BIOL591: Introduction to Bioinformatics (2003)
Program to superimpose one sequence on a known structure

I. Our starting point: Threading a sequence through a known structure

By now you've probably used Protein Explorer to look at the known 3-dimensional structure of UDP-glucose dehydrogenase (UDPGD) from *Streptococcus pyogenes*, and a thing of beauty it is. Unfortunately, it does not tell you how to prevent the similar enzyme from *Mesorhizobium loti* from precipitating when it is overexpressed in *E. coli*. The first step is to see the three dimensional structure from the *M. loti* enzyme. That structure has not been determined, but if the sequences of UDPGD from the two bacteria are similar enough, one may presume that the two structures are also similar.

If the structures are indeed similar, we could use the alignment to affix the sequence of the *M. loti* enzyme to the structure of the *S. pyogenes* enzyme. What does this get us? It doesn't tell us anything new about the ***structure*** UDPGD from *M. loti*, because we ***presume*** the structure to be the same as that of the structure of the homologous enzyme in *S. pyogenes.* The analysis does tell us, however, where specific amino acids are located if the structure is basically applicable. We can use this mapping to tell us the location of altered amino acids in mutants of UDPGD we isolated that are moderately more soluble. And from that knowledge, we might be able to predict what additional mutations would affect solubility, perhaps to a greater extent than any we isolated.

The first step is to get an alignment between UDPGD sequences from the two bacteria. Clustal (see link in unit web page) takes two or more sequences and returns the best alignment of the sequences it can find.

SQ1. Get the sequences of UDPGD from *Streptococcus pyogenes* and *Mesorhizobium loti* in FastA format (if you haven't already) or convert the sequences from some other format into FastA format. Concatenate them into one file.

SQ2. Download Clustal (if you haven't done so already) and upload the file you created in SQ1. Then do a complete alignment. Save the aligned sequences (Save as) in NBRF/PIR format (which is FastA format but allowing gaps).

You can now use the aligned sequences as in put to ThreadProtein.pl (available on the unit web page). That program is ready to go, as soon as you get the files you need (and change the **Files** section of the program so that the file names are correct).

SQ3. Find and download the PDB file containing the structure of UDPGD from *S. pyogenes.*

SQ4. Download, modify, and run ThreadProtein.pl

SQ5. Upload the resulting PDB file into Protein Explorer (see link on unit web page).

Stop the protein from spinning, get into QuickViews, toggle the water out of existence, and you should see a three-dimensional structure with four colors. What do these colors mean?

SQ6. Examine the documentation for ThreadProtein.pl. What *do* the colors mean? Which color corresponds to which type of amino acid residue? To answer this, you might

click on a residue (its identity will appear in the command box) or select a chain and change its color.

You can isolate different chains by selecting the others and changing their display to ball+stick.

SQ7. What generality can you draw regarding the location of amino acids in common between *S. pyogenes* and *M. loti*. How do you explain this generality?

SQ8. What generality can you draw regarding the location of amino acids that are insertions or deletions in one sequence relative to the other? How do you explain this generality?

II. The workings of ThreadProtein.pl

II.A. Modules

In a moment you will modify ThreadProtein.pl so that you can identify the positions of new mutations in UDPGC in the same way you can now identify replacements, insertions, and deletions. First, however, it might be useful to learn a bit about how the program works. Before getting to the strategy of the program, notice two things unusual about this program. Every other program this semester has had a **Libraries and Pragmas** section consisting of a single line:

```
use strict;
```

The section in this program has three `use` statements.

To see a second, more subtle difference, one that is intimately related to the first, go to the **Main Program**, which opens:

```
Read_FastA_sequences ($alignment_file);
```

By now you immediately recognize this as a call to a subroutine called Read_FastA_sequences. So you scroll down to the **Subroutine** section and… no subroutine! How can this call work if there's no subroutine telling Perl what to do?

You've no doubt guessed the solution to this mystery. The `use FastA_module` statement must somehow make `Read_FastA_sequences` meaningful, even in the absence of a subroutine in ThreadProtein. Examine FastA_module.pm and you'll see how: Except for the first four magical lines and a new **Initialization** section, FastA_module contains just a collection of subroutines, including the one we were looking for. FastA_module is an example of a Perl ***module*** (indeed, the ***pm*** suffix means "Perl module").

