METAPROGRAMMING BIOINFORMATICS

IN THE POSTGENOMIC ERA

by

Nathaniel Tobias Ohler

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Sciences

MONTANA STATE UNIVERSITY
Bozeman, Montana

April 2006

APPROVAL

of a thesis submitted by

Nathaniel Tobias Ohler

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the Division of Graduate Education.

Brendan Mumey

Approved for the Department of Computer Science

Michael Oudshoorn

Approved for the Division of Graduate Education

Dr. Joseph J. Fedock

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Montana State University, I agree that the Library shall make it available to borrowers under rules of the Library.

If I have indicated my intention to copyright this thesis by including a copyright notice page, copying is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for extended quotation from or reproduction of this thesis in whole or in parts may be granted only by the copyright holder.

Nathaniel Ohler

April, 2006

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

ABSTRACT

The number of bioinformatics programs available is continuously growing, along with the knowledge required to run each individual program. As more programs become available, more complex combinations of these programs are being used by scientists. Workflow engines attempt to remove repetitive procedures from these combinations by saving and executing any group of the programs as one larger “meta-program”. The output of one program is automatically directed to the input of another thereby creating a “flow” of “work”. The first part of this project is the design and development of an easy to use workflow editor that maintains the usability of the original programs and allows for the relatively simple addition of new programs. The result is a workflow editor that currently allows access to over 170 bioinformatics programs each with a simple, common, and descriptive interface.

In the second part of this project a specific workflow application for protein antibody imprinting is examined. Part of the workflow is a program that produces a series of potential alignments for an antibody and a protein, has already been developed. An additional program is developed which uses a greedy search algorithm to select the “best” set of alignments. This alignment selection problem is shown to be NP-Complete. These programs represent a real world example of a bioinformatics workflow.

INTRODUCTION

Overview

Computational scientists are faced with an ever-growing number of programs developed to solve bits and pieces of different puzzles. Multiple programs may be required to solve any one problem, and tedious steps are often required to use the output of one program as the input to another. These steps, while individually small, may slow down a process significantly when it is to be repeated tens, hundreds, or thousands of times. The process of connecting the programs must be manually repeated every time a new input must be analyzed. As though this were not enough, every new program's interface must be learned, and the scientist must have an advanced knowledge of the program's environment for manipulating the inputs and outputs to match.

To this end, much work has been done to develop workflow engines, software programs that connect the inputs and outputs of different programs (Oinn et al 2000). This allows a user to create a workflow of processes that may be saved and repeated using different inputs with minimal difficulties. Many such engines exist, and a few have been developed with computational science programs in mind. However, these engines require an advanced knowledge of the programs being connected and have a bit of a learning-curve for a scientist unfamiliar with workflows or advanced computer operations.

The main goal of this project was to improve upon existing resources to build a workflow editor that would be easy for a scientist to use. This led to a few requirements: First and foremost, it was to be web-based, so as to remove any need for downloading or installing software. It had to allow for the addition and deletion of tasks in a straightforward manner, and lastly, it had to maintain some form of documentation for each of the individual programs as they were loaded by the user. The last requirement was added due to the overwhelming number of bioinformatics programs available, leading us to believe few users would be fully versed in all of the available programs.

Additionally, it was recognized that to continue to be useful the workflow editor would need to accept new programs without any additional programming. To fulfill this need, we looked for workflow engines that could connect to web services (Hashmi 2004).

Research on Existing Methods

Several formats for storing workflows already exist. XML business process languages, such as WSBPEL, WSFL, and XPDL, are already in use within the business community, for a few years now, though none has risen as the absolute standard (Krishnan et al 2002, Van der Aalst 2003). As a testament to this lack of standard, the European myGrid community has settled on their own XML workflow language, SCUFL (Simple Conceptual Unified Flow Language) (Stevens et al 2004). And finally, a few scientific workflow engines used non-XML based scripting languages and petri nets.

A few web-based workflow editors already exist. GenePattern appears relatively robust, though it does not provide access to web services, and is not open source (Reich

2005). Pise provides excellent user interfaces, allows for the easy addition of new programs, but only has minimal piping capabilities (Letondal 2001).

Several non-web-based workflow editors also existed, most notably Dagman, Wildfire, and Taverna. Dagman was tied to the Condor system, which made it too system specific for our needs (Thain 2005). Wildfire used a scripting language (GEL) (Tang et al 2005, Chua 2005). Taverna had an internal workflow engine called Freefluo, and used an XML-based language intended for scientific workflows, XScufl (Oinn et al 2004). Another notable system was Pegasys, which used directed acyclic graphs (DAG) similarly to Dagman (Shah et al 2004). Finally, additional engines Bossa and YAWL (Yet Another Workflow Language) based their workflows on petri-nets (Van der Aalst et al 2005).

Program	Flow Format	Web-based	Open Source
GenePattern	Unknown	yes	no
Pise	XML	yes	yes
Taverna/Freefluo	XML	no	yes
Wildfire	GEL	no	yes
Dagman	DAG	no	no
Bossa	Petri-net	no	yes
YAWL	Petri-net	no	yes
Pegasys	DAG	no	yes

Table 1. A comparison of existing workflow editors and engines.

The following chapters explain the design of the workflow system, the creation of the alignment filtering program and its subsequent workflow, additional use cases for the workflow system, and a summary of what was achieved.

2. DESIGN AND IMPLEMENTATION

Overview of Metaprogrammer

The design of a meta-programming system entails three main parts. 1. Access, both user-interface and program-interface, to a library of programs. 2. An editor for combining the programs into one large meta-program. 3. An execution engine to run the individual programs while tying together their inputs and outputs efficiently. To this end, we used an altered version of Pise, a home-brewed workflow editor, and Freefluo, respectively.

Individual Program Interface

We used Pise as a basis for interfacing with the individual programs. By altering Pise's script generation, we could quickly convert hundreds of programs into web services while maintaining each program's individual user-interface. This approach provided a straight forward way to add new programs without additional coding.

Workflow Editor

We chose Ajax (Asynchronous JavaScript And XML) as the language base for the workflow editor (Garrett 2005). Ajax programming has been around for a few years, though it and its namesake have just come into the spotlight in the past year, with notable web interface implementations by Yahoo and Google (Paulson 2005). It allows a web page to exchange information with a server and update itself without refreshing the page.

This provides a browser-based environment that can easily load the Pise forms and still allow for the appearance of a seamless workflow editor.

Workflow Engine

We chose Freefluo as the workflow engine for several reasons (Oinn 2005). First, it is the engine used by Taverna and thus is continuously supported and used by the global bioinformatics community (Taverna uses Freefluo as its workflow engine) (Oinn et al 2003). It also allowed us to save our workflows in an XML format. Since XML is heavily used throughout the software industry, this allowed us to leverage existing XML routines, rather than have to write new code for a proprietary or little used format.

Alterations and Additions to Pise

Pise's original design used XML files, which contained input and output descriptions for a given program, to create html forms and cgi scripts for that program (Letondal 2005). To add a new program interface, one simply created an XML file with the inputs and outputs of the program. Running Pise with the XML file would then generate the web interface, providing a relatively simple way to make new programs available.

