Software for Simulating and Analyzing Markov Chains

Bellcore TM-ARH-020-376

Adam L. Buchsbaum[1] Milena Mihail[2]

September, 1991

[1]Work conducted while the author was a summer student in the Mathematics, Information Sciences and Operations Research Division. Author's current address: Dept. of Computer Science, Princeton University, Princeton, NJ 08544-2087.
[2]Bell Communications Research, Morristown, NJ.

Contents

1 Introduction

Recent research into randomized algorithms has focused on the rate of convergence of various Monte Carlo methods to the true values desired. In many circumstances however, theoretical conclusions about the convergence rates of proposed algorithms either remain open or involve very sophisticated techniques. For this reason, a package of software has been written to simulate various randomized algorithms that follow a certain general outline. It is the hope that the results of these simulations will help direct efforts toward devising theoretical bounds for the algorithms' performance.

1.1 Overview of Random Processes

The rest of this report describes the software that was written to perform and analyze these simulations. An initial overview of the random processes being simulated is first necessary, though. In general, a random process will consist of a (probably large) state space (of size s) and a relation that describes which states are reachable from any state in the space. At any time, the process will be in a state and will randomly (uniformly among all the possibilities) choose an adjacent state to which to move (or might stay in the current state). How many such random steps are necessary to leave the process in any state with probability $1/s$ independent of the initial state measures the rate of convergence, and it is this number that the simulation helps discover.

Since the state space will be prohibitively large, some implicit representation must be used. Therefore, we model the random process in the following way. Consider a *ground set* E of *ground elements*. The state space will be some family $\mathcal{B} \subseteq 2^E$. Any $B \in \mathcal{B}$ is a *basis*; thus, each basis corresponds to a state, and moving from basis to basis will mimic the state transitions. There must exist some efficient algorithm for testing whether or not $B \in \mathcal{B}$ without searching $\mathcal{B}$ explicitly (as we assumed that $\mathcal{B}$ is too large to fit in main memory) and some practical way of moving from basis to basis.

For example, let $G = (V, E)$ be an undirected graph of n vertices and m edges. A subgraph $G' = (V, E')$, $E' \subseteq E$ is *connected* if and only if there exists a *path* (a series of *adjacent* edges, where two edges are adjacent if they share a vertex) between any pair of vertices in G'. We wish to count the number of connected subgraphs of some particular size of G. The ground set is precisely E, the set of edges, and $\mathcal{B}$ is the set of connected subgraphs of the desired size of G. Whereas $\mathcal{B}$ is too large to search to see if any subgraph B is a member, checking to see if B is connected and of the correct size (and therefore a member of $\mathcal{B}$) is straightforward and can be done in time linear in the number of edges in B. It is interesting to note that while testing connectivity is a linear time problem, calculating the number of connected subgraphs of G is complete for the class $\#\mathcal{P}$.

What is left is to describe how to move from basis to basis. A step consists of flipping a coin and, if it turns up heads, making a move, otherwise staying in the current basis. Assuming that a move is indeed to be made, it is effected by selecting uniformly at random an element from the ground set and an element from the current basis and replacing the latter by the former. If the new test-basis is indeed a basis, it becomes the new current basis; otherwise, the previous current basis is restored. Pseudo-code for this follows:

```
comment $B$ is the basis, $E$ is the ground set
comment rndbit() returns 0 or 1, each with probability 1/2
comment rndelt() returns an element chosen unformly at random from a given set
if (rndbit() == 0) {
    $b$ := rndelt($B$)
    $e$ := rndelt($E$)
    $B$ := $B \backslash \{b\} \cup \{e\}$
    if ($B$ is no longer a basis)
        $B$ := $B \backslash \{e\} \cup \{b\}$
}
```

In our graph example, a move would entail removing an edge from the basis and replacing it by an edge from the original graph. If the new test-basis is connected and of the correct size, it becomes the new basis; otherwise, the old basis is restored. In this way, bases are "adjacent" if they differ by one edge swap.

