TESTER(1) **TESTER(1)**

NAME

tester − A analyzer and statistics generating program for the assembler output

SYNOPSIS

tester [-L #] [-c] [-h] [-p] [-o <output file>] [-d <dna file>] <root file name>

DESCRIPTION

Tester reads the output files from the assembly program and measures how well the assembly worked based on a number of statistics.

The –L option specifies at which level the tester program should be run. The tester program has three different levels. Level 1 (the default level) assumes that no simulation information is available. Thus only general statistics about the size and structures of the assembly may be generated. Levels 2 and 3 assume the existence of a dna file as generated by the celsim program. This file will be generated when celsim is invoked using the –d option. Level 2 assumes that the original DNA strand is known, but there is no information on its internal repeats. This will be true when celsim is given an existing DNA strand to shotgun. Level 3 assumes that the dna file contains not only the original DNA sequence, but also the positions of the repeats within it.

The –c option makes the tester open a slew of celagram windows. This allows the user to view the histograms in a graphical format. The files used by celagram are named <root_file_name>.tst#.cel. The tester program removes these files before exiting. It does wait, however, until the user closes the last celagram window created to be closed before completing. (This is to insure that a data file is not closed before it is read.) There is a time penalty for using this option, but it may be worth it to see the data graphically displayed.

The –h option causes to create the celagram data files talked about above. If used with the –c option the only difference is the files (<root>.tst#.cel) are not removed before the program exits. This option allows one call celagram to view some of the data graphically, without having to deal with all of the windows. The drawback to this option is that the naming convention of the files is not very frienly.

The –p option tells tester to send its output to stdout. Normally tester sends its output to <root_file_name>.tst. The –p option converts tester into a pipe. This option overrides the –o option.

The –o option allows the user to specify an output file other than <root_file_name>.tst

The –d option allows the user to specify a dna file. Normally tester looks for the file <root_file_name>.dna to be used as the dna file. This option is only meaningful if the level is set above 1.

If the string used for the <root file name> ends in ".cns", this extension is removed and the prefix is treated as the root name. The string ".cns" is appended to the root file name and tester attempts to open this file. If this attempt fails, tester then attempts to open the file without the extension. Thus a file whose name does not end in ".cns" may be used as input, though this file is assumed to be the one output by the consensus module of the assembler.

The program will normally be called using only the –L, -c, or -h options. For example:

```
tester -L2 a004
```

Using this call tester opens the files a004.cns and a004.dna for input and sends its output to a004.tst.

```
tester -c a004
```

Using this call tester opens the file a004.cns, sends it's output to a004.tst, and opens many celagram windows to display some of the results.

FUTURE WORK

Unfortunately I ran out of time before I could finish my list of work on the tester program. I will list here what needs to be done and some idea of how I had planned to go about doing it.

1. The contig checking needs to be revised.

Right now I look at the simulator coordinates of the fragments and the extreme ends of a contig to get the starting and ending location for the contig. The difference between these numbers gives the actual contig length. This is compared with the length of the consensus. If these two numbers differ by too much, the contig is marked BAD (in the code the bad flag of the contig structure is set to –1) and a message is printed out. Then the simulator coordinate of each fragment in the contig is examined to determine if it lies between the starting and ending locations of the contig. If it does not, then a count (also the bad flag in the contig structure) is incremented. In the end this count tells how many fragments within the contig are misplaced.

The above scheme works well under most situations. The problem would be if the Repeat Resolver misplaces a fragment at the end of a contig. While the contig would essentially still be good, the starting or ending location would be way off and thus the contig would end up looking bad. I had planned to resolve this by using code similar to that which sets the collapse field in the unitig structure. Each fragment would get a 'vote' as to where in the genome the contig came from. The majority rules. I would then take the simulator coordinates from the two extreme fragments that voted with the majority, extend those numbers as needed to cover the consensus, and use those as my starting and ending locations. Notice this will make the length check no longer applicable, so I would probably still treat as BAD contigs with too many fragments out of place or perhaps those that required too large an extension to the extreme fragments to cover the consensus.

2. The calls to celegram need to be made interactive.

I hate that I didn't get to this, but oh well. This is pretty much self-explanatory. After the calculations are done a simple menu should be displayed allowing the user to select which celegram files to display. The only nasty thing here is that currently the celegram files are named by tacking a number onto a common root. While this makes the creation, deletion, and any handling of ALL the files easy to code and maintain, it means the file names will change when new files are added. Also the file names will be different based on what level the tester is run at. The solution is more work when creating and updating the interactive portion, changing to the file names to ones that depend on the data input which means more work when creating and deleting the files, or perhaps adding a code to the beginning of a file (maybe 30 chars in from the title screen) which tells the program which data file it is.

3. Histogram of Contig Lengths as Requested by Mark

Mark wants a histogram of the lengths of the contigs in base pairs. This would be a 2 minute addition except for the fact that I currently don't store information on contigs which aren't placed. I did this because I didn't believe non-placed contigs to be interesting (they are just unitigs with a new name after all) and I wanted to save some memory. One could change the global variable Contigs from an array of pointers to ContigT structures (these pointers are set to NULL for non-placed contigs) to an array of ContigT structures and read in all contigs. If this is done an extra field will need to be added to distinguish the placed contigs and all code which deals with contigs will have to be adjusted accordingly. If I had done this I think I would have just added a special array to store the lengths of all

contigs as that's the only bit of information needed for this histogram. (Notice that the length field of the ICM message includes the –'s, so some work will be required to get the correct length.) Mark also wanted an identical histogram except that instead of each bucket containing the number of contigs of that size, it would contain the sum of the lengths of the contigs of that size. (See Granger if you need further clarification.) This histogram will be a bit of a pain as so far all histograms are handled by the CalcStats & PrintStats procedures which won't work for this.

4. Command Line Switch to Turn of Sequence Comparison

As you can see from the code I wrote my own banded, dynamic programming routine to compare the consensus with the original sequence. Since it's banded it uses linear time/space to do the comparison, but it does add a bit of time for large files. Thus I've toyed with the idea of allowing it to be turned off to save compute time. (Actually the lion's share of time is in the I/O, not the sequence comparison)

5. Histogram of Quality Values of Missed Calls

Speaking of the sequence comparison, you may wonder why with a banded matrix I used linear space instead of constant space. This was to save time (backtracking) to get the actual alignment. The only reason I can think of to want the actual alignment is to look at the quality values of the missed calls.

6. Breakdown of Types of Missed Calls

No one asked for this, but I thought it would be interesting. This would simply be a count of how many miss calls were insertions, deletions, and substitutions. Is our assembler biased toward making a specific type of mistake? If one calculates number 5 above, this would be a simple add on.

7. Histogram of Quality Values of All the Consensus

This would actually be very easy to add. If it had been higher on my list (where I would have seen it before now) I would have done it already.

8. Report on How Many Unitigs are Actually Splitable

Talk to Knut about this one. He basically wants to know how many of the invalid unitigs his routine has a chance of splitting. Also tell him I'm sorry I didn't get this in.

AUTHORS

Eric Anson:

Created July 7, '99