To have Pise generate the web interface from the XML files, there are three main steps:

1. Configuration

```
./configure --prefix=/home/pise
```

2. Make

```
make program=<program name>
```

3. Install

```
make install=<program name>
```

Some modifications were required to make Pise better serve the needs of a workflow. First, with multiple forms required on one web page, generation of a complete web page was no longer required. Simply the form portion of the page would be needed. Second, since another program would be interfacing the programs, instead of a direct web user, the cgi scripts would function better as web service.

The first step was very straight forward. The script that generated the html files, make-html.pl, was used as the basis for the script make-form.pl, and altered to generate only the form portion of the html page. This provided us with html forms that could easily be added to and removed from an Ajax web page.

The second step required more reworking. The cgi scripts provided visual feedback that was no longer required. The script make-cgi.pl was used as a template for the script make-service.pl and alterred to return the direct output of the program, rather than a web page. The make-html.pl was used again, this time as the basis for make-wsdl.pl. This file was then changed to create a WSDL file describing how to operate against the web service (Cerami 2002). (WSDL files are commonly used in conjunction with web services, so a computer program may simply check the WSDL file to determine how to communicate with the web service.)

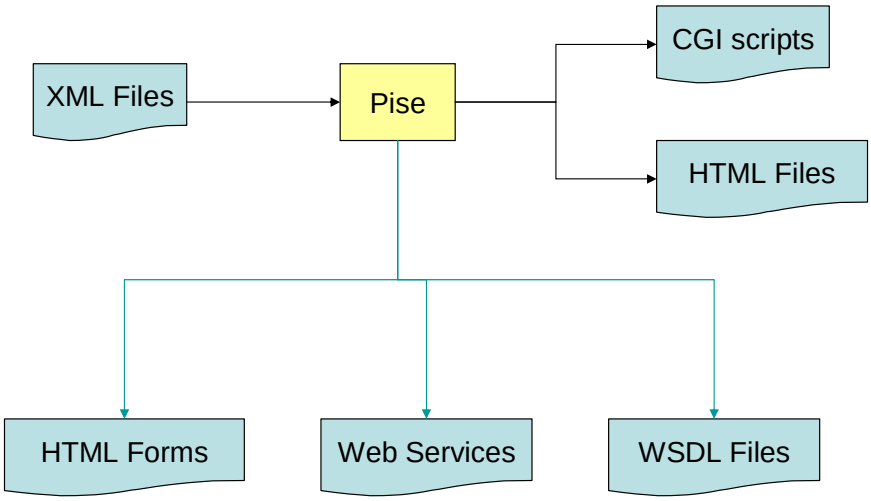

Figure 1. A simplified view of the Pise file generation process. The blue arrows indicate paths that were added to the process by the new perl make files.

Finally, the main makefile and configuration files for Pise were altered to include the new make perl scripts. Including the new features in this manner allowed us to maintain Pise's three step file generation process exactly as is.

With these new modifications in place, we are able to easily generate user-friendly forms for the workflow editor and web services for every bioinformatics program supported by Pise, currently well over 200 programs. Additionally, new programs may be added to the line-up simply by creating an XML file and running it through the modified Pise program.

To expedite the process of adding new programs to the set available for the workflow editor, a web application was developed to handle nearly any input for generating XML files for Pise. (The 'pipe' parameters were excluded as they would be redundant with the workflow program.) The application was required to be very

dynamic, since each program has a differing number of parameters and each parameter has differing attributes. To this end a form was developed that could easily be extended by the user to encompass any combination of parameters. The user can add additional parameters of Pise approved types to the form and hide or show the advanced attributes of a parameter without reloading the web page. Filling out this web form becomes the only step an end user may need to submit a program for inclusion into the set available to the workflow editor. The generated XML file may then be included in Pise's library and when Pise is rerun, it will build the necessary web services and forms for the given program.

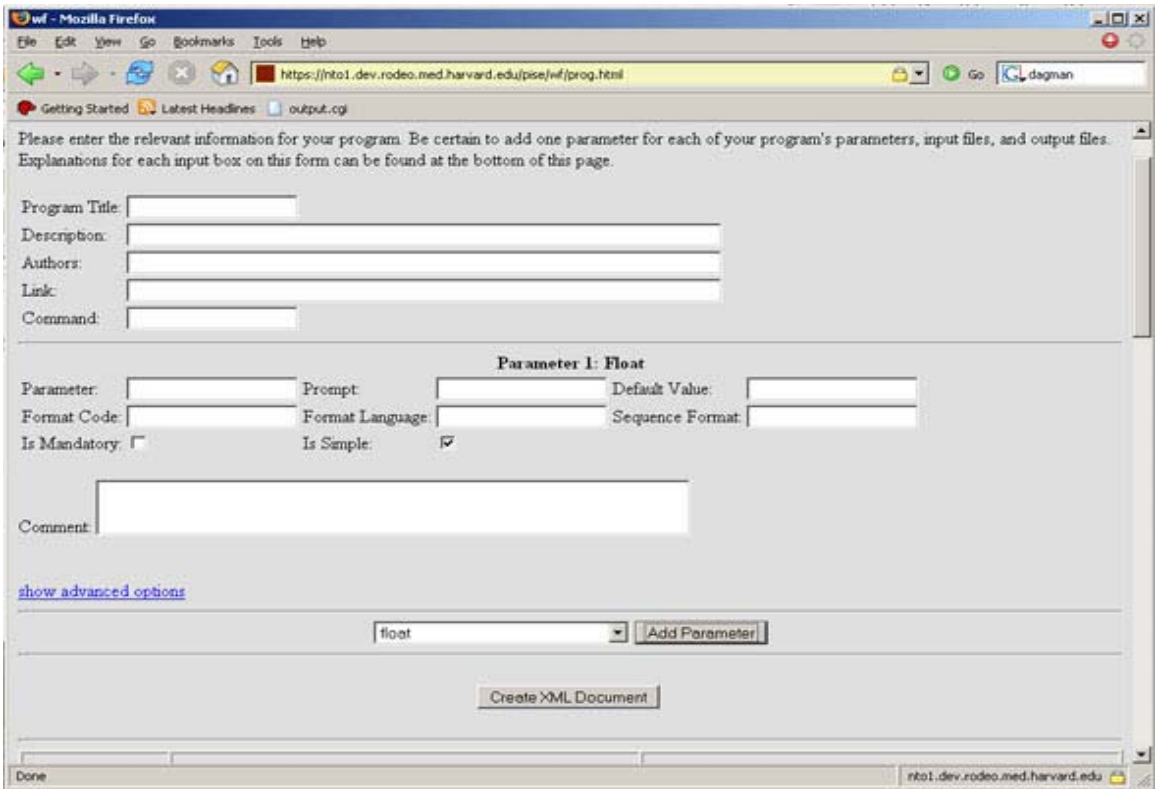

Figure 2. A screenshot of the Program Description XML generator.

The Workflow Editor

The Workflow Editor has three main functions: provide the tab interface; retrieve program information from the server; and XSCUFL workflow processing. Workflow processing refers to saving, loading, and sending the XSCUFL files for execution.

The tab interface seamlessly hides and displays tabs as they are selected or deselected. It also allows a user to add or delete any tab. When a new tab is generated, it appears with a list of programs to choose from. Selecting one of these programs generates the appropriate form for the given program. A save option allows the current order of tabs, with their appropriate forms, to be saved. Once saved, this workflow may be loaded from a web browser, or run against the Freefluo work engine. The workflows are saved as XSCUFL files, which are the native format for Freefluo (Oinn et al 2004). The Workflow Editor has functions for converting the XSCUFL files back into web forms.

