VCU Bioinformatics and Bioengineering Summer Institute

Maintaining Continuity in a Genome Project

Notes: Using Blast to compare sets of protein: Parsing massive output

I. Overview of problem

The Scenario presents a common problem for those who try to keep up with genome sequencing projects (both the organizers and the users of the information): How do you integrate information derived from an earlier version of the sequence with a later version? It may sound pretty easy. Even the earliest version of the sequence is 99% correct, though incomplete. What's the problem?

Imagine that you, an archaeologist from the 30th century, recover from a sea-worn chest shreds of thousands of copies of *Moby Dick*, from a boat evidently carrying the cargo of a book publisher. You lovingly piece together the text as best you can. You treasure each of the resulting pages and pour over them, writing liberally in the margins. A colleague, who uses even more shreds from the chest, is able to do better, recreating whole chapters. This version makes more sense, and you want to copy your notes to it.

Seems straightforward enough, just a matter of comparing each of your pages with pages from the more complete copy. But, no. The two versions sometimes differ in how smudged words are interpreted. Worse, the pages of the two versions are not always assembled in the same way (owing in part to the ancient author's penchant for repetition -- is it 20 pages or 40 devoted to the characteristics of the color white?).

Since you can't compare whole pages, you try sentences instead, taking each sentence of your version and searching the other version until you find a match. This method is still imperfect: sometimes even sentences are mangled and partially match multiple sites in the other version, and sometimes your sentences find no match at all. (Perhaps there were some contaminating fragments of *Flipper, the Happy Dolphin*, printed by the same publisher).

We will use the same method to match the content of one genomic assembly with a later, better assembly, recognizing that the attempt will not be completely satisfactory. The strategy, then, is to compare each protein sequence predicted from the sequence of the 2002 assembly of the *Streptococcus sanguis* genome with those predicted from the recently assembled revised sequence. Once protein matches have been made, annotation of protein predicted from the earlier sequence can be transferred to the annotation of the new sequence.

The new sequence of *S. sanguis* has not been analyzed yet, however, and for this research simulation we'll use two available versions of the genome of another organism, *Cryptospiridium*. From the lessons learned in this test case, we (or at least one of us) may proceed to the real case when the set of predicted protein becomes available.

II. *Blast*: A sequence comparison tool

The workhorse in this strategy is *Blast* (for **B**asic **L**inear **A**lignment **S**equence **T**ool), a program that compares sets of sequences and finds the best alignments. It is the most widely used of any bioinformatics program, usually in one of its many online incarnations (e.g., the implementation made available by the National Center for Biotechnology Information). For our special purpose, however, it is more convenient to run *Blast* on our own computers, where we may exert more control over the process.

Now is the time to download *Blast* from NCBI, configure a database composed of the set of proteins from a May 2003 assembly of *Cryptospiridium*, and use *Blast* to compare this database with the set of proteins from an October 2003 assembly.

Today's webpage provides instructions on:

> How to download *Blast*
> How to set up a database of protein sequences for use by *Blast*
> How to run *Blast*

The question arises, should you set up the database using the set of protein from the earlier assembly or the later assembly? To answer this question, consider the problem at hand. You want to connect work done on proteins found from an earlier assembly to proteins of the new assembly. The way *Blast* (actually *BlastAll*) works is to take each protein in a file (the protein is called the query sequence) and compare it against each protein of the database (called the subject). If a match is found, it is reported. If not, *Blast* says "No match found". If every protein predicted from the earlier assembly is also predicted from the later assembly, then it doesn't matter which set is used to make the database. You might think that a later assembly ***ought*** to have every protein predicted by the earlier assembly. Don't count on it. Either way will work, but I advise you make the database from the later assembly.

SQ1. Suppose that some proteins you have annotated are *not* predicted in the new assembly. How will the results differ if you use the proteins of the old assembly as the database instead of as the query set?

OK, do it.

