—22 October, 2009

Dale Hunscher, MSI
University of Michigan Medical School
South Wind Design, Inc.



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.
01 AUG 2009	DALE HUNSCHER	0.85	Accumulated bug fixes; added top-level domain analysis features.
22 OCT 2009	DALE HUNSCHER	0.90	Many additions and improvements, including but not limited to page content checker callback functions for filtering pages and setting custom properties, and enhancements to use of custom properties to create custom Pajek partitions and vectors, and to add custom attributes to GUESS vertices.

TABLE OF CONTENTS

FIGURES

TABLES

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 process, often referred to as ***spidering the Web,*** creates a directed graph that may loop back on itself, and is therefore in may instances a true network rather than an acyclic graph. It is often useful to create such networks when studying the structural and dynamic properties of the World Wide Web and its content. For example, spidering can be used to determine 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 four formats: a printable text hierarchy; a simple two-column edge network containing URL-to-URL edge pairs; Pajek project and network files; and GUESS network files. Other formats can be added easily.

The two-column edge network can be imported into a number of different network analysis and visualization programs, such as the freeware NodeXL macro package for Microsoft Excel. 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 each of these programs from the Web at the addresses shown in the following table.

TABLE 1 – SUPPORTED NETWORK ANALYSIS PROGRAMS

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.
NodeXL	Formerly called ".NetMap", this is a freeware macro library for Excel 2007 and a set of .NET libraries that can be used to build custom applications. Its home page can be found on the Web at http://www.codeplex.com/NodeXL. It was developed by Marc Smith and others at

	Microsoft Research, and is maintained by a globally distributed team of Microsoft Research staff, academicians, and others. Marc Smith is now with the Connected Action Consulting Group, found on the Web at http://www.connectedaction.net/.

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 network theory and turned me on to Python programming. 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.

TABLE 2 – EXAMPLE PROGRAMS

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).
Urltree4blogs	UrlTree.BuildUrlTree	Demonstrates use of the host name option for domain networks.
Urlforest1	UrlTree.BuildUrlForest	Builds a vanilla URL forest from a hard-coded list of URLs.
Urlforest4blogs	UrlTree.BuildUrlForest	Builds a URL forest from a hard-coded list of URLs, using host names instead of domain names in the domain networks; also demonstrates how to generate a single Pajek domain network as opposed to a full project.
Phantomroot1	UrlTree. BuildUrlForestWithPhantomRoot	Retrieves the National Library of Medicine's MedlinePlus melanoma portal and creates a URL forest from its outlinks, then writes a Pajek project.
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.
Sitemap1	UrlTree.BuildUrlTree	Shows how to create a site-map network.
Writing output for consumption by network visualization and analysis programs		
writepajeknetworkfile	UrlTree.BuildUrlTree	Shows how to generate a single Pajek URL network file instead of an entire project. Also shows how to generate individual partition files.
Generateguessnets1	UrlTree.BuildUrlTree	Generates two network files for GUESS, one for

Example program	Network build method	Comments
		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.
Writepairfile	UrlTree.BuildUrlTree	Demonstrates generation of a two-column URL-to-URL edge network, a format that can be read by many network analysis programs.
Saving and reloading networks		
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.
Controlling what gets included in the network		
ignorabletext1	UrlTree.BuildUrlTree	Shows a simple approach to filtering useless URLs out of the network before they are added, by choosing to either ignore URLs or to truncate spidering at URLs.
Includeexclude.py	UrlTree.BuildUrlTree	Shows how to use the library's parameterized relevance algorithm as a page checker callback function, to decide whether a URL will be included in or excluded from the URL and domain networks.
Advanced operations		
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.
redirects1	UrlTree. 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.
netsmerged1	AOLTree. BuildUrlTreeWithPlaceholderRoot and AOLTree. BuildUrlForestWithPhantomRoot	Shows how to generate and merge multiple trees and/or forests into the same network.
netsmerged2	AOLTree. BuildUrlTreeWithPlaceholderRoot and	Same as netsmerged1, but adds partitions and vectors for top-level domain analysis.

Example program	Network build method	Comments
	AOLTree. BuildUrlForestWithPhantomRoot	
netsmerged3	AOLTree. BuildUrlTreeWithPlaceholderRoot and AOLTree. BuildUrlForestWithPhantomRoot	Same as netsmerged2, but adds visualization of click probability by position in result set.
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.
Linktree1	GoogleLinkTree.BuildUrlTree	Use Google's link facility for tracking *inbound* links to a page, to see by what routes on the Web a given page can be reached.
Creating custom partitions and vectors		
customproperties1	UrlTree.BuildUrlTree	Shows how to create Pajek partitions and vectors and/or add attributes to GUESS vertices using custom properties based on analysis of the URL.
Customproperties2	UrlTree.BuildUrlTree	Shows how to create custom Pajek partitions and vectors and/or add attributes to GUESS vertices using custom properties based on analysis of the page content.
Customproperties3	YahooTree. BuildUrlTreeWithPlaceholderRoot	A real-world example program showing off the use of custom properties based on analysis of the URL to generate additional Pajek partitions and vectors.
Partition1	UrlTree.BuildUrlTree	Demonstrates two more ways to create additional Pajek partitions from node properties.
flaggedpartition	UrlTree.BuildUrlTree	Shows how to create a custom partition in which URLs from a user-defined list are identified if they appear in the network.
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 API key and

Example program	Network build method	Comments
		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 author's publications in PubMed).
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 construct a network showing te connections between a gene and related proteins, nucleotides, and SNPs.
Ncbilinkstree2	NCBILinksTree. BuildUrlForestWithPhantomRoot	Same program as ncbilinkstree1, 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, a processing commonly referred to as *spidering*:

```
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.

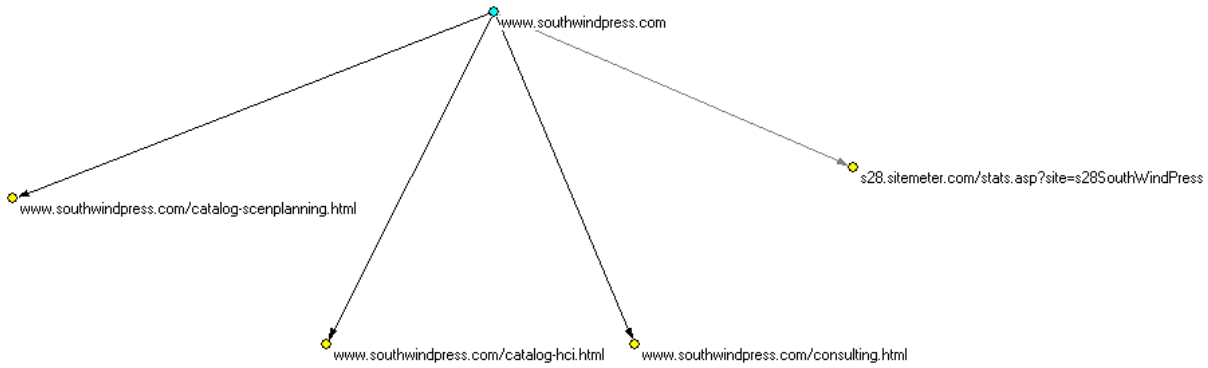


FIGURE 1 – A SIMPLE 2-LEVEL URL NETWORK

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:

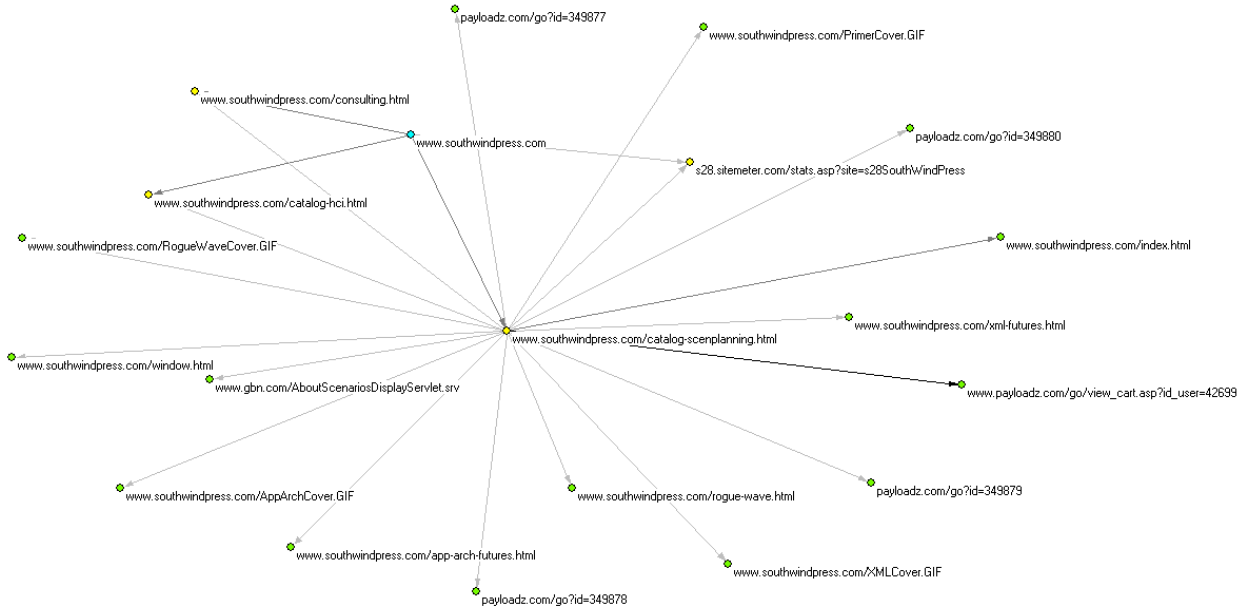


FIGURE 2 – SAME NETWORK INCREASED TO 3 LEVELS.

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:

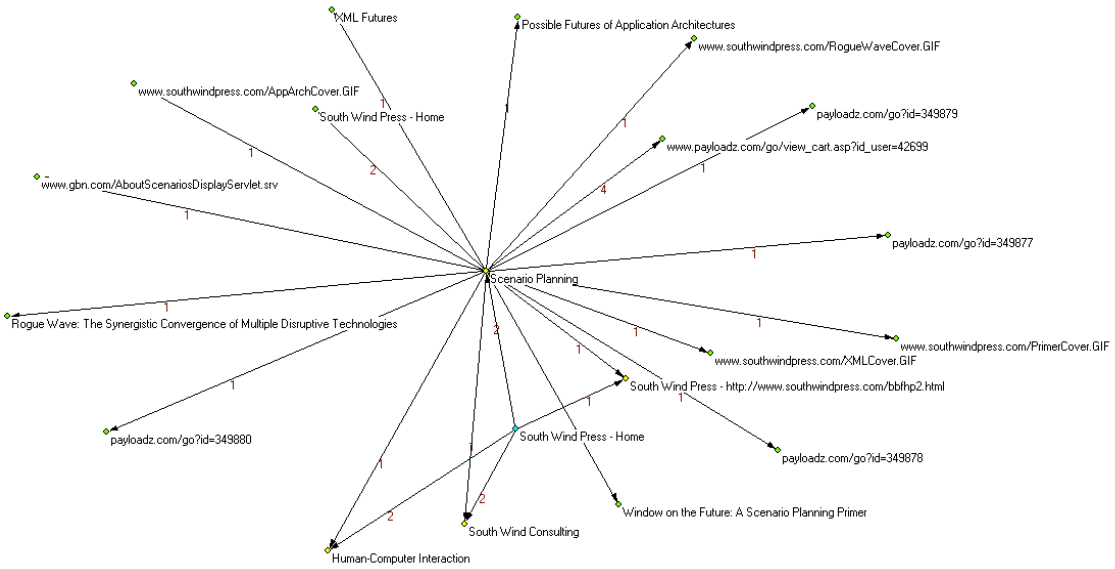


FIGURE 3 – 3-LEVEL NETWORK WITH PAGE TITLES FOR NODE NAMES INSTEAD OF URLS

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.

Performance Caveat

Be aware that turning on this feature (or using the custom page content checker callback feature we'll encounter in chapter 10) slows things down considerably when spidering large networks, because leaf-node pages must be partially read to get the page title. If this is not turned on, leaf-node URLs are not used in HTTP GET operations.

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')
```

This new example is found in 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:

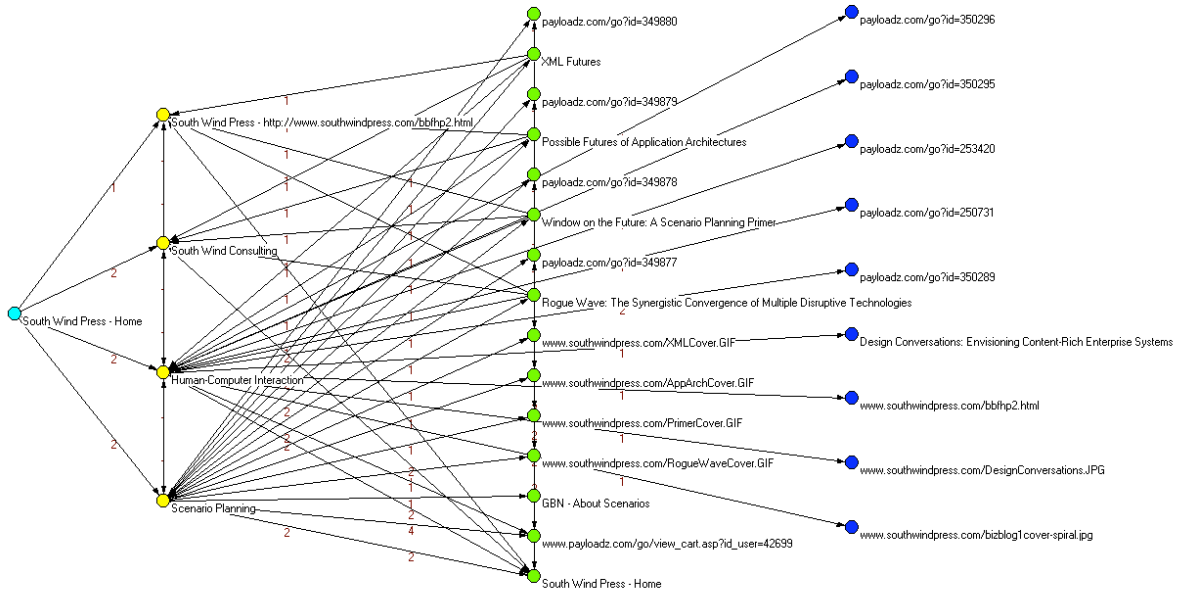


FIGURE 4 – A SITE MAP NETWORK

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 can 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 8.3, 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:

