The Mathematica® Journal

Random Walks on the World Wide Web

Todd Silvestri

This article presents *RandomWalkWeb*, a package developed to perform random walks on the World Wide Web and to visualize the resulting data. Building upon the package's functionality, we collected empirical network data consisting of 35,616 unique URLs (approximately 133,500 steps). An analysis was performed at the domain level and several properties of the web were measured. In particular, we estimated the power-law exponent γ for the in- and out-degree distributions, and obtained values of 2.10 ± 0.09 and 2.36 ± 0.1, respectively. These values were found to be in good agreement with previously published results.

1 Introduction

The World Wide Web (WWW), commonly referred to as simply "the web," is a vast information network accessible via the Internet. Initially proposed in 1989 [1], the web grew out of the work of Tim Berners-Lee while at the European Organization for Nuclear Research, known as CERN.

Two software technologies form the core of the web, namely the HyperText Markup Language (HTML) and the Hypertext Transfer Protocol (HTTP). The HTML (or source) of a web page contains elements known as tags that describe the content of the document. For instance, the anchor < a > tag defines a hyperlink, or link, to another document. Each file (or resource) on the web is identified by a Uniform Resource Locator (URL). A browser or other user agent may request a file via HTTP by specifying its URL. The response—typically the requested file—is again transmitted by HTTP from the web server to the client.

The topology of large-scale complex networks, such as the web, can be explored using graph theoretic methods (see [2] and references therein). Specifically, the web can be viewed as a directed graph, where the web pages are vertices and the hyperlinks are edges. Unfortunately, two problems exist due to the nature of the web: (1) it cannot be indexed (or mapped) in its entirety; and (2) analyzing the corresponding graph would be highly computationally intensive. In fact, a recent announcement [3] suggests that the web may contain at least 10^{12} unique URLs.

Despite its sheer size and complexity, it is possible to extract meaningful measures that can be used to quantify the web's structure. Using the method of random walks, one can sample the network and examine a subgraph of the web.

In this method, one begins at a specified URL. The client requests the web page at that URL, and the server responds with the document's HTML. Next, the client extracts all URLs from the web page and chooses one of them at random. Again, the client makes a request for the document at the chosen URL. If the server cannot be reached or the web page cannot be found, the client simply chooses another URL at random. The entire process is then repeated a finite number of times.

The focus of this article is on the application of random walks to the study of the World Wide Web. In Section 2, we provide a brief overview of *RandomWalkWeb*, a package developed to perform random walks on the web and to visualize the resulting data. Next, in Section 3, we build upon the package's functionality and use it to perform a random walk to sample the web. The collected empirical network data is then analyzed and several properties of the web are estimated. Finally, in Section 4, we provide a summary of our work and give our concluding remarks.

2 A Brief Overview of *RandomWalkWeb*

In this section, we give a brief overview of *RandomWalkWeb* and demonstrate some of its functionality.

The *RandomWalkWeb* Package (www.mathematica-journal.com/data/uploads/2013/09/RandomWalkWeb.zip) is built upon the graph and network functionality introduced in *Mathematica* 8. In addition, connectivity to the web is provided through *.NET/Link* and requires the .NET Framework 2.0 (or higher).

The package consists of 28 public symbols covering the following four areas: (1) data collection and visualization; (2) web page components; (3) operations on URLs; and (4) message logging. Each symbol is fully documented and can be easily accessed through the *Mathematica* help system.

We begin by loading the package.

```
Needs["RandomWalkWeb`"]
```

A single random walk on the web can be executed, as described in Section 1, by using the package's namesake, `RandomWalkWeb`. The start (or origin) URL is specified, along with the maximum number of steps n_s to be taken.

```
RandomWalkWeb["http://wolfram.com/", 10] // Column
```

```
http://wolfram.com/
http://products.wolframalpha.com/
http://www.wolfram.com/mathematica/how-mathematica-made-
  wolframalpha-possible.html
http://www.stephenwolfram.com/
http://www.wolframalpha.com/
http://www.wolframalpha.com/examples/
http://www.wolframalpha.com/examples/Geochronology.html
http://www.wolframalpha.com/input/?i=continental+map+for+the
  +Karoo+ice+age&amp;lk=3
http://www.wolframscience.com/
http://www.wolframalpha.com/
http://www.wolfram.com/
```

`RandomWalkWeb` returns a list of successfully visited URLs, displayed here as a column. If the function is evaluated from a notebook-based front end, the walk's progress is displayed in the window status area. In the event that it reaches a URL with zero valid outgoing links, `RandomWalkWeb` will attempt to backtrack at most one step. The function may exit prematurely; that is, the number of steps returned is less than n_s, if all previous hyperlinks have been exhausted.

One may wish to perform multiple random walks from the same URL. This can be accomplished by using `PerformRandomWalks`. Like `RandomWalkWeb`, we specify the start URL and the maximum number of steps to be taken. Additionally, we pass the number of random walks n_w to be performed as the function's second argument.

```
PerformRandomWalks["http://wolfram.com/", 3, 8]
```

```
3
```

The value that is returned indicates the number of successfully exported data files.

