Report Number: WUCS-90-37

1990-10-01

DNA Mapping Algorithms: The DNA Simulator

Authors: Will Gillet and John Heidemann

This report documents the intent and use of a suite of programs for simulating the production of DNA restriction fragment data, as might come from the biological laboratory doing work in DNA mapping. This suite includes programs for (a) creating a random strand of DNA, (b) creating random clones given a strand of DNA, (c) taking a clone and applying a restriction enzyme to create restriction fragments, and (d) creating a nucleotide map of how the clones relate to one another within the original DNA strand. Besides this fundamental software, there are a number of a programs for introducing different forms of random error (nre, nce) intro the restriction fragments produced, and aggregating and "filtering" the clones in different ways to select those with appropriate properties.

DNA MAPPING ALGORITHMS: THE DNA SIMULATOR

Will Gillett and John Heidemann

WUCS-90-37

October 1990

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

This work was supported by the James S. McDonnell Foundation under Grant 87-24.

ABSTRACT

This report documents the intent and use of a suite of programs for simulating the production of DNA restriction fragment data, as might come from a biological laboratory doing work in DNA mapping. This suite includes programs for (a) creating a random strand of DNA, (b) creating random clones given a strand of DNA, (c) taking a clone and applying a restriction enzyme to create restriction fragments, and (d) creating a nucleotide map of how the clones relate to one another within the original DNA strand. Besides this fundamental software, there are a number of programs for introducing different forms of random error (`nre`, `nce`) into the restriction fragments produced, and aggregating and "filtering" the clones in different ways to select those with appropriate properties.

TABLE OF CONTENTS

LIST OF FIGURES

1. Introduction

This report documents the intent and use of a suite of programs for simulating the production of DNA restriction fragment data, as might come from a biological laboratory doing work in DNA mapping. This suite includes programs for (a) creating a random strand of DNA (`dnamake`), (b) creating random clones given a strand of DNA (`dnaclone`), (c) taking a clone and applying a restriction enzyme to create restriction fragments (`dnafrag`), and (d) creating a nucleotide map of how the clones relate to one another within the original DNA strand (`dnamap`). Besides this fundamental software, there are a number of programs for introducing different forms of random error (`nre`, `nce`) into the restriction fragments produced, and aggregating and "filtering" the clones in different ways to select those with appropriate properties.

This software is decomposed into a number of stand-alone programs so that each can be used separately, or in concert with one another. For example, the standard scenario is to create a random DNA strand (`dnamake`), produce random clones (`dnaclone`), produce fragments for those clones (`dnafrag`), introduce random error into the fragments of the clones (`nre`), and then supply that fragment information to whatever mapping activities are applicable. However, the DNA used to produce clones need not have been produced by `dnamake`. For instance, it may have been produced by hand or supplied by any other external means, such as extraction from a database. Before the fragment information is produced for the clones, the user can apply a number of different "filters" for extracting the clones desired. It may also be that normal random error is desired (`nre`) in a specific situation or that normal correlated error (`nce`) is desired. All of these programs can be applied independently as desired.

The general style used by the software is somewhat eclectic; this is caused by the historical evolution of the software. Specifically, input to most of the programs can be done interactively at the keyboard, or the input can be supplied by command line arguments. In the command line form, the standard `-letter` notation is used to supply the values of the parametric

input. This form of input is most often used when the software is being used in a batch environment, e.g., in a long script of activities. For any *required* input arguments that are not supplied on the command line, the software will interactively request the input from the user. In this interaction, a line is printed requesting the desired information (along with a default). The user may either enter the information, or enter RETURN (in which case the default is used).

In most cases a file is written, indicating the values of the input arguments which were used. This file is useful for at least two reasons. First, it documents the parameters from which the software produced its output. Second, the file is written in such a format that it can be used as direct input (using the `command < filename` option) if exactly the same parameters are desired in a subsequent application of the software.

The remainder of this report will present the intent and use of each of the specific programs available. A running example is used throughout the report to supply continuity for comparison across different forms of processing.

2. Basic Simulation

2.1. Dnamake

The program `dnamake` creates a random strand of DNA of a specific length and a specific ratio of A-T/C-G base pairs. Obviously, an underlying random number generator is used. Its original seed can be set (via a command line argument) or the seed can be randomly selected based on the state of the operating system (the default if not supplied on the command line).

The command line arguments are as follows:

```
Command Line Options:
      -s <int>    set random number generator seed
      -l <int>    length of dna strand to be created
      -r <string> rootfilename
      -p <int>    percentage
```

The above presentation was created by executing `dnamake -h`. Each of the programs for which command line arguments are applicable has a `-h` (help) argument which prints out a legend of what command line arguments are available.