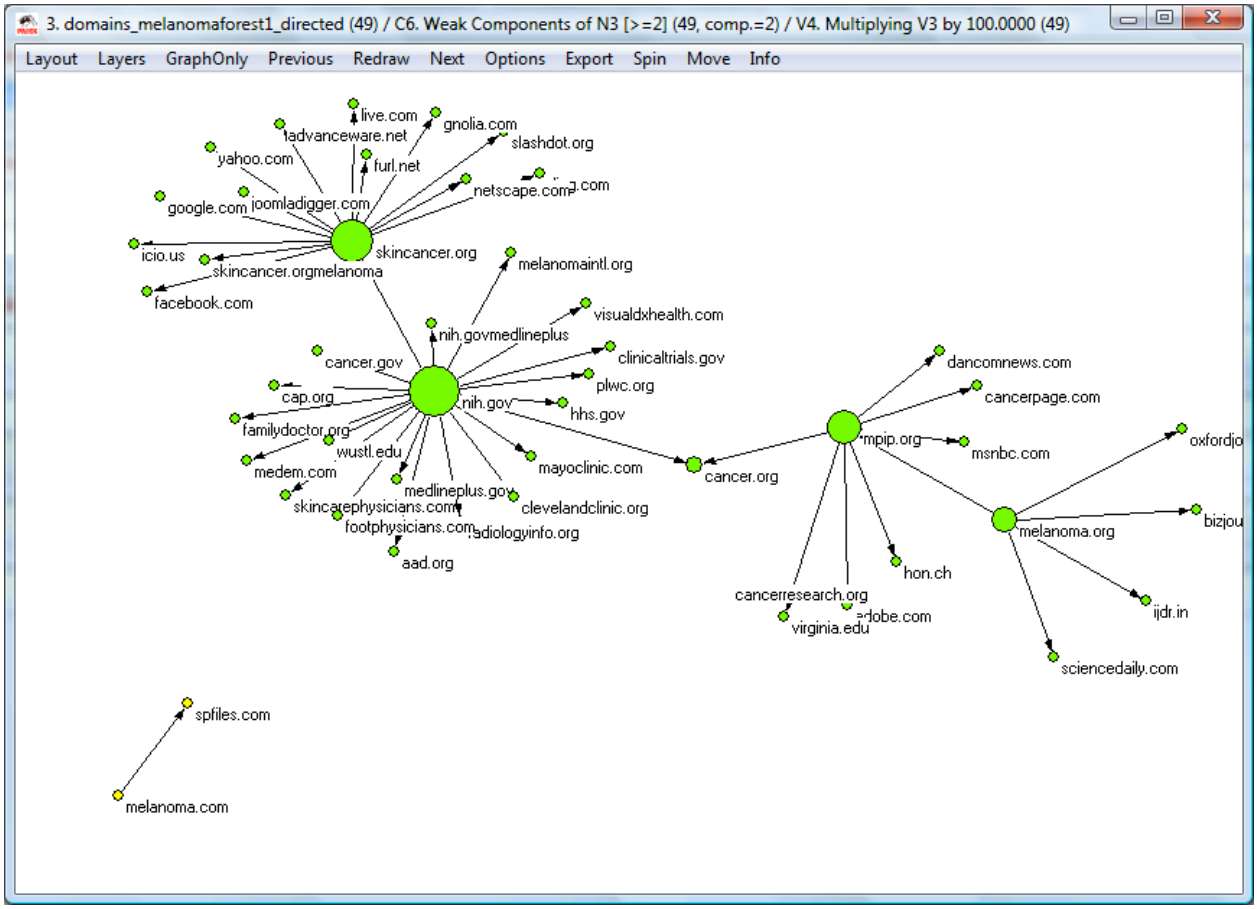


FIGURE 5 – A FOREST NETWORK

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 10.1, Ignoring specific URLs and URL types .

```
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:

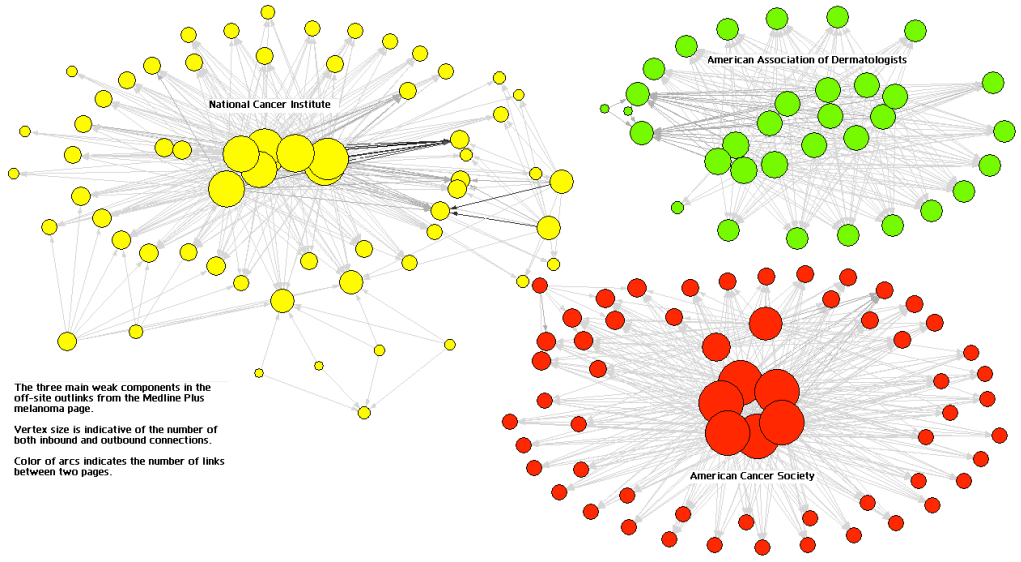


FIGURE 6 – FOREST NETWORK GENERATED FROM PHANTOM ROOT (CLEANED AND DRESSED UP A BIT)

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:

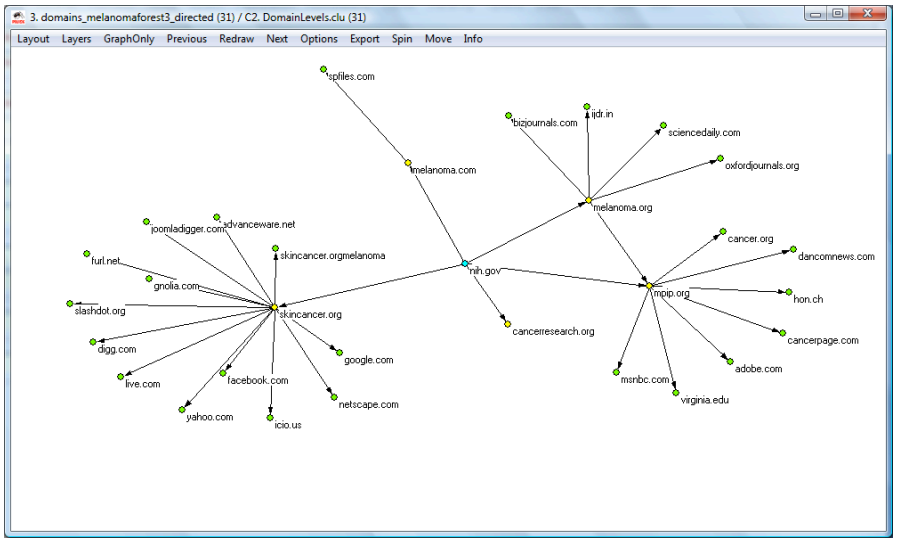


FIGURE 7 – NETWORK WITH PLACEHOLDER ROOT NODE

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 PAJEK PROJECT FILES

In the examples above, we already 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. In many cases, having all these in one file is the easiest way to do network analysis in Pajek.

5.3 GENERATING PAJEK PARTITIONS AND NETWORKS IN SEPARATE FILES

It is often useful to work with a single network and/or partition out of the set created in the Pajek project file. The member function UrlTree.WritePajekNetworkFile takes care of writing the network. Its arguments allow you to specify the base of the file name (it will always end in '.net' when written to disk), the network name as seen by Pajek, whether the URL or domain network is to be written, and whether the network is directed or undirected. The example below illustrates how to use this function, along with the partition file-writing feature. If this is all you want to do and you don't need separate partition files, you can skip to the example code.

Partitions are a bit more complicated, but we'll work our way through a step at a time. By default, UrlNet's UrlTree and most of its derivative classes create the data used to generate a Pajek partition 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. The level of each node is stored in a *property* associated with the node. Properties are name-value pairs stored in a dictionary available on each UrlNet object, and will be discussed in greater detail in chapter 7, in section 7.1.1.1, The property list. If you need the level partition in a file of its own and don't need any other partitions, the example code shows you how.

Geek alert

From here to the example code, we're discussing a capability that requires some degree of sophistication in Python programming. Many readers don't possess this level of skill, and of that group, many will never need that level of skill, so don't hesitate to skip ahead if you've already got the info you need and can get by with the functions demonstrated in the example. You can always come back and read this part if and when you are in need of this detail.

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

5.3.1 Method #1: When property value equals partition number

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 example shown here produces one of the same networks and also one of the same partitions as those written to the Pajek project file in UrlTree.WritePajekFile.

```
# writepajeknetworkfile.py
from urlnet.urltree import UrlTree
net = UrlTree()
net.BuildUrlTree('http://www.southwindpress.com')
# write URL network file
net.WritePajekNetworkFile('urltreenet1', 'urltreenet1urls',  rlNet = True)
# write levels
partition net.WritePajekPartitionFile('urltreenet1UrlLevels','URLLevels',\
                    'level', rlNet=True,valueDict=None)
```

It's also possible to do the same things for the domain network (continuation of the same program):

```
# write domain network file
net.WritePajekNetworkFile('urltreedomnet1', 'urltreenet1domains', \
             rlNet = False)
# write levels partition
net.WritePajekPartitionFile('urltreenet1DomainLevels','DomainLevels', \
                     'level',  rlNet=False,valueDict=None)
```

In this method, it is often 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)'. UrlNet can't read your mind, and it's possible that neither can you a few months after you generate the partition, so this kind of decoding can come in very handy.

The example program shown here is writepajeknetworkfile.py. There is an older example program showing the use of lower-level partition-writing functions, called partition1.py, but the methods shown in that example are considered deprecated for external use. The functions in the new example are easier to use and accomplish all the same things.

5.3.2 Method #2: Look that up in your Funk & Wagnall's

The second method is a bit more complicated, but not much. It is intended for use in situations when the property to be used for the partition has a value domain that is different from, but can be mapped to, the desired partition numbers. This is the case, for example, when the property values are string,

Boolean, or floating-point values rather than integers. Even when the property values are integers, you may want to map the values in the property's value domain to a different set of integer values.

In addition to the arguments passed in method #1, 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.

For example, if the domain of property value 'gender' on nodes in the network consisted of three string values, 'male', 'female', and 'no response', the dictionary might look like this:

```
myDict = { 'male' : 0, 'female' : 1, 'no response' : 2,  }
```

The invocation of UrlTree.WritePajekPartitionFile would then look like this:

```
partition net.WritePajekPartitionFile('urltreenet1UrlLevels','Gender responses',\
                    'gender', rlNet=True,valueDict=myDict)
```

This method is the one to use 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 automatically uses the dictionary key-value pairs to do this for you.

The example program shows how to do this by mapping the level property value domain to a set of arbitrary integers:

```
myDict = { 0 : 123, 1 : 234, 2 : 456, }
# write levels partition using the dictionary approach
net.WritePajekPartitionFile('urltreenet1DomainLevelsUsingDict','DomainLevels', \
            'level',  rlNet=False,valueDict=myDict)
```

The output looks like this:

```
*Partition DomainLevels (0=123,1=234,2=456)
*Vertices 10
     123
     456
     456
     234
     456
     456
     456
     456
     456
     456
```

5.4 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:

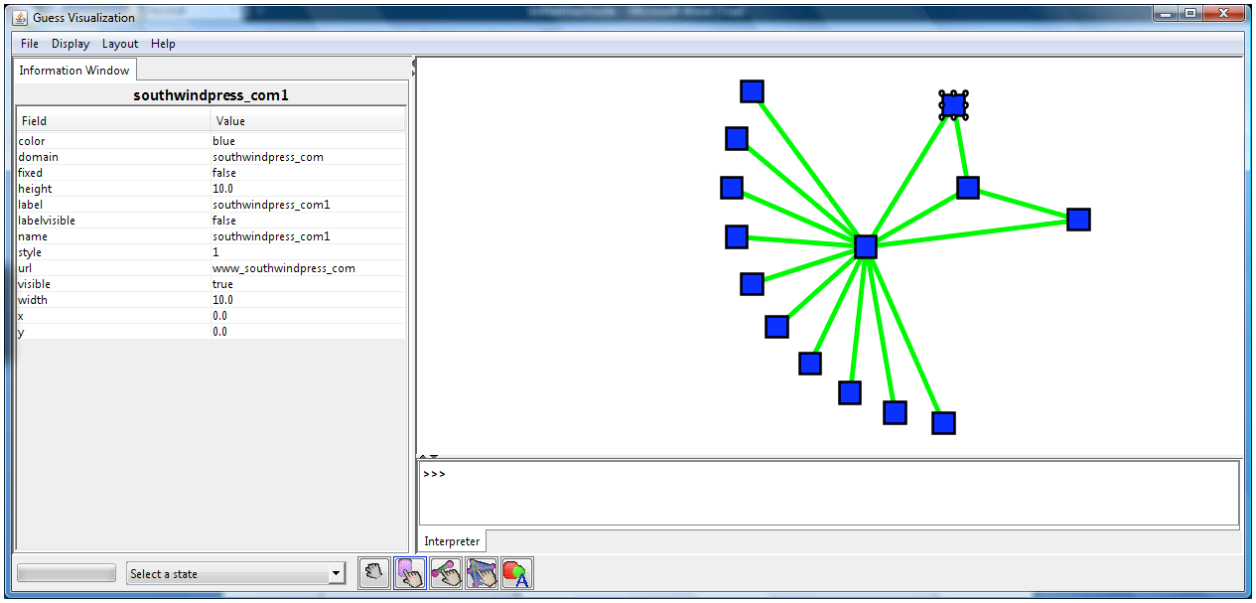


FIGURE 8 – 2-LEVEL NETWORK IN GUESS

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.

5.5 WRITING A TWO-COLUMN URL-TO-URL EDGE NETWORK

Many of the network analysis programs handle the most simple expression of a network, the two-column edge network, in which the data structure consists of a vertex name, a separator of some sort, and another vertex name. The first name is the origin of the arc, if the network is directed, and the second name is the destination of the arc. If the network is undirected, the first and second names are interchangeable. If edge/arc weights are important, multiple lines with the same origin-destination name pairs represent them.

Here's a sample program, writepairfile.py, that shows the UrlNet facilities for writing this kind of network output:

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

```
# write URL network file
net.WritePairNetworkFile('urltree1', 'urltree1urls',  rlNet = True)

# write domain network file
# in this one we change the default delimiter (tab) to a bunch of spaces
# and exercise the other optional arguments as well
net.WritePairNetworkFile('urltree1',
                         'urltree1domains',
                          rlNet = False, # do domains instead
                         uniquePairs = True,
                         delimiter = '        ')
```

The second net.WritePairNetworkFile invocation shows all the possible arguments. By setting the rlNet argument to False, we trigger generation of a domain network. By setting uniquePairs = True, we ensure that only one line per pair is generated, which is desirable if we don't mind losing arc/edge weight information. By setting the delimiter argument to some text string, in this case eight spaces, we override the default delimiter, a tab character. Here's what output of the second invocation:

```
southwindpress.com       gbn.com
southwindpress.com       payloadz.com
southwindpress.com       sitemeter.com
sitemeter.com       southwindpress.com
sitemeter.com       n4g.com
sitemeter.com       giantbikes.net
sitemeter.com       wonkette.com
sitemeter.com       simplyrecipes.com
sitemeter.com       lazygirldesigns.com
sitemeter.com       sbnation.com
```

5.6 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
        rl.com
       del.icio.us
       google.com
       live.com
       facebook.com
        rlNet a.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
        rlNet a.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.

5.7 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 (where the custom originated), 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

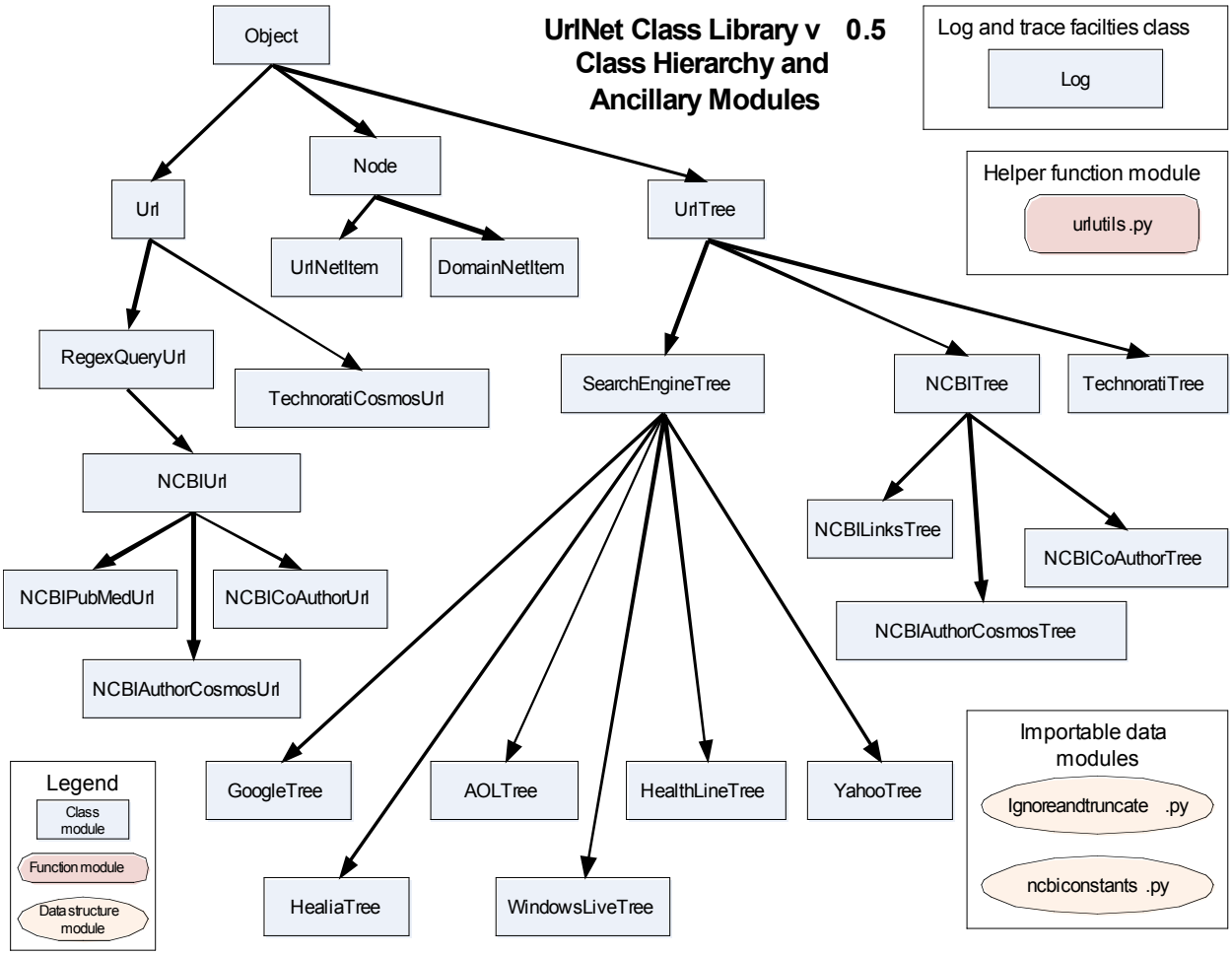


FIGURE 9 – URLNET CLASS AND MODULE HIERARCHY

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 class 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 (see discussion and list of properties in section 7.4, Properties used in UrlNet, on page 43).

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, thus far illustrated only in the examples in chapter 3 that demonstrated substitution of page titles for URLs in naming vertices, 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.

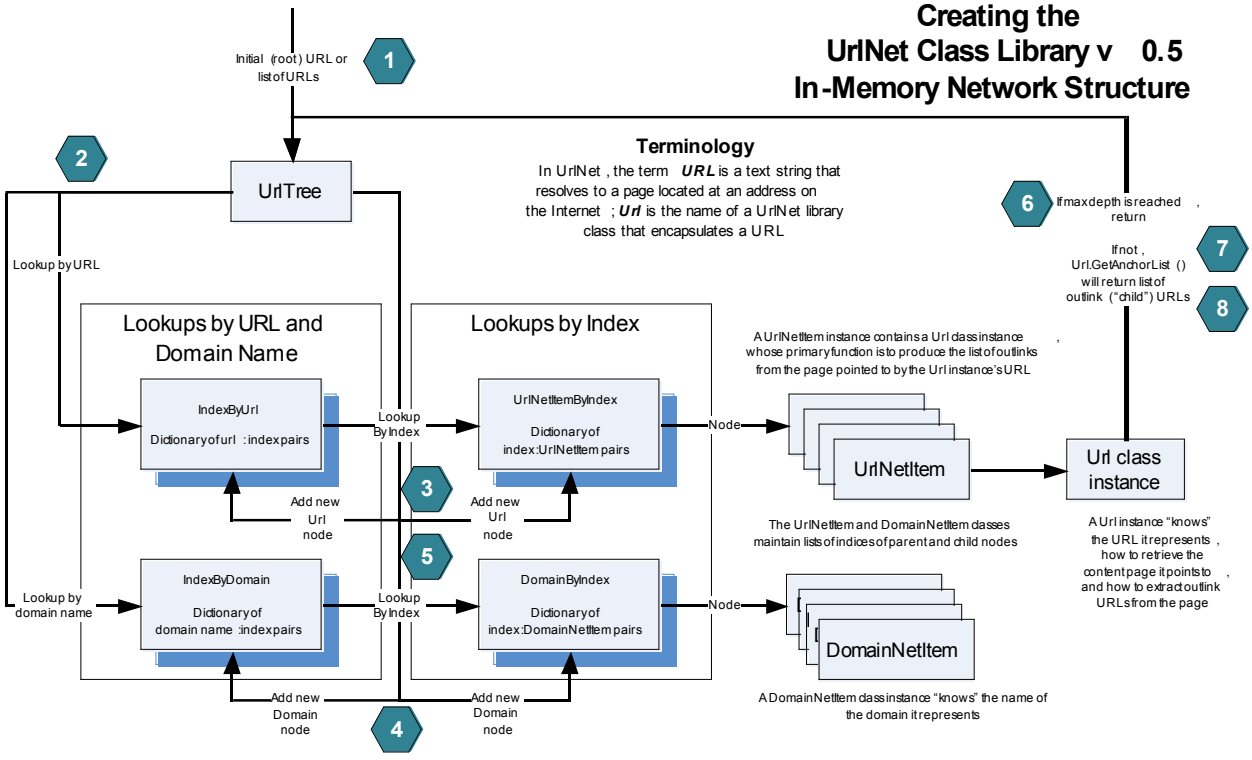


FIGURE 10 – SCHEMATIC OF THE NETWORK GENERATION ALGORITHM

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. If the URL doesn't match anything in the ignorableText list (whose usage is more fully described in section 10.1.1), 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, OR if the URL matches an item in the truncatableText list (more fully described in section 10.1.2), return without retrieving the page and harvesting its outlinks. This ends recursion for this node.

7. If the conditions in the previous step did not apply, 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. About half of the library's 10,000+ 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 and noting the time at which each occurred, it can also be used to time function execution 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.

CAVEATS:

- The log module variables named here are global to the module, therefore cannot be nested transparently.
- If you set altfd to a file handle multiple times in the same program, make sure the files you open have different pathnames each time you open the handle, lest you overwrite a file closed earlier in the program.

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.

7.4 PROPERTIES USED IN URLNET

More experienced programmers among the readers of this document may notice that in many cases property-setting is used when a constructor argument might seem more appropriate; at the very least it may seem as if there is no rational basis for using one versus the other to pass information to the internal functions of an object instance or to the delegatee classes associated with an object (e.g., to UrlNetItem and Url class instances created during the spidering processes and stored in the collections of the UrlNet or UrlNet-derived network class instance. This is a valid insight, but there is an explanation for this seemingly arbitrary inconsistency.

At some point in designing derivative classes from UrlNet, in particular when doing research using the search engine class subtree, managing the lengthy list of constructor arguments began to be tedious and error-prone. Moreover, many of the arguments being passed to network class constructors were deliberately intended to be forwarded to classes to which the network class delegated some work.

Harkening back to my LISP/CLOS programming days, I began to rely more extensively on the property getting and setting methods of the Object class at the root of the hierarchy. In keeping with the software engineering principle of information hiding, it seemed better to provide delegate classes with a getter function they could invoke on the network class instance rather than a variable they could read directly. This affords the network class a degree of security by allowing it to override the property setter function and programmatically decide which callers can actually set values, although I have never used this security feature in practice.

In retrospect, I can posit a rule that I should have applied from the very beginning: arguments to constructors that almost always will take the default value should be replaced by properties set in the constructor. This eliminates the need for derived classes to repeat all these arguments in their constructors in order to keep them exposed to the outside world.

Though I sincerely believe in this rationale for use of properties versus constructor arguments, it is no doubt confusing to others at times, keeping track of which properties are used in any given scenario, especially since they control behaviors that can be combined in myriad ways. To help reduce the degree of puzzlement, I present here a table of properties in use at the time this version of the manual is written. I hope it helps you. They are presented in alphabetical order by property name.

TABLE 3 – PROPERTIES USED IN URLNET

Property	Library files where used	Examples	What it does (and other notes on its use)
additionalDomainAttrs	urltree.py		A list of names of user-defined attributes stored in each DomainItem's property list. These can be used to create Pajek partitions and are automatically added to GUESS networks.
additionalUrlAttrs	searchenginetree.py,		A list of names of user-defined attributes stored in

Property	Library files where used	Examples	What it does (and other notes on its use)
	urltree.py		each UrlNetItem's property list. These can be used to create Pajek partitions and are automatically added to GUESS networks.
Current_page	relatedurl.py, url.py		If the UrlTree-derived instance's 'save_page_as_property_value' property value is not None or False, this property on the Url object stores the text of the page retrieved at this location.
Email	ncbi*.py		For use in NCBI API calls.
Exclude_patternlist	urlutils.py	includeexclude.py	Regex patterns to identify pages to exclude based on content.
Exclude_patternlist_flags	urlutils.py	includeexclude.py	Flags to use in the applications of exclusion regex patterns (see re module for definition of useful flags).
filterToBasicReadingLevel	healiatree.py	none	Can be used in collector programs for the Healia search engine to test its ability to limit results to a basic reading level.
filterToKeep	regexqueryurl.py, urltree.py		used in Top-Level Domain (TLD) processing to identify which TLDs to keep when processing a search engine result set.
Findall_args	aoltree.py, bingtree.py, googletree.py, yahootree.py, healiatree.py, healthlinetree.py, googlelinktree.py, windowslivetree.py, regexqueryurl.py, collectorutils.py	*-stopsmoking-collector.py, searchengine.py, regexqueryurl1.py, searchengine2.py	Flags argument to pass to re.findall, for example to make the search case-insensitive. See re module documentation for details.
forProfitUrlTLD	urltree.py	internal	Boolean value indicating whether the URL is published by a for-profit entity, as determined by its top-level domain or by a TLDExceptions entry,
getTitles	googelinktree.py, relatedurl.py, url.py, urltree.py	generateguessnets1.py, logging*.py, printhierarchy*.py, redirects1.py, sitemap1.py, urltree3.py	Must be set to True to trigger acquisition of page titles if the network-output routines are to use titles instead of URLs in URL networks.
Include_patternlist	urlutils.py	includeexclude.py	Regex patterns to identify pages to include based on content.
Include_patternlist_flags	urlutils.py	includeexclude.py	Flags to use in the applications of inclusion regex patterns (see re module for definition of useful flags).
Includeexclude_level	urlutils.py	includeexclude.py	Level at which to start applying include/exclude rules. Defaults to 1 to avoid inadvertently excluding the root node and thereby producing an empty tree.
Level	ncbi*tree.py, urltree.py		Set on each UrlNetItem and DomainItem to indicate the first level (with zero being the root level) on which the URL or domain is encountered in the network.
Max_domain_prob	searchenginetree.py		Used in search engine result set processing to keep track of the highest probability of the user encountering the URL.
NCBI_IdCountLimit	ncbiurl.py		Used in NCBI searches to limit the number of entity IDs to be returned in queries that return ID sets.
NCBIResultSetSizeLimit	ncbi*.py		Used in NCBI searches to limit the size of result sets from its retrieval APIs.
nextUrlClass	aoltree.py, bingtree.py, googletree.py, yahootree.py, healiatree.py, healthlinetree.py, googlelinktree.py, windowslivetree.py, regexqueryurl.py	internal	Used in SearchEngineTree-derived classes to store the class (Url or one of its subclasses) to use in processing network levels below the root (query) node.
nodeLengthLimit	urlutils.py	ncbilinkstree*.py	Limit length of node (vertex) name when generating networks.

Property	Library files where used	Examples	What it does (and other notes on its use)
numSearchEngineResults	aoltree.py, bingtree.py, googletree.py, yahootree.py, healiatree.py, healthlinetree.py, googlelinktree.py, windowslivetree.py, searchenginetree.py, regexqueryurl.py	internal	Used in SearchEngineTree-derived classes to store the size of the search engine result set to be retrieved in querying the search engine. Must be a legal value for the particular search engine.
Pos_prob	searchenginetree.py		Used in UrlNetItem-derived instances in the networks of SearchEngineTree-derived classes to store the highest probability of the node being encountered by a user navigating the result set "forest".
regexPattern	aoltree.py, bingtree.py, googletree.py, yahootree.py, healiatree.py, healthlinetree.py, googlelinktree.py, windowslivetree.py, regexqueryurl.py, collectorutils.py	*-stopsmoking-collector.py, searchengine.py, regexqueryurl1.py, searchengine2.py	Pattern, or list of patterns, used to identify search results in search engine result-set spidering. Each regex expression is used in a call to re.findall.
reportButDontDie	ncbiurl.py		In NCBI classes, if this property is True, report errors on NCBI API calls but don't throw an Exception if NCBI returns an error.
Request-headers	relatedurl.py, url.py, urlutils.py, urltree.py		Baseline dictionary of request headers to use in HTTP GET operations. Default is empty dictionary.
reverseArcOrEdgeDirection	urltree.py, googlelinktree.py	none	Inverts the tree. This was put in early on as a result of failure to heed the 'YAGNI' principle (you ain't gonna need it), and has never been tested to my knowledge. Caveat factor!
Save_page_as_property_value	relatedurl.py, url.py		If present in the UrlTree-derived network instance's property list and not set to False or zero, tells the Url-derived instances in the network to store the retrieved page in its property list under the name 'current_page'.
SEQueryFileName	regexqueryurl.py, searchenginetree.py	regexqueryurl1.py	If present, file path to which a copy of the search engine result urls is to be written. Useful for debugging and other purposes.
Sleeptime	ncbi*.py, url.py, urlutils.py, urltree.py, relatedurl.py, technoraticosmosurl.py		How many seconds to sleep between HTTP GET operations.
technoratiKey	technoratitree.py, technoraticosmosurl.py		Your Technorati API key, for use in using APIs from Technorati to spider blog cross-reference networks.
Timestamp	searchenginetree.py, urltree.py, urlutils.py		Start time of the network-building tree class construction; useful for constructing unique filenames
title	url.py, urlutils.py	internal	Records the title of the page in the NetItem-derived class.
TLDExceptions	urltree.py, collector-utils.py	BingTLD.py, GoogleTLD.py, YahooTLD.py	Identifies exceptional cases in tracking of top-level domain properties—for-profit entities using the dot-org TLD, for example.
TrackTLDProperties	urltree.py, collector-utils.py	BingTLD.py, GoogleTLD.py, YahooTLD.py, BreakdownByTLD.py	Trigger use of top-level domain tracking properties. Unless this is set to True, the other TLD properties have no effect.
typeSpecificQueryFormatter	searchenginetree.py		In SearchEngineTree-derived classes, this property can be set to the name of function to be used to format the search engine's query URL.
urlTLD	urltree.py	internal	Numeric representation of a node's TLD type for use in building Pajek partitions.
urlTLDVector	urltree.py	internal	For a UrlNetItem instance (aka network node), stores

Property	Library files where used	Examples	What it does (and other notes on its use)
			the author-defined value representing the value to ain information seeker of a particular TLD type.
User-agent	relatedurl.py, url.py, urlutils.py, urltree.py		user-agent string to use in HTTP GET operations.
WriteEFetchRawOutput	ncbiurl.py	ncbilinkstree1.py ncbilinkstree2.py ncbilinkstree3.py	Set this property to a file name/path to trigger writing the NCBI eFetch service's output to disk. If your program or function invokes this multiple times, be sure to change the file name between calls to the NCBI eLink service.
WriteELinkRawOutput	ncbiurl.py	ncbilinkstree1.py ncbilinkstree2.py ncbilinkstree3.py	Set this property to a file name/path to trigger writing the NCBI eLink service's output to disk. If your program or function invokes this multiple times, be sure to change the file name between calls to the NCBI eLink service.
WriteESearchRawOutput	ncbiurl.py	ncbilinkstree1.py ncbilinkstree2.py ncbilinkstree3.py	Set this property to a file name/path to trigger writing the NCBI eSearch service's output to disk. If your program or function invokes this multiple times, be sure to change the file name between calls to the NCBI eLink service.
WriteESummaryRawOutput	ncbiurl.py	ncbilinkstree1.py ncbilinkstree2.py ncbilinkstree3.py	Set this property to a file name/path to trigger writing the NCBI eSummary service's output to disk. If your program or function invokes this multiple times, be sure to change the file name between calls to the NCBI eLink service.
LogEFetchRawOutput	ncbiurl.py	ncbilinkstree1.py ncbilinkstree2.py ncbilinkstree3.py	Set this property to a file name/path to trigger writing the NCBI eFetch service's output to the log file (but only if logging is turned on). If your program or function invokes this multiple times, be sure to change the file name between calls to the NCBI eLink service.
LogELinkRawOutput	ncbiurl.py	ncbilinkstree1.py ncbilinkstree2.py ncbilinkstree3.py	Set this property to a file name/path to trigger writing the NCBI eLink service's output to the log file (but only if logging is turned on). If your program or function invokes this multiple times, be sure to change the file name between calls to the NCBI eLink service.
LogESearchRawOutput	ncbiurl.py	ncbilinkstree1.py ncbilinkstree2.py ncbilinkstree3.py	Set this property to a file name/path to trigger writing the NCBI eSearch service's output to the log file (but only if logging is turned on). If your program or function invokes this multiple times, be sure to change the file name between calls to the NCBI eLink service.
LogESummaryRawOutput	ncbiurl.py	ncbilinkstree1.py ncbilinkstree2.py ncbilinkstree3.py	Set this property to a file name/path to trigger writing the NCBI eSummary service's output to the log file (but only if logging is turned on). If your program or function invokes this multiple times, be sure to change the file name between calls to the NCBI eLink service.

8 ADVANCED CONTROLS AND OPERATIONS

8.1 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'. This property only affects output; the full URL is used when engaged in the spidering process.

8.2 USING HOST NAMES IN LIEU OF DOMAINS FOR VERTEX NAMES

Domain networks can be a handy high-level view of a URL network, but sometimes, for example when studying blogs, a domain network isn't quite good enough. For example, there are bazillions of blogs under domains such as typepad.com, blogspot.com, and wordpress.com. UrlNet has a feature that can help here—use of the full host name rather than the domain name when building the domain network. To take a typical blog URL as an example, in http://hunscher.typepad.com/, the host name is everything between the scheme prefix (in this case 'http://') and the next occurrence of a slash (following '.com' in this example)—'hunscher.typepad.com'. The domain name is 'typepad.com'.

Here's a domain network with two blog umbrella domains, typepad.com and blogspot.com (red nodes):

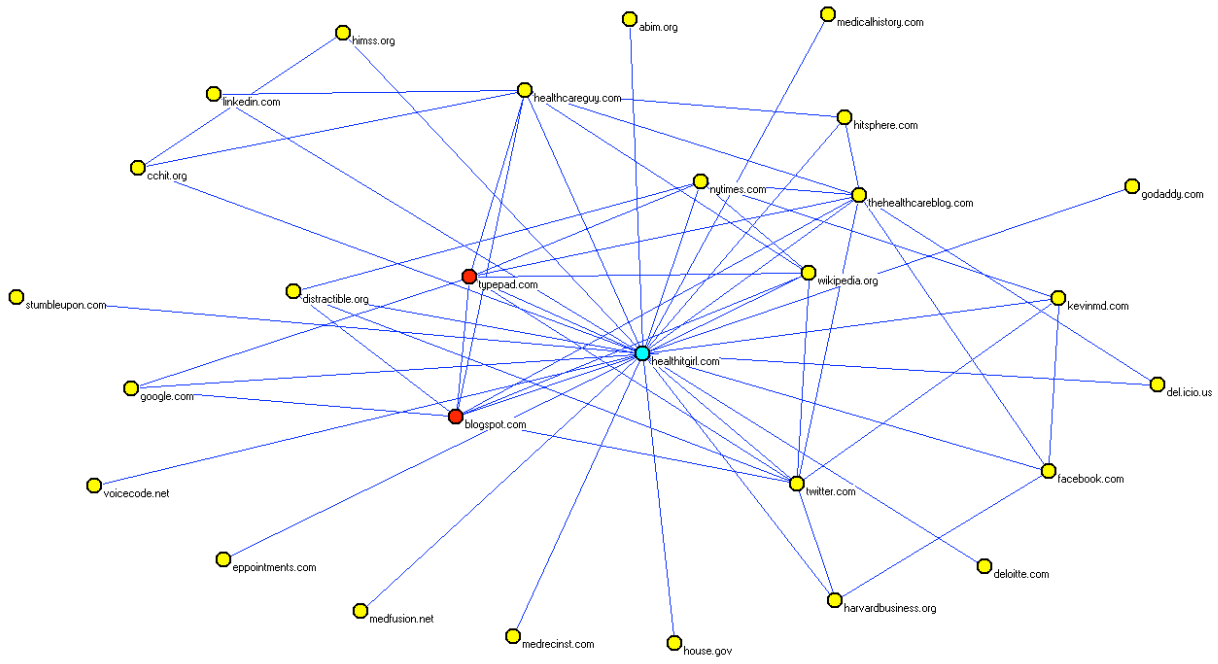


FIGURE 11 – EXAMPLE DOMAIN NETWORK (DEFAULT, VERTEX LABELS ARE DOMAIN NAMES)

To trigger use of host names rather than domain names when building a URL network, simply add _useHostNameForDomainName = True as an argument to the constructor of the UrlTree. The example program urltree4blogs.py, the relevant part of which is shown here, shows how:

```
# urltree4blogs.py
from urlnet.urltree import UrlTree
net = UrlTree(_useHostNameForDomainName = True)
net.BuildUrlTree('http://healthitgirl.com/')
net.WritePajekFile('urltree4blogs', 'urltree4blogs')
```

The only difference between the script that generated the above diagram and the script that generates the diagram shown below is the use of the _useHostNameForDomainName constructor argument. The resulting network uses host names instead of domain names. Now we can see the host names of the Typepad and BlogSpot blogs in this diagram (red nodes).

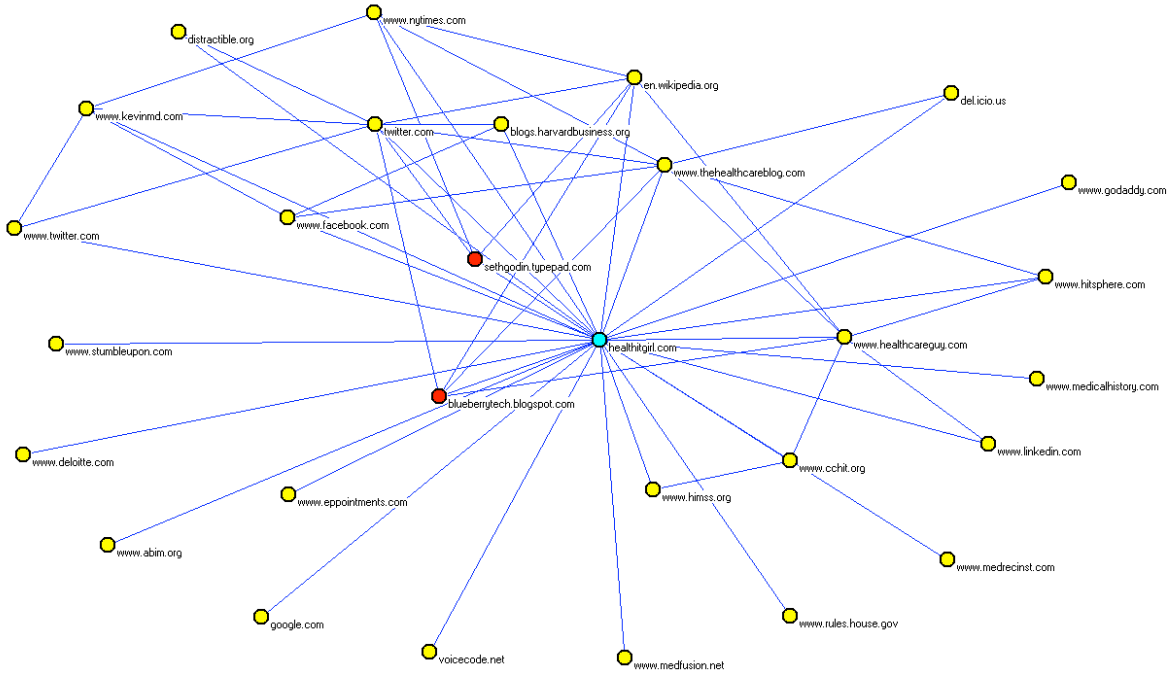


FIGURE 12 – SAME NETWORK, USING HOST NAMES AS DOMAIN NAMES

Forestry for blogs…

There's another example program, urlforest4blogs.py, that demonstrates the use of host names in a URL forest scenario.

8.3 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.

8.3.1 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.

8.3.2 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:

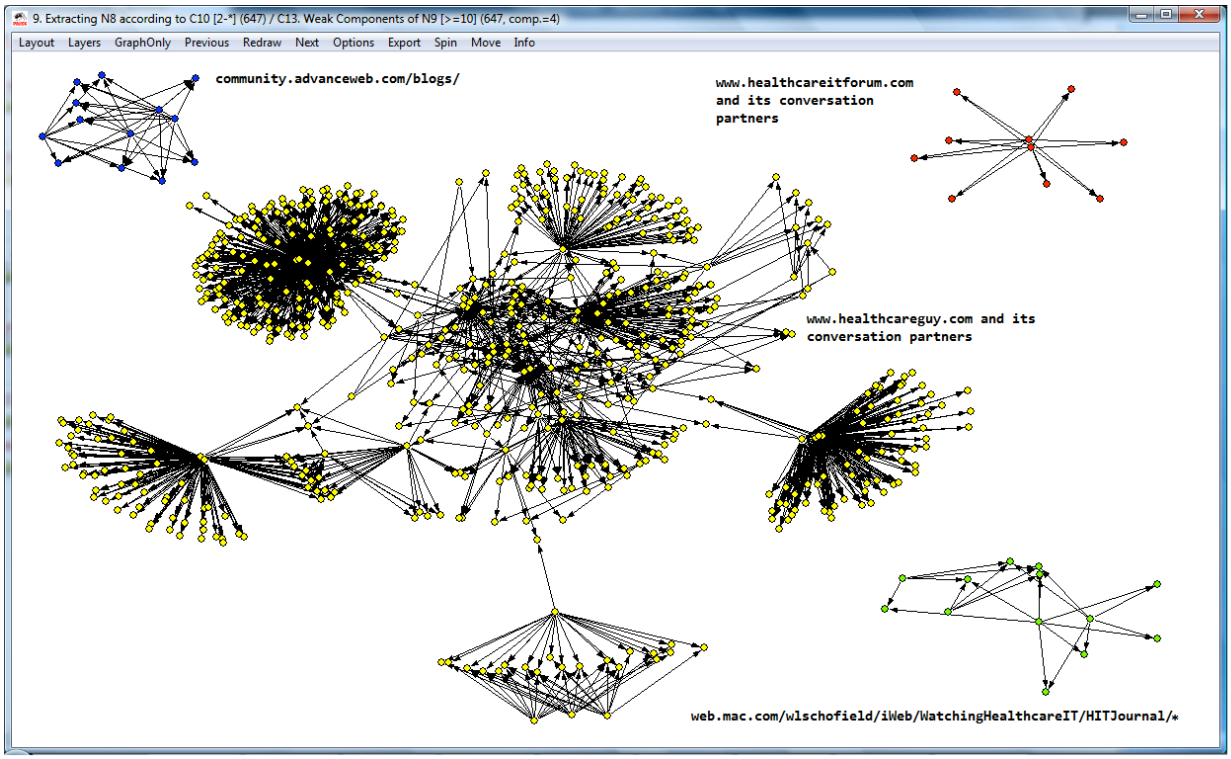


FIGURE 13 – A NETWORK GENERATED USING REDIRECTS

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.

8.4 Combining 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").

8.4.1 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 12, 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)
```

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 '.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. Properties are used to trigger the one-time-only behavior and preserve the functionality of the constructor arguments for use in the remainder of the program execution.

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 12, 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.

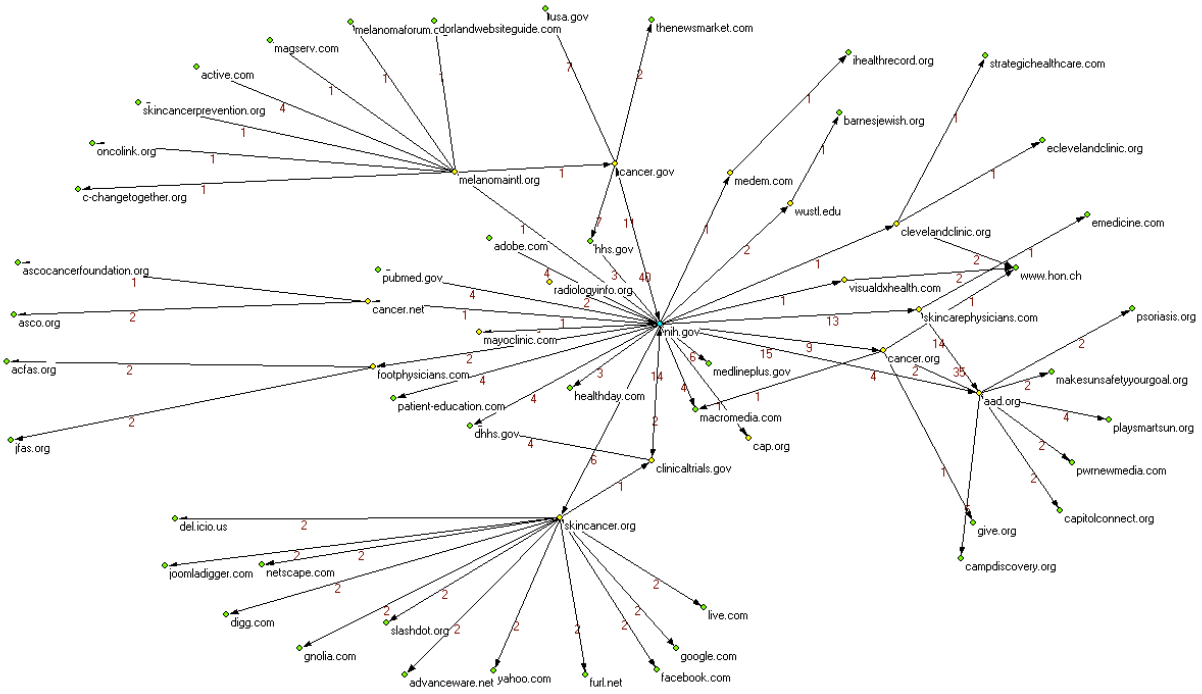


FIGURE 14 – REGEXQUERYURL IN ACTION: A SEARCH ENGINE RESULT SET DOMAIN NETWORK

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.

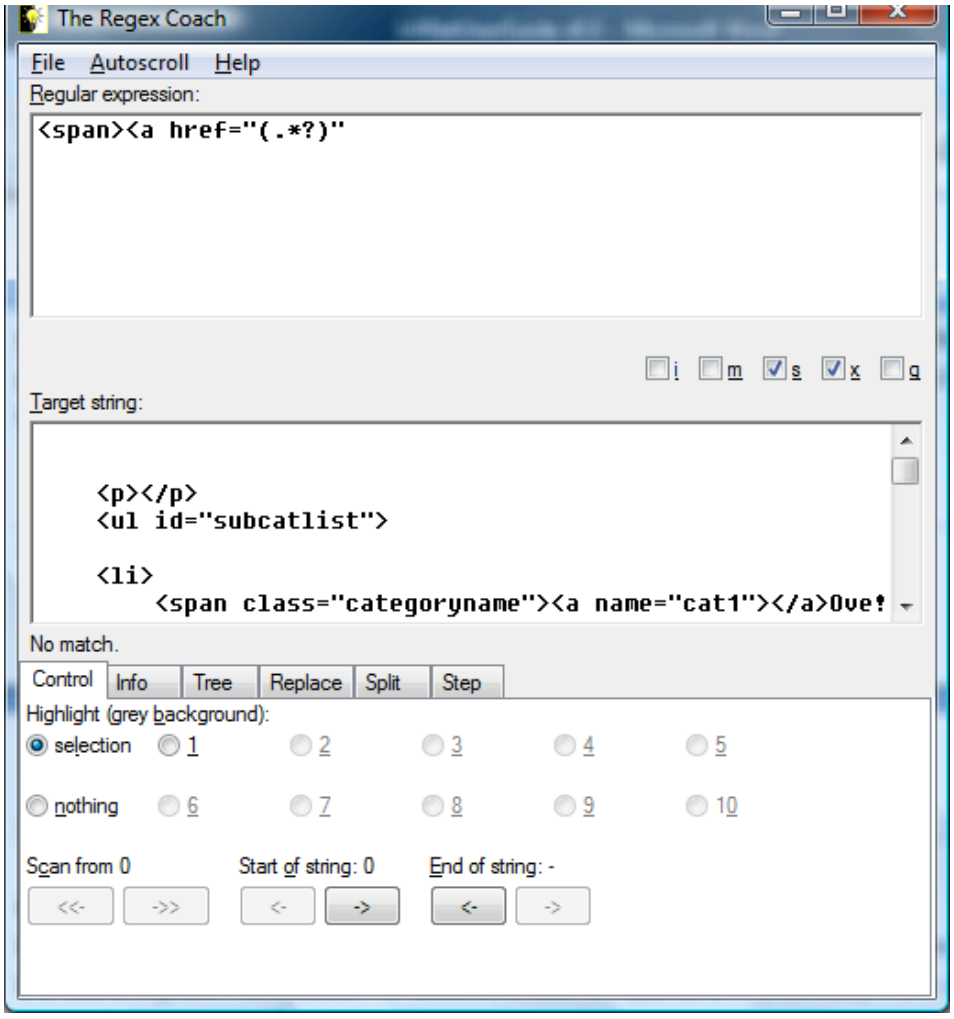


FIGURE 15 – THE REGEX COACH

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 DYNAMIC APPLICATION OF INCLUSION AND EXCLUSION CRITERIA

UrlNet users will often want to filter their networks as early as possible in the spidering process, because on many pages the outlinks lead to pages irrelevant to the task at hand. In this chapter, we'll see a couple of ways to accomplish this. The first way applies simplistic criteria based on the URL itself, e.g., to filter out specific sites or pages. This method involves no coding at all on your part, just adding strings to a list.

The second method is more sophisticated, and offers a lot more flexibility. We provide a default data-driven relevance checker that again requires no custom coding on your part, just maintaining one or two lists and setting some properties. We'll also see how you can build your own content-based relevance-checking function, which can be as simple or as complicated as you wish.

10.1 IGNORING SPECIFIC URLS AND URL TYPES (OR THEIR CHILDREN)

We can easily exclude URLs from the network, or allow the URL in but exclude its outlinks by truncating search at the URL, using one of the methods described in this section. All we need to do is tell UrlNet's tree-building algorithm what to look for when deciding whether to ignore or truncate the URL.

10.1.1 IGNORANCE CAN BE BLISS

We can provide the UrlTree constructor with a sequence or list of text strings to look for in child URLs, and when it finds a match (simple text search or optionally regular-expression)[2] on any one of these text strings, it will exclude the URL and its entire subtree from the generated network. We do this by passing the list of ignorable text strings to the UrlTree constructor. Here's a brief example, using a modification of the code we examined back 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 we saw in Figure 2, minus three URLs:

[2] Don't be frightened by the term "regular-expression match". If you don't want to deal with regular expressions, you don't have to—it's a feature you have to turn on if you want to use it, as we will see shortly. The advantage to regular expressions for the faint of heart is that you can make the text-match process case-insensitive if you so desire, or force a Unicode comparison, by setting the flags for the re.search function as we'll see below.

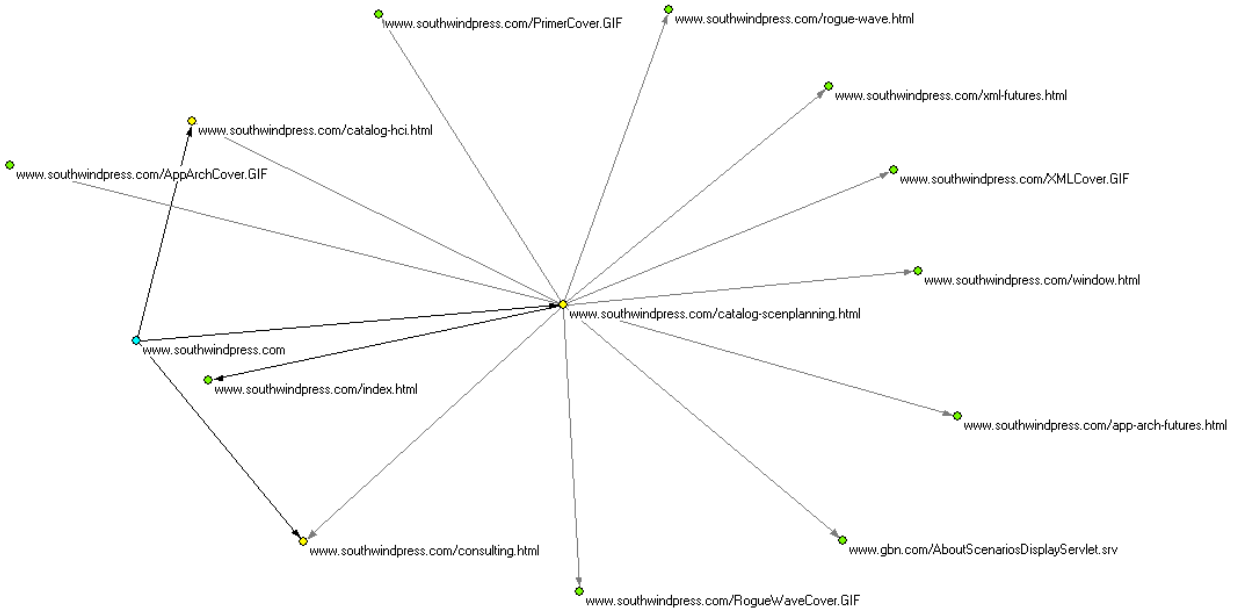


FIGURE 16 – IGNORING SOME URLS

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 (or nodes in the case of a forest) 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.

A default list of ignorables is found in the textToIgnore list in the urlnet.ignoreandtruncate.py module. It contains my personal list of pesky URLs that keep popping up in my consumer health information search networks—URLs for ad banner generators, web analytics, the Acrobat Reader download, and many more. If you want to use it, you must import it:

```
# import the default ignorables list
from urlnet.ignoreandtruncate import textToIgnore
```

You can add your own custom text patterns using a Python statement like this:

```
textToIgnore.append('microsoft.com') # ignore the Evil Empire
```

You can also remove items from the default textToIgnore list:

```
textToIgnore.remove('yahoo.com') # don't ignore Yahoo
```

You trigger use of this list either by passing it as the _ignorableText argument to the constructor of a UrlTree-derived class, or by passing the list as an argument to the UrlTree.SetIgnorableText() function.

```
# either this:
net = UrlTree( _ignorableText = textToIgnore )

# or this:
net.SetIgnorableText(textToIgnore)
```

10.1.1.1 Turning on pattern-matching using regular expressions

Regular-expression use is off by default, because it turns certain common characters into regex commands unless they are escaped by prefixing with a backslash (\). When installing your list of ignorables, you must use the second method shown above, the call to set.SetIgnorableText(), if you want to turn on regular expression matching. To turn it on, pass as the second argument (searchArgs) either a zero, or one of the constants defined in the Python standard re module (for example, re.I or re.IGNORECASE to make the pattern-matching case-insensitive).

```
# at the top of your program, where you do your imports, import the re module
# you need to do this only if you will be using one of the defined flags, like
# re.IGNORECASE

import re

# use plain-vanilla regex pattern matching
net.SetIgnorableText(textToIgnore, 0)

# make the search case-insensitive

net.SetIgnorableText(textToIgnore, re.IGNORECASE)  # or just use re.I

# use Unicode pattern-matching

net.SetIgnorableText(textToIgnore, re.UNICODE)  # or just use re.U
```

10.1.2 Truncating outlink spidering

It's also possible to truncate outlink spidering at URLs in much the same manner as ignoring them. With truncation, the URL that matches the pattern is included in the network, but its outlinks are not followed even if the URL is at a level that is not at the maximum depth for the tree or forest. Why would you want to do this? The URLs I want to truncate are of interest themselves but are rife with irrelevant outlinks, such as YouTube video URLs and Amazon book listing pages.

Truncation operates pretty much the same as ignorable text. Once again, UrlNet provides a default list, which you can use as a starting point if you don't want to create your own from scratch:

```
# import the default truncatables list
from urlnet.ignoreandtruncate import textToTruncate
```

You can add your own custom text patterns using a Python statement like this:

```
textToTruncate.append('bn.com') # truncate the Barnes & Noble booksite
```

You can also remove items from the default textToTruncate list:

```
textToTruncate.remove('youtube.com') # don't truncate YouTube URLs
```

You trigger use of this list either by passing it as the _ignorableText argument to the constructor of a UrlTree-derived class, or by passing the list as an argument to the UrlTree.SetIgnorableText() function.

```
# either this:
net = UrlTree( _truncatableText = textToTruncate )

# or this:
net.SetTruncatableText(textToTruncate)
```

Caveat

Be sure to make your additions and deletions from the list ***before*** calling net.SetIgnorableText() or net.SetTruncatableText(), or passing either list as a constructor argument.

10.1.2.1 Turning on pattern-matching using regular expressions

In truncation as in use of ignorables, regular-expression use is off by default. When installing your list of truncatables, you must use the second method shown above, the call to set.SetTruncatableText(), and pass it a second argument (searchArgs) if you want to turn on regular expression matching. To turn it on, pass as the second argument searchArgs either a zero, or one of the constants defined in the Python standard re module (for example, re.I or re.IGNORECASE to make the pattern-matching case-insensitive).

```
# at the top of your program, where you do your imports, import the re module
# you need to do this only if you will be using one of the defined flags, like
# re.IGNORECASE

import re

# use plain-vanilla regex pattern matching

net.SetTruncatableText(textToIgnore, 0)

# make the search case-insensitive

net.SetTruncatableText(textToIgnore, re.IGNORECASE)  # or just use re.I

# use Unicode pattern-matching

net.SetTruncatableText(textToIgnore, re.UNICODE)  # or just use re.U
```

10.1.3 INGORE/TRUNCATION EXAMPLE

To see these features in action, take a look at the example program ignorabletext1.py, provided for hands-on experimentation.

10.2 APPLYING INCLUSION/EXCLUSION CRITERIA BASED ON PAGE CONTENT CHECKER

In the previous section, we looked at how we can ignore URLs based on text patterns in the URL itself. That's handy when you know about particular URLs that have no bearing on your work yet appear frequently, such as the ubiquitous Get Acrobat Reader link. UrlNet even provides a list of such URLs you can use as a basis for your personalized list of ignorables.

Oftentimes, though, we don't know whether we want to include a page or leave the page out of the network until we look at its content. We may want to include only pages that match a particular text pattern (or one of a set of such patterns); or we may want to exclude pages that match a particular text pattern (or one of a set of such patterns); or possibly, we may want to apply both inclusion and exclusion criteria. UrlNet provides a way to do exactly this out-of-the-box using a fairly simple algorithm in the function urlnet.urlutils.CheckInclusionExclusionCriteria(), but can be readily extended to employ more sophisticated relevance-determination techniques. We'll refer to relevance-determination routines as *inclusion/exclusion checkers*.