A brief note on the GUI references is included here. A tab is an individual menu item displayed horizontally. A pane is the display revealed by the tab. As such, each pane has one tab and vice versa. A collection of tabs and panes is referred to as a hypercard. An active tab or pane is the one in the given hypercard that is currently visible. Only one pane and its respective tab should be active at any given time.

Initially, the user sees one pane with an option of loading an existing workflow or adding a tab to create a new one. Each tab that is added shows a list of available programs in its pane. Only the active pane (the last one the user selected) is visible at any

given time, though inactive tabs are visible but greyed out. The user may continue to insert new tabs or select a program from the given list.

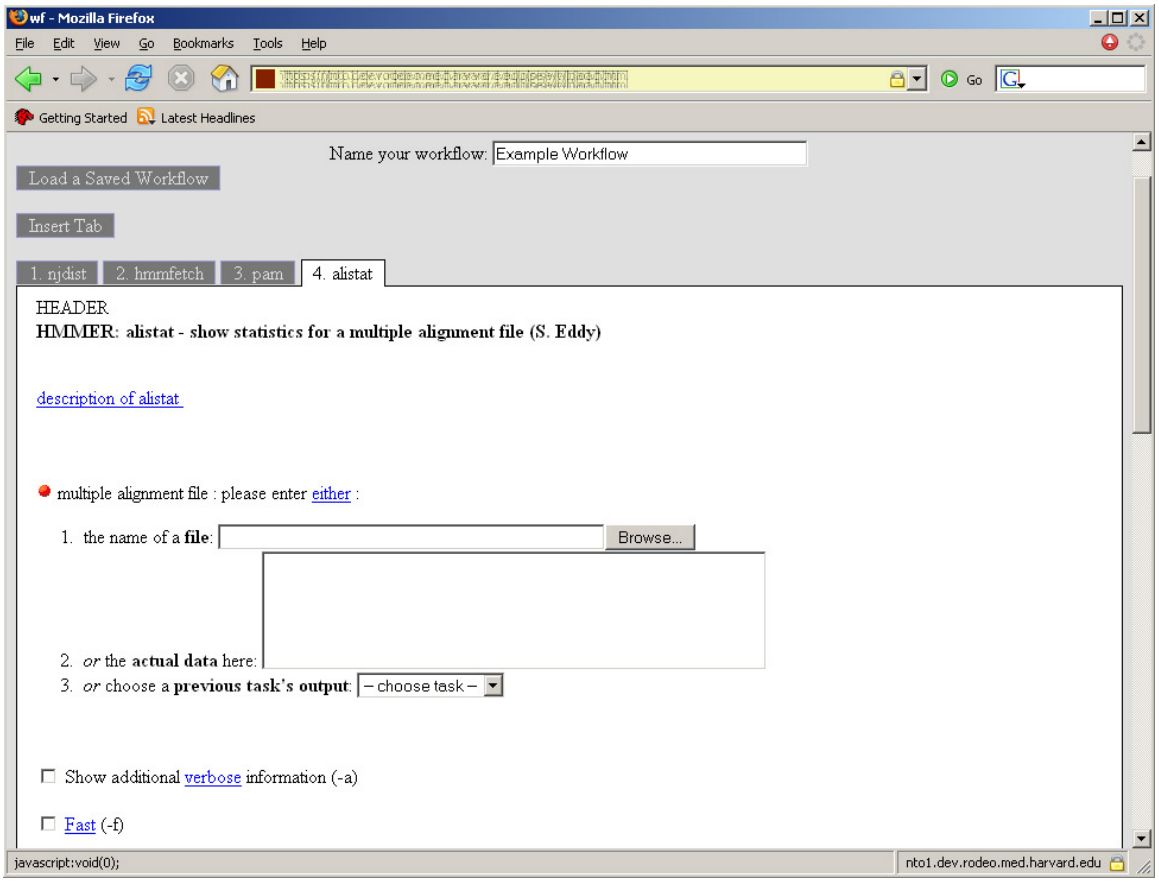

Figure 3. An example of the Workflow Editor's tab interface.

Once a user selects a program, the list of programs is replaced with the input form for the given program. On this input form, wherever sequence or file information is inputted, the user also has the option of choosing the output of a previous task. This step is the key to the workflow editor, as it links the outputs and inputs of the programs together.

A user may switch between programs in the workflow by selecting the appropriate grey tab. This will hide the current program and display the selected program (or list of programs if the user has not selected one yet). The same program may occur multiple times in a workflow, though each occurrence is treated as a separate instance, and the tabs and input options are numbered to uniquely identify each occurrence.

A user may also delete programs from the workflow, though an alert box appears if the user attempts to delete a program with output that is currently linked to another program.

Finally, the user has the option to save a workflow. The workflow is saved along with all current data in an XSCUFL file. (A workflow may be saved with or without input data.) When reloaded, any saved data is loaded into the form boxes as well, though saved files will now appear as data in the text area boxes.

For executing the saved workflows, a Java Server Page (JSP) was created based on one provided by the DALEC project (Oinn et al 2005). The JSP receives the XSCUFL file and uses the taverna java library to execute it through freefluo. Freefluo then handles the calling of each individual web service and the data passed between the services. The output returned by Freefluo is then saved locally so it may be recalled at any time.

A few additional perl scripts were written to present information from the server, rather than relying on purely text based files. These scripts return lists of existing workflows, programs available for inclusion in a workflow, and the outputs for any given program. The script for retrieving the outputs of any given program reuses Pise’s

generated libraries so the addition of new programs does not require any additional information to be inserted at this step.

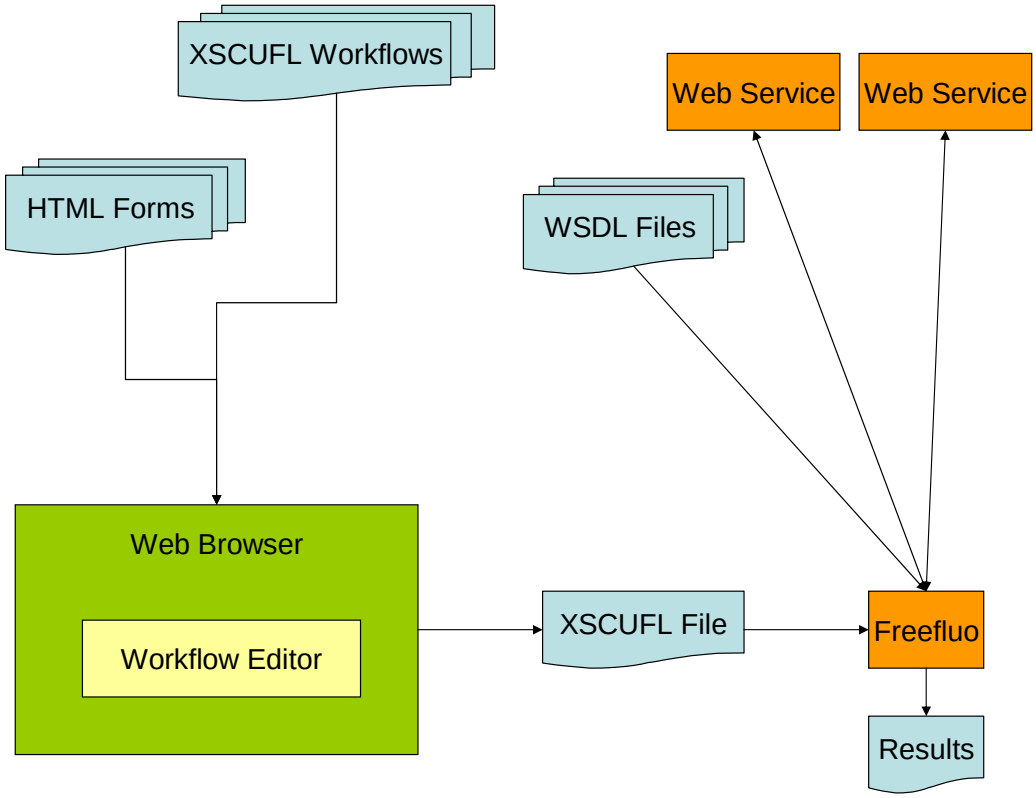

Figure 4. An overview of the entire workflow editing and executing process.