An interactive session might look something like the following:

```
dna% dnamake
Starting dnamake
Root filename (DNA/test)? YAC/yac
DNA length (10000)? 100000
Percent of DNA that is A or T (50)? 35
Generating DNA of length 100000...done.
Ending dnamake
dna%
```

Any argument supplied on the command line will not be interactively requested from the user; any *required* argument not supplied on the command line will be interactively requested from the user. The standard mode of using this software is to be either completely command line driven or completely interactive; however, it is possible to mix the two modes.

The file produced echoing the parametric inputs will be (in this case) `YAC/yac.dnamake.input`. Its content is:

```
YAC/yac
100000
35
```

The DNA strand of 100000 base pairs is placed in the file (in this case) `YAC/yac.dna`. In general the *rootfilename* is used as a base name in all of the software. Different `.extensions` are appended to this name and used as the file name for different output data.

2.2. Dnaclone

The program `dnaclone` is used to create a set of random clones from a strand of DNA. This is done by simulating a partial digestion of the original DNA strand with a restriction enzyme, usually one which recognizes a short sequence (often 4 base pairs long). A partial digestion of a DNA strand means that when the DNA is digested, the restriction enzyme is not allowed to cleave the DNA at *every* restriction site, but instead it is only allowed to cleave at a *small percentage* of the restriction sites (randomly selected). In the biological world, this is achieved by limiting the time that the restriction enzyme is allowed to interact with the DNA and chemical inhibitors.

After the DNA has been randomly cleaved, specific substrands of the DNA (called clones) are randomly selected for fragmentation. In the biological world, this essentially is done by dilution. In the simulation, a random number generator is used to select random restriction sites (at which the cloning restriction enzyme is allowed to cleave the DNA) and then randomly select a clone which corresponds to the substrand of DNA occurring between two randomly selected restriction sites.

In the biological world, the clone selection process puts certain restrictions on the size of the clones that can be produced. This is caused by the packaging mechanism used by the λ-vector, by which the clones are reproduced. In the simulation, all clones are produced whether or not they fall within these bounds; however, they are "marked" (good, long, and short) to indicate whether or not they are appropriate to use.

In general, in the activity of DNA mapping, there must be enough clones present in the clone pool so that the entire original DNA strand (or at least most of it) is "covered" by the clones. Since the mapping is performed by discovering overlap between the clones, it is actually necessary to have enough clones to "cover" the original strand several times, and because of the statistical possibility of having many clones cluster from one region and only a few from another

region, it is desirable to have a high redundancy factor for the clones, usually in the range of 5 to 10. The redundancy factor is the average depth of coverage of the clones over the original strand. Specifically, if the original DNA strand to be mapped has X base pairs, a redundancy factor of 5 implies that the sum of the lengths of all clones must be approximately 5X. Sometimes it is known what minimal number of clones is desired; other times only a general redundancy factor is known. The software allows both of these requirements to be input.

The command line arguments are as follows:

```
Command Line Options:
      -s <int>    set random number generator seed
      -r <string> rootfilename
      -e <string> cloning enzyme sequence
      -m <int>    minimum number of clones to generate
      -f <int>    redundancy factor
      -n <int>    numerator of chance of clone site being cleaved
      -d <int>    denominator of chance of clone site being cleaved
      -l <int>    lower limit on clone length to be produced
      -u <int>    upper limit on clone length to be produced
      -a          process both orig and like clones in redun anal
      -o          process all clones as if they were original
```

An interactive session might look something like the following:

```
Starting dnaclone
Root filename (DNA/test)? YAC/yac
Clone restriction enzyme (atat)? aatt
Minimum number of clones to generate (25)? 32
Minimum redundancy factor (an integer) (5)? 7
Clone cleavage chance (15/1000)?
Minimum clone length (in base pairs) (0)? 10000
Maximum clone length (in base pairs) (+inf)? 22000
Examining dna...length 100000.
Locating clone cleavage sites...84 sites.
Generating at least 32 clones, redundancy at least 7x...289 are unique.
Ending dnaclone
```

The input DNA is extracted from the file *rootfilename*.dna, and the output (a description of the clones) is placed in the file *rootfilename*.clones. The positions of the clone sites are written into the file *rootfilename*.clone.sites. A sample of the output placed in the file