1.2 What's Counted

After each move (or set of moves) the simulator checks the current basis to see which ground elements are members of it, and it keeps a count of how many times each element has been present as well as how many bases have been sampled (with multiplicity). At various stages this data is output, and the user can thus see the progression over time of the ratio of number of bases (connected subgraphs in our example) sampled containing a certain element (edge) to the total number of bases sampled. The convergence of this ratio to its true value is a measure of the original convergence discussed earlier.

For the rest of this report, we assume the application being simulated is the one from our example, namely walking through the set of connected subgraphs of some graph. Indeed, the software is currently configured for this application.

2 File Formats

This section describes the formats of the input and output files used by the software.

2.1 Graph Input File

The initial input file describes some graph, either generated by the user or by some other program (see §4). This is a file of ASCII integers, the first of which is the number n of vertices in the graph. After this number, the rest of the file consists of pairs of integers which denote which edges are present. E.g. pair "2 3" says that there is an edge between vertices 2 and 3. Vertices are numbered from 0 to $n - 1$. Since the graph is undirected, only one pair need be present for each edge; i.e. "3 2" need not be (and, in fact, must not be) present in the file if "2 3" is. White space in the file is completely ignored except to separate integers.

For example, the following would be the contents of a file for the complete graph on 4 vertices:

```
4
0 1
0 2
0 3
1 2
1 3
2 3
```

Some of the programs refer to edges by numbers, in which case the edges are numbered, starting at 0, in the order they appear in the input file.

2.2 Data Output File

The other file used by the programs serves as both an output file and an intermediate file; e.g. some of the analysis programs read the output from the simulation programs. This is a binary file that contains the data as described in §1.2.

The file starts with two integers, the number of vertices n and the number of edges m, respectively, in the input file. These are passed along into the output file to save the programs that read it the work it would take to calculate them or the user the responsibility of giving the numbers to those programs.

Following these two numbers are "lines" of data. Each line contains an initial tag integer which identifies which program wrote the data and possibly information about the sampling rate under which the data was written. The tags will be fully explained in §5. Following the tag are m integers, one per edge, giving the current counts (as of the particular sample) of occurrences of the respective edges in the bases. The line then ends with one integer which is the count of bases sampled so far.

From this, one calculates that an output file containing data from s samples will consist of $2 + s(m + 2)$ integers.

3 Syntax

Most of the following sections describe component programs of the software. A word on syntax is necessary to understand them. We use the standard UNIX[1] syntax for describing the usage of a command:

- Items in **boldface** are to be entered literally.
- Items in [square brackets] are optional.
- Items in {curly braces} separated by vertical bars (|) represent a set of possible entries, one of which is to be used.
- Items in *italics* are further arguments to options.
- Items in `typewriter style` represent values to be substituted in the command line.
- Ellipses (...) denote zero or more additional arguments of the same type as what immediately precedes them.

Optional arguments may appear in any order but must all occur before any required arguments; required arguments must occur in the order in which they appear in the usage synopsis.

For example, the usage synopsis for a fictional command *repeat* which takes an optional argument **-n** that changes the default number of repetitions to its own argument x and then repeats additional command line entries that many times would be:

repeat [**-n***x*] `arg` ...

The usage synopses and argument explanations for all the commands listed in the sections below are summarized in Appendix A.

4 Graph Generation

In some cases the user will want to write a graph file manually (thus the reason for the ASCII format of the graph files). However, in most situations s/he will likely want to create a number of graphs of different sizes but of the same general structure; in these cases, s/he can easily write a program to generate the graph files. The section describes three such programs provided by the package. They all output the graph to standard output.

4.1 Random Graphs

The program *gengraph* will output a random graph with a given *edge probability*, i.e. the probability that any edge will be present. Its usage is as follows:

gengraph [**-c**] [**-e***max*] `n numer denom`

[1]UNIX is a Registered Trademark of AT&T Bell Laboratories.