Modules are great. They enable you to use and reuse proven code without need for cutting and pasting. Never again will you need to search through programs for a routine that reads FastA files. Just bring in FastA_module and you're ready to go. They have another benefit. ThreadProtein is less cluttered than it would otherwise be because housekeeping functions have been dropped into modules. What remains is code that's specific to the task of threading protein.

As you write generally useful subroutines for one program or another, it is a good practice to package it up into a module so that you can easily reuse the code later. Soon you'll find that much of programming is piecing together capabilities you've accumulated for a new purpose. Much easier than writing from scratch!

You may surmise that there are other people in the world using Perl, even others using it for bioinformatics. If so, then there's probably lots of useful modules on different computers, often

just the one you need. Wouldn't it be wonderful if there were libraries you could visit to check out a module when you need it?

Well, there are. In particular, there's a useful library full of modules of bioinformatic interest, called BioPerl. (see web site links page).

Bringing in modules seems like inserting the code into your program, but there are significant differences. First, the module can't access variables in your program and you can't access variables in the module (at least not by the usual means). This is good. Neither entity can accidentally alter the values of variables when it shouldn't. The only part of the module you can normally access is what is exported to you, e.g. the three subroutines of FastA_module and the three hashes of AA_module, via the declaration of the special variable @EXPORT.

Second, modules are ***initialized*** prior to use. Your program never calls Read_aa_info in AA_module (and it couldn't even if it wanted to, since the subroutine was not exported). Nonetheless, that subroutine is invoked during the initialization of the module, once and only once. This behavior makes modules ideal for taking care of informational entities (like databases) whose workings need not concern the main program.

II.B. The threading process

Scroll to the **Main Program** of ThreadProtein to read the overall strategy of the program. The claim is that the output PDB file is the same as the input PDB file, except for ATOM lines. This is more easily seen than explained, so…

SQ9. Open both the input and output PDB files simultaneously and scroll down in each window so that you can see the first 20 or so ATOM lines (roughly 10% of the way through the file). How do the two files differ? What do the values in the first six columns signify? Where did the missing amino acids go in the output file?

SQ10. Scroll down the C chain of the output PDB file. Why is it that all the other chains have multiple atoms per amino acid but the C chain has only one? Why is it that the program assigned such a large number to all the residues in this chain?

So somehow the program figures out which amino acids are common between the two sequences, which are different, which are insertions, which are deletions, and then apportions them to the four chains. In the **Main Program** you can read that all that information is stored somehow in a variable called $comparison_summary. How is that done?

SQ11. Modify ThreadProtein.pl so that $comparison_summary is printed out immediately after it is calculated. It might be helpful to see on the same screen the sequences that are being compared, so print those two out as well. Pause the program at that point and see if you can determine what all the symbols in $comparison_summary mean, relative to the two sequences.

Armed with that insight, look now at the subroutine that actually constructs the new PDB file. Look at it just for the overall logic – never mind for now the nuts and bolts of what it actually does.

SQ12. The variable $residue_state seems to be pretty important in directing events. What is contained in this variable? Where did it come from?

III. Modification of ThreadProtein to set apart positions of known mutations

III.A. Overview of problem

Finally we come to our appointed task: writing a program that will produce a PDB file readable by Protein Explorer and enable PE to make visible altered positions of mutant UDPGD from *M. loti* that are able to partially avoid precipitation. You can see the list of the mutated amino acid in each of the seven mutants by going to the unit web page. Some have a single mutation, others multiple. Of course, there's no way of knowing which of the mutations is responsible for the observed decrease in precipitation or if more than one mutation is responsible (in those cases where a mutant protein has multiple mutations).

The first thing to realize is that this is a very similar problem to the one that ThreadProtein has already solved. In the end, Protein Explorer doesn't care if we divide amino acids by match vs replacement or wild-type vs mutant. It will color whatever categories we set up, so long as they're set up properly. If we can get the mutation information into the same format ThreadProtein understands, we might be able to get the job done with relatively minor modifications.

What needs to be modified?

1. First, we need a variable to hold the mutation information and a subroutine to read the file into that variable.
2. We need a subroutine to incorporate that information into the variable used by ThreadProtein to hold the bases for partitioning the amino acid residues into different chains.
3. We need to alter the subroutine that builds the new PDB file to be cognizant of the new symbol we'll define to represent mutations.

That should do it.