YAC/yac.clones is shown below:

```
83810    96893     69   80   original       0  good
    0     1625     -1    0   original 1000000 short
22926    92897     24   74   original 1000001  long
10839    16545     14   17   original 1000002 short
    0    95554     -1   78   original 1000003  long
    0    25369     -1   28   original 1000004  long
59922    92082     53   72   original 1000005  long
61284   100000     54   84   original 1000006  long
21774   100000     22   84   original 1000007  long
    0    21774     -1   22   original       1  good
29349    49484     30   42   original       2  good
    0    19221     -1   20   original       3  good
61284   100000     54   84       like 1000006  long
50268   100000     45   84   original 1000008  long
```

As can be seen, the data in this file are organized into 7 columns, one row for each clone. The first two columns specify the start and end position of the clone in terms of the index into the base pair sequence of the original DNA strand. The third and fourth columns specify the same information, but the index used indicates the index of the restriction site (defined by the cloning restriction enzyme). The fifth column indicates whether or not the clone produced is an original clone (never before produced by the simulation) or one which has been previously generated. The sixth column is essentially a unique name (or identifier) for the clone. Note that *original*, *good* clones are numbered sequentially starting at 0. *Like* clones "point" to their predecessor. *Bad* clones (i.e., *long* and *short* clones) are numbered sequentially starting at a very large number (1,000,000). The seventh column indicates whether the clone falls into the length specifications given as input parameters. In this running example, 388 clones were produced. 287 of these clones were unique, and only 43 of these were *good* (i.e., neither short nor long).

2.3. Dnafrag

The program `dnafrag` is used to produce the restriction fragment lists that would be produced by applying a set of restriction enzymes to a clone in a complete digestion (i.e., a digestion where all restriction sites are cleaved). The set of clones to which this complete

digestion is applied are the *good, original* clones found in the `.clones` file. "Partial" fragments (those which are on the end of the clone and do not actually occur between two restriction sites) are not reflected.

In the biological world, the lengths of the fragments are determined by a process known as electrophoresis. In this process, the information for fragments of length less than 400 base pairs and longer than 8000 base pairs is effectively lost. The simulator allows the specification of upper and lower fragment length boundaries.

This software is somewhat primitive, and does not recognize that DNA consists of complementary strands. Thus, in order to specify one *conceptual* restriction enzyme, two *physical* restriction enzymes must be specified to the software (one being the complement of the other). For instance, in order to specify one restriction enzyme, say *Hin*dIII, the two restriction enzymes, say `aagctt` and `ttcgaa`, must be input.

The command line arguments are as follows:

```
Command Line Options:
      -r <string> rootfilename
      -e <int>    # of enzymes to use
      -0 <string> first restriction enzyme
      -1 <string> second restriction enzyme
      .
      .
      .
      -9 <string> tenth restriction enzyme
      -l <int>    lower limit on clone length to be produced
      -u <int>    upper limit on clone length to be produced
      -a          process both (all) orig and like clones
```

An interactive session might look something like the following:

```
dna% dnafrag
Root filename (DNA/test)? YAC/yac
Number of fragment restriction enzymes (1)? 2
Fragment restriction enzyme #1 (aaattt)? aagctt
Fragment restriction enzyme #2 (cccggg)? ttcgaa
Minimum fragment size (0)? 400
Maximum fragment size (+inf)? 8000
Examining dna...length 100000.
Locating frag cleavage sites...12 sites.
Locating frag cleavage sites...12 sites.
Reading clone information...done.
Determining fragment lists for 43 unique clones...done.
```

The input for `dnafrag` is extracted for the file *rootfilename*`.clones`, and the output is placed in files of the form *rootfilename*`.frag.`*n*, where *n* is the "name" of the clone extracted from the *rootfilename*`.clones` file. The positions of the restriction sites are written into a file named *rootfilename*`.frag.sites`. A sample of the output placed in `YAC/yac.frag.1` is shown below:

```
2022 513
2318 2535
3738 4853
1972 8591
1377 10563
```

Note that there are two columns in this file. The first column indicates the length of the restriction fragment; the second column indicates the position of the start of the fragment (in terms of the index of the base pair). This second column can be used in two ways. First, it gives each fragment in the original DNA strand a unique "name" so that the same fragment occurring in different clones can be determined uniquely. Second, it orders the fragments (for instance, when the fragments are sorted in some other order than they actually appear in the clones) within the original DNA strand.

2.4. Dnamap and Dnamapcomp

The program `dnamap` is used to print a nucleotide map of how the clones relate to one another and the original DNA strand. The restriction sites associated with (potential) clone ends and (potential) fragment ends within the clone are marked.