It then constructs a graph of n nodes with edge probability $\frac{numer}{denom}$. If **-c** is specified, it ensures that the resulting graph is connected (if the generated graph isn't, the program will generate a new one, repeating until a connected graph is constructed.) The **-e** option specifies the maximum number of edges that the graph may have (the procedure here is similar to the case of **-c**.)

4.1.1 Complete Graphs

Note that no special program is required to generate a complete graph. To output a complete graph on n vertices one issues the command

```
gengraph n 1 1
```

4.2 Hypercubes

A *hypercube* is a graph on 2^k vertices for some value of k in which two vertices are *adjacent* (have an edge between them) if and only if their vertex indices' binary representations differ by only bit. The command *genhcube* generates a hypercube and takes the value of k as its argument. Its usage is thus:

```
genhcube k
```

4.3 Tori

A *torus* is a graph on n^2 vertices for some value of n in which, if one thinks of the vertices laid out as an $n \times n$ grid, two vertices are adjacent if and only if they are neighbors on the grid. For this purpose, the vertices on either end of a grid row or column are neighbors; i.e. the grid "wraps around" in both dimensions. One can picture a grid superimposed on a donut. The command *gentorus* generates a torus and takes the value of n as its argument. Its usage is thus:

```
gentorus n
```

5 Simulation

The programs described in this section each take as input a graph file and produce as output a data file containing the results of their simulations.

5.1 Random Walking

The main simulation tool is the program *rwalk*, which takes its name from its original purpose: simulating a random walk on a very large graph. The large graph consists of a vertex for each connected subgraph, two of which are adjacent if their respective subgraphs differ by one edge swap, as explained previously. In effect, the random process simulates a traversal of this graph, its goal being to finish at a uniformly random vertex independent of the start vertex; in effect, given an initial connected subgraph, the process constructs a random one.

By default, *rwalk* uses minimum spanning trees (i.e. connected subgraphs of size $m-1$) and samples the basis separately every n, m, m^2, and m^3 steps for an n vertex m edge graph. The initial basis is constructed via a breadth-first-search of the graph starting at vertex 0. *Rwalk* outputs a data line every 10 samples, sampling 10000 times at each "rate." Data lines are tagged 0, 1, 2, and 3, respectively, depending on the sampling rate.

The following options modify *rwalk*'s behavior:

- **-a** Rather than walking over all connected subgraphs of a fixed size, walk over all connected subgraphs. In this case, the algorithm of §1.1 is slightly modified to allow the removal and addition of edges from/to the basis, as shown further in Appendix B.

-b Output data in binary format, as described above. The default ASCII output is meant only for human interpretation for runs on small graphs.

-c*true* The true value of the ratio of bases containing a particular edge to all bases traversed is $true/1000$. This is used in conjunction with the **-e** option.

-d Output copious debugging information to standard error. Not for the light of heart.

-e*edge* Collect and output data for this edge only. Run the program until the average relative error (over all samples) of the calculated ratio for this edge against the true value (given via the **-c** option) is less than one per cent (by default; see **-r**). The output file will contain 1 for the value of m, and each line will contain only the tag, the count for the edge in question, and the basis count.

-E*edge* With **-e**, compare the ratios of the two given edges and run until the average relative error as compared against the true value (again, as given by **-c**) falls under the threshhold (modified by **-r**).

-o*spo* Change the default number of samples per output line to *spo*.

-p*plus* Add *plus* extra edges to the initial basis. Thus, unless **-a** is specified, the random walk will be over connected subgraphs of size $n - 1 + plus$.

-r*aerr* In conjunction with **-c** and **-e** (and possibly **-E**), run until the average relative error for samples taken at each rate is less than *aerr* per cent.

-R*rate* Do not output data for sampling rate *rate* (0 for n, 1 for m, 2 for m^2, or 3 for m^3). This is particularly useful if given for rate 3, as it will speed the program by letting it stop after all the samples for the finer rates are taken. One instance of this option may be given for each rate.