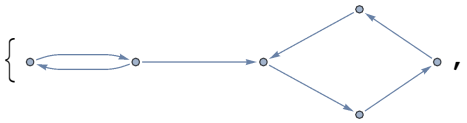
The root directory used to store the data files is specified by `$BaseDataDirectory`. By default, this is set to the current working directory. For each unique URL passed to `PerformRandomWalks`, a folder is created in the root data directory whose name is a 32-character, hexadecimal-formatted MD5 hash of that URL. The successfully visited URLs from each walk are exported as separate human-readable plain text files. The name of each file is a combination of a label, specified by `$DataFilePrefix`, and the walk number.

In the next example, we examine previously collected network data (available at www.mathematica-journal.com/data/uploads/2013/09/RW_Data_1.zip). We begin by specifying both `$BaseDataDirectory` and `$DataFilePrefix`.

```
$BaseDataDirectory =
  FileNameJoin[{NotebookDirectory[], "RW_Data_1"}];
$DataFilePrefix = "2012-09-28_RW";
```

The data can be easily imported and visualized by using `RandomWalkGraph`. Here, we are interested in the first seven steps extracted from the second random walk.

```
RandomWalkGraph["http://wolfram.com/", {2}, 7]
```

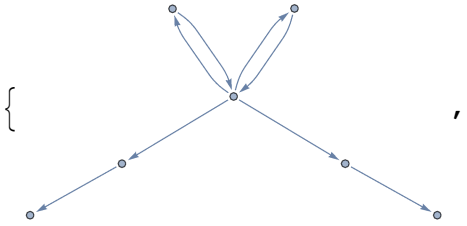


```
{{1, mathematica-journal.com}, {2, stephenwolfram.com},
 {3, wolframalpha.com}, {4, wolfram.com},
 {5, wolfram-media.com}, {6, wolframscience.com}}}
```

The first part of the returned list is a `Graph` object, while the last part contains a list of enumerated vertices. All graphs returned by `RandomWalkGraph` are simple directed graphs; that is, they contain neither loops nor multiple edges.

Similarly, one can construct a graph from multiple data files. Here, we combine all steps from the first and third random walks.

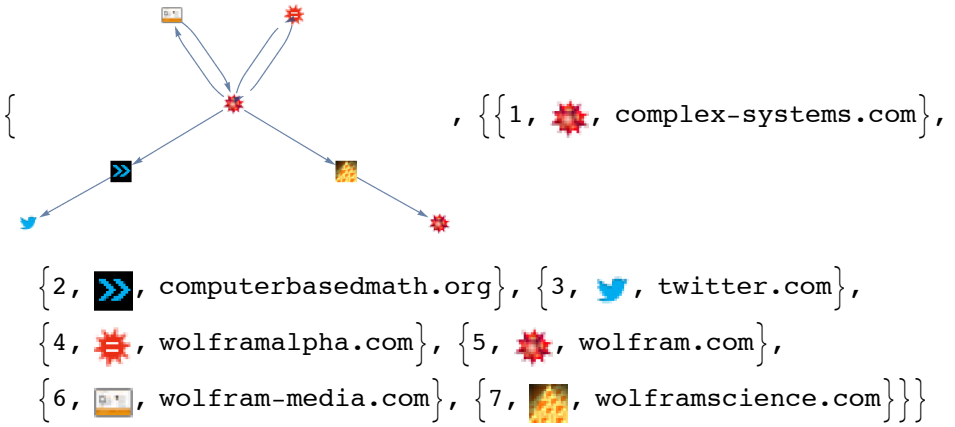
```
RandomWalkGraph["http://wolfram.com/", {1, 3}, 8]
```



```
{{1, complex-systems.com}, {2, computerbasedmath.org},
 {3, twitter.com}, {4, wolframalpha.com}, {5, wolfram.com},
 {6, wolfram-media.com}, {7, wolframscience.com}}}
```

The graphs can be visually enhanced by using the `VertexIcon` option. With `VertexIcon` set to `True`, `RandomWalkGraph` attempts to download each vertex's associated favorite icon and uses it in place of the default vertex shape. `RandomWalkGraph` also accepts the same options as the built-in function `Graph`.

```
RandomWalkGraph["http://wolfram.com/", {1, 3}, 8,
 VertexIcon → True, VertexSize → Medium]
```



■ 3 Properties of the Web

In this section, we build upon *RandomWalkWeb*'s functionality and use it to perform a random walk to sample the web. The collected empirical network data is then analyzed and several properties of the web are estimated.

A Note on Timings

The timings reported in this section were measured on a custom workstation PC using the built-in function `AbsoluteTiming`. The system consists of an Intel® Core™ i7 CPU 950 @ 4 GHz and 12 GB of DDR3 memory. It runs Microsoft® Windows™ 7 Professional (64-bit) and scores 1.23 on the *MathematicaMark8* benchmark.