The command line arguments are as follows:

```
Command Line Options:
      -r <string> rootfilename
      -c <string> clonefilename (full name)
```

An interactive session might look something like the following:

```
dna% dnamap
Root filename (DNA/test)? YAC/yac
```

A portion of the output produced by `dnamap` is shown in Figure 1. Note that in this partial output, one clone end (designated by the C) and one restriction fragment end (designated by the F) occur. The number to the left is the index of the nucleotide; the letter after the colon (:) indicates which nucleotide is present; the vertical lines with numbers interspersed represent the presence of a clone. The number interspersed in the line is the clone number extracted from the `.clones` file. Note that clones 27 and 32 start in this region of the map.

Input to `dnamap` comes from a variety of files. The base pair sequence comes from the file *rootfilename*`.dna`. The (potential) clone end information comes from the file *rootfilename*`.clone.sites` (this is a file produced by `dnaclone` behind the scenes). (The `-c` option is used to eliminate some of the restriction sites marked as potential clone ends. Instead of *all* potential clone ends being marked, only those which actually occur as clone ends in the designated clone file are marked.) The extent of each clone is extracted from the file *rootfilename*`.clones`. The (potential) fragment end information comes from the file *rootfilename*`.frag.sites` (this is a file produced by `dnafrag` behind the scenes). Output from

```
2525: a       1     |     |     |     |     9     |
2526: g       |     |     |     |     8     |     |
2527: c       |     |     |     3     |     |     |
2528: g       |     |    29     |     |     |     |
2529: g       |    12     |     |     |     |    16
2530: c       1     |     |     |     |     9     |
2531: c       |     |     |     |     8     |     |
2532: t       |     |     |     3     |     |     |
2533: t       |     |    29     |     |     |     |
2534: c       |    12     |     |     |     |    16
2535: g  F    1     |     |     |     |     9     |
2536: a       |     |     |     |     8     |     |
2537: a       |     |     |     3     |     |     |
2538: t  C    |     |    29     |     |     |     |    27     |
2539: t       |    12     |     |     |     |    16     |     |
2540: g       1     |     |     |     |     9     |     |     |
2541: a       |     |     |     |     8     |     |     |     |
2542: g       |     |     |     3     |     |     |     |    32
2543: c       |     |    29     |     |     |     |    27     |
2544: g       |    12     |     |     |     |    16     |     |
2545: c       1     |     |     |     |     9     |     |     |
2546: c       |     |     |     |     8     |     |     |     |
2547: c       |     |     |     3     |     |     |     |    32
2548: c       |     |    29     |     |     |     |    27     |
2549: a       |    12     |     |     |     |    16     |     |
```

Figure 1: Example Output from dnamap

dnamap goes to stdout.

The output from dnamap is extremely voluminous; one line is printed for each nucleotide in the original DNA strand. Thus, if there are 100,000 base pairs in the DNA strand to be mapped, there are 100,000 lines in the output; it is very difficult to extract useful information from such a detailed map. Thus, a second piece of software (dnamapcomp) has been produced to compress this type of map.

The program dnamapcomp is used to compress this large base pair map by selecting and retaining only those lines (and a small region around them) which constitute a "significant change" in the map, i.e., a clone end or a fragment end. A portion of a compressed map, in the same region as that shown is Figure 1, is shown in Figure 2. Note the large gaps in index number at the left. Only regions of interest (change) are retained. The uncompressed map of

```
2533: t       |     |    29     |     |     |     |
2534: c       |    12     |     |     |     |    16
2535: g   F   1     |     |     |     |     9     |
2536: a       |     |     |     |     8     |     |
2537: a       |     |     |     3     |     |     |
2538: t   C   |     |    29     |     |     |     |    27     |
2539: t       |    12     |     |     |     |     |    16     |     |
2540: g       1     |     |     |     |     |     9     |     |     |
2898: a       |     |    29     |     |     |     |     |    27     |
2899: a       |    12     |     |     |     |     |    16     |     |
2900: t   C   1     |     |     |     |     |     9     |     |     |     |    42
2901: t       |     |     |     |     |     8     |     |     |     |    17     |
2902: c       |     |     |     |     3     |     |     |     |    32     |     |
3329: a       |    12     |     |     |     |     |    16     |     |     |     |
```

Figure 2: Example Output from dnamapcomp

this DNA strand contains 100,000 lines. The compressed map contains only 538 lines.