-sa, b, c, d Take a samples at rate n, b at rate m, c at rate m^2, and d at rate m^3. Those values omitted will be 0 (for n) or the previous given value (for the others); e.g. **-s,1000,,50** will result in 0 samples at rate n, 1000 at rates m and m^2, and 50 at rate m^3.

-t*min* Stop the program (if it hasn't stopped already) after *min* minutes.

-u Rather than traverse the connected subgraphs, traverse the disconnected subgraphs. This option implies subgraphs of all sizes. Its initial basis is the null basis.

Rwalk's usage synopsis is thus:

rwalk [**-a**] [**-b**] [**-c***true*] [**-d**] [**-e***edge*] [**-E***edge*] [**-o***spo*] [**-p***plus*] [**-r***aerr*] [**-R***rate*] [**-s**a, b, c, d] [**-t***min*] [**-u**]

5.1.1 Internal Buffer Flushing

Rwalk buffers its output so that it won't try to write an enormous amount of data in small pieces at the beginning. Due to this buffering, if the system crashes during a run, whatever data has been collected but not actually output will be lost. To avoid this, for long runs on an unreliable machine, it is advisable to flush *rwalk*'s buffer forcibly every so often. One does this by sending a terminate signal (SIGTERM) to the rwalk process. A script exists to do this periodically; see §8.3.

5.1.2 Caveat Rwalker

Rwalk was designed as a research tool, and its plethora of options is a result of the desire to simulate random walks on graphs in a variety of modes (some of which were devised after building of the software commenced). Some of the options are interdependent, e.g. the **-c** and **-e** options must be given together, and the **-r** option is meaningless without the other two. Because of the profusion of options, the program makes no attempt to ensure that the collection of options given on the command line is in fact legitimate. If some needed options are left out (e.g. **-e** given but not **-c**), *rwalk*'s behavior is undefined.

5.2 Exhaustive Searching

If the input graph is small enough, an exhaustive search can be performed to calculate the true ratios for the graph. The program *exhaust* effects this search. It takes optional **-b** and **-d** arguments, both of which have the same meaning as they do for *rwalk*. The final command line argument is the size of the connected subgraphs to search exhaustively. *Exhaust*'s usage is:

exhaust [**-b**] [**-d**] `size`

Exhaust produces only one line of output data (after the vertex and edge counts), tagged with a -1.

5.3 A Simple Monte Carlo Algorithm

The naïve Monte Carlo algorithm for random walks is to generate repeatedly subgraphs with edge probability 1/2 and, if each in turn is connected, update the ratios appropriately. Why this algorithm is not sufficient by itself is beyond the scope of this report. When the algorithm produces good data, however, it is a useful comparison again *rwalk*.

The program *monte* implements this method and realizes the following options:

-b As for *rwalk*.

-d As for *rwalk*.

-e*edge* Generate ratios for the given edge only.

-o*spo* Output a data line every *spo* steps, default 1.

-p*plus* Generate random subgraphs of $n-1+plus$ edges; the default is to generate random subgraphs of all sizes.

-s*steps* Run for *steps* steps, default 1000.

-u Maintain the ratios for disconnected subgraphs (of all sizes).

Its usage is

monte [**-b**] [**-d**] [**-e***edge*] [**-o***spo*] [**-p***plus*] [**-s***steps*] [**-u**]

Monte tags its output lines with -3's.

6 Analysis

The programs of §5 produce simulation data that must later be analyzed. Two programs are currently available to aide this process.

6.1 Extracting Data

The main quantity considered during the production of the software package was the ratio of the number of times some edge appeared in a sampled basis to the total number of bases sampled. The program *extract* synthesizes this information on a per edge per sampling rate basis from the input data and produces it on its output in a form suitable for plotting (cf. §6.2).

Extract requires two command line arguments, the first of which is the edge desired, the second of which is the sampling rate desired (the tag information produced by the simulation program). It produces on its output a sequence of pairs of integers which correspond to (x, y)-coordinates on a plot of the data being produced. The first number of each pair merely counts from 0 in increments of 1 (the x-coordinate on the plot). The second number of each pair is n such that the ratio so far of the number of times the edge has appeared in sampled bases to the number of bases sampled is $n/1000$. The y-coordinates thus range from 0 through 1000.