3. ALIGNMENT SELECTION FILTER

Alignment Selection Problem

Given an antibody that binds with a known protein, typically fifty to a hundred probes are bound to the antibody. The probes mimic a portion of the protein's epitope (the binding site of the antibody). An alignment refers to the matching of a peptide probe's binding positions relative to the primary protein sequence. These alignments are not straight-forward, as they may be discontinuous. This is due to the nature of protein folding. Discontinuous positions in a protein sequence may in fact be adjacent to each other in the three dimensional space of the folded protein. If a probe represents such points in a fold, when it is matched against the sequence it will appear as a discontinuous alignment.

Currently, the EPIMAP program produces a set of groups of alignments (Mumey 2003). Each group represents one probe that bound to the given antibody. Within an individual group, EPIMAP may have predicted one or several possible 'best scoring' alignments between the probe and the protein. The goal is to choose one alignment from each group in the set such that the chosen set of alignments best agree with each other.

A probe is generally a sequence of around ten residues and a protein is a sequence of up to a couple hundred proteins. Given a probe, the alignment places these residues along side a position in the protein. If the probe and protein positions are continuous (i.e. positions 2 and 3 on the probe fall next to each other on the protein), then a "run" is said

to have occurred. These runs may occur in a forward or backward direction, just as long as the positions are all continuous.

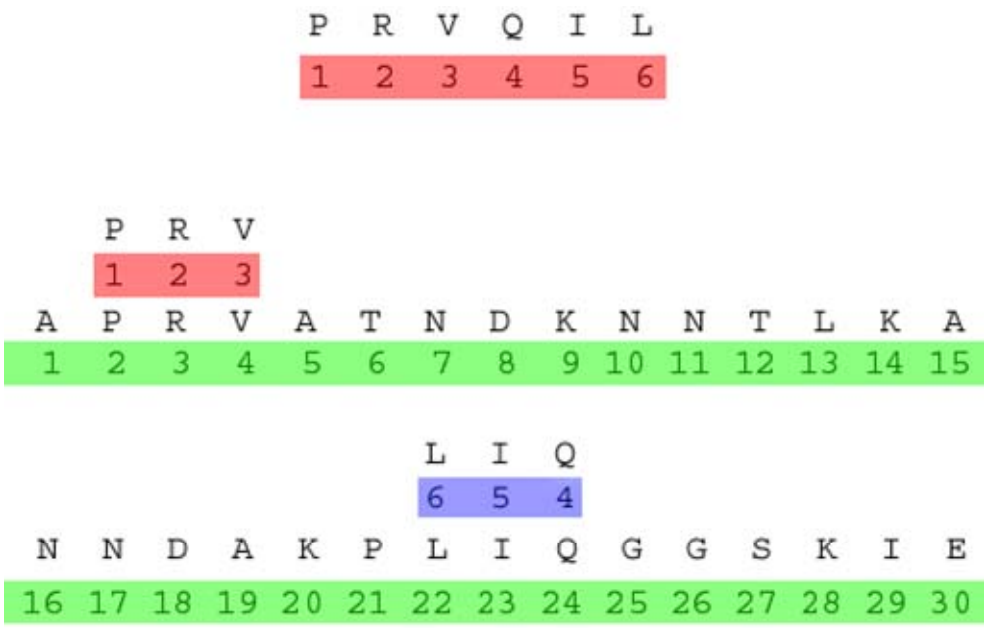

Figure 5. An example alignment. The probe PRVQIL is aligned against a made up protein. Two runs occur in opposite directions.

It is important to note the positions of the probes are numbered arbitrarily, and therefore the numbering system is purely positional and not directional. (The numbering system could have started on either side of the antibody.) This makes the definition of a direction of a forward or backward direction ambiguous, making it difficult to compare one alignment to another. To this end, we introduce the concept of complement alignments. An alignment's complement is another alignment with identical positions which are considered to be in the opposite direction. So if a given alignment calls 1, 2, 3 forward than its complement would call 1, 2, 3 backward. However, for a specific alignment, if 3, 2, 1 is considered forward than 4, 5, 6 within that alignment must be

considered backward to maintain consistency of direction for that alignment. Thus a run of positions 1, 2, 3, 6, 5, 4 may be considered forward than backward for one alignment and backward than forward for another. Using both an alignment and its complement, one can compare alignments with arbitrary directions while maintaining directional consistency within the individual alignments.

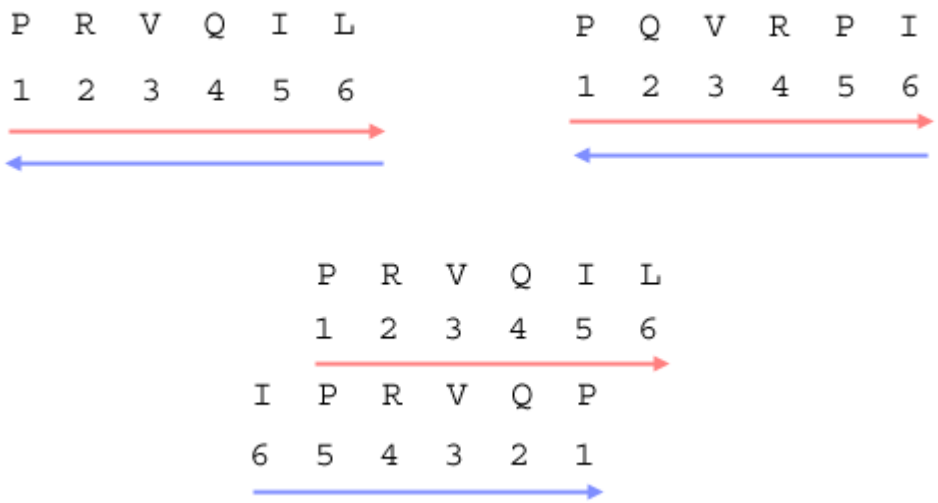

Figure 6. The probe PRVQIL is aligned with the antibody PQVRPI. In this case, the red and blue arrows represent the complements and their individual concept of forward. Had the complement not been used, the positions would still align but the directions would not. It should be noted the complements of the selected directions would match as well.