The command line arguments for `dnamapcomp` are as follows:

```
Command line options:
      -c[number] print number lines around each line that marks a
            clone boundary.  Default is 2.
      -f[number] print number lines around each line that marks a
            fragment boundary.  Default is 2.
      -m[number] search for clone or fragment boundary marker up
            to the numberth column.  Default is 16.
```

The program `dnamapcomp` is a filter in the classical UNIX sense. Input comes from `stdin`; output goes to `stdout`.

3. Fragment Manipulation

There are a number of programs available for massaging the `.frag` files and the fragments in them. In the activity of extracting the data during the process of electrophoresis, the fragments become sorted by their length. Since this is the normal way that data will be obtained, it is not appropriate for the mapping software to know what the actual order of the fragments is in the clone. Thus, there is software to sort the simulated restriction fragments by length.

In measuring the length of the restriction fragments after electrophoresis has taken place, the standard problem of measurement error is introduced into the data. There is software for introducing such measurement error to the simulated lengths in several different ways.

3.1. Addnums and Addnums_and_sort

In general, most of the software requires a header line in the `.frag` files indicating how many fragments are present. Sorting can also be performed during this process. The shell file `addnums` simply adds a header line to the file indicating the number of fragments present. The shell file `addnums_and_sort`, in addition, performs a sort during the process.

There is no command line mode for this software. An interactive session might look something like the following:

```
dna% addnums_and_sort
starting addnums_and_sort
root pathname?YAC
YAC
run id?yac
yac
filetype?frag
frag
YAC/yac.frag.0
changing YAC/yac.frag.0
changing YAC/yac.frag.1
changing YAC/yac.frag.2
changing YAC/yac.frag.3
        .
        .
        .
changing YAC/yac.frag.40
changing YAC/yac.frag.41
changing YAC/yac.frag.42
ending addnums_and_sort
```

This software concatenates together the three arguments of *rootpathname*, *runid*, and *filetype*. To this it appends an integer, starting at 0, and uses this as the name of a file to be manipulated. Output is returned in the same file from which the input was extracted. The

integer is incremented by one (1) and the process is attempted again. The process ends when the file corresponding to the file name constructed does not exist.

An example of the output produced by `addnums_and_sort` is the new content of the file YAC/`yac.frag.1`, shown below.

```
N= 5
3738 4853
2318 2535
2022 513
1972 8591
1377 10563
```

Note the new header line (`N= 5`) and that the restriction fragments are sorted in descending order, by length.

3.2. Introducing Error

There are two programs which allow the introduction of measurement error: `nre` (normal random error) and `nce` (normal correlated error). Each introduces errors, based on a normal distribution, into the fragments of a `.frag` file. Each can be used independently of the other or in concert with the other.

The command line arguments for `nre` are as follows:

```
Command Line Options:
      -s <int>    set random number generator seed
      -r <string> rootdirectory
      -i <string> runid
      -p <float>  percent spread
```

An interactive session of `nre` might look something like the following:

```
dna% nre
Root directory (DNA)? YAC
Run id (test)? yac
Percent (1.000000)? .75
root_runid is YAC/yac
NOTE:  Can't open old frag file YAC/yac.frag.43
sd1,2,3 and frag_cnt_tot are 89.000000, 127.000000, 132.000000, 132
STAT:  Errors falling within 1, 2 and 3 sd's:
       67.424240, 96.212120, 100.000000
DONE
```

The program `nre` introduces normal random error into the fragments with a standard deviation which is a percentage of the length of the specific fragment; this percentage is an input parameter. The program `nce` introduces error across all the fragments of the clone in a correlated way. Specifically whatever normal error is selected for insertion is introduced to *all* of the fragments of the clone (as a multiplicative factor).

Input to `nre` come from the `.frag` files. Again, numbering starts at 0, and processing ends when the file corresponding to the generated file name does not exit. Output from `nre` is placed into corresponding `.rfrag` files.

After applying `nre` to the data previously presented, the file `YAC/yac.rfrag.1` might contain the following.

```
N=5
3733 4853
2315 2535
2043 513
1978 8591
1362 10563
```

The command line arguments for `nce` are as follows:

```
Command Line Options:
      -s <int>    set random number generator seed
      -r <string> rootdirectory
      -i <string> runid
      -p <float>  percent spread of corr error
```

An interactive session of nce might look something like the following:

```
dna% nce
Root directory (DNA)? YAC
Run id (test)? yac
Percent spread of correlated error (1.000000)? .75
root_runid is YAC/yac
NOTE:  Can't open old frag file YAC/yac.frag.43
sd1,2,3 and clone_cnt_tot are 33.000000, 43.000000, 43.000000, 43
STAT:  Errors falling within 1, 2 and 3 sd's:
      76.744186, 100.000000, 100.000000
DONE
```

Input to nce comes from the .frag files. Again, numbering starts at 0, and processing ends when the file corresponding to the generated file name does not exit. Output from nce is placed into corresponding .crfrag files.