□ 3.1 Data Collection

▪ *3.1.1 Random Walk with Jumps*

Let us define a new function called `RandomWalkWithJumps`.

```
RandomWalkWithJumps[
  originURL_String?AbsoluteLinkQ,
  numJumps_Integer /; numJumps ≥ 0,
  numSteps_Integer /; numSteps ≥ 1
 ] :=
```

```
Module[{$FunctionName = "RandomWalkWithJumps",
  dataDirectory, i, numWalks = numJumps + 1, pt,
  url = originURL, successfulURLs, totalNumSteps = 0,
  dataFileName, urlHistory = {}, totalUniqueURLs},
 LogMessage["DEBUG", "Entering " <> $FunctionName <> "."];

 dataDirectory =
  FileNameJoin[{$BaseDataDirectory,
    IntegerString[Hash[originURL, "MD5"], 16]}];
 If[! FileExistsQ[dataDirectory],
  CreateDirectory[dataDirectory];
 ];

 For[i = 1, i ≤ numWalks, i++,
  If[$Notebooks,
   pt = PrintTemporary["Walk " <> ToString[i] <> " of " <>
        ToString[numWalks] <> ": " <> url];
  ];
  LogMessage["INFO", "Walk Number: " <> ToString[i] <>
    " (of " <> ToString[numWalks] <> ")"];

  successfulURLs = RandomWalkWeb[url, numSteps];

  If[successfulURLs =!= $Failed,
   totalNumSteps += Length[successfulURLs] - 1;,
   (* else *)
   successfulURLs = {};
  ];

  dataFileName =
   FileNameJoin[{dataDirectory,
     $DataFilePrefix <> ToString[i] <> ".txt"}];
  Export[dataFileName, successfulURLs];

  urlHistory = Union[urlHistory, successfulURLs];
  totalUniqueURLs = Length[urlHistory];

  If[totalUniqueURLs > 0,
   url = RandomChoice[urlHistory];,
   (* else *)
   Break[];
  ];

  If[$Notebooks,
   NotebookDelete[pt];
  ];
 ];

 LogMessage["DEBUG", "Exiting " <> $FunctionName <> "."];
 {totalNumSteps, totalUniqueURLs}
]
```

The function behaves like `PerformRandomWalks` (see Section 2) except that, instead of returning to the start URL, `RandomWalkWithJumps` "jumps" to a URL chosen at random from the walk's history. From there, the function attempts to perform an additional n_s steps, and the process is repeated. The number of walks completed is given by $n_w = n_j + 1$, where n_j is the specified number of jumps.

Before using `RandomWalkWithJumps` to collect empirical network data, we set up a few package parameters.

```
$BaseDataDirectory =
  FileNameJoin[{NotebookDirectory[], "RW_Data_2"}];
$DataFilePrefix = "2012-10-22_RW";
```

Next, we change the location of the log file (see Appendix).

```
$LogFileName =
  FileNameJoin[{$BaseDataDirectory, "messages.log"}];
```

Finally, we specify a generic user agent string (see Appendix).

```
$UserAgent = "Mozilla/5.0 (compatible; RWW/1.4)";
```

For this particular walk, our goal is to perform a total of 150,000 steps.

```
nj = 599;
ns = 250;
nw = nj + 1;

ns * nw

150 000
```

We now evaluate `RandomWalkWithJumps`.

```
RandomWalkWithJumps["http://wolfram.com/", nj, ns]

{133 505, 35 616}
```

The function exits after nearly 26 hours on the web. The returned list shows that approximately 89% of the requested number of steps was completed. Additionally, we see that 35,616 unique URLs were visited.