Inclusion/exclusion criteria checking is just one reason you may want to analyze page content. We'll see another example of page content analysis mentioned in two places: section 10.5, Another use for a custom page content checker, in this chapter on page 68; and in more detail in section 11.3, Setting custom properties based on page content, found on page 74.

UrlNet provides a multi-purpose *callback mechanism* for checking page content. If you install a page content checker callback function, it is invoked early in processing a URL for inclusion in the network, before any data structure is allocated for the URL and domain items. It is called a callback mechanism because the function you provide is called from within a library function you have invoked: you call the library function, and the function calls back to your application code to execute the function you provide. Your custom function focuses almost entirely on content analysis of one page at a time; the tree-building algorithm takes care of the business of spidering the Web for you, and the results of your analysis can emerge at the end in the form of Pajek partitions and vectors or as enhancements to GUESS networks.

You install a page checker callback function using the following code:

```
# use the UrlNet-provided simple inclusion/exclusion checker
net.SetPageContentCheckerFn(MyVeryOwnCheck4InclusionExclusionCriteria)
```

If you do not install your own, the default is urlutils.CheckInclusionExclusionCriteria.

Return Address

You'll notice that MyVeryOwnCheck4InclusionExclusionCriteria is not followed by its own set of parentheses enclosing its arguments, which would be the case if the function were being invoked. Instead, the above example installs the callback function by passing the *address* of the MyVeryOwnCheck4InclusionExclusionCriteria function to the UrlTree class's member function SetPageContentCheckerFn.

10.3 A SAMPLE PROGRAM

Here's the code for a program demonstrating the default inclusion/exclusion checker algorithm's features:

```
from urlnet.urltree import UrlTree
from urlnet.urlutils import CheckInclusionExclusionCriteria
import re # for search flags

# in case we need to be polite, we could set this to 1 or 2 (for 1 or 2 seconds'
# sleep between HTTP GETs.
SLEEPTIME = 0

# go 2 deep
MAXLEVEL = 2

# first, do a simple tree without any inclusion/exclusion checking
net = UrlTree(_maxLevel=MAXLEVEL,_sleeptime=SLEEPTIME)
net.BuildUrlTree('http://www.southwindpress.com')

# write URL network file: we can compare to the next set to see what's different
net.WritePajekNetworkFile('urltreeinclexcl1', 'urltreeinclexcl1', \
                            rlNet = True)


# Now test inclusion/exclusion patterns

# The include_patternlist and exclude_patternlist property values,
# if present, must evaluate to a list or sequence. Items in the
# list are 'regular expressions' – the simplest form of which is
# a text string. Regular expressions support wildcards and many other
# subtle search tweaks. Search Google for "python regular expressions"
# to learn more about them.

# re-create the UrlTree instance.
Net = UrlTree(_maxLevel=MAXLEVEL,_sleeptime=SLEEPTIME)

# We will use the UrlNet-provided default inclusion/exclusion checker

# Now we do the properties that establish the rules for checking inclusion
# and exclusion.

# First, we set the level at which checking is to start. In actual fact,
# if this call were omitted, the level would be set to its default of 1 anyway.
# This SetProperty call is included to show you how to set it.

Net.SetProperty('includeexclude_level', 1)

# For readability's sake, I put each item in the lists on its own
# line. Python doesn't care about the extra newlines and spaces,
# as long as the indentation is consistent.
```

```
Incl_patternlist = [
    'catalog',
    ]

net.SetProperty('include_patternlist',incl_patternlist)

# pass in the re module flag that says to ignore case, re.IGNORECASE;
# re.I would also work. 'catalog' in the pattern list should match 'CATALOG'
# in the HTML text. This way, it will match 'catalog' or 'Catalog'
# or any similar mixed-case form.

Net.SetProperty('include_patternlist_flags',re.IGNORECASE)

# try to exclude at least one vertex.

Excl_patternlist = [
    'discovering tacit assumptions',
    ]

net.SetProperty('exclude_patternlist',excl_patternlist)

net.SetProperty('exclude_patternlist_flags',re.IGNORECASE)

# Now we build the tree-network again with include/exclude checking in place.
# We should get a lot fewer nodes.
Net.BuildUrlTree('http://www.southwindpress.com')

# write URL network file
net.WritePajekNetworkFile('urltreeinclexcl2', 'urltreeinclexcl2', \
                              rlNet = True)
```

10.3.1 STEPS TO CREATE A PROGRAM OF YOUR OWN

You can always modify the code in the example above to suit (includeexclude.py). If you want to build from scratch, here's the step-by-step guide.

1. Import the inclusion/exclusion page checker function from the urlutils module (or write your own).
2. Create a UrlTree instance (or derived class instance, such as GoogleTree, etc.).
3. Set the properties (see next section).
4. Install your page checker function (if you don't want to use the default function).
5. Build the network using BuildUrlTree or one of the other network-building functions.
6. Generate output using one of the standard output functions (e.g., WritePajekFile).

That's all there is to it.

10.3.2 PROPERTIES TO BE SET

The following table shows the properties that need to be set when employing UrlNet's default inclusion/exclusion checker algorithm, which is one instance of a page content checker function. These properties are specific to that function, but for consistency's sake, your custom page content checker function should test for and respect the includeexclude_level property as shown in the sample program.

TABLE 4 – PROPERTIES USED BY THE DEFAULT RELEVANCE-CHECKER FUNCTION

Property	Description
includeexclude_level	The level in the tree to begin inclusion/exclusion criteria processing. Defaults to 1 if not set.
Include_patternlist	List of regular-expression patterns, at least one of which must be found in the URL's page text for the URL to be *included* in the network.
Include_patternlist_flags	Flags to modify the algorithm of re.search when processing the *include* pattern list. For example, re.IGNORECASE can be used to tell re.search to ignore case.
Exclude_patternlist	List of regular-expression patterns; if at least one of these is found in the URL's page text, the URL is to be *excluded* in the network.
Exclude_patternlist_flags	Flags to modify the algorithm of re.search when processing the *exclude* pattern list. For example, re.IGNORECASE can be used to tell re.search to ignore case.

10.4 UNDER THE HOOD: THE OUT-OF-THE-BOX INCLUSION/EXCLUSION CHECKER

Here's the library's own routine for checking relevance. It returns the Boolean value True if the page is relevant (or didn't need to be checked, or couldn't be retrieved). It returns False if the page was retrieved and checked, and then found to be irrelevant.

If you aren't interested in building your own checker or understanding exactly how the default algorithm works, you can skip it altogether.

```
Def CheckInclusionExclusionCriteria(network,theUrl,level):

    NO_INCLEXCL = 42 # This is just a magic number for use in
                     # the context of this function only

    log = Log('CheckInclusionExclusionCriteria','url=%s' % (str(theUrl)))

    try:
        # if this routine were not set up as the inclusion/exclusion
        # checker we would never get here. This check is a way to
        # exit quickly if the user didn't provide any criteria.
        # UrlTree's constructor sets this member variable to False by default.
        If network.check4InclusionExclusionCriteria == NO_INCLEXCL:
            return True

        if network.check4InclusionExclusionCriteria == True:
            network.check4InclusionExclusionCriteria = False
```

We are using a member variable on the UrlTree instance to keep track of whether we have initialized our inclusion/exclusion algorithm by getting the necessary properties, and whether that check resulted in successful initialization. Network.check4InclusionExclusionCriteria can take one of three values:

True	We need to check for the necessary data for our inclusion/exclusion algorithm
False	We have checked and the initialization was successful.
NO_INCLEXCL	This constant is defined local to this function, and is simply a flag to say we did check for the necessary data for our inclusion/exclusion algorithm, and the check failed, so don't try checking again. This avoids pointless repetition of calls to network.GetProperty().

```
            Network.includeexclude_level = network.GetProperty('includeexclude_level')


            if network.includeexclude_level == None:
                network.includeexclude_level = 1
```

All the properties we use in this algorithm are specific to the algorithm and you need not use them if you write your own, but includeexclude_level is a good one to include in your custom checker, for the following reasons. In tree searches, you usually don't want to exclude the root, so the starting level would ordinarily be 1. If using a placeholder root and you want to include the whole forest, set the level to 2. If processing a forest as the top level, you may want to set the level to zero to ensure that any irrelevant URLs in the top-level forest are weeded out.

```
            Network.include_patternlist = network.GetProperty('include_patternlist')
            network.include_patternlist_flags = \
                network.GetProperty('include_patternlist_flags')
            network.exclude_patternlist = network.GetProperty('exclude_patternlist')
            network.exclude_patternlist_flags = \
                network.GetProperty('exclude_patternlist_flags')

            if network.include_patternlist == None \
                        and network.exclude_patternlist == None:
                network.check4InclusionExclusionCriteria = NO_INCLEXCL
                return True # no regex searches needed
```

In the above section, we got all the property values we need for our algorithm. Include_patternlist contains a list of text strings (as does exclude_patternlist). Each of these is treated as a 'regular expression' by the algorithm, using the search function in Python's standard module called *re*.

If we weren't able to get the necessary property values, we don't want to check for them again. In fact, we really want to turn off inclusion/exclusion checking altogether for the reasons cited above. We set network.check4InclusionExclusionCriteria to NO_INCLEXCL, and return True to tell the caller to continue processing this URL. In the first two lines below, we also return True if we haven't hit the level where checking is to be done.

```
        If level < network.includeexclude_level:
            return True # no regex searches needed
```

Next, in the code below, we use the standard routine in urlutils.py to get the page. This ensures that all necessary housekeeping is done. The checker routine is called early in the processing of a URL, so we will save the page for later to avoid doing a second GET, and if there is an HTTP error we will also save that to warn off attempts to retrieve the page later. The GetHttpPage function resides in urlutils.py, as does this checker function. When you invoke GetHttpPage from your own code module, you will need to import GetHttpPage from urlutils, or import urlutils and then invoke the function as urlutils.GetHttpPage.

```
        page = GetHttpPage(network,theUrl)
        if not page:
            return True # press on regardless


        # ignore anything that is not html, xhtml, or xml
        if not re.search('<html',page,re.IGNORECASE):
            if not re.search('<xhtml',page,re.IGNORECASE):
                if not re.search('<xml',page,re.IGNORECASE):
                    return True
```

Now we parse the page to get textual content. The DumbWriter instance will try to put in line breaks, and we don't want that to complicate our search, so we tell it to make the lines REALLY long.

```
        strIO = StringIO.StringIO() # to hold the page text
        parser = HTMLParser(AbstractFormatter(DumbWriter(file=strIO,maxcol=32767)))
        parser.feed(page)
        data = strIO.getvalue()
```

Now we get to the heart of the algorithm. We cycle through the inclusion patternlist to see if page matches at least one pattern the list. Network.include_patternlist_flags is used in the re.search call if it is not None. It is likely to be set to re.I (capital letter 'i') if the caller wants a case-insensitive search. If we get a match we stop looping: only one match is required in order to include the page (i.e. to consider it relevant). If we cycle through the list and don't have any pattern match, we return False to tell the caller the page is not relevant.

```
        Found = None
        if network.include_patternlist:
            for inclusion_pattern in network.include_patternlist:
                if network.include_patternlist_flags:
                    found = \
                       re.search(inclusion_pattern,data,
                              network.include_patternlist_flags)
                else:
                    found = re.search(inclusion_pattern,data)
                # all we need is one item to trigger inclusion;
                # if we found one we can quit looping
```

```
                if found != None:
                    break
            # if we didn't find at least one of the inclusion patterns, scrap the page
            if found == None:
                return False
```

If we get here, *EITHER* there was no include_patternlist, *OR* at least one of the inclusion patterns was found in the text. Now we check for exclusion criteria and cycle through them if they are present, doing as we did for inclusion patterns, with one variation: if any of the exclusion patterns are found, we return False to indicate that the page is to be excluded from the network.

```
        If network.exclude_patternlist:
            for exclusion_pattern in network.exclude_patternlist:
                if network.exclude_patternlist_flags:
                    found =
re.search(exclusion_pattern,data,network.exclude_patternlist_flags)
                else:
                    found = re.search(exclusion_pattern,data)
                # we only need one match on an exclusion pattern to trigger
                # scrapping the page.
                If found != None:
                    return False
```

If we get here, *EITHER* there was no include_patternlist *OR* there was a match on at least one of the inclusion patterns; *AND EITHER* there was no exclude list, *OR* none of the patterns in the exclusion list were found in the page. Return True. Following that is our exception handling code. If an exception occurs, we return False.

```
        Return True

    except Exception, inst:
        theError = 'CheckInclusionExclusionCriteria: ' \
                  + str(type(inst)) + '\n' + str(inst) + '\non URL ' + theUrl
        network.SetLastError ( theError )
        log.Write(theError)
        #print theError
        return False
```

That's all there is to it.

10.5 Another use for a custom page content checker

There's another reason why you might want to create a custom routine that can be installed as the page content checker, even if you don't plan to filter out any pages (or in addition to filtering). You can use your custom routine to analyze the page content, for example to determine the number of occurrences in the page text of terms of interest to you. You can then set properties that are used to create Pajek

partitions or vectors, or to add additional attributes to GUESS network nodes (vertices), as described in chapter 11, Setting custom properties , found below starting on page 70.

It's a bit tricky, because the UrlNet internal data structures that encapsulate the URL and domain entities do not exist at the time the page content checker is invoked. Your custom routine needs to set properties on the UrlTree-derived network instance that will provide the UrlNet tree-building algorithm with the set of name-value pairs representing the properties you want to set on the URL or domain node. This technique is described in section 11.3, Setting custom properties based on page content, on page 74 below.

11 SETTING CUSTOM PROPERTIES TO CREATE ADDITIONAL ANALYTICAL DATA

Using UrlNet's simplest methods, you get a Pajek partition (or GUESS node attribute) for a URL or domain's level in the tree, with zero being the root level. It is often desirable to create your own partitions or to add attributes to GUESS network nodes, based on analysis of the URL or the page content.

In past releases of the library, this was typically accomplished by creating a Url subclass, overriding the default argument on the UrlTree constructor to tell the network building algorithm to use the new Url subclass. For users unfamiliar with object-oriented concepts and/or Python programming, this was a daunting prospect. [3]

There are now some easier ways for you to generate Pajek partitions and vectors of your own, or to add additional attributes to GUESS vertices. You can do so by setting properties on URL and domain items based on characteristics of the URL, or based on inspection of the page content.

11.1 DATA STRUCTURES USED IN CUSTOM PROPERTY-SETTING ACTIVITIES

Specific data structures are used in custom property-setting, whether based on URL or page content inspection. The data structures are stored in two properties on the UrlTree-derived network class instance, properties that are used by the tree-building algorithm to control custom property-setting: additionalUrlAttrs and additionalDomainAttrs.

If present, the additionalUrlAttrs property value should be a dictionary containing two to six values:

TABLE 5 – DICTIONARY STRUCTURE USED IN PROCESSING CUSTOM PROPERTIES

Dictionary entry name	Description
attrName	The name of a property on each UrlNetItem and/or DomainNetItem instance in the network. Required for both Pajek and GUESS networks.
PorVName	Name of Pajek partition or vector. Required for Pajek networks.
doPartition	True for partition, False for vector. Required for Pajek networks.
Default	Default for use in partition or vector if the item does not have a value set for the target property. Optional, but if present, used for both Pajek and GUESS networks.
Dict	For Pajek partitions only, dictionary for use as described in section 5.3.2, Method #2: Look that up in your Funk & Wagnall's. Ignored for vectors.
Datatype	The Java data type to be used for the attribute on vertices in the GUESS network. Required for GUESS networks.

[3] I know what you are thinking. I appear to be blind to the possibility that my code is overly arcane and/or complex. The truth is that I just don't want to admit it, except in a footnote that nobody will read. Well, *almost* nobody…

To emphasize the required entries, Pajek partition and vector generation requires attrName, PorVName, and doPartition. GUESS networks require attrName and datatype. GUESS and Pajek output routines will use default if present, and dict is optional for Pajek partitions.

11.2 Setting custom properties based on URL inspection

11.2.1 Writing and installing custom property-setter function

The data structure described above is used to auto-generate Pajek partitions and vectors, and to auto-add attributes to GUESS vertex definitions. The easiest way to set custom property values is by using a custom URL properties function. Custom property-setter functions for URLs and/or domains installed on the UrlTree or UrlTree-derived instance before building the network are automatically invoked after the URL item (or domain item) has been added to the network data structure.

11.2.2 Custom properties based on URL inspection: A complete example

Here

```
#!/usr/bin/env python
# $Id$
'''
Demonstrate the use of custom property setter.
'''
from urlnet.urltree import UrlTree
from urlnet.log import Log, logging

import math

def MyCustomPropertySetter(net,item,urlToAdd):
    log = Log('MyCustomPropertySetter',urlToAdd)
    logging = True
    '''
    Set some properties for demonstration purposes
    '''

    item.SetProperty('length', len(urlToAdd) )        # integer property
    item.SetProperty('sqrt',math.sqrt(len(urlToAdd))) # float property

def main():
    try:
        net = UrlTree(_maxLevel=2)

        # set custom URL property setter.
        Net.SetCustomUrlPropertiesFn(MyCustomPropertySetter)

        # The same custom properties setter works for domains; this
        # might not be the case for other applications.

        Net.SetCustomDomainPropertiesFn(MyCustomPropertySetter)

        # These are meaningless properties used for demonstration purposes only:
        # length of the urlToAdd argument and square root of the length of the
        # urlToAdd argument.

        PropertyDictList4Urls = \
            [
                {
```

```
                    'attrName'    : 'length',
                    'PorVName'    : 'URL Length',
                    'doPartition' : True,
                    'default'     : 9999998,
                    'datatype'    : 'INT',
                },
                {
                    'attrName'    : 'sqrt',
                    'PorVName'    : 'Square root of length of URL',
                    'doPartition' : False,
                    'default'     : 0.00001,
                    'datatype'    : 'DOUBLE',
                },
            ]

        net.SetProperty('additionalUrlAttrs',PropertyDictList4Urls)

        # this list is identical to the one above, but is duped
        # so you can use this example as a template for your real
        # work, in which URLs and domains might be handled differently.

        PropertyDictList4Domains = \
            [
                {
                    'attrName'    : 'length',
                    'PorVName'    : 'URL Length',
                    'doPartition' : True,
                    'default'     : 9999998,
                    'datatype'    : 'INT',
                },
                {
                    'attrName'    : 'sqrt',
                    'PorVName'    : 'Square root of length of URL',
                    'doPartition' : False,
                    'default'     : 0.00001,
                    'datatype'    : 'DOUBLE',
                },
            ]

        net.SetProperty('additionalDomainAttrs',PropertyDictList4Domains)


        net.BuildUrlTree('http://www.southwindpress.com/')

        # the Pajek project will contain an additional partition and vector,
        # and the GUESS networks will contain an additional two attributes
        # per vertex
        net.WritePajekFile('customproperties1','customproperties1')
        net.WriteGuessFile('customproperties1Urls', doUrlNetwork = True)
        net.WriteGuessFile('customproperties1Domains', doUrlNetwork = False)

    except Exception,e:
        print str(e)



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

11.2.3 A BONUS EXAMPLE: FLAGGING SPECIFIC URLS IN A NETWORK

Sometimes, as you generate the same network repeatedly while engaged in analytical activities, it may be desirable to flag a set of URLs when they appear in the network, for example so you can spot them quickly when visualizing a large network. The example program flaggedpartition.py shows an easy way to do this using a custom property-setter function. It's less than a hundred lines of code, even with a liberal dose of white space and comments, and should be easy to modify to suit your needs.

11.2.4 EFFECTS ON OUTPUT NETWORK FILES

Once the property-setter function is installed and the data structure properties are initialized, execution happens automatically. The output is different depending on what output functions are called. The example produces the following in a Pajek project (abbreviated here):

```
*Network urls_customproperties1_directed
*Vertices 81
       1 "www.southwindpress.com"
       2 "www.southwindpress.com/catalog-scenplanning.html"
       3 "www.southwindpress.com/index.html"
       4 "www.southwindpress.com/catalog-hci.html"
       5 "www.southwindpress.com/consulting.html"
. . .