After applying nce to the data previously presented, the file YAC/yac.crfrag.1 might contain the following.

```
N=5
3749 4853
2324 2535
2027 513
1977 8591
1381 10563
```

4. Clone Manipulation

There are a number of programs for manipulating the .clones file and aggregating the .frag files to which the remaining clones correspond. Specifically, some important activities are to: (a) separate clones into *good* clones and *bad* (i.e., *long* and *short*) clones (separate_clones), (b) sort the clones in terms of their occurrence in the original DNA strand (sort_clones), (c) determine which of the clones are subclones of one another (dnasubclone), and (d) concatenate .frag files together based on a specific set of clones of interest (dnamergeclone).

4.1. Separate_clones

The shell script `separate_clones` separates the clones in a `.clones` file into *good* clones and *bad* clones.

There is no command line mode for `separate_clones`. An interactive session of `separate_clones` might look something like the following:

```
dna% separate_clones
Starting separate_clones
filename?YAC/yac.clones
YAC/yac.clones
Ending separate_clones
```

Input comes from the file *filename*. Output is placed in two files: *filename*.`good` and *filename*.`bad`. Given the data of the running example, the 43 good clones will be placed in the file `YAC/yac.clones.good`, and the 345 bad clones will be placed in the file `YAC/yac.clones.bad`. The content of `YAC/yac.clones.good` is shown in Figure 3.

4.2. Sort_clones

The shell script `sort_clones` sorts the clones in the order in which they appear in the original DNA strand.

There is no command line mode for `sort_clones`. An interactive session of `sort_clones` might look something like the following:

```
dna% sort_clones
filename?YAC/yac.clones.good
```

Input comes from the file *filename*. Output is placed in *filename*.`sorted`. The content of `YAC/yac.clones.good.sorted` is shown in Figure 4.

83810	96893	69	80	original	0	good
0	21774	-1	22	original	1	good
29349	49484	30	42	original	2	good
0	19221	-1	20	original	3	good
5436	26786	10	29	original	4	good
3546	22739	7	23	original	5	good
42599	59922	38	53	original	6	good
61284	82948	54	68	original	7	good
0	21182	-1	21	original	8	good
0	10839	-1	14	original	9	good
22739	36437	23	36	original	10	good
45852	64187	40	56	original	11	good
0	16977	-1	18	original	12	good
39489	55012	37	48	original	13	good
55012	72281	48	60	original	14	good
23609	36437	26	36	original	15	good
1625	16429	0	16	original	16	good
2900	16977	4	18	original	17	good
82348	100000	66	84	original	18	good
75924	95622	63	79	original	19	good
83810	100000	69	84	original	20	good
35971	55266	35	49	original	21	good
30503	49484	31	42	original	22	good
5423	19059	9	19	original	23	good
16429	29349	16	30	original	24	good
86960	100000	70	84	original	25	good
4986	16429	8	16	original	26	good
2538	16429	3	16	original	27	good
16977	30503	18	31	original	28	good
0	15041	-1	15	original	29	good
50268	64441	45	57	original	30	good
44698	59922	39	53	original	31	good
2538	19059	3	19	original	32	good
5436	21182	10	21	original	33	good
3331	19059	5	19	original	34	good
74449	92311	62	73	original	35	good
22926	44698	24	39	original	36	good
23609	39489	26	37	original	37	good
42599	53524	38	47	original	38	good
77243	93497	65	75	original	39	good
82509	100000	67	84	original	40	good
7441	29349	11	30	original	41	good
2900	21774	4	22	original	42	good