> ***Warning:*** *the running of Blast using these large protein sets will take a long time. The exact time will depend on the speed of your machine, but I'd count on a few hours. Why not spend that time going through the rest of these notes?*

You wanted to know which proteins in the old assembly matched which proteins in the new assembly. *Blast* has run for the required few yours, you have the answer you seek... somewhere in a file of some tens of million characters. You ***could*** go through the file by hand, picking out the proteins you're looking for, but since you can't seem to find a free 10-year block anywhere in your schedule, you need another approach.

Still easy. You've come up with some free software, *BlastParser* (see today's web site), that automates the process of doing just what you would do if you had the time: go through the output of *Blast* and pick out just the names of each protein used in the comparison and what the comparison found, in brief. Unfortunately, the program was written to work on comparisons of two different sets of protein, not yours. You can't expect a free program to do exactly what you

want, but it shouldn't be too difficult to modify it slightly to work on your comparisons. You hope.

III. Figuring out what *BlastParser* produces

For starters, why not try running the program? Well, that could be dangerous. I wouldn't run a program someone handed me without looking at it to see if it did anything I might regret later. But trust me on this one and download *BlastParser* to your favorite directory, run `perl blastparser.pl`, and stand back!

[a moment of silence while you run the program]

... not as satisfying as one might have hoped, but not unexpected either:

```
Can't open 71vsnps.txt: No such file or directory
```

(If you got instead a message indicating that the parsed output was successfully stored, then you can skip the next paragraph)

As I said, *BlastParser* was not written to work on ***your*** *Blast* comparison but on a different one. Perl said it couldn't open something called `71vsnps.txt`. I'll bet you don't have that file in your directory. Moral: When you get a program, always ask for an accompanying data file on which the program runs successfully. Go back to today's webpage, download `71vsnps.txt` to your directory and run the program again.

This time you should get a message indicating a successful completion of the task. Check out the directory, looking for the newly created file called `71vsnps-out.txt`. If you take a look at it, you'll see that it contains some bioinformaticky looking things, gene names and such. That's good. But you'll need to understand what the output means, and, most important, you'll need to get the program to work on YOUR file, i.e. the output of blasting one *Cryptospiridium* protein set against another.

We can look at the program -- we ***will*** look at the program --, but it might be easier to compare the input and the output to get an idea of what comes from what. So bring up the input file `71vsnps.txt` (Fig. 1) and the output file `71vsnps-out.txt` (Fig. 2).

Putting the two together, I get the following color coded interpretation:

1. **Query name**: Some symbolic identification of a gene used to probe the database
2. **Query description**: Something in plain English
3. **Query length**: the length of the query sequence
4. **Subject name**: symbolic identification of a gene within the database similar to the query
5. **Subject description**: something in plain English [A description *ought* to be there, even though it's not]
6. **Subject length**: the length of that protein
7. **E-value**: The probability that a match of this quality might have arisen by chance

Items 4-6 are repeated if there is more than one match and omitted if there aren't any matches.

```
BLASTP 2.1.3 [Apr-1-2001]

Reference: Altschul, Stephen F., Thomas L. Madden, Alejandro A. Schaffer,
Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman (1997),
"Gapped BLAST and PSI-BLAST: a new generation of protein database search
programs",  Nucleic Acids Res. 25:3389-3402.

Query= all0001 ID:6715 {-311 <--- 918} unknown protein
         (409 letters)

Database: Npunctiforme_protein
           7432 sequences; 2,397,140 total letters

>Contig647.revised.gene32.protein
          Length = 438

 Score =  256 bits (655), Expect = 3e-069
 Identities = 169/438 (38%), Positives = 235/438 (53%), Gaps = 45/438 (10%)

Query: 2   KFVRNIVILLSSLAAVLTVCENANAGKGSLNVGAK-IPPGQERQLLQYQLQNDGKGLHRI 60
           K VR   I  SSLA  LT C++A A    LN+G   I  G E + LQYQ+ N  + L+++
Sbjct: 13  KAVRFATIAFSSLAVGLTTCQSAFA---QLNIGRYGIQQGLESEYLQYQINN--QNLNQM 67
```

etc for many megabytes... [I cut it off at about 50 kbytes]

Fig. 1. Output from *Blast*. First few lines from the file `71vsnps.txt`. The colors are not in the original file but are added here to aid comparison with the output file (**Fig. 2**).

```
all0001 ID:6715 {-311 <--- 918} unknown protein 409  647.032   438  3e-069
all0002 ID:2 {981 <--- 1718} unknown protein    245  647.030   213  6e-060
```

etc for many more lines

Fig. 2. Output from *BlastParser*. First two lines from the file `71vsnps-out.txt`. The colors are not in the original file but are added here to aid comparison with the input file (**Fig. 1**). Each field is separated by a tab (not shown).