3.2 Analysis

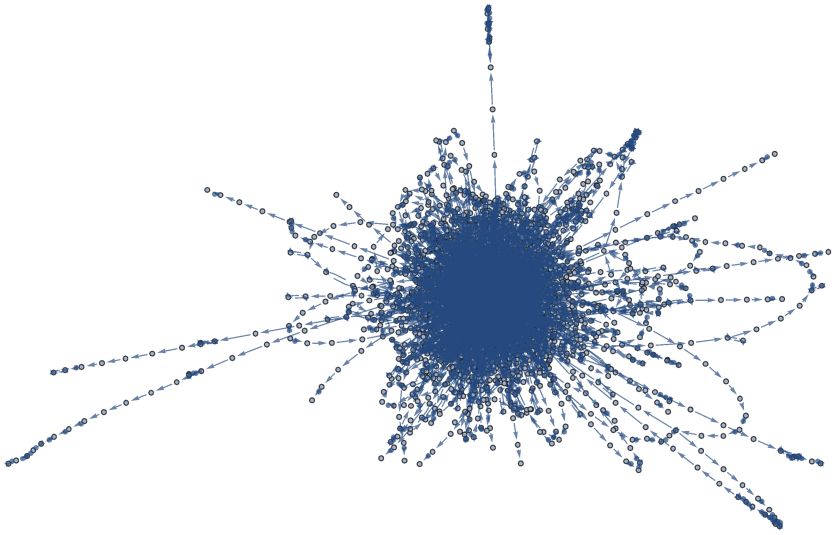
3.2.1 Data Import and Visualization

We proceed by importing the collected empirical network data (available at www.mathematica-journal.com/data/uploads/2013/09/RW_Data_2.zip) using `RandomWalkGraph`. The built-in function `Range` is used to generate a complete list of file numbers.

```
{graph, enumeratedVertices} =
  RandomWalkGraph["http://wolfram.com/", Range[nw], ns];
```

It takes approximately 68 seconds for the function to return the `Graph` object and list of enumerated vertices.

```
graph
enumeratedVertices
```



```
{{1, 0713hb.com}, {2, 100kin10.org},
 {3, 104.fr}, {4, 12321.org.cn}, {5, 123rj.com},
 {6, 1394ta.org}, {7, 163.com}, {8, 1800postcards.com},
 {9, 1915studios.com}, {10, 193.71.77.27}, <<4308>>,
 {4319, zope.org}, {4320, zq24.com}, {4321, zunicore.com},
 {4322, zupuk.com}, {4323, zuqiubifen.com},
 {4324, zvab.com}, {4325, zvooq.ru}, {4326, zweitehand.de},
 {4327, zynga.com}, {4328, zz-police.com}}
```

Together, they contain all the information needed to perform a domain-level analysis.

■ *3.2.2 Basic Measures*

Using *Mathematica*'s built-in functions, we extract a few basic graph measures from our data.

Let n be the number of vertices (i.e., domain names) in the graph.

```
n = VertexCount[graph]
```

```
4328
```

Similarly, let m be the number of edges (i.e., links).

```
m = EdgeCount[graph]
```

```
9665
```

In general, the degree k_i of vertex i is a count of the edges attached to it. For example, we can use `VertexDegree` to get the number of links connected to a given domain name.

```
VertexDegree[graph, "wolfram.com"]
```

```
1
```

The mean vertex degree c of a graph is given by

$$c = \frac{1}{n} \sum_{i=1}^{n} k_i. \tag{1}$$

If a vertex is not specified, `VertexDegree` returns a list of degrees for all vertices in the graph. Calculating c from our empirical network data is then straightforward.

```
c = N[Mean[VertexDegree[graph]]]
```

```
4.46627
```

Here, we report the mean absolute deviation from c.

```
N[MeanDeviation[VertexDegree[graph]]]
```

```
3.62206
```

■ 3.2.3 In- and Out-Degree Distributions

For a directed graph, a vertex has both an in- and out-degree equal to the number of ingoing and outgoing edges, respectively.

Let p_k^{in} be the fraction of vertices in a directed graph that have in-degree k. Similarly, let p_k^{out} be the fraction of vertices with out-degree k. We define two functions to calculate these quantities.

```
pIn[g_?DirectedGraphQ, k_Integer /; k ≥ 0] :=
 Count[VertexInDegree[g], k] / VertexCount[g]

pOut[g_?DirectedGraphQ, k_Integer /; k ≥ 0] :=
 Count[VertexOutDegree[g], k] / VertexCount[g]
```

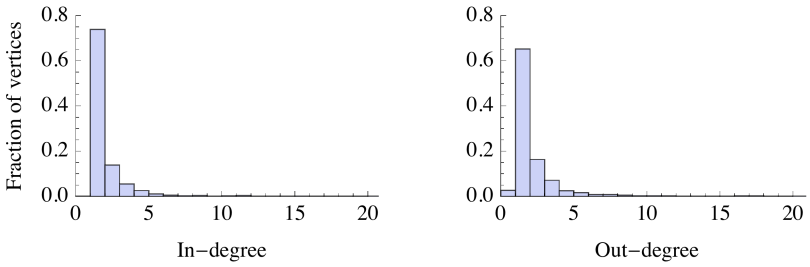
Both p_k^{in} and p_k^{out} can be viewed as the probability that a vertex chosen at random will have in- and out-degree k, respectively. For example, using our empirical network data, we can calculate the probability of randomly choosing a domain name with seven ingoing links.

```
N[pIn[graph, 7]]
```

```
0.0030037
```

We use the built-in function `Histogram` to visualize the degree distributions.

```
hInDegreeDistribution =
  Histogram[VertexInDegree[graph], {1}, "Probability",
   Frame → {{True, False}, {True, False}},
   FrameLabel → {"In-degree", "Fraction of vertices"},
   PlotRange → {{0, 20}, {0, 0.8}}];
hOutDegreeDistribution =
  Histogram[VertexOutDegree[graph], {1}, "Probability",
   Frame → {{True, False}, {True, False}},
   FrameLabel → {"Out-degree"},
   PlotRange → {{0, 20}, {0, 0.8}}];
GraphicsRow[{hInDegreeDistribution, hOutDegreeDistribution}]
```