Figure 3: Good Clones

0	21774	-1	22	original	1	good
0	21182	-1	21	original	8	good
0	19221	-1	20	original	3	good
0	16977	-1	18	original	12	good
0	15041	-1	15	original	29	good
0	10839	-1	14	original	9	good
1625	16429	0	16	original	16	good
2538	19059	3	19	original	32	good
2538	16429	3	16	original	27	good
2900	21774	4	22	original	42	good
2900	16977	4	18	original	17	good
3331	19059	5	19	original	34	good
3546	22739	7	23	original	5	good
4986	16429	8	16	original	26	good
5423	19059	9	19	original	23	good
5436	26786	10	29	original	4	good
5436	21182	10	21	original	33	good
7441	29349	11	30	original	41	good
16429	29349	16	30	original	24	good
16977	30503	18	31	original	28	good
22739	36437	23	36	original	10	good
22926	44698	24	39	original	36	good
23609	39489	26	37	original	37	good
23609	36437	26	36	original	15	good
29349	49484	30	42	original	2	good
30503	49484	31	42	original	22	good
35971	55266	35	49	original	21	good
39489	55012	37	48	original	13	good
42599	59922	38	53	original	6	good
42599	53524	38	47	original	38	good
44698	59922	39	53	original	31	good
45852	64187	40	56	original	11	good
50268	64441	45	57	original	30	good
55012	72281	48	60	original	14	good
61284	82948	54	68	original	7	good
74449	92311	62	73	original	35	good
75924	95622	63	79	original	19	good
77243	93497	65	75	original	39	good
82348	100000	66	84	original	18	good
82509	100000	67	84	original	40	good
83810	100000	69	84	original	20	good
83810	96893	69	80	original	0	good
86960	100000	70	84	original	25	good

Figure 4: Sorted Good Clones

4.3. Dnasubclone

The program `dnasubclone` determines which clones are superclones (i.e., are not contained in any other clone of the set of clones being considered) and which are subclones.

The command line arguments for `dnasubclone` are as follows:

```
Command Line Options:
      -r <string> rootfilename
```

An interactive session of `dnasubclone` might look something like the following:

```
dna% dnasubclone
Starting dnasubclone
Root filename (DNA/test.clones)? YAC/yac.clones.good
Sorting clone information...done.
Ending dnasubclone
```

Input is taken from the file *rootfilename*. Output is placed in two files: *rootfilename*.super and *rootfilename*.sub. The content of `YAC/yac.clones.good.super` is shown in Figure 5, and the content of `YAC/yac.clones.good.sub` is shown in Figure 6.

```
    0    21774   -1   22   original     1  good
 3546    22739    7   23   original     5  good
 5436    26786   10   29   original     4  good
 7441    29349   11   30   original    41  good
16977    30503   18   31   original    28  good
22739    36437   23   36   original    10  good
22926    44698   24   39   original    36  good
29349    49484   30   42   original     2  good
35971    55266   35   49   original    21  good
42599    59922   38   53   original     6  good
45852    64187   40   56   original    11  good
50268    64441   45   57   original    30  good
55012    72281   48   60   original    14  good
61284    82948   54   68   original     7  good
74449    92311   62   73   original    35  good
75924    95622   63   79   original    19  good
82348   100000   66   84   original    18  good
```

Figure 5: Superclones

```
     0    21182    -1   21   original     8  good
     0    19221    -1   20   original     3  good
     0    16977    -1   18   original    12  good
     0    15041    -1   15   original    29  good
     0    10839    -1   14   original     9  good
  1625    16429     0   16   original    16  good
  2538    19059     3   19   original    32  good
  2538    16429     3   16   original    27  good
  2900    21774     4   22   original    42  good
  2900    16977     4   18   original    17  good
  3331    19059     5   19   original    34  good
  4986    16429     8   16   original    26  good
  5423    19059     9   19   original    23  good
  5436    21182    10   21   original    33  good
 16429    29349    16   30   original    24  good
 23609    39489    26   37   original    37  good
 23609    36437    26   36   original    15  good
 30503    49484    31   42   original    22  good
 39489    55012    37   48   original    13  good
 42599    53524    38   47   original    38  good
 44698    59922    39   53   original    31  good
 77243    93497    65   75   original    39  good
 82509   100000    67   84   original    40  good
 83810   100000    69   84   original    20  good
 83810    96893    69   80   original     0  good
 86960   100000    70   84   original    25  good
```

Figure 6: Subclones

4.4. Dnamergeclone

The program `dnamergeclone` is used to aggregate a set of `.frag` files (specified by a specific `.clones` file) into a single file.