Extract takes one optional argument, **-e***true*, which informs the program that the true ratio for the edge being extracted is $\frac{true}{1000}$. In this case, each y-coordinate is n such that the relative error of the ratio at its sample to the true value is $n/1000$. *Extract*'s usage is

extract [**-e***true*] `edge rate`

6.2 Plotting Data

To plot the data from *extract* (cf. §6.1), a general X Window System[2] plotting utility, *xplot*, was built.

With no arguments, *xplot* takes as input a sequence of (x, y)-coordinate pairs of integers. It produces an X window which contains the plot of the coordinates, successive coordinates connected by lines unless a pair's x-coordinate is less than the previous x-coordinate, in which case a new plot begins (i.e. is overlaid atop the previous plot). *Xplot* automatically resizes and redraws the plot any time the plot's window is resized or reexposed.

Xplot's usage is

xplot [**-{c|C}***cmd*] [**-f***font*] [**-g***x, y*] [**-t***file*] [**-v**]

and its arguments are as follows:

-c*cmd* After any plot's window (cf. §6.2.1) is exposed, run the UNIX command *cmd* via the shell (waiting for the *cmd* to exit unless it is put into the background). This is useful for invoking commands that, for example, print the contents of a designated window, thus allowing some mechanism for obtaining hard copies of plots.

-C*cmd* Like **-c**, only exit *xplot* after *cmd* runs.

-f*font* With the **-t** option, use *font* for the title font.

-g*x, y* Overlay a grid on the plot. Vertical grid lines occur every x ticks, and horizontal lines every y ticks. If either is missing or 0, no lines in the respective orientation will be generated; if x is missing, a comma (,) must still precede y. See §6.2.2 to change the grid interactively.

-t*file* Prints the contents of *file* at the upper left of the plot.

-v Give verbose output in the command window as to what *xplot* is doing.

6.2.1 Button Commands

In this and the following sections, *current point* refers to the *xplot* screen coordinate of the mouse over the *current plot*, that plot which has most recently been exposed. *Xplot* responds to mouse button clicks as follows:

1. Clicking button 1 once at any two locations on the current plot will cause *xplot* to produce a new plot (in a new window) of the region of the current plot between the x-coordinates of the points over which the mouse sat during the button clicks. Whatever grid and title were on the old plot will be placed on the new plot at well. This option is useful for zooming in on small areas of a plot. It only functions properly for "simple" plots, i.e. those which have no overlaid sections.

2. Clicking button 2 will cause *xplot* to output to its command window the sample number and y-coordinate of the current point.

3. Clicking button 3 will cause *xplot* to destroy the current plot. If this was the original plot, *xplot* itself will exit (destroying any other plots as well).

[2]X Window System is a Trademark of the Massachusetts Institute of Technology.

6.2.2 Keyboard Commands

Xplot also accepts one keyboard command in any plot window. This is the **g** command, entered as follows:

$$\mathbf{g}x, y$$

It causes the grid over the particular plot to be redrawn as per the parameters given. The meaning of those parameters is the same as for the **-g** argument to *xplot.*

6.3 Calculating Convergence

The program *arerr* takes as input a data file and calculates at which sample each rate for which data is presented converged to some specified value. Its usage is

```
arerr edge realval goal1 ... goaln
```

Edge is the edge for which the convergence data is to be collected; *realval* is the true value of the ratio for this edge. The *goal* arguments (given in per cent) must be given in decreasing order. *Arerr* will then determine and print a summary of the samples at which each rate achieved each goal.