Does *BlastParser* do what you need it to do? Actually, it may do too much. You want to know the ***best*** match between the two sets of protein. You're probably not very interested in the second and third best matches provided by *BlastParser*. And you may not be sure at the moment exactly what to do with those E-values. For now, you decide to accept the output format as-is.

SQ2. Speculate on what you might do with the information produced by *BlastParser*, given the concerns described in the Scenario.

SQ3. Speculate exactly what, in plain English, the E-value "3e-069" might signify.

III. Modifying *BlastParser*

The time has come to make the program work on the file *you* generated from Blast. So, it will be necessary to go into *BlastParser* and modify it to look for your file instead of the one it was built for. Let's do it.

Use *PerlEditor* or *Notepad* to examine *BlastParser* and try to find the section of code that identifies the name of the file to be parsed. Happily, the program very kindly put neon lights around the pertinent section, and right below the comment

```
#### OPEN THE BLAST FILE
```

there's a very suspicious two lines of code:

```
my $blastPath = "71vsnps.txt";
open BLAST, "<$blastPath" or die "Can't open $blastPath: $!\n";
```

Obviously, that's where the error message you got came from.

SQ4. Change $blast_path so that it is assigned the correct file name, i.e. the one created when YOU ran Blast.

Now run the program again, and examine the results... That's more like it! But wait... Comparing the output with the file produced by BlastAll (do the comparison), it looks like I'm getting the query name, the query description, and the query length, but nothing for the subject. Could it be that there are NO matches between the two sets of protein? I doubt it. More likely, the program runs fine with the other Blast output, but there's something perhaps subtly wrong that we need to fix before it can run with OUR Blast output. Perhaps you were afraid of that.

No time like the present. Let's try to get a minimal idea how this program works and what might need fixing. First, go to the Main Program section.

```
$line = <BLAST>;                    # Read first line
while (defined $line) {             # Only if not end of file
   if ($line =~ /^Query=/) {        # If 'Query=' matches beginning of line...
       Write_previous_query_info();  # ... end previous query record
       Start_new_query();}           # ... and begin a new one
   elsif ($line =~ /^>Contig/) {    # If '>Contig' matches beginning of line...
       Record_subject() }            # ... save hit information
   $line = <BLAST>;}                # Read next line
```

There's much here that's mysterious, but so long as I don't get dragged down by that, there's also much here that I recognize. For example, `/^Query=/`. Never mind the `/` and `^`, I recognize `Query=`. That appears in the output whenever query information is presented. And I recognize `>Contig`. That appears in the output -- the output that *BlastParser* was designed for -- and is followed by subject information. BUT `>Contig` DOESN'T APPEAR IN *YOUR* BLAST OUTPUT! That's clearly a problem. You need to figure out what signals the presence of subject information in your output and modify the program accordingly.

SQ5. Have any idea how to change the program so that it recognizes the beginning of the subject line?

Even if you make an appropriate change, you're going to find that the program STILL doesn't work. Why? You'll have to look at the parts of the program that actually do the work: the routines called `start_new_query` and `record_match`. There's a lot mysterious here, but with surprisingly little explanation, you can make a good deal of sense out of this code. The important thing is to figure out what questions you need to ask:

- **How is relevant information in the input file recognized?**
  Answer: Through *regular expressions*, the code between two / as in:
  ```
  ($subject_length) = ($line =~ /Length = (\d+)/);
  ```

- **How is the recognized information in the input file captured?**
  Answer: Through *capturing within regular expressions*, as in:
  ```
  $subject_description = $3;
  ```

It seems that the time is ripe to learn about regular expressions. The next set of notes will take you through this topic.