*Partition URL Length
*Vertices 81
      29
      55
      40
      46
      45
. . .

*Vector Square root of length of URL
*Vertices 81
5.38516480713
7.4161984871
6.32455532034
6.78232998313
6.7082039325
. . .
```

And in a GUESS network, this is the result:

```
nodedef>name VARCHAR, url VARCHAR,domain VARCHAR, theLevel INT, length INT, sqrt DOUBLE
southwindpress_com1,"www_southwindpress_com","southwindpress_com",0,29,5.3851
southwindpress_com2,"www_southwindpress_com_catalog_scenplanning_html","southwindpress_com",1,55,7.4161
southwindpress_com3,"www_southwindpress_com_index_html","southwindpress_com",2,40,6.3245
southwindpress_com4,"www_southwindpress_com_catalog_hci_html","southwindpress_com",1,46,6.7823
southwindpress_com5,"www_southwindpress_com_consulting_html","southwindpress_com",1,45,6.7082
. . .
```

Output generated by processing custom properties is bolded in the above sample output.

11.2.5 Bonus Example: Classifying nodes according to top-level domain

As part of my ongoing research into the quality of consumer health information on the Web, I developed a data collection program that uses the custom property-setting mechanisms described above. I developed a simplified version of this program and have included it in the examples as customproperties3.py. It is fairly well commented and very similar in structure to the example program shown above, but set in the context of a good bit more industrial-strength infrastructure.

11.3 Setting custom properties based on page content

Chapter 10, Dynamic application of inclusion and exclusion criteria, on page 58 above, describes a method for filtering pages based on inclusion/exclusion criteria. It uses a technique much like that described here to apply a custom algorithm during the tree-building process, only it is applied earlier, before the URL or domain item has been added to the network.

The custom page content checker approach provides you with an easy-to-grasp opportunity to analyze the page content and create network metadata that can be used in partitions and vectors and GUESS vertex attributes. This analysis can be done as part of or in lieu of determining whether the URL should be filtered out of the network. If you are just setting properties and not filtering pages, your custom page content checker should always return True so the URL and domain nodes are added to the network. After the URL or domain item is added to the network, the tree-building algorithm will look for a property whose value, if present, indicates the properties to be set on the item (URL or domain). These properties are UrlPagePropsToSet for URL items, and DomainPagePropsToSet for domain items.

Here is an example of the code you would include in your page content checker for setting URL properties:

```
urlPropertyDict = { 'wordcount' : 3427, 'keyword_occurrences' : 65, }
net.SetProperty('UrlPagePropsToSet', urlPropertyDict)
```

For domain properties, everything is the same except for the property name:

```
domainPropertyDict = { 'wordcount' : 3427, 'keyword_occurrences' : 65, }
net.SetProperty('DomainPagePropsToSet', urlPropertyDict)
```

There is no limit to the number of properties that can be set in this manner. Here's an example program that shows how it works (customproperties2.py):

```
# customproperties2.py
from urlnet.urltree import UrlTree
from urlnet.urlutils import GetHttpPage
from urlnet.log import Log

import re # for search flags
import math # for square root
```

First, we'll make a little utility function that creates dictionaries for us. The syntax of dictionaries is a bit tricky for me, so I'll put it in one place and get it right. We'll call this multiple times in our page checker function. The two dictionary key names, 'length' and 'sqrt', are important, because they must mesh with the 'attrName' values in the property definition dictionaries we'll encounter in the main function.

```
Def makeDictionary(length,sqrt):
    ‘’’
    This function provides a single method for creating a dictionary
    from the calculated property values. In our simple example, this
    works for both URL and domain item properties, but that would
    not always be the case; if they were different, there would be separate
    functions for URLs and domains.
    ‘’’
    return { ‘length’ : length, ‘sqrt’ : sqrt, }
```

Now we'll code our page checker callback function. It's always going to return True unless there's an exception, so the page will never be rejected by our function.

The function takes three arguments: the UrlTree-derived network instance, the URL we are processing, and the level on which we are operating. The first things we do are to initialize some default values for our early exits from the function; create an instance of the Log class in case we want to turn on logging and trace from the main program; and start an exception-handling wrapper.

```
Def SetPageBasedPropertyValues(network,theUrl,level):
    ‘’’
    This function shows the use of UrlNet’s inclusion/exclusion checker
    protocol to analyze page content and set properties based on the
    results of the analysis. Our analysis in this case is trivial, but
    it provides a framework in which you can put your own analytical
    features.
    ‘’’
    # default values for use in setting some crazy properties
    length = 0
    sqrt = 0.0

    # create a log in case the program decides to turn logging and trace on.
    Log = Log(‘SetPageBasedPropertyValues’,’url=%s’ % (str(theUrl)))

    try:
```

In chapter 10, Dynamic application of inclusion and exclusion criteria, starting on page 58, the default inclusion/exclusion level was set to 1. In this case, we want to include the root node in the "analysis" we are doing here, so we'll set the default to zero. If this page checker function were re-used in a program that generates a URL tree network with a placeholder root, we would want to set the includeexclude_level property to 1. This section of code gives us the option to set the starting level from outside. You should always include this level-checking code in your page checker functions, to give yourself the choice of level as you re-use the function.

```
Network.includeexclude_level = network.GetProperty('includeexclude_level')
if network.includeexclude_level == None:
    network.includeexclude_level = 0
    network.SetProperty('includeexclude_level',0)

if level < network.includeexclude_level:
    network.SetProperty('UrlPagePropsToSet', makeDictionary(length,sqrt) )
    network.SetProperty('DomainPagePropsToSet', makeDictionary(length,sqrt) )
    return True # no regex searches needed
```

We use the standard routine urlutils.GetHttpPage to get the page, ensuring that all necessary housekeeping is done. The checker routine is called early in the processing of a URL, so we will save the page for later to avoid doing a second GET, and if there is an HTTP error we will also save that to warn off attempts to retrieve the page later. If we don't get a page for some reason, we'll set the properties to the default values initialized above (fudged a bit for the domain properties, as you can see below)

```
page = GetHttpPage(network,theUrl)
if not page:
    network.SetProperty('UrlPagePropsToSet', makeDictionary(length,sqrt) )

    # fudge the domain values so they are a little different...
    network.SetProperty('DomainPagePropsToSet', \
                 makeDictionary(length*3,sqrt/2.0) )
    return True # press on regardless
```

We'll ignore any page that is not recognizably html, xhtml, or xml.

```
Head = page[:500]
#print str(head)
if not re.search('<html',head,re.IGNORECASE):
    if not re.search('<xhtml',head,re.IGNORECASE):
        if not re.search('<xml',head,re.IGNORECASE):
            network.SetProperty('UrlPagePropsToSet', \
                makeDictionary(length,sqrt) )
            network.SetProperty('DomainPagePropsToSet', \
                makeDictionary(length*3,sqrt/2.0) )
            return True
```

We'll set our faux property values to the length of the page and the square root of the length, for use in a Pajek partition and vector respectively (or in GUESS, an integer and a floating-point attribute on the vertex). We're fudging the values a bit for the domain properties, just to make them look different.

This is a good time to point out an important labor-saving fact. ***The only code in this function that is not boilerplate code are the definition of the makeDictionary help function (one line of code), calls to that function (four lines), and the code that calculates property values (in our trivial case, two lines of code).*** In other words, there are only seven lines of situation-specific code in the page-checker function, and about two dozen more in the main function where we define the property dictionaries UrlNet uses to auto-process your custom properties (and those two dozen lines are boilerplate in structure if not in

content). All in all, about 30 lines of code here will buy us two Pajek partitions and two vectors, and/or two new attributes on the vertices of both URL and domain networks in GUESS.

Of course your analytical work may require many more lines of code than my two lines shown below, but that's where the vast bulk of your effort is going to go—*into the work that is uniquely yours*. UrlNet does just about everything else for you.

```
        ############ YOUR ANALYTICAL CODE GOES HERE! #############
                                                                  #
        length = len(page)                                        #
        sqrt = math.sqrt(len(page))                               #
                                                                  #
        ############################################################

        network.SetProperty('UrlPagePropsToSet', \
            makeDictionary(length,sqrt) )

        network.SetProperty('DomainPagePropsToSet', \
            makeDictionary(length*3,sqrt/2.0) )

        return True
```

To close out the page checker callback function, we handle exceptions by writing to the log, setting the lastError member variable on the network, and returning False to truncate handling of this page.

```
    Except Exception, inst:
        theError = 'SetPageBasedPropertyValues: %s\n%s\non URL %s' \
              % ( str(type(inst)), str(inst), theUrl )
        network.SetLastError ( theError )
        log.Write(theError)
        return False
```

Now we come to the 'main' attraction, if you will pardon the double entendre. We set some constants, start an exception-handling wrapper, and instantiate a UrlTree object. We put all the good stuff in a function called main(), which will be executed by the very last line in the program.

```
Def main():
    # in case we need to be polite...
    SLEEPTIME = 0
    MAXLEVEL = 2
    try:
        net = UrlTree(_maxLevel=MAXLEVEL,_sleeptime=SLEEPTIME)
```

Now come the couple dozen lines of code mentioned above that is specific to this program, as opposed to boilerplate. We need to set up two lists, one for URLs and one for domains, with property definitions the output-writing functions will need to

```
        ############### YOUR DEFINITIONS GO BELOW HERE #############
        PropertyDictList4Urls = \
            [
                {
```

```
                'attrName'    : 'length',   # must match name in makeDictionary()
                'PorVName'    : 'Page length',
                'doPartition' : True,
                'default'     : 9999998,
                'datatype'    : 'INT',
            },
            {
                'attrName'    : 'sqrt',     # must match name in makeDictionary()
                'PorVName'    : 'Square root of length of page',
                'doPartition' : False,
                'default'     : 0.00001,
                'datatype'    : 'DOUBLE',
            },
        ]

    # this list is identical to the one above, but is duped
    # so you can use this example as a template for your real
    # work, in which URLs and domains might be handled differently.
    PropertyDictList4Domains = \
        [
            {
                'attrName'    : 'length',
                'PorVName'    : 'Page length',
                'doPartition' : True,
                'default'     : 9999998,
                'datatype'    : 'INT',
            },
            {
                'attrName'    : 'sqrt',
                'PorVName'    : 'Square root of length of page',
                'doPartition' : False,
                'default'     : 0.00001,
                'datatype'    : 'DOUBLE',
            },
        ]
    ################ YOUR DEFINITIONS GO ABOVE HERE #############

    # Set properties that tell the output functions how to handle
    # your custom properties.
    Net.SetProperty('additionalUrlAttrs',PropertyDictList4Urls)
    net.SetProperty('additionalDomainAttrs',PropertyDictList4Domains)

    # install the custom page checker
    net.SetPageContentCheckerFn(SetPageBasedPropertyValues)
```

Now we spider the Web to build our networks (URL and domain). The page checker callback function will be invoked for every URL we encounter. After we finish building the network tree structures, we write a Pajek project and two GUESS networks in the normal way. They will contain partitions and vectors (Pajek) and/or additional vertex attributes (GUESS) corresponding to our custom properties set by the page setter callback function.

```
    Net.BuildUrlTree('http://www.southwindpress.com')

    # write Pajek project
    net.WritePajekFile('customproperties2', 'customproperties2')
    # write GUESS network files
    net.WriteGuessFile('customproperties2Urls', doUrlNetwork = True)
```

```
        net.WriteGuessFile('customproperties2Domains', doUrlNetwork = False)
    except Exception, e:
        print str(e)
```

Below we have the grand finale, the code in which Python actually runs our main() function. This is the 'street-legal' way to write an executable python program.

```
If __name__ == '__main__':
    main()
```

That's all there is to a custom page-checker callback function that sets properties based on page content. Our callback could also be filtering pages based on include/exclude criteria, in which case we would need to handle that in much the way the urlutils.CheckInclusionExclusionCriteria() works. If its functionality is sufficient for your purposes, you can call it from your own page-checker callback function; you would do this at the beginning of your function, and if it returned False, you would follow suit immediately and not bother trying to set properties.

12 GENERATING TREES AND FORESTS FROM SEARCH ENGINE QUERY RESULT SETS

12.1 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.

12.1.1 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.

12.1.2 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.

12.1.3 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=\"(.*?)\"')
```

12.1.4 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.

12.1.5 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 16, 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
```

12.2 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:

TABLE 6 – URLNET'S SEARCH ENGINE CLASSES

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.
	GoogleLinkTree	googlelinktree	A way to look at the network of inbound links to a URL, one of the primary data structures used in algorithms for computing URL relevance. The sample program linktree1.py demonstrates it use.
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,

Search Engine	UrlNet class	Module	Notes
			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.
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. But this may be deprecated soon, if not already by the time you read this, because Windows Live is being replaced by Bing.
Bing	BingTree	bingtree	This code may be fragile, due to Bing's newness. I'll try to update it if Bing's result set page format changes.
Yahoo!	YahooTree	yahootree	Yahoo! Provides another relatively simple and straightforward example of how we handle search engine result sets.

For your convenience…

In all of these, there are two builder functions that produce a tree from a search engine result set (e.g., GoogleTree.BuildUrlTree and GoogleTree.BuildTreeWithPlaceholderRoot) and another builder function that produces a forest (e.g., GoogleTree.BuildForestFromPhantomRoot). The tree functions will replace the URL of the root node item and the domain name of the root domain node item with a token that shows the search engine and the query (e.g., 'Bing—stop smoking'). This makes your network visualizations less cryptic that the obscure, highly decorated search engine query URLs.

12.3 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.

13 ANALYZING REACHABILITY AND FINDABILITY

Purveyors of products, services, and information over the Web are well-advised to determine the findability of their Web presence via organic search (commercial search engines ranking query results by relevance heuristics). There is a well-defined workflow for this type of investigation. The first step is determining the queries one's target market segment is most likely to employ (*semantic analysis*). The second step is to determine whether one's target pages are reachable at all from the result set (*reachability analysis*), and the third step is to determine the likelihood an individual in the target audience will detect the "information scent" of the target pages, follow his or her metaphorical nose, and find the site (*findability analysis*).

The semantic analysis step in this workflow may or may not be able to leverage network analysis, but we'll leave that question open for the time being. In this chapter, we'll look at how network analysis and visualization can help with reachability and findability.

The example script we'll look at here, searchengine2.py, employs the AOLTree class in a variation on the example we looked at in section 12.3. This program closely emulates the data collection algorithm I employed in my Master's thesis work at the University of Michigan School of Information.[4]

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.

The merged network shows us the degree of connection between 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
```

[4] 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.

```
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)
```

14 COMPARING AND CONTRASTING SEARCH ENGINE RESULTS FOR SYNONYMOUS QUERIES

A well-known barrier to obtaining relevant results in search and retrieval is the "Vocabulary Problem".[5] In simplified terms, the Vocabulary Problem posits that keyword assignment to documents is an inadequate means of supporting search, because different people express the identical problem in different terms. A lot of empirical research over the years has demonstrated the validity of this assertion. Network analysis using UrlNet-generated trees and forests provide a visual means of analyzing the problem in specific situations.

The example program netsmerged1.py shows how to use UrlNet to investigate the Vocabulary Problem as it manifests in search engine result sets. In this example, we look at how AOL responds to three popular and synonymous query phrases used to find information on how to quit smoking. If you look way down at the bottom at the main() function call (not shown here), you'll see how to switch it between producing trees and forests.

```
# These two constants are the possible values for the whichBuilder argument to main()
PLACEHOLDER = 'PLACEHOLDER'
PHANTOM = 'PHANTOM'

def main(whichBuilder):

    . . .   # the same working directory and log setup code as in the previous example

    goAhead = True
    query = None

    try:
        # combine 'quit smoking', 'stop smoking', and 'smoking cessation'
        # query result trees
        net = AOLTree(_maxLevel=1,
                      _workingDir=workingDir,
                      _resultLimit=10
                      )
        net.SetIgnorableText(textToIgnore)
        net.SetTruncatableText(textToTruncate)
        ret = True
        badQuery = None
        for query in ('quit smoking','stop smoking','smoking cessation'):

            # set a query filename root that tells us which builder method and
            # query were used. This will be used in writing some debugging info
            # to disk in our working directory.

            net.SetFilenameFromQuery('netsmerged1-%s-%s' % (str(whichBuilder),query))

            # NOTE: the following line is crucial to making the merge work
            # as expected. Without it, the search URL is treated as an
```

[5] G. W. Furnas , T. K. Landauer , L. M. Gomez , S. T. Dumais. "The Vocabulary Problem in Human-System Communication: an Analysis and a Solution." Communications of the ACM (1987) 30:11 pp. 964-971.

```
            # ordinary URL and all links on that page are processed, not just
            # the links in the search results. Does nothing the first time
            # thru the loop.

            net.RestoreOriginalUrlClass()

            root = \
              'http://search.aol.com/aol/search?s_it=comsearch40&query=%s&do=Search' \
                  % re.sub(' ','+',query)
            if whichBuilder == PLACEHOLDER:
                ret = net.BuildUrlTreeWithPlaceholderRoot(root,query)
            elif whichBuilder == PHANTOM:
                ret = net.BuildUrlForestWithPhantomRoot(query)
            else:
                goAhead = False
                break
            if not ret:
                badQuery = query
                break

        if goAhead == False:
            print 'main(whichBuilder) requires whichBuilder ' + \
                   'to be PHANTOM or PLACEHOLDER-nothing built.'
        elif not ret:
            print 'main(whichBuilder) %s builder function failed for query "%s".' \
                 % (whichBuilder, badQuery )
        else:
            qry = query
            query = 'done building net'
            netname = 'netsmerged1-%s' % str(whichBuilder)
            net.WritePajekFile(netname,netname)

    except Exception,e:
        if query == None:
            myLog.Write(str(e)+'\n on constructor invocation\n')
        elif query == 'done building net':
            myLog.Write(str(e)+'\n during Pajek output file generation\n')
        else:
            myLog.Write(str(e)+'\n on ' + query + ' query\n')
```