A match is said to occur when two alignments have runs in the same direction (increasing or decreasing) at the same position on the protein. The position on each probe is irrelevant, as long as both runs are either both increasing or both decreasing. If multiple probes align at the same position, some increasing and some decreasing, then the larger sum of matches is used. For instance, if five antibodies at position B are increasing, and three antibodies at position B are decreasing, then the score for position B is five. A match is possible for every position on the protein, though there may only be as many

matches as there are positions on the probe and possibly fewer if the probes do not align with the same protein positions.

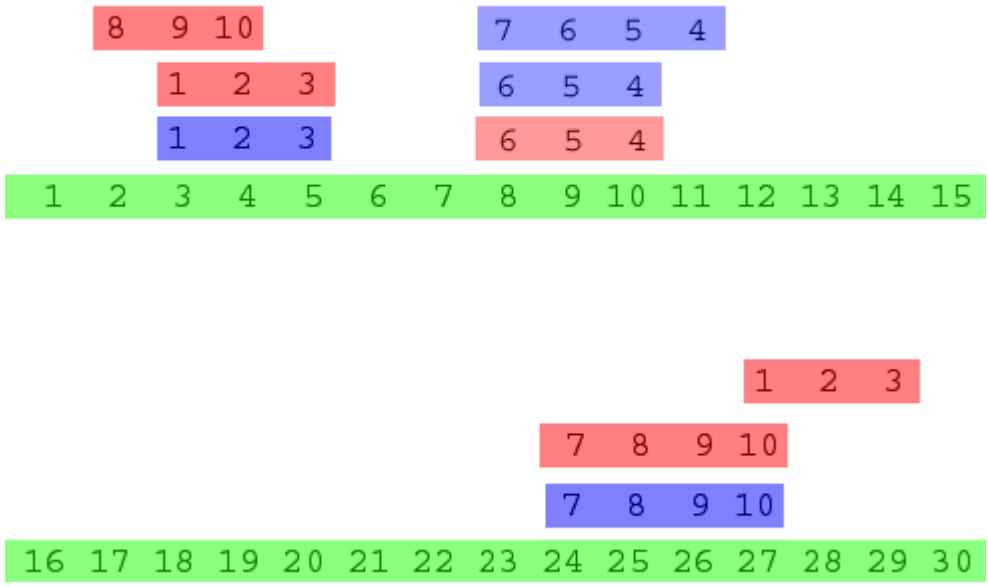

Figure 7. The positions of three probes are shown aligned against the positions of a protein. Here red represents a forward direction and blue represents a backward direction. The bottom two probes are complements. The top probe agrees best with the upper complement (middle probe).

Thus, as stated previously, the goal is to select the set of alignments (one from each group of alignments) that has the most agreement (summing the matches) with all of the alignments in all of the groups.

NP-Completeness of the Alignment Problem

A reduction from 3SAT is used to show the NP-completeness of the alignment selection problem, which we well refer to as ASEL (Garey 1979). To begin with, we formally define the ASEL problem:

> Given a set of S probes aligned over X protein positions, the score of the set is defined as follows: For every protein position i, any continuous probe positions in the set S aligned to the position i are summed into one of two categories: increasing or decreasing. The score for position i is the greater of these two sums. The overall score of the set S is the sum of all of the position scores. The ASEL problem is thus: Given a set of groups of probe alignments, chose one probe alignment from each group such that the score of the set of selected probe alignments is maximized.

The first step is to convert input for 3SAT to input for our supposed ASEL solver. Given the set of literals from the 3SAT input, each literal is considered an alignment, while each clause is considered two continuous positions on the protein. If a literal occurs in a clause, then the corresponding positions in the protein are considered to have an increasing run occurring in them. Likewise, if the complement of a literal occurs, then the corresponding positions in the protein are considered to have a decreasing run occurring in them. The literals are grouped in sets of two, such that each literal and its complement are included in one set.

We then include two additional sets, each with one alignment spanning the length of the protein in an increasing manner. The purpose of this is twofold. First, since each set is an alignment and its complement, there would be two maximum alignment solutions. The addition of increasing sets nullifies one of the solutions, favoring the

maximum solution with more alignments in the increasing (or true) direction. Secondly, as shown later, it will provide a clear threshold at the point of satisfiability in our scoring mechanism.

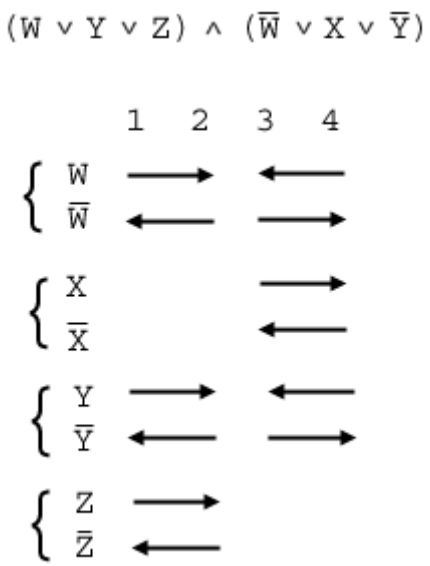

Figure 8. An example of input for 3SAT converted into input for ASEL. Positions 1 and 2 represent the first clause, while 3 and 4 represent the second clause. Note the inverted variables have opposing directions to their counterparts. The brackets represent the individual groups of alignments.

#→	score
0	6
1	6
2	8
3	10

Figure 9. The four rows represent the four possible combinations of the number of true literals in a clause (or forward facing directions in a pair of positions). 0 and 1 have the same ASEL score since both possibilities have 3 pairs of directional matches (when the two additional forward arrows are taken into account). The resultant ASEL scores are not useful, since they do not distinguish between unsatisfied (0) and satisfied (1, 2, 3) clauses.

At this point, the satisfied and unsatisfied clauses are still indistinguishable to the ASEL solver. By definition, any given clause is satisfied if at least one of its literals is true. So, we produce a system of alterations to our pairs of positions such that satisfied clauses will always produce greater scores than unsatisfied clauses.

Each pair of positions on the protein is turned into nine new pairs. (Each pair of positions represents an individual clause with a run for each literal within the clause.) The new pairs each represent a perturbation to the original pair of positions. The first three pairs each contain one run reversed from the original. The second three pairs just repeat the first three pairs, while the last three pairs each contain two literals reversed from the original. As shown by the chart, the resulting ASEL scoring will produce a 60 for any unsatisfied clause and a 66 for any satisfied clause.