■ *3.2.4 Power-Law Degree Distributions*

Evidence suggests [4] that the in- and out-degree distributions of the web exhibit power-law behavior. Here, we attempt to reproduce those results using our collected empirical network data. We proceed by defining the in-degree cumulative distribution function (CDF):

$$P_k^{\text{in}} = \sum_{j=k}^{k_{\max}^{(i)}} p_j^{\text{in}}, \tag{2}$$

where $k_{\max}^{(i)}$ is the maximum vertex in-degree of the graph and p_j^{in} is the fraction of vertices with in-degree j, as defined earlier. A similar expression can be written for the out-degree CDF, P_k^{out}.

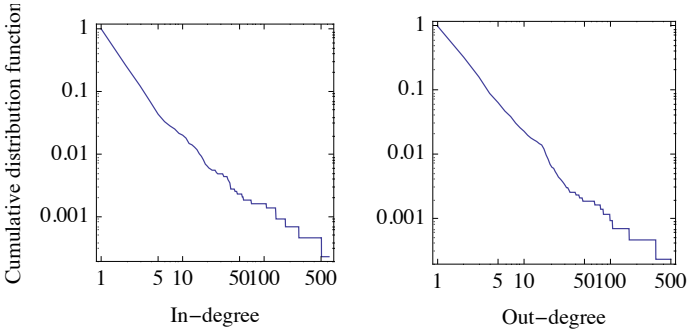
```
InDegreeCDF[g_?DirectedGraphQ, k_Integer /; k ≥ 0] :=
 Sum[N[pIn[g, j]], {j, k, Max[VertexInDegree[g]]}]

OutDegreeCDF[g_?DirectedGraphQ, k_Integer /; k ≥ 0] :=
 Sum[N[pOut[g, j]], {j, k, Max[VertexOutDegree[g]]}]
```

Next, we use both functions to generate data spanning their entire degree domains.

```
dataInDegreeCDF = Table[{k, InDegreeCDF[graph, k]},
    {k, 0, Max[VertexInDegree[graph]]}];
dataOutDegreeCDF = Table[{k, OutDegreeCDF[graph, k]},
    {k, 0, Max[VertexOutDegree[graph]]}];
```

The resulting CDF data is visualized using the built-in function `ListLogLogPlot`.

```
lllpInDegreeCDF = ListLogLogPlot[dataInDegreeCDF,
    AspectRatio → 1, Frame → True,
    FrameLabel →
     {"In-degree", "Cumulative distribution function"},
    Joined → True, PlotRange → Full];
lllpOutDegreeCDF = ListLogLogPlot[dataOutDegreeCDF,
    AspectRatio → 1, Frame → True, FrameLabel → {"Out-degree"},
    Joined → True, PlotRange → Full];
GraphicsRow[{lllpInDegreeCDF, lllpOutDegreeCDF}]
```



The log-log plots reveal approximate power-law behavior in the degree distributions of the web.

Now, let us assume that the degree distributions are proportional to $k^{-\gamma}$ for $k \geq k_{\min}$, where $k_{\min}$ is some minimum degree for which the power law holds. Following [5], we use the maximum likelihood estimator (MLE) $\hat{\gamma}$ to estimate the power-law exponents:

$$\hat{\gamma} \simeq 1 + N\Big[\sum_{i=1}^{N} \ln \frac{k_i}{k_{\min} - \frac{1}{2}}\Big]^{-1}, \tag{3}$$

where N is the number of vertices with degree $k_i \geq k_{\min}$. This approximation remains accurate, provided $k_{\min} \gtrsim 6$. The standard error on $\hat{\gamma}$ is given by

$$\sigma = \frac{\hat{\gamma} - 1}{\sqrt{N}}. \tag{4}$$

We encapsulate equations (3) and (4) in the following function.

```
PLExponentEstimated[
  g_?DirectedGraphQ,
  kmin_Integer /; kmin ≥ 6,
  distType_String /;
   StringMatchQ[distType, {"InDegree", "OutDegree"}]
 ] :=
 Module[{$FunctionName = "PLExponentEstimated",
   vertexDegrees, ki, M, exponentMLE, expStandardError},
  Switch[distType,
   "InDegree",
   vertexDegrees = VertexInDegree[g];,
   "OutDegree",
   vertexDegrees = VertexOutDegree[g];
  ];

  ki = Select[vertexDegrees, # ≥ kmin &];
  M = Length[ki];

  exponentMLE =
   1 + M * Total[Map[N[Log[# / (kmin - 1 / 2)]] &, ki]]^-1;
  expStandardError = (exponentMLE - 1) / Sqrt[M];

  {exponentMLE, expStandardError}
 ]
```

Evaluating `PLExponentEstimated` yields an estimate of the power-law exponent.

```
PLExponentEstimated[graph, 6, "InDegree"]
```

```
{2.10333, 0.0922647}
```

```
PLExponentEstimated[graph, 6, "OutDegree"]
```

```
{2.36154, 0.0957979}
```

Here, the minimum value for $k_{\min}$ was used. We see that $\gamma_{\text{in}} = 2.10 \pm 0.09$ and $\gamma_{\text{out}} = 2.36 \pm 0.1$ for the in- and out-degree distributions of the web, respectively. These values are in good agreement with those reported in [4].