Here's the network of URLs generated by the program, using the setting to produce a three-root tree network:

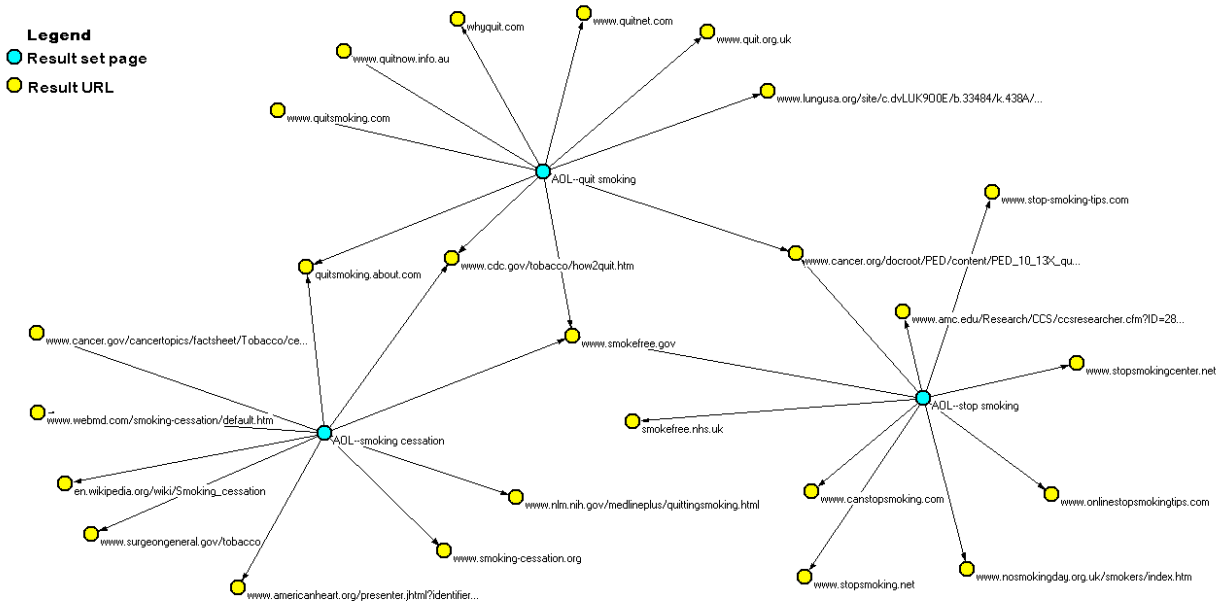


FIGURE 17 - THREE-ROOT DIRECTED URL NETWORK CONTRASTING THREE SYNONYMOUS QUERIES (BLUE=QUERY, YELLOW=RESULT ITEM)

14.1.1 VISUALIZING THE NETWORK IN TERMS OF POTENTIAL VALUE TO THE INFORMATION SEEKER

It would be really useful to visualize this in terms of the relative value of the results. Fortunately, we can determine link value relatively reliably using the top-level domain of each URL and domain name (e.g., ",com", ".org", etc.).

According to the original planning for the Internet, these had specific meanings and were therefore intended to be reserved for specific types of domains: dot-com for commercial entities; dot-edu for educational institutions; dot-org for non-profit organizations; dot-gov for governmental units; dot-mil for military; dot-net for Internet service providers; et cetera.

As shown in section 11.2.5, A bonus example: flagging specific URLs in a network, found above on page 74, UrlNet makes this relatively easy to do. Using code from customproperties3.py, I modified a copy of netsmerged1.py to gather the data automatically as the three networks are generated and merged. This modified version is the example program netsmerged2.py. It produces the following result:

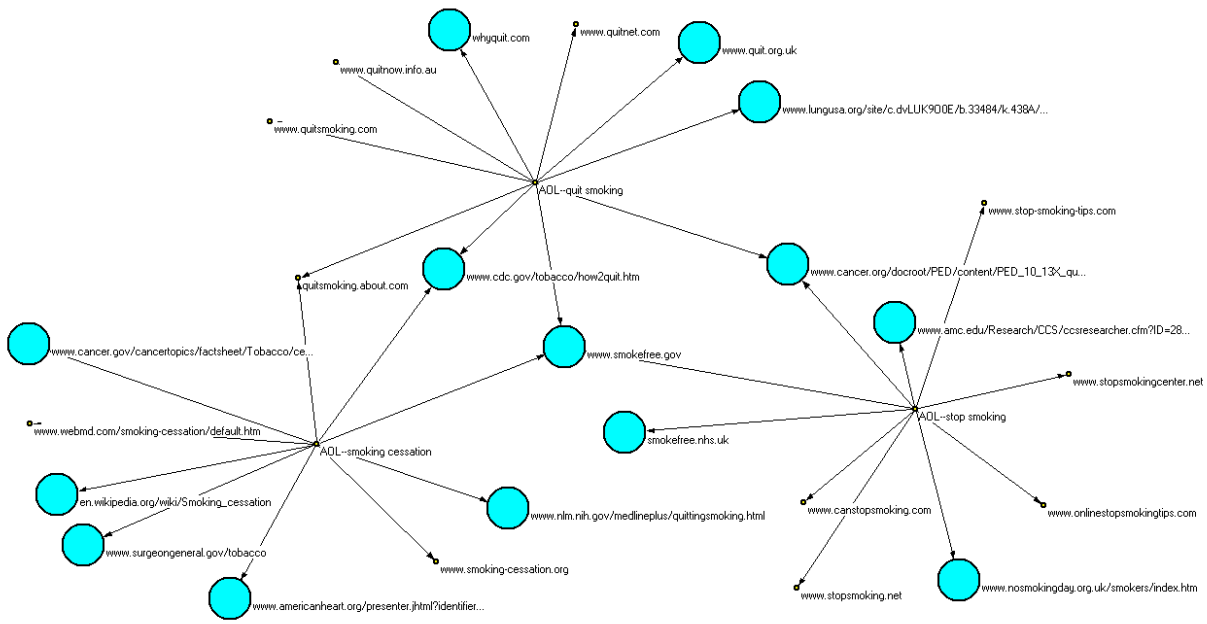


FIGURE 18 – SAME URL NETWORK WITH VERTEX SIZE = TLD VALUE COLOR = PROFIT STATUS (BLUE=NON-PROFIT, YELLOW=FOR-PROFIT)

In this visualization, a vector whose values represent my estimation of the relative value of the different top-level domains determines the size of nodes. The vector values come from this table in tldconstants.py:

```
# author-defined relative value of different TLD types.
V_DOTCOM  = 0.01
V_DOTORG  = 0.5
V_DOTNET  = 0.01
V_DOTEDU  = 0.5
V_DOTGOV  = 0.5
V_DOTADM  = 0.01
V_DOTMIL  = 0.01
V_OKDOTCOM = 0.25
# imposter
V_DOTFAKE = 0.01
# unknown
V_DOTUNK  = 0.01
```

You may have noticed that there are exceptions to the general rules. One dot-com domain, whyquit.com, is a labor-of-love site without advertising, which in my mind puts it in the dot-org category. Conversely, smokingcessation.org is actually a commercial site masquerading as an NGO, hence is classified as equivalent to a dot-com.

As you can see, nonprofit TLDs (governmental, NGO, and educational) are valued the highest. There are some dot-coms that provide relatively high-value content at the expense of bombarding the visitor with ads, e.g., mayoclinic.com (not shown in this diagram). These are rated at half the value of the high-value

TLDs. All others are rated at one-twentieth the value of the top TLDs. The vector is used along with the partition that classifies vertices by TLD type; the vector determines the size and the partition the color of the vertices in the diagram.

This visualization gives the impression that the search results produce high-quality results: governmental, NGO, and educational sites are associated with the majority of vertices. Factoring in click probability based on position in the result set (see section 12.1.1, Estimating click-through probabilities, above on page 80) gives a somewhat different picture.

14.1.2 Investigating click probability

In the next diagram, the vector used to size the vertices is based on the probability of the information seeker clicking on each result URL, as determined by position in the result set and using probabilities per position from a large topically-related data set.

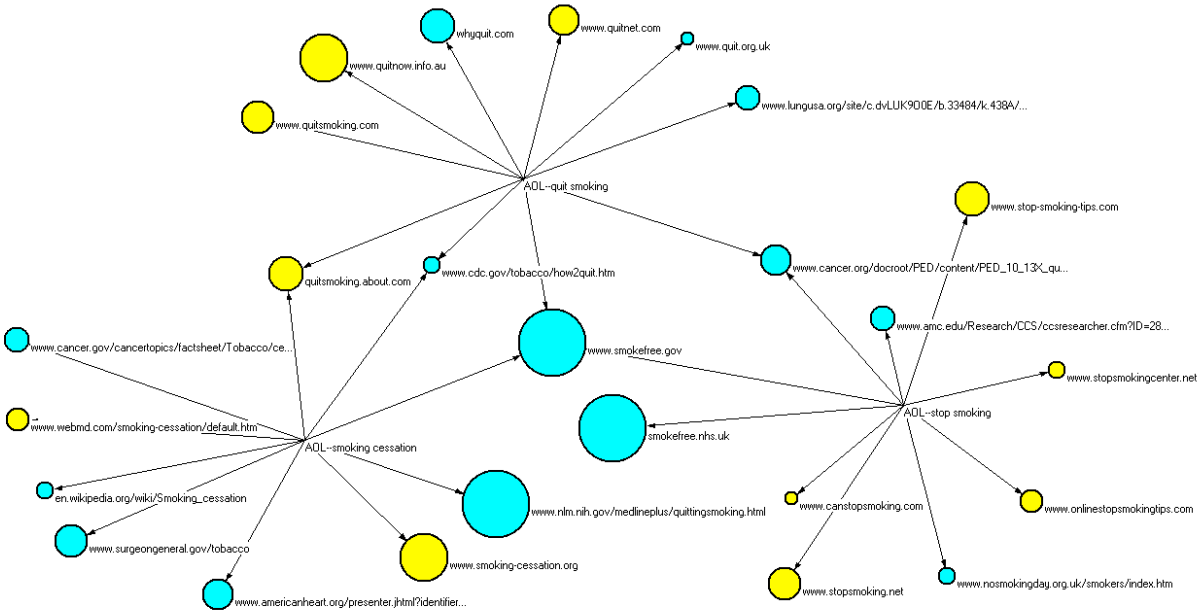


FIGURE 19 - SAME URL NETWORK WITH VERTICES SIZED BY CLICK PROBABILITY

The largest vertices, the number-one item in the results returned by each of the three synonymous queries, are indeed high-value. Smokefree.gov, the # 1 result of the quit-smoking query, is the second result of the stop-smoking query and third in the smoking-cessation query results. This is a good start. The rest of the picture, though, is quite mixed. The second result for smoking cessation is the dot-org that is actually a dot-com. Dot-coms form the majority of the remaining “above-the-fold” (top five) results in the three result sets.

14.1.3 The value of network visualization

Contrasting the previous two diagrams, we can see that no one factor is going to give us a useful measure of the value of search engine results. We can see the importance of click position simply by looking at the two diagrams in terms of the change in the proportional color balance, with blue representing non-profit vertices and yellow for-profit. Although the three result sets contain a majority of non-profit results, there is a disturbingly high likelihood that information seekers will visit and rely on information from for-profit sites.

This kind of insight is impossible to derive from the first of the three diagrams, and difficult to discern by simply looking at numbers in a spreadsheet. Network visualization is a powerful tool, leveraging our visual acuity in order to augment our cognitive capabilities.

15 Working with Web Service APIs, Part one: Technorati

15.1 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[6] 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.

15.2 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).

15.2.1 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.