The command line arguments for `dnamergeclone` are as follows:

```
Command Line Options:
      -c <string> clonefilename
      -f <string> fragmentfilename
```

An interactive session of `dnamergeclone` might look something like the following:

```
dna% dnamergeclone
Starting dnamergeclone
Clone filename (DNA/test)? YAC/yac.clones.good.super
Fragment filename (DNA/test.frag)? YAC/yac.frag
Ending dnamergeclone
```

All the *fragmentfilenames* corresponding to clones present in the file *clonefilename* are aggregated into the file *fragmentfilename*.aggreg. This file can be used by DNA mapping software for defining the set of clones (and their fragments) to be processed. A portion of the file YAC/yac.frag.aggreg is shown in Figure 7.

```
N= 5,clone=cl1
3738 4853
2318 2535
2022 513
1972 8591
1377 10563

N= 3,clone=cl5
3738 4853
1972 8591
1377 10563

N= 2,clone=cl4
1972 8591
1377 10563

N= 3,clone=cl41
4618 24238
1972 8591
1377 10563

N= 1,clone=cl28
4618 24238

N= 4,clone=cl10
4618 24238
3855 30935
1809 28856
1343 34790
```

Figure 7: Portion of an Aggregated File

5. Scenarios

This suite of programs can be used in a variety of ways. For instance, it may be desirable to create a DNA strand, create random clones, and then perform complete digestions of these clones in 3 different ways (first using *Hin*dIII, second using *Eco*RI, and third using a combination of both, referred to as RH). This might be achieved by creating a directory into which all mapping data is to be deposited (e.g., CL32). This directory might have three different subdirectories (HINDIII, ECORI, and RH) intended to contain the `.frag` files for each of the separate digestions. A scenario for obtaining these results might be achieved by executing the following UNIX shell script.

```
# Make the DNA itself
dnamake -s32 -l100000 -rCL32/cl32 -p50
# Make the clones from the DNA
dnaclone -s32 -rCL32/cl32 -eatat -m32 -f10 -n15 -d1000 -l14000 -u25000
# Separate the good and bad clones from each other
separate_clones <<END
CL32/cl32.clones
END
# Find the superclones of the good clones
dnasubclone -r CL32/cl32.clones.good
# Do detailed stuff for HINDIII
#     create frag files
dnafrag -rCL32/cl32 -e2 -0aagctt -1ttcgaa -l400 -u8000
#     sort the fragments and add numbers
addnums_and_sort <<END
CL32
cl32
frag
END
#     introduce normal random error into the frag files
nre -s32 -r CL32 -i cl32 -p 0.75
#     move the results to the subdirectory
mv CL32/cl32.dnafrag.input CL32/HINDIII
mv CL32/cl32.frag.* CL32/HINDIII
mv CL32/cl32.rfrag.* CL32/HINDIII
# Do detailed stuff for ECORI
#     create frag files
dnafrag -rCL32/cl32 -e2 -0gaattc -1cttaag -l400 -u8000
#     sort the fragments and add numbers
addnums_and_sort <<END
CL32
cl32
frag
END
#     introduce normal random error into the frag files
nre -s32 -r CL32 -i cl32 -p 0.75
#     move the results to the subdirectory
mv CL32/cl32.dnafrag.input CL32/ECORI
mv CL32/cl32.frag.* CL32/ECORI
mv CL32/cl32.rfrag.* CL32/ECORI
# Do detailed stuff for RH
#     create frag files
dnafrag -rCL32/cl32 -e4 -0aagctt -1ttcgaa -2gaattc -3cttaag -l400 -u8000
#     sort the fragments and add numbers
addnums_and_sort <<END
CL32
cl32
frag
END
#     introduce normal random error into the frag files
nre -s32 -r CL32 -i cl32 -p 0.75
#     move the results to the subdirectory
mv CL32/cl32.dnafrag.input CL32/RH
```

```
mv CL32/cl32.frag.* CL32/RH
mv CL32/cl32.rfrag.* CL32/RH
#     make an aggregate frag file for the good superclones for ECORI
dnamergeclone -c CL32/cl32.clones.good.super -f CL32/ECORI/cl32.frag
#     make an aggregate rfrag file for the good superclones for ECORI
dnamergeclone -c CL32/cl32.clones.good.super -f CL32/ECORI/cl32.rfrag
#     make an aggregate frag file for the good superclones for HINDIII
dnamergeclone -c CL32/cl32.clones.good.super -f CL32/HINDIII/cl32.frag
#     make an aggregate rfrag file for the good superclones for HINDIII
dnamergeclone -c CL32/cl32.clones.good.super -f CL32/HINDIII/cl32.rfrag
#     make an aggregate frag file for the good superclones for RH
dnamergeclone -c CL32/cl32.clones.good.super -f CL32/RH/cl32.frag
#     make an aggregate rfrag file for the good superclones for RH
dnamergeclone -c CL32/cl32.clones.good.super -f CL32/RH/cl32.rfrag
```

Note that in this scenario all the DNA strand and clone information remains in the top-level directory (CL32), while all the fragment information is transferred to the appropriate directory (HINDIII, ECORI, or RH) corresponding to the particular restriction enzyme used.