7 Examples

Here we give two examples for using the above programs. The first will perform a random walk on a graph in *graphfile* of 1000 samples at sampling rate m, outputting data every sample. Assuming the graph has six vertices, the random walk will be over spanning trees (connected subgraphs of five edges). It will then perform a similar naïve Monte Carlo simulation and compare the results of each for edge 0 (note some of the command line options are unnecessary, as they simply pass the default parameters):

```
rwalk -b -R0 -R2 -R3 -s1000 -o1 <graphfile >rwalk.data
monte -b -s1000 -o1 -p5 <graphfile >monte.data
(extract 0 1 <rwalk.data ; extract 0 -3 <monte.data) | xplot
```

The second example will run a random walk on the same graph but for connected subgraphs of size eight and only for edge 3, whose true ratio is known to be 767/1000; it will execute the random walk, outputting data every five samples, until the average relative error of all the samples is below two per cent and then plot the results for sampling rate m^2:

```
rwalk -b -c767 -e3 -o5 -p3 -r2 <graphfile | extract -e767 0 2 | xplot
```

Note that the edge argument to *extract* is 0 and not 3; this is because *rwalk* run with the **-e** flag produces output for only the one edge, and thus as far as *extract* is concerned, there is only one edge, number zero, in the graph.

8 Handy Utilities

This section describes some smaller programs and shell scripts contained by the package. They were written to implement some of the more repetitive tasks involved in simulating large sets of graphs. Those utilities implemented as shell scripts have a colon (:) as the first character of their names. Some shell scripts have variable definitions towards the beginning instructing them whence to read graph files and where to put data files.

8.1 Binary Files

The simulation programs of §5 write binary data files as output. The three programs described here deal with these files on a lower level than that of data generation and analysis.

8.1.1 Writing Them Manually

It is occasionally desirable to write a binary data file manually, without aid of one of the simulators. For example, a regularly structured graph (such as a complete graph, torus, or hypercube) might have well known edge-occurrence-to-sampled-bases ratios for all sizes of connected subgraphs. An "output file" for these correct values can be written by hand to be used by some of the automated analysis tools of §8.5. The program *bwrite* takes as input a list of white-space-separated integers and writes them in binary format to its output.

8.1.2 Converting Them

Sometimes a lot of simulations will be run simultaneously, one or two each on a different machine. Some of these machines may have different byte orders; e.g. machines made by Digital Equipment Corporation have reverse byte order compared to those of Sun Microsystems. The program *cvt* reads a binary data file on standard input and reverses the byte order of each 32-bit word, writing the updated data to standard output.

8.1.3 Fixing Them

Experience with running long simulations provides anecdotal evidence that, when the output files actually reside on another machine (via a network file system), the likelihood is significant that a few data samples might be lost or garbled. (This may, of course, simply be a symptom of the actual network used for the trials.) To cope with this unpleasantness there exists a program *fixdata* which attempts to massage partially garbled data files by removing problem samples. Its takes a data file on input and writes a cleaned version to its output, its usage being:

fixdata [**-r***rate*] ... [**-s***skip*] ...

The **-r***rate* argument tells *fixdata* that *rate* is present as a tag in the data file (more than one such argument can be given); the **-s***skip* argument tells *fixdata* to remove sample *skip* (counting from 0). The latter argument might be used in the case where *xplot* reveals a spurious sample (a large spike, e.g.) whose sample number can be isolated by zooming in and using button 2 (cf. §6.2.1.)

Fixdata works by reading samples from the input and checking to ensure that each tag is valid. Upon detecting an invalid tag, the program searches for a valid tag starting at the next integer after the bad tag, throwing out integers until it discovers a valid tag. Usually, *fixdata* is invoked twice: once with a set of **-r** options to remove garbaged samples and then again, after an *xplot*, to remove any data spikes that may have been introduced. This scheme is of course not the only one possible, but its simplicity and observed ability to cope with the type of data failures encountered make it palatable. *Fixdata* prints informative messages about its actions to the standard error output.

8.2 Setting Up Complete Graphs

The script *:Kwrite* prepares data files for a complete graph. Its usage is

:Kwrite `n`

where n is the number of vertices in the graph. *:Kwrite* constructs data files with basenames (file names in the data directory, as defined in the script) "pe," where $p \in [0..m-n]$ ($m = n(n-1)/2$ is the number of edges) is the number of extra edges (above $n-1$, the number needed for minimum spanning trees) in the graph represented by that data file. The data files each contain one line of data, tagged by -1 (like those output by *exhaust* — cf. §5.2), which represents the true ratio values for each edge.