15.2.2 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,
```

[6] 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.

15.2.3 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.

15.2.4 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.

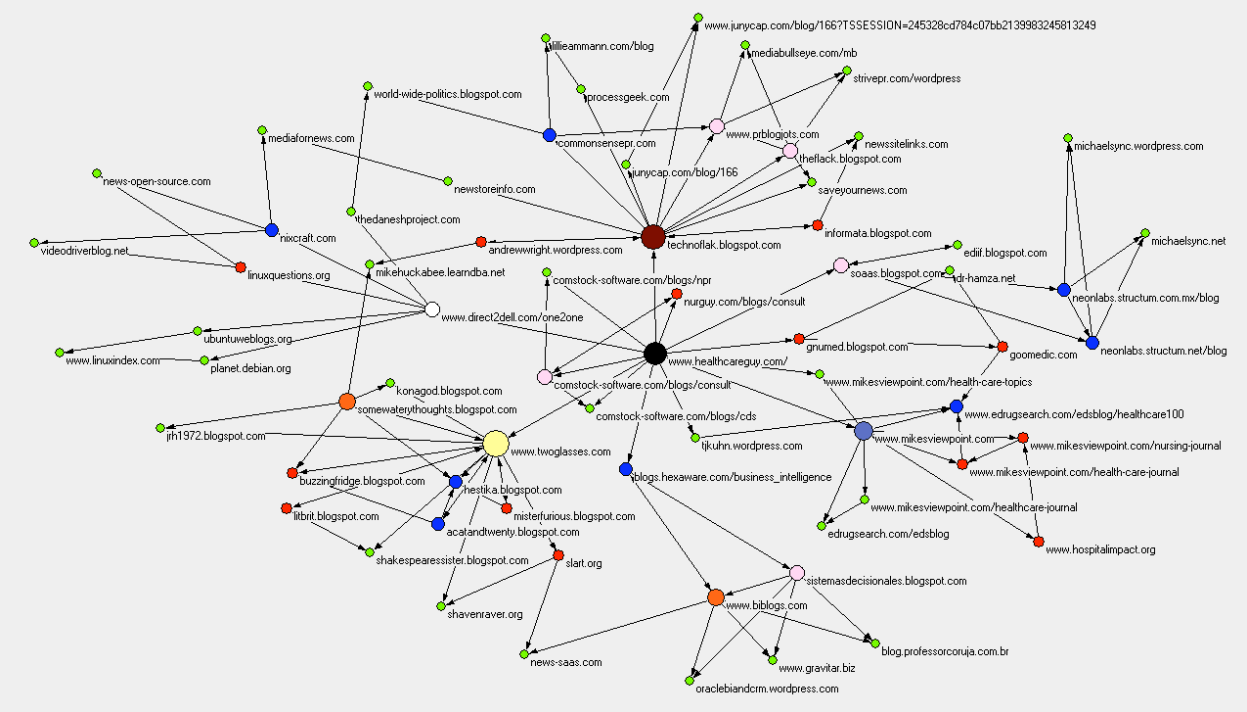
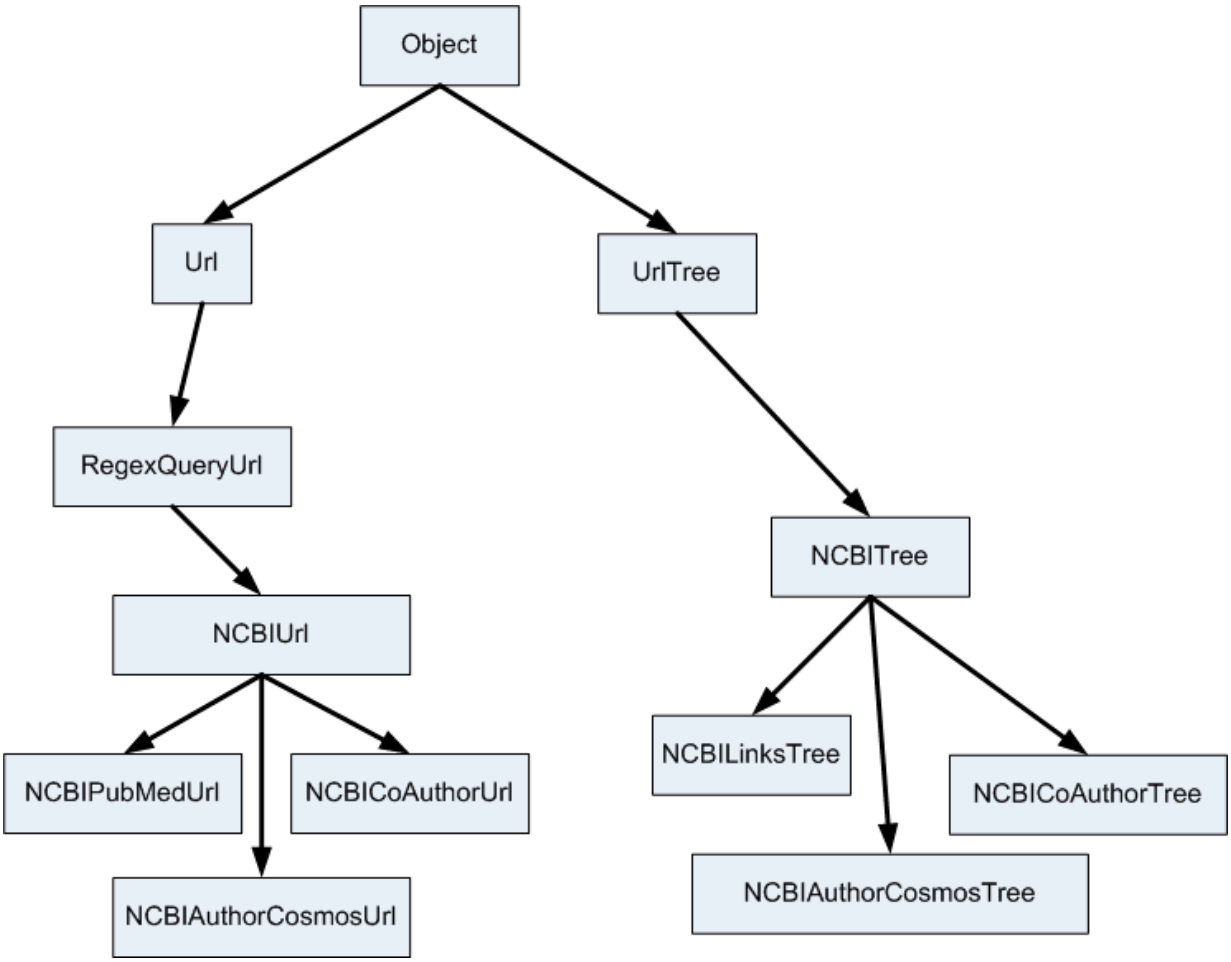


FIGURE 20 – A TECHNORATI COSMOS NETWORK

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

16.1 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.



16.2 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.

16.2.1 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:

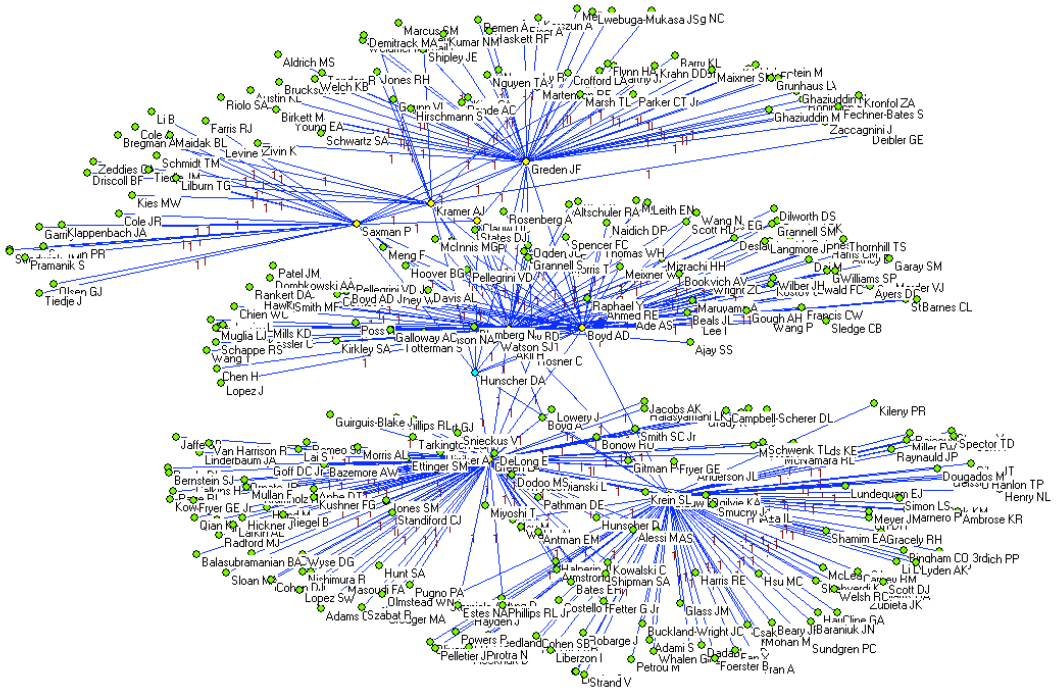


FIGURE 21 - NCBI-CO-AUTHOR NETWORK

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.

16.2.2 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 22 - NCBI Author cosmos on page 105 was created in Pajek, using the cosmos of Dr. Victor Strecher, a University of Michigan investigator.

In Figure 23 on page 106, 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 24 on page 107 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.

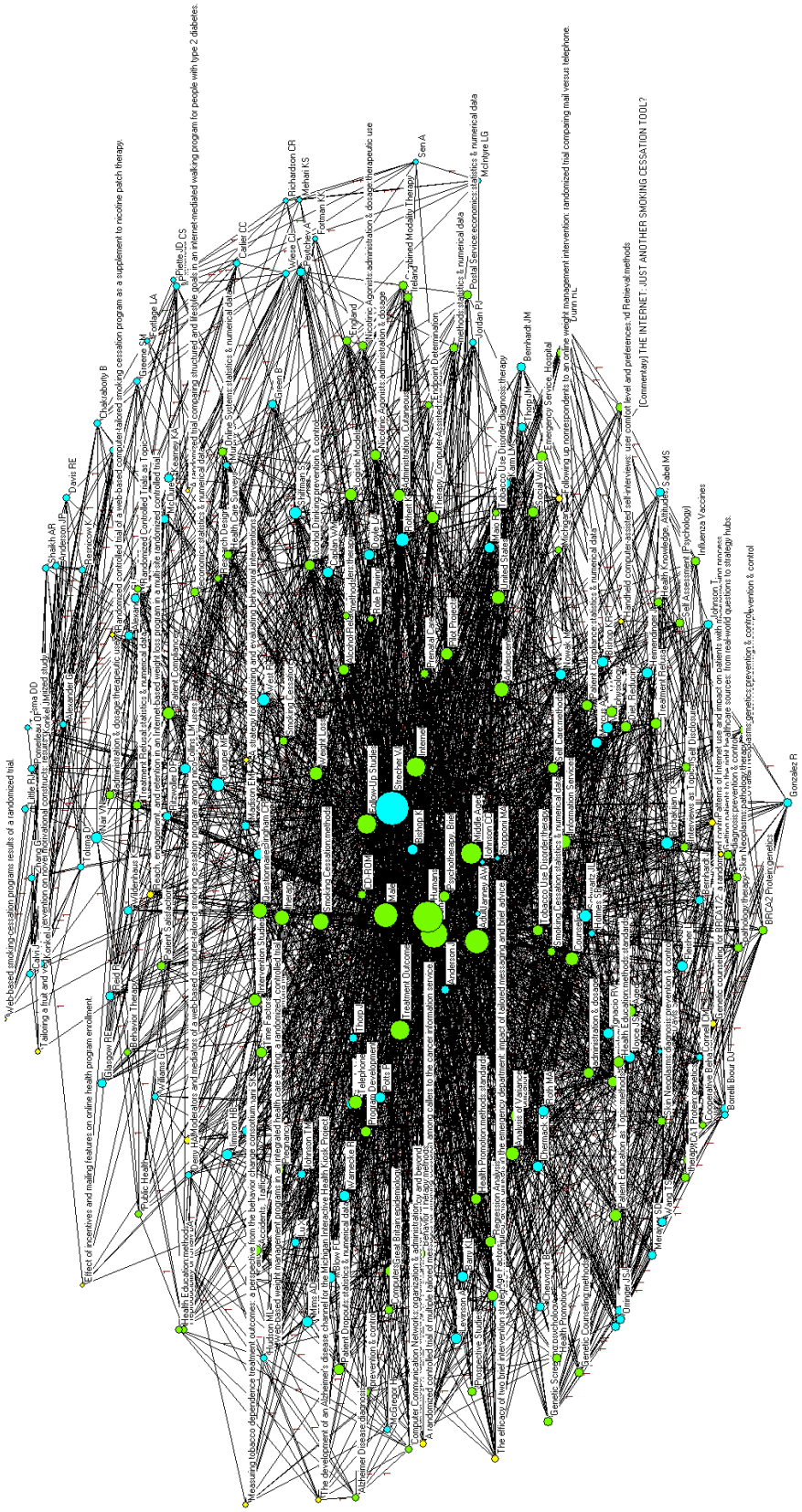


FIGURE 22 - NCBI AUTHOR COSMOS

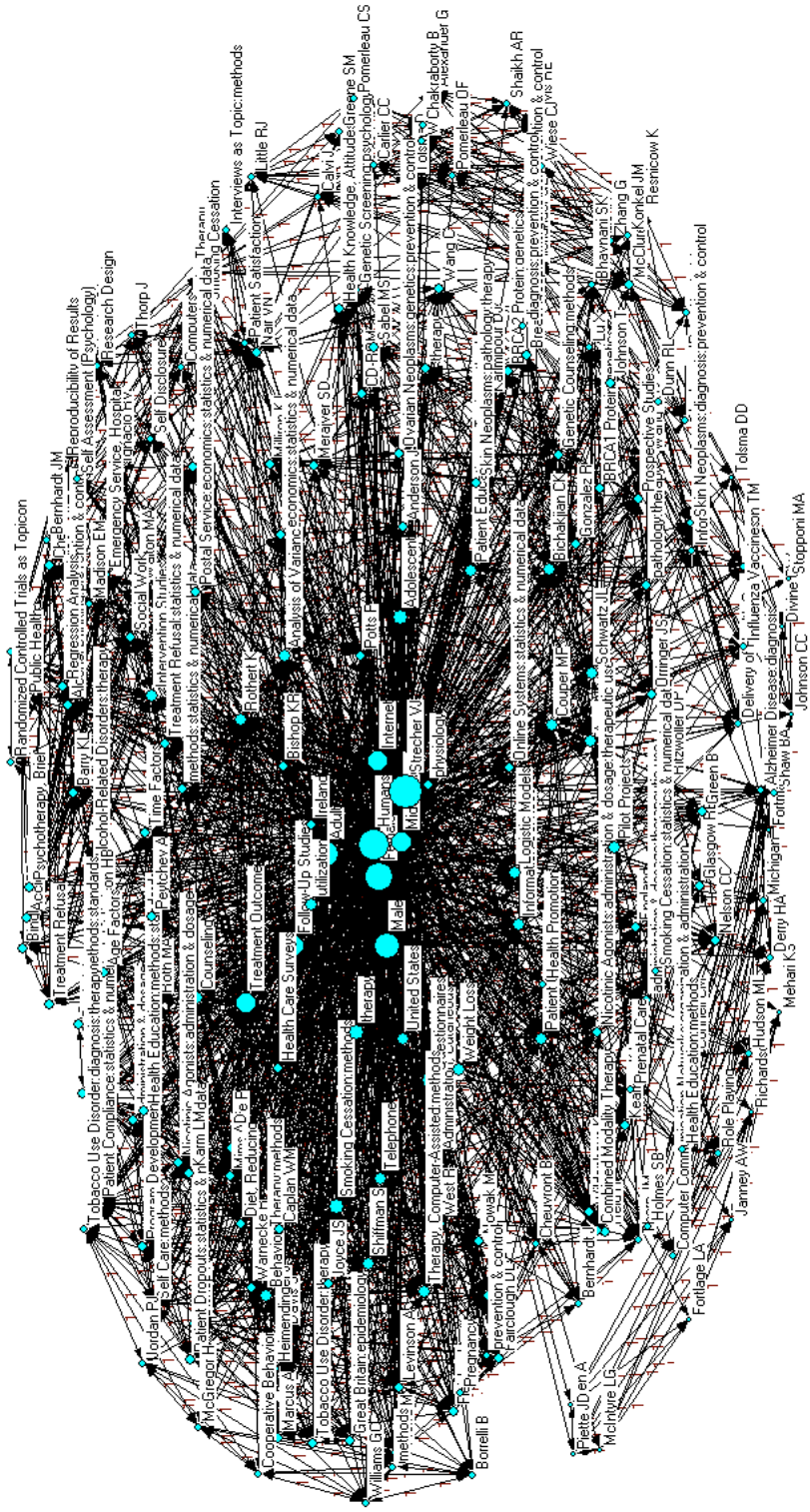


FIGURE 23 – AUTHORS AND MESH TOPICS

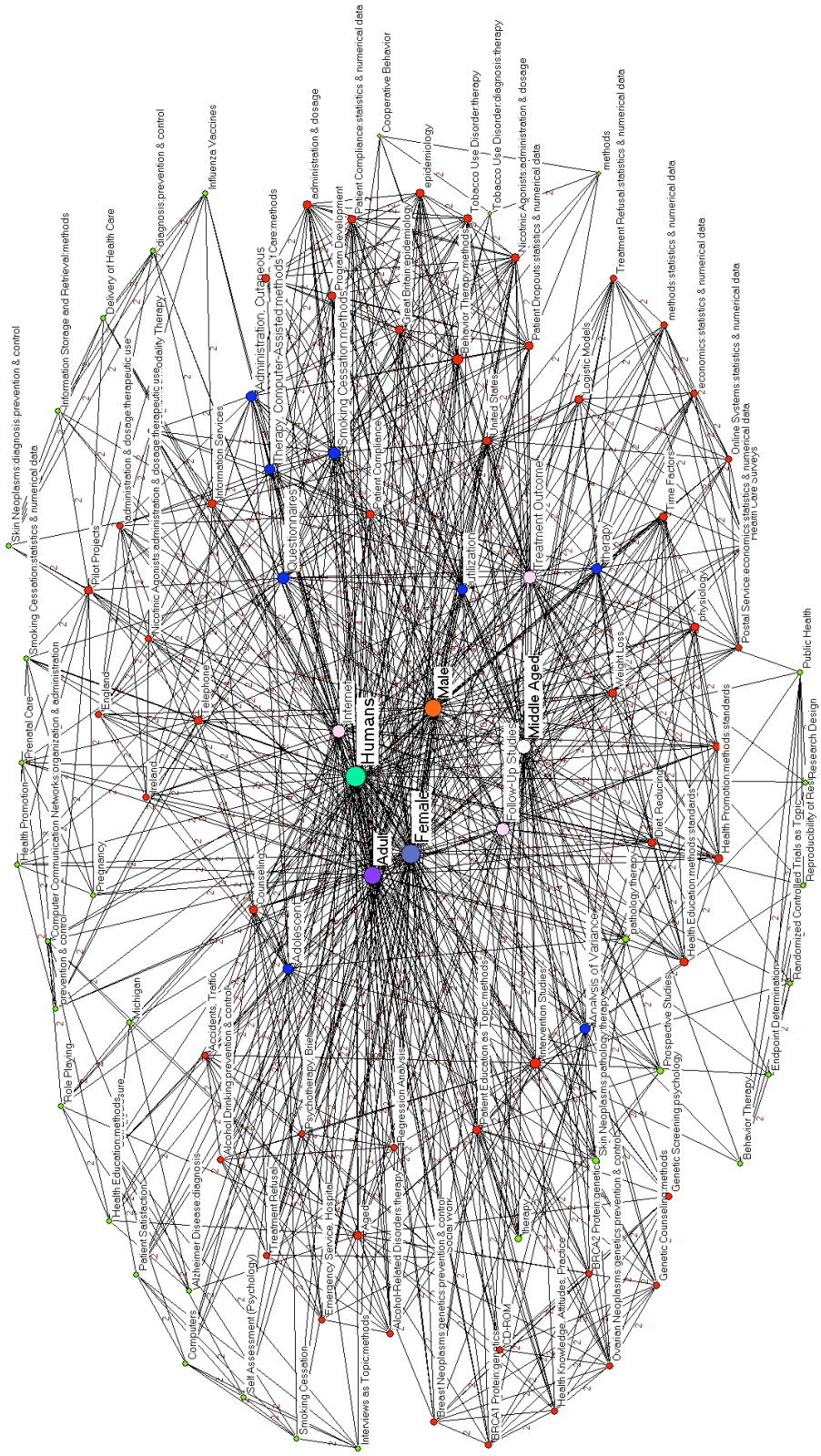


FIGURE 24 - MESH TOPICS NETWORK

16.2.3 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:

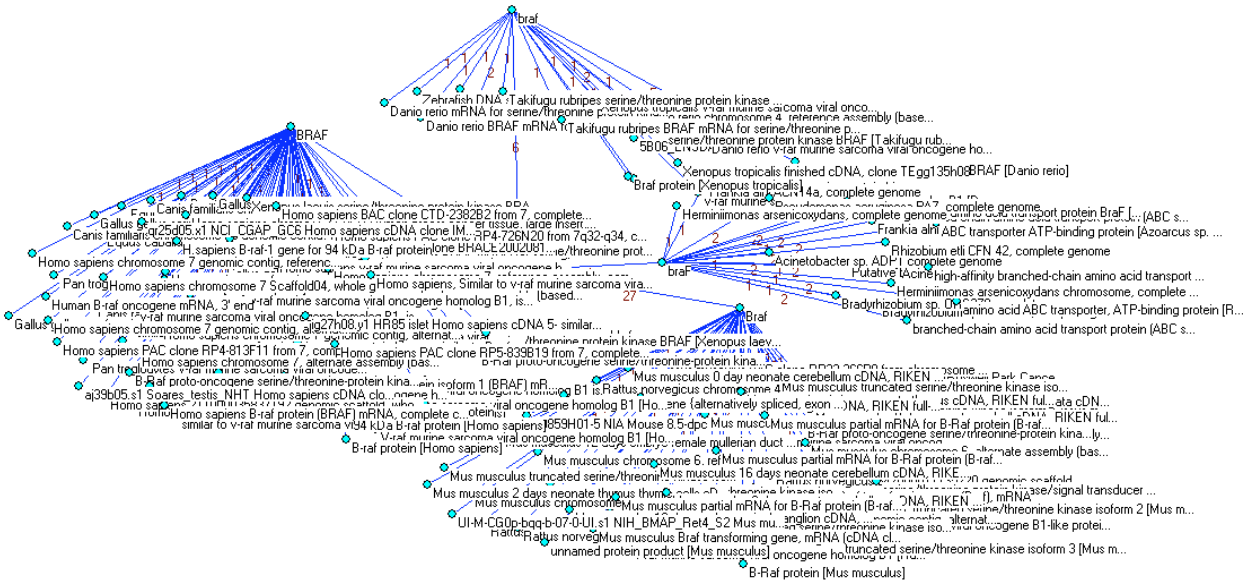


FIGURE 25 - LINKS BETWEEN GENE, PROTEIN, NUCLEOTIDE, AND SNP DATABASES REGARDING GENE 'BRAF'

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)
. . .
```

Helpful NCBI Query Debugging Feature

If you wish to see the raw output of the NCBI Web Services, set the properties 'WriteEFetchRawOutput', 'WriteELinkRawOutput', 'WriteESearchRawOutput', and/or 'WriteESummaryRawOutput' on your NCBILinkTree network to a file name/path to which it can be written.

```
net.SetProperty('WriteELinkRawOutput', 'elinkoutput-raw.txt')
net.SetProperty('WriteESearchRawOutput', 'esearchoutput-raw.txt')
net.SetProperty('WriteEFetchRawOutput', 'efetchoutput-raw.txt')
net.SetProperty('WriteESummaryRawOutput', 'esummaryoutput-raw.txt')
```

To see the data written to the log file instead, which will happen only when you choose to turn on the logging feature as described in section 7.3 above, set an equivalent set of properties as follows:

```
net.SetProperty('LogELinkRawOutput', True)
net.SetProperty('LogESearchRawOutput', True)
net.SetProperty('LogEFetchRawOutput', True)
net.SetProperty('LogESummaryRawOutput', True)
```

The ncbilinkstree*.py examples set all four of these, though each of them employ only a subset of the four NCBI services used in UrlNet. It's worth setting all four when using the NCBI classes, so you'll be sure to get the information you need to understand what's happening.

Turning on logging and writing the log to a file, as shown in the ncbilinkstree*.py examples, will allow you to see the actual URLs used in the NCBI service invocations.

Here's the network resulting from this investigation:

FIGURE 26 - LINKS BETWEEN PROTEIN, GENE, AND PUBMED WITH RESPECT TO THE PROTEIN DUTPASE

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

16.3 CAVEATS

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

17 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. 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. This provides a nice lead-in to the next flaw:
- **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. Please educate me.
- **Eat its own dog food.** The library should be able to read common network serialization formats, for example the output streams it writes. It doesn't. It bothers me, but not enough to do anything about it yet. At least you can now save a network and reload it later, in case you want to write output of a different type.
- **More output formats.** I've added one new format in this release, which outputs pairs of node names (URLs or domains) representing edges in the network. That format is read by a number of network analysis programs, but there are others I haven't covered. I don't own a copy of UCINet, a fairly popular program, because it is expensive and I would have to fork over the money out of my own pocket. As a result, the library doesn't write files in UCINet's format. I have the specification and some examples, but I don't have a way to test, and I won't put out code I haven't tested.
- **Better support for GUESS.** It is an abomination that I know so little about GUESS. I think it really annoyed Dr. Adamic and I value her opinion highly, so I may do something about this sooner than some of the others.
- **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.
- **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 while back 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).[7] I will fix or remove the in-module test code sometime in the future; meanwhile, my focus is on the examples and the manual.

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

[7] In my circle, fragments of deprecated code like these are referred to as dust bunnies, named for the balls of lint, pet-hair, and who-knows-what-else that appear underneath furniture between Hooverings. Like their namesake, deprecated code fragments are useless and annoying, but not particularly harmful.