■ 3.2.5 Distribution of Top-Level Domains

We examine the distribution of top-level domains (TLDs) in our data. Examples of TLDs include `com`, `net`, and `org`.

First, the domain names are extracted from the list of enumerated vertices. We then extract the last part of each domain name and use `EffectiveTLDNameQ` (see Appendix) to filter the results.

```
domainNames = enumeratedVertices[[All, 2]];
tlds = Select[Map[Last[StringSplit[#, "."]] &, domainNames],
  EffectiveTLDNameQ]
```

```
{com, org, fr, cn, com, org, com, com, com, fr,
 <<4304>>, org, com, com, com, com, com, ru, de, com, com}
```

Next, we get the total number of TLDs and tally the list.

```
totalNumTLDs = Length[tlds];
talliedTLDs = Tally[tlds]
```

```
{{com, 2557}, {org, 490}, {fr, 28}, {cn, 49}, {jp, 131},
 {ch, 18}, {it, 18}, {net, 150}, {ru, 25}, {nl, 39},
 <<73>>, {lu, 1}, {cz, 1}, {sh, 1}, {am, 1}, {ro, 1},
 {travel, 1}, {jobs, 1}, {va, 1}, {pro, 1}, {mq, 1}}
```

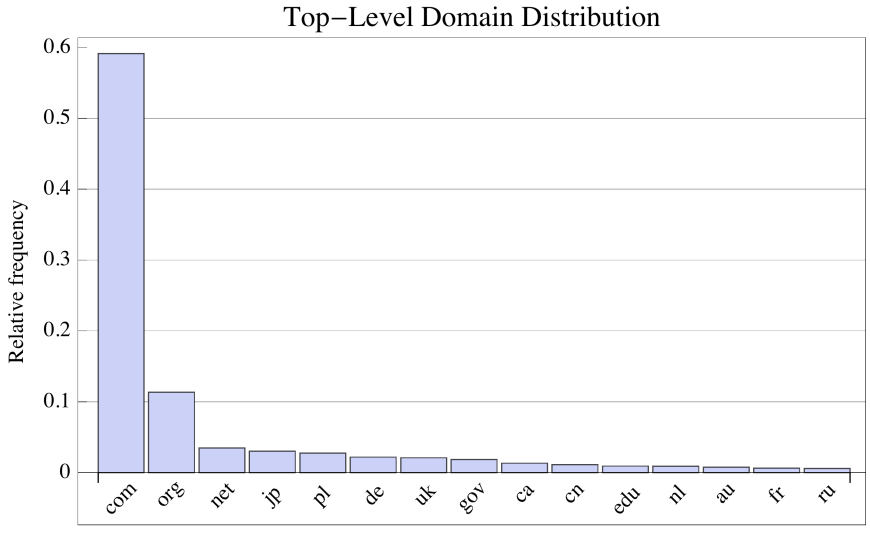
The relative frequency of each TLD is then calculated and the data is sorted—both numerically and alphabetically.

```
relFreqTLD = Map[{First[#], N[Last[#] / totalNumTLDs]} &,
   talliedTLDs];

relFreqTLD = GatherBy[Sort[relFreqTLD, Last[#1] > Last[#2] &],
   Last];
relFreqTLD = Flatten[Map[Sort[#] &, relFreqTLD], 1]
```

```
{{com, 0.591351}, {org, 0.113321}, {net, 0.0346901},
 {jp, 0.030296}, {pl, 0.0275208}, {de, 0.0219704},
 {uk, 0.0210453}, {gov, 0.0185014}, {ca, 0.0134135},
 {cn, 0.0113321}, <<73>>, {ps, 0.000231267},
 {ro, 0.000231267}, {sh, 0.000231267}, {to, 0.000231267},
 {tr, 0.000231267}, {travel, 0.000231267}, {tt, 0.000231267},
 {uy, 0.000231267}, {va, 0.000231267}, {xxx, 0.000231267}}
```

Finally, we visualize the resulting data using the built-in function `BarChart`.

```
BarChart[relFreqTLD[[ ;; 15, 2]],
 ChartLabels → Placed[relFreqTLD[[All, 1]], Axis,
   Rotate[#, π / 4] &], Frame → True,
 FrameLabel → {"", "Relative frequency"},
 FrameTicks → {{Union[{0, 1}, Range[0.1, 0.9, 0.1]], None},
   {None, None}}, GridLines → {None, Automatic},
 PlotLabel → "Top-Level Domain Distribution",
 PlotRange → {All, {-0.06, 0.6}}]
```



Here, the relative frequencies of the top 15 TLDs are compared.