8.3 Flushing *Rwalk*'s Buffers

As mentioned in §5.1.1, upon receiving the SIGTERM signal, *rwalk* flushes its output buffers. The script *:doflush* has usage

:doflush `pid`

and sends the SIGTERM signal to process *pid* (assumed to be an *rwalk* process) every fifteen minutes. The motivation behind this method of periodic flushing is twofold: (1) Any periodic flushing method will entail the use of some signal, either sent from the outside (as in the current method) or generated internally by *rwalk* (e.g. a SIGALRM alarm signal); (2) a script is used to send the SIGTERM signals periodically rather than having *rwalk* realize another command line argument for this purpose to allow the user total freedom over the frequency of flushing (e.g. irregular flushing intervals might be desired, making implementation of flushing via command line argument impractical.)

8.4 Automated Graph Simulation

The script *:dograph* runs an *rwalk* simulation on an input graph for each valid subgraph size. Its usage is

dograph [-e] `arch graph rwalk-opts ...`

Arch is the machine architecture (e.g. vax[3], sun3, etc.) under which *rwalk* is running, *graph* is the basename of the graph file, and *rwalk-opts* are additional options for each *rwalk* command. *:Dograph* runs an *rwalk* command for each valid subgraph size (from $n-1$ through $m-1$), putting the output of each walk in a data file of basename "*p*r," where *p* is as in §8.2. If the **-e** option is given, *:dograph* also runs *exhaust* on the input graph for each valid subgraph size, putting the output in data files of basename "*p*e."

8.5 Automated Plotting ...

Once a graph has been simulated, two methods of automated plotting are available.

8.5.1 ... of One Simulation

:Doplot expects the *p*r and *p*e files described in §8.2,8.4 to exist. Its usage is

:doplot [-p] `graph plus rate` **{b|w}**

where *graph* is the basename of the graph input file, *plus* is the number of additional edges of the desired simulation's subgraph size, and *rate* is the sampling rate from which the data is to be taken. The final argument instructs *:doplot* to plot the data for the best/worst edge (that edge with the best/worst relative error to the true ratio value). *:Doplot* calculates which edge to plot and then invokes *xplot* (cf. §6.2) to plot the relative error of the selected edge. Given the **-p** option, *:doplot* invokes *xplot* with a command line argument that causes a window printing program to run after the plot is exposed, allowing the user to obtain a hard copy of the plot.

8.5.2 ... of a Set of Simulations

Built around *:doplot* (cf. §8.5.1), *:exgraph* plots data for a number of different subgraph sizes. Its usage is

:exgraph `graph`

where *graph* is the basename of the graph input file. *:Exgraph* invokes *:doplot* to plot the best edge from the simulation of subgraph size $n-1$ and the worst edges from the simulations of subgraph sizes $n-1+(i*d)$ where $d=(m-n)/4$ and $i \in [1..3]$. It does this for each sampling rate of *rwalk*; thus, *:exgraph* generates a total of sixteen plots. *:Doplot* is called with the **-p** option, so that hard copies of these plots can be produced.

[3]VAX is a Trademark of the Digital Equipment Corporation

Appendix A — Summary of Command Line Arguments

A table explaining the command line arguments to all of the programs described herein follows. The “Section” column indicates in which section the command is fully documented; an asterisk (*) in the “Opt” column indicates that the designated argument is optional. Optional arguments may appear in any order but must all occur before any required arguments; required arguments must occur in the order in which they appear in the usage synopsis.