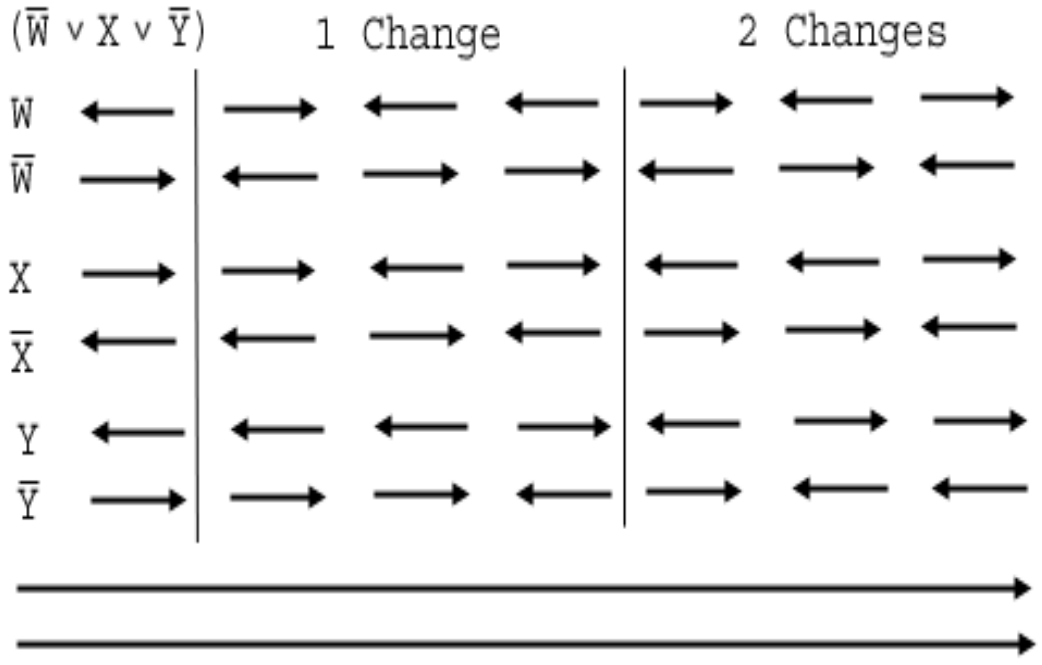

Figure 10. The original alignment created by the clause, and the six new pairs of positions created through alterations. (Nine new pairings occur when the three pairs in the '1 Change' category are used twice.) The two full length arrows represent the two additional alignments mentioned above.

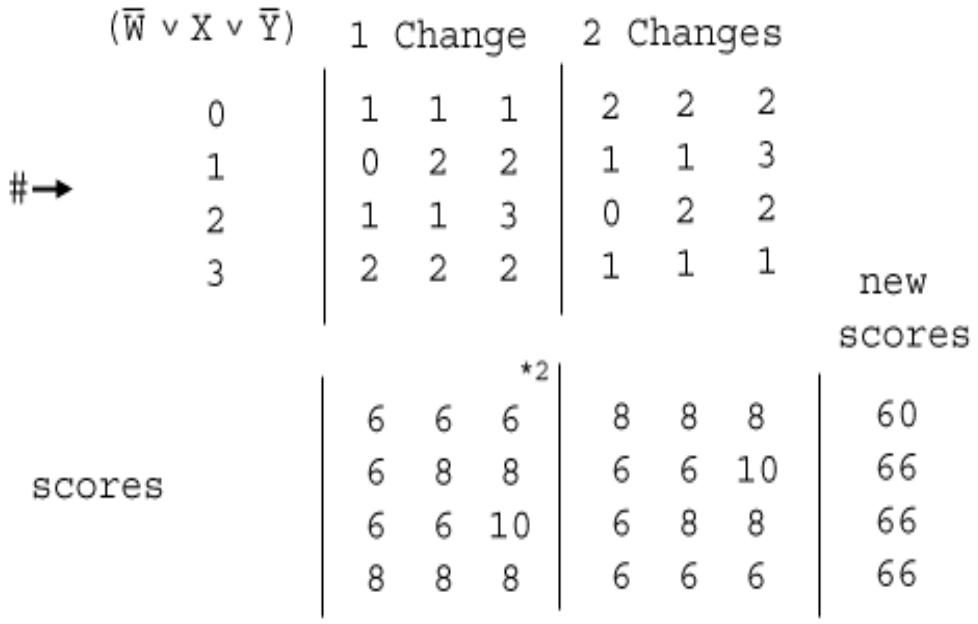

Figure 11. The top columns represent the possible new number of forward facing arrows given the number of changes to the original combination. The bottom columns represent their respective ASEL scores. (Note the '1 Change' category is multiplied by two since each '1 Change' pairing occurs twice.) The resulting final ASEL scores return 60 if the original clause was not satisfied and 66 otherwise.

These sets are then given as input to an ASEL solver. The sets are grouped such that ASEL may only choose only alignment for each pair of literal complements. By choosing one alignment from a pair of literal complements, the literal chosen may be considered to be true, along with all positions in the alignment with increasing runs. Conversely, any positions within the alignment with decreasing runs may be considered false. As shown before, the maximum score possible, 66N, is achievable if, and only if, every clause is satisfied. If the ASEL solver returns a score of 66N we will know the clauses are all satisfied. If the ASEL solver returns any other score, than there is no combination of literals that may satisfy all of the clauses. Thus an ASEL solver is capable of solving 3SAT, an NP-complete problem.

Greedy Search Algorithm

Since the alignment filtering problem is NP-complete, thus unsolvable within our constraints, a greedy search algorithm is used to achieve a ‘best guess’ answer.

For our problem, the sets of alignments given are predictions from the EPIMAP program. Since only the continuity, and not the direction, of the antibody positions is known within these alignments, the complement of each alignment must also be included when the alignments are read into the program.

To prepare for the algorithm, we assign each probe to a level of a tree. Each node at any given level represents one alignment for that level’s probe. A node has one child for every alignment in the following level’s probe. So, if probe 1 has 3 alignments and probe 2 has 8 alignments, than level 1 will have 3 nodes each with 8 children, giving level 2 a total of 24 nodes.

Each node also contains a score. The score represents a sum of the scores for each position. Each position’s score is the higher tally of increasing or decreasing runs at the given position for all of the alignments of all of that node’s ancestors and the node itself. If a node has two ancestors, then a maximum score of three is possible for any given position, with a maximum node score of three times the number of positions.

A greedy search algorithm is then used to seek out a good solution. Each level is assigned a bounded priority queue. Initially, a greedy depth first approach is taken. Starting at the top, the children of the lowest scoring node are used to fill the next level’s priority queue. The process is repeated down the tree, and then started over again at the

top, each time following the current best node. It should be noted that if the priority queues were not bounded, this process would attempt to follow every solution. However, due to the bounds, when the limit of a priority queue is exceeded at any level, nodes on that level that are deemed less likely to contain a good solution, i.e. nodes that have a lower score, are thrown out.