III.B. Reading in mutation information

We need to understand what information we're given regarding the mutant enzymes. The file, UDPGD-mutants.txt (available from the unit web page), describes mutations according to a common convention. Each mutation is given in the form: `XXX ddd YYY`, where `XXX` and `YYY` are the three-letter codes of the wild-type and mutant proteins, respectively, and `ddd` is the position of the amino acid. Each line begins with an identification number of the mutant.

How much of this information do we need? There might be some value in giving the set of mutations in each mutant a different color, but for now I'm going to combine all the mutations into one set. So the mutant ID number is of no consequence. The position of the mutant amino acids is the main value, since that's what I'll use to define a new subdivision of the protein (i.e. a new chain), to be colored separately from the other amino acids. What about the ***names*** of the amino acids? The name of the wild-type amino acid is useful information, because I can use it as a cross check, to make sure that the it agrees with the amino acid given by the amino acid sequence of *M. loti* enzyme. But that's an extra. Grabbing that information will complicate the subroutine. You can decide if you want to save that information or throw it away. The name of the mutant amino acid could be useful in a different way. It would be nice to be able to visualize alternatively the original amino acid then the mutant amino acid. That is still another

complication. In the interest of simplicity, I've decided for the first draft to use only the positional information.

Having defined what we want, writing the subroutine boils down to the kind of parsing we did weeks ago. What kind of regular expression will capture everything we want?

SQ13. Write a subroutine that will read the file UDPGDH-mutants.txt and print out the coordinate of each mutation. Once that works, modify it so as to save those coordinates in an array.

III.C. Incorporating the position of mutations into the summary of changes

ThreadProtein incorporates information about the alignment into a summary of changes in the subroutine called Analyze_alignment. My first thought is to use this as a model for a second subroutine that inpororates information about the mutation positions into the same summary. So, just as Analyze_alignment proceeds via a foreach loop through a sequence (actually two) containing the information it wants, my subroutine will proceed through the array in which I put all the positional information.

What I want to do then is very simple: put a new symbol in every position where a mutation occurred, just as Analyze_alignment puts, for example, a symbol in every position where an insertion occurs. If it weren't for one complication, this part of the program would be relatively simple. The complication is that I have the mutations in coordinates of the *M. loti* enzyme. ThreadProtein wants to see everything in terms of coordinates of the aligned sequences, with gaps included.

When such a complication arises, there are two general approaches. One is to write the subroutine such that it recognizes and handles the complication. The other is to write a separate function that transforms what I know into what I want to know and then write a much simpler subroutine. I think the second approach is both easier to code and easier to understand when you're reading the program (e.g. weeks from now when you've forgotten what you did). So I suggest you begin this part of the problem by writing a subroutine that takes as input the coordinate of the mutation and returns the coordinate of the mutation relative to the aligned sequences. There are many ways of accomplishing this.

SQ14. Write a subroutine that will read translate one coordinate system into another: from *M. loti* UDPGDH coordinates to *S. pyogenes*/*M. loti* alignment coordinates.

SQ15. Write a subroutine analogous to Analyze_alignment that incorporates mutant positional information into the same variable that contains information about insertions and deletions.

III.D. Modifying the routine that builds the new PDB file

Here again, what we want to do is very analogous to what ThreadProtein already does. The subroutine Create_new_structure_file goes through the string that summarizes the status of each residue in the alignment and partitions the residues amongst four different chains. We want to add one more chain, built in a way very similar to the way that one of the chains is already built.

SQ16. Which condition (match, replacement, insertion, deletion) is most similar to mutating an amino acid? Which section of Create_new_structure_file would be the best model for the task at hand?

SQ17. Modify Create_new_structure_file so that the PDB file produced has a fifth chain, containing those amino acids that were observed to be mutated when selecting for increased solubility of UDPGD.

You may complete the last study question but feel a bit uneasy. How do you know that the right amino acids were flagged? Protein Explorer will be of limited use in testing, because if you click on an amino acid (i.e. those you identify as in the new, fifth chain) it will report to you the amino acid position in *S. pyogenes* coordinates! It will help a lot if we had a concordance, enabling us to go back and forth between coordinate systems.

SQ18. Print out for each mutation the position of the amino acid in *S. pyogenes* coordinates and in *M. loti* coordinates. *Hint: if you use the handy function called True_size, you can accomplish this task in a single line!*

That's a lot of programming! Even if each step is not overwhelming, the irritating details add up. Let's see how far we can get by the end of class tomorrow. Go as far as you can beforehand.