4 Conclusion

In this article, we have presented *RandomWalkWeb*, a package developed to perform random walks on the World Wide Web and to visualize the resulting data. Building upon the package's functionality, we collected empirical network data consisting of 35,616 unique URLs. A domain-level analysis was performed and several properties of the web's structure were measured. We examined the in- and out-degree distributions and verified their approximate power-law behavior. The power-law exponents were estimated to be $\gamma_{\text{in}} = 2.10 \pm 0.09$ and $\gamma_{\text{out}} = 2.36 \pm 0.1$, in good agreement with previously published results.

The *RandomWalkWeb* Package relies upon the graph and network functionality introduced in *Mathematica* 8. In addition, the package was designed to take advantage of the client-server communication features provided by the .NET Framework. The choice to use *.NET/Link* affects only a small number of package functions, and it would be a straightforward task to reimplement those functions to utilize other technologies, e.g., *J/Link*.

RandomWalkWeb can also be improved and expanded in many different ways. For instance, one could modify the code to allow functions like `RandomWalkWeb` to be evaluated on parallel subkernels. Another possibility would be to construct a full-featured *Mathematica*-based crawler capable of exploring the web's structure more methodically.

Finally, since *Mathematica* forms the foundation of Wolfram|Alpha, one could easily imagine the web-based computational knowledge engine returning graph theoretic answers to users' queries regarding the World Wide Web.

Appendix: Design of *RandomWalkWeb*

This appendix provides some details on the design and implementation of the *RandomWalkWeb* Package. Readers are strongly encouraged to review the fully documented source code.

Web Page Components

GetSource

The first operation in performing a random walk on the web is to request and obtain the HTML (or source) of a web page. The most straightforward way to accomplish this is to use the built-in function `Import`.

```
Import["http://wolfram.com/", "Source"]
```

```
<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
  xml:lang="en" lang="en">
<head>

<link rel="shortcut icon"
  href="/common/images2003/f…unction displayIMG(){
        // Insert no-flash image into the flash div
        document.getElementById('flash').innerHTML
  = '<a href="/system-modeler/"><img
  src="/images/homepage2010/noflash-wsm.png"
  border="0" /></a>';
    }
/* ]]> */
</script>
</body>
</html>
```

There is, however, at least one drawback to this method. A website may be configured to serve different content to different devices (e.g., mobile versus desktop). Various methods exist for detecting the type of device making the request.

One technique in particular involves the server parsing the User-Agent HTTP header sent by the client. The client software uses this header to identify itself to the server during requests. For instance, if we pass a URL to `Import` and evaluate it using *Mathematica* 8, a server would see `Mathematica / 8.0.4.0.0 PM / 1.3.1` as its user agent string. Unfortunately, this string is immutable.

To circumvent this constraint, *RandomWalkWeb* implements its own HTML import function called `GetSource`. The function uses *.NET/Link* to communicate with the .NET runtime. During HTTP requests, `GetSource` transmits the string assigned to `$UserAgent`.

```
$UserAgent
```

```
Mathematica/8.0.4.0 RWW/1.4
```

We can request and obtain the HTML of a web page using `GetSource`.

```
GetSource["http://wolfram.com/"]
```

```
{http://wolfram.com/,
 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML
   1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

```
<html xmlns="http://www.w3.org/1999/xhtml"
   xml:lang="en" lang="en">
<head>

<link rel="shortcut icon"
   href="/common/images2003/f…nction displayIMG(){
        // Insert no-flash image into the flash div
        document.getElementById('flash').innerHTML
   = '<a href="/system-modeler/"><img
   src="/images/homepage2010/noflash-wsm.png"
   border="0" /></a>';
    }
/* ]]> */
</script>
</body>
</html>
}
```

The first part of the returned list is the responding address. Typically, this address is the same as the requested URL. However, it may differ due to one or more redirects. The last part contains the HTML of the web page.

Let us now set `$UserAgent` to mimic a popular mobile device.

```
$UserAgent =
  "Mozilla/5.0 (iPhone; CPU iPhone OS 5_0 like Mac
    OS X) AppleWebKit/534.46 (KHTML, like Gecko)
    Version/5.1 Mobile/9A334 Safari/7534.48.3";
```

This time we pass a list of URLs to `GetSource` and inspect the responding addresses.

```
urls = {"http://bing.com/", "http://facebook.com/",
   "http://weather.com/", "http://yahoo.com/"};
GetSource[urls][[All, 1]]

{http://m.bing.com/?mid=10006,
 http://m.facebook.com/?refsrc=http://www.facebook.com/&_rdr
 , http://m.weather.com/,
 http://m.yahoo.com/?.tsrc=yahoo&mobile_view_default=true}
```

Here, we see that `GetSource` follows the redirects and obtains the HTML from mobile-specific addresses. Having the ability to modify the user agent string allows us to perform random walks on the so-called "mobile web."

□ Operations on URLs

▪ *DomainName*

RandomWalkWeb contains several functions that perform useful operations on URLs. One in particular, `DomainName`, is used both in standalone form and internally by `RandomWalkGraph` and related functions. As its name suggests, it extracts the domain name from the specified URL.