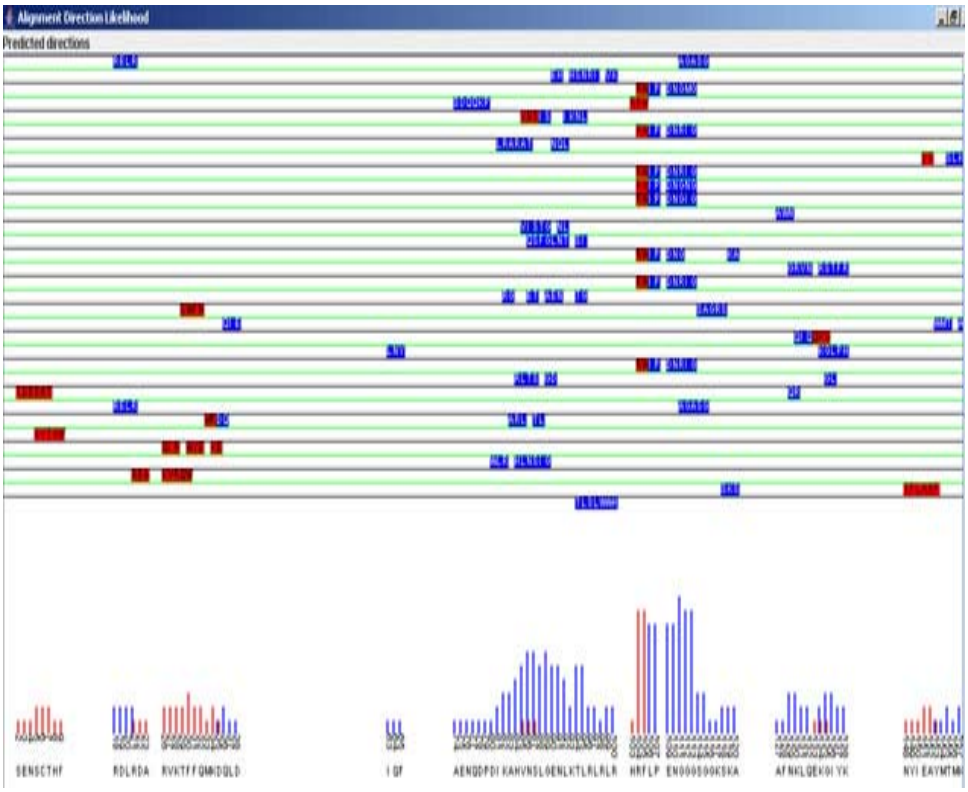

Figure 12. An example of the graphic output of the alignment selection program. The rows represent the probes and their selected directions. The histogram below shows the protein and the number of matching probes.

A final solution is achieved when all of the bounded priority queues are empty. The final result is a set of alignments, one from each group provided, with a high set of directionally matching runs. The program then produces the selected set of alignments as a graphic image along with a histogram of the matches or as an ASCII text file.

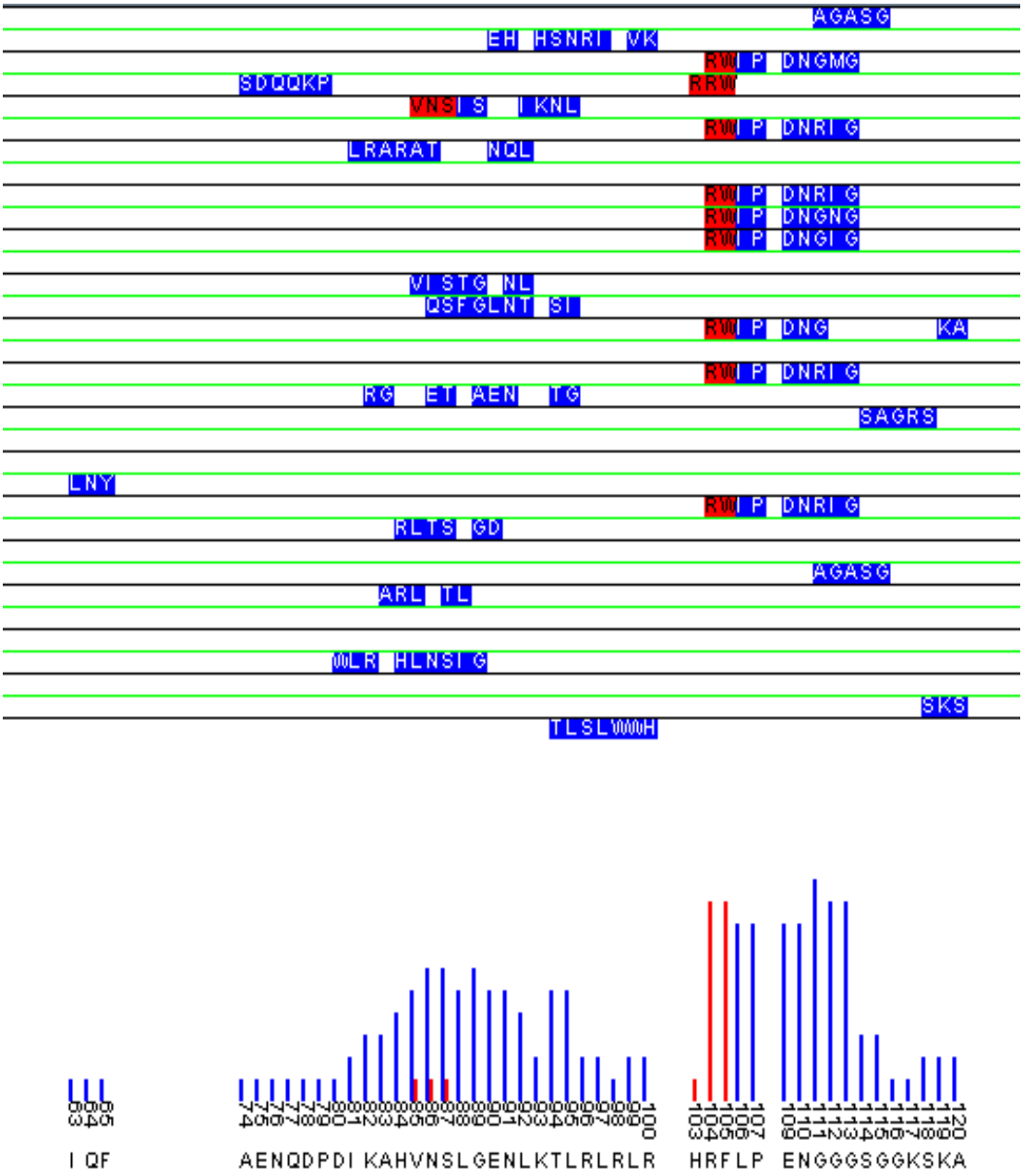

Figure 13. A cropped example of the graphic output of the alignment selection program. In this example, a structural twist or turn is evident from the significant change in direction at positions 105-106. It is also evident that position 108 is most likely buried just beneath the surface, as it has no apparent bindings while positions 107 and 109 have high binding affinities.

4. ADDITIONAL USE CASES

The alignment filtering programs are just one form of workflow among multitudes that may benefit scientists. A few more examples that have been successfully completed using Metaprogrammer are given here.