Command	Section	Arg	Opt	Meaning
gengraph	§4.1	-c	*	Require graph to be connected
		-e*max*	*	Allow no more than *max* edges in graph
		n		Construct graph of n vertices
		numer		Use edge probability ...
		denom		... $\frac{numer}{denom}$
genhcube	§4.2	k		Construct hypercube of 2^k vertices
gentorus	§4.3	n		Construct torus of n^2 vertices
rwalk	§5.1	-a	*	Simulate over all subgraph sizes
		-b	*	Output data in binary format
		-c*true*	*	With -e, true ratio for edge is $true/1000$
		-d	*	Output lots of debugging information to stderr
		-e*edge*	*	Collect and output data only for edge *edge*
		-E*edge*	*	With -e, compare ratios of the two given edges
		-o*spo*	*	Output one data line every *spo* samples
		-p*plus*	*	Construct initial basis of size $n - 1 + plus$
		-r*aerr*	*	With -c and -e, run until average relative error is less than $aerr\%$
		-R*rate*	*	Don't output data for sampling rate *rate*
		-sa, b, c, d	*	Take a samples at sampling rate n, etc.
		-t*min*	*	Stop simulation after *min* minutes
		-u	*	Traverse disconnected subgraphs; implies -a
exhaust	§5.2	-b	*	Output data in binary format
		-d	*	Output lots of debugging information to stderr
		size		Run over subgraphs of *size* edges
monte	§5.3	-b	*	Output data in binary format
		-d	*	Output lots of debugging information to stderr
		-e*edge*	*	Generate ratios for given edge only
		-o*spo*	*	Output one data line every *spo* samples
		-p*plus*	*	Generate random subgraphs of $n - 1 + plus$ edges (all sizes by default)
		-s*steps*	*	Run for *steps* steps
		-u	*	Maintain ratios for disconnected subgraphs of all sizes
extract	§6.1	-e*true*	*	The true ratio for the edge is $true/1000$
		edge		Extract data for edge *edge*
		rate		Extract data for sampling rate *rate*
xplot	§6.2	-c*cmd*	*	Run *cmd* via the shell after exposing each plot
		-C*cmd*	*	Like -c, but exit after running the command
		-f*font*	*	Print title in *font*
		-gx, y	*	Overlay a grid every x x-ticks and y y-ticks
		-t*file*	*	Print contents of *file* as a plot title
		-v	*	Give verbose output as to program's actions
arerr	§6.3	edge		Edge for which to calculate convergence
		realval		Real value for this edge
		goal ...		Successive convergence goals
bwrite	§8.1.1			NO ARGUMENTS
cvt	§8.1.2			NO ARGUMENTS
fixdata	§8.1.3	-r*rate*	*	Expect sampling rate *rate* in data file
		-s*skip*	*	Skip sample *skip*
:Kwrite	§8.2	n		Generate true data files for K_n
:doflush	§8.3	pid		Periodically flush *rwalk* process *pid*'s buffers
:dograph	§8.4	-e	*	Also run *exhaust* on input graph
		arch		*Rwalk* running on machine with architecture *arch*
		graph		Basename of input graph file
		rwalk-opts		Any additional arguments to *rwalk* (may be empty)
:doplot	§8.5.1	-p	*	Run *xplot* with argument to print plots
		graph		Basename of input graph file
		plus		Plot selected edge for simulation of $(n - 1 + plus)$-edge subgraphs
		rate		Plot selected edge for sampling rate *rate*
		b|w		Select edge with best/worst relative error to true ratio value
:exgraph	§8.5.2	graph		Basename of input graph file

Appendix B — Random Walks over Subgraphs of All Sizes

The **-a** and **-u** options to *rwalk* cause the program to simulate a random walk over subgraphs of all sizes. In this case, the general procedure of §1.1 must be slightly modified to allow the basis to grow and shrink at each move. To effect this, the ground set and basis each contain a dummy element. Let B be the randomly selected basis element and G the randomly selected ground element of any move. If G but not B is the dummy element (and the newly constructed test-basis is a valid basis) then the basis shrinks by one element; if B but not G is the dummy element then the basis grows; if neither B nor G is the dummy element then the basis remains the same size; if both B and G are dummy elements then no move takes place.

The pseudo-code of §1.1 remains the same if one considers the dummy element to be the null member of any set.