```
url = "http://www.example.net/dir1/page1.html";
DomainName[url]

example.net
```

At the heart of `DomainName` lies a list of known effective TLDs, or public suffixes, that is imported from a data file and stored in memory during the package's initialization. Examples of effective TLDs include `com`, `co.uk`, and `nj.us`. The file is located in the Data folder under the package's root directory. It consists of a base list [6] augmented by user-specified additions.

We can evaluate `$ETLDNInfo` and inspect the last part of the returned list to determine the number of effective TLD names in the data file.

```
$ETLDNInfo // Last

6065
```

`DomainName` works by first splitting the hostname into a list of components.

```
StringSplit[Hostname[url], "."]

{www, example, net}
```

Next, the function takes the last part of the returned list and uses `EffectiveTLDNameQ` to test whether the string is a known effective top-level domain. If it is, the next-to-last part of the list is prepended to the string and joined with a dot (a period). Again, the string is tested. The process of growing and testing the string continues until `EffectiveTLDNameQ` gives `False`. The result is the domain name.

If the effective TLD cannot be determined, `DomainName` returns the equivalent of `Hostname`.

```
DomainName["http://www.example.zzz/dir1/page1.html"]

www.example.zzz
```

```
Hostname["http://www.example.zzz/dir1/page1.html"]

www.example.zzz
```

If message logging is enabled (see later in this appendix), `DomainName` logs the error and hostname for later review. The user can then decide whether to add the missing effective TLD to the appropriate section of the data file.

□ Message Logging

▪ *LogMessage*

During code development, it is often necessary to examine detailed error messages (e.g., .NET exceptions) or to trace the execution path of a function. This can be especially difficult if the function is called repeatedly hundreds or even thousands of times. For these reasons, several of the functions in *RandomWalkWeb* have been designed to write error, informational, and debug-level messages to a plain text file.

By default, messages are written to `messages.log` located in the directory given by `$TemporaryDirectory`. Evaluating `$LogFileName` shows the fully qualified name of the log file.

```
$LogFileName

C:\Users\$UserName\AppData\Local\Temp\messages.log
```

One is free to change the name or location of the file. Also, message logging can be disabled entirely by assigning an empty string to `$LogFileName`.

We use `LogMessage` to write a message to the log file, specifying its level of importance.

```
msg = "$DebugLogging = " <> ToString[$DebugLogging];
LogMessage["INFO", msg]
```

We can verify that the message was appended to the file.

```
ReadList[$LogFileName, String] // Last

[2012-10-10 19:44:12.003] INFO  : $DebugLogging = False
```

Acknowledgments

The author is grateful to his family for their unwavering patience and support.

References

[1] T. Berners-Lee. "Information Management: A Proposal." (May, 1990) www.w3.org/History/1989/proposal-msw.html.

[2] M. Newman, *Networks: An Introduction*, Oxford, UK: Oxford University Press, 2010.

[3] J. Alpert and N. Hajaj. "We Knew the Web Was Big..." *Google Official Blog* (blog, Google, owner). (Jul 25, 2008) googleblog.blogspot.com/2008/07/we-knew-web-was-big.html.

[4] A. Barabási, R. Albert, and H. Jeong, "Scale-Free Characteristics of Random Networks: The Topology of the World-Wide Web," *Physica A*, **281**(1–4), 2000 pp. 69–77. doi:10.1016/S0378-4371(00)00018-2.

[5] A. Clauset, C. Shalizi, and M. Newman, "Power-Law Distributions in Empirical Data," *SIAM Review*, **51**(4), 2009 pp. 661–703. doi:10.1137/070710111.

[6] Mozilla Foundation. "Public Suffix List." (Aug 2, 2012) publicsuffix.org.

T. Silvestri, "Random Walks on the World Wide Web," *The Mathematica Journal*, 2013. dx.doi.org/doi:10.3888/tmj.15-9.

About the Author

Todd Silvestri received his undergraduate degrees in physics and mathematics from the University of Chicago in 2001. As a graduate student, he worked briefly at the Thomas Jefferson National Accelerator Facility (TJNAF), where he helped to construct and test a neutron detector used in experiments to measure the neutron electric form factor at high momentum transfer. From 2006 to 2011, he worked as a physicist at the US Army Armament Research, Development and Engineering Center (ARDEC). During his time there, he cofounded and served as principal investigator of a small laboratory focused on improving the reliability of military systems. He is currently working on several personal projects.

Todd Silvestri
New Jersey, United States
todd.silvestri@optimum.net