Emma, Prophecy, Prophet

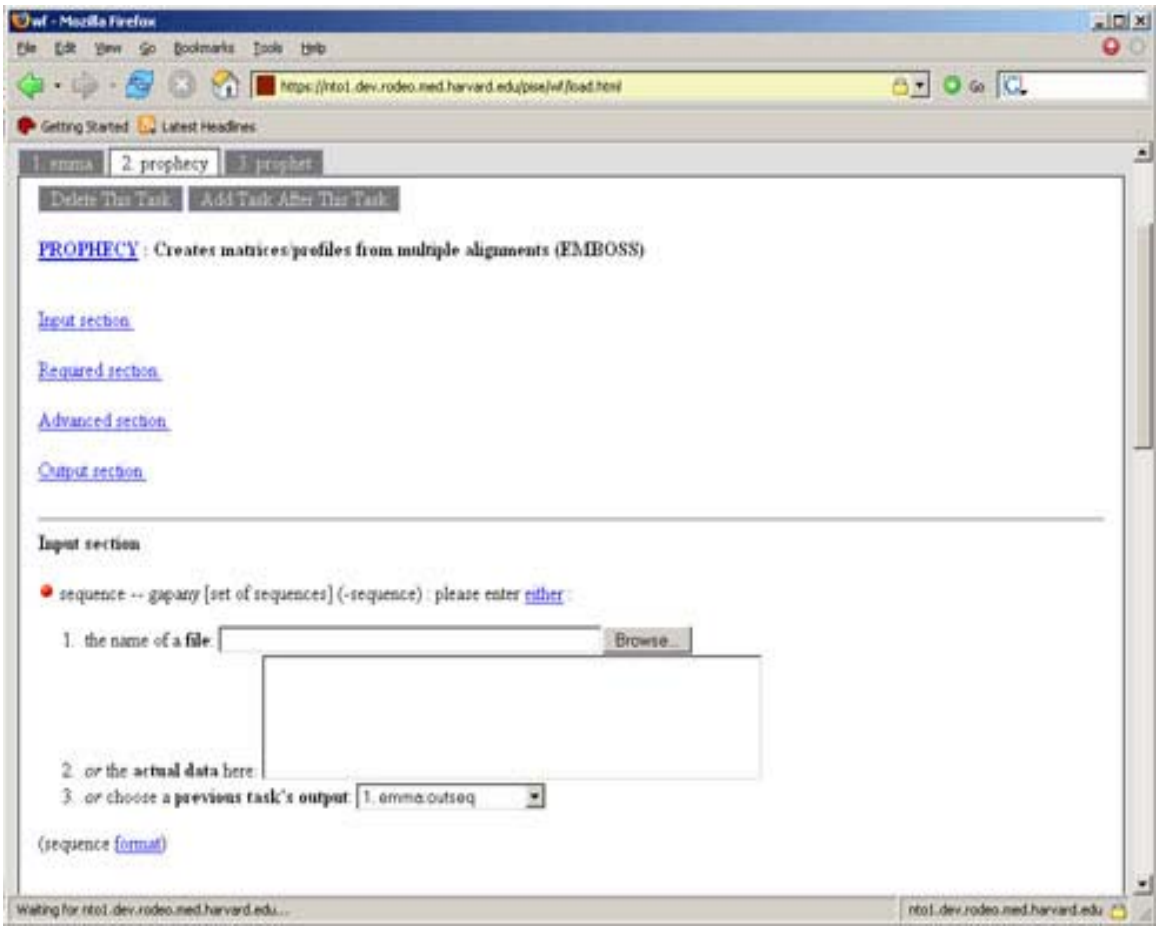

Figure 14. A workflow consisting of emma, prophecy, and prophet.

To begin with, emma is the emboss interface to ClustalW, and as such will align multiple sequences (Faller 2005). The multiple sequence alignments identify both conserved amino acids and amino acids with similar properties that may be substituted for one another. Ideally, a scientist would align a group or family of related sequences.

The output of emma, a multiple sequence alignment, is then passed to prophecy where it is turned into a profile matrix (Bleasby 2005-1). The profile matrix is a representative set of probabilities of amino acid positions. An amino acid that is found to be highly conserved at a given position in the multiple sequence alignment will have a high probability for that position, while other amino acids will be given a near zero probability of occurring at that position. This provides a scoring mechanism for checking other sequences against this multiple sequence alignment.

Finally, the program prophet takes two inputs: the output of prophecy (the profile matrix) and a sequence. Prophet then tests the sequence against the profile matrix to produce a score of the similarity of the sequence to the profile matrix (Bleasby 2005-2). If the match is significant, the scientist may then reason the sequence belongs to the family of sequences previously aligned in the emma program.

Emma, Hmmbuild, Hmmalign

This workflow again begins with Emma to produce a multiple sequence alignment from a set of sequences. The output of Emma is then piped to the input of hmmbuild. Hmmbuild uses the given multiple sequence alignment to create a profile

hidden Markov model (HMM) (Eddy 2003). Much like the profile matrix from Prophet, the profile HMM provides a scoring system for the multiple sequence alignment.

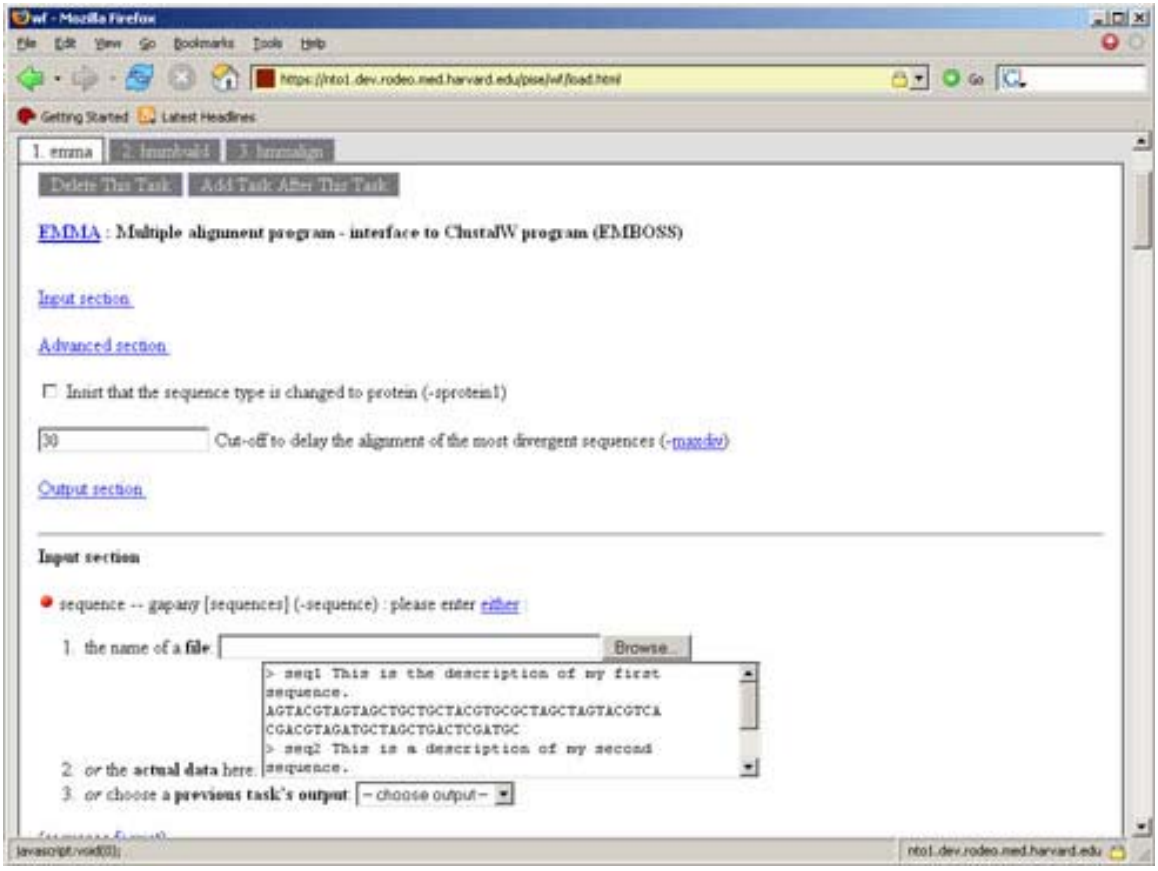

Figure 15. A workflow consisting of emma, hmmbuild, and hmmalign.

Finally, the output profile HMM from Hmmbuild is passed to Hmmalign, along with a set of sequences. The sequences given are aligned to the profile HMM and a multiple sequence alignment is outputted (Eddy 2003).

Emma, Prophecy, Prophet, Hmmbuild, Hmmalign

The two examples given show how the Metaprogram can easily turn multi-step processes into one-click workflows. To take this point a bit further, we can combine the two workflows into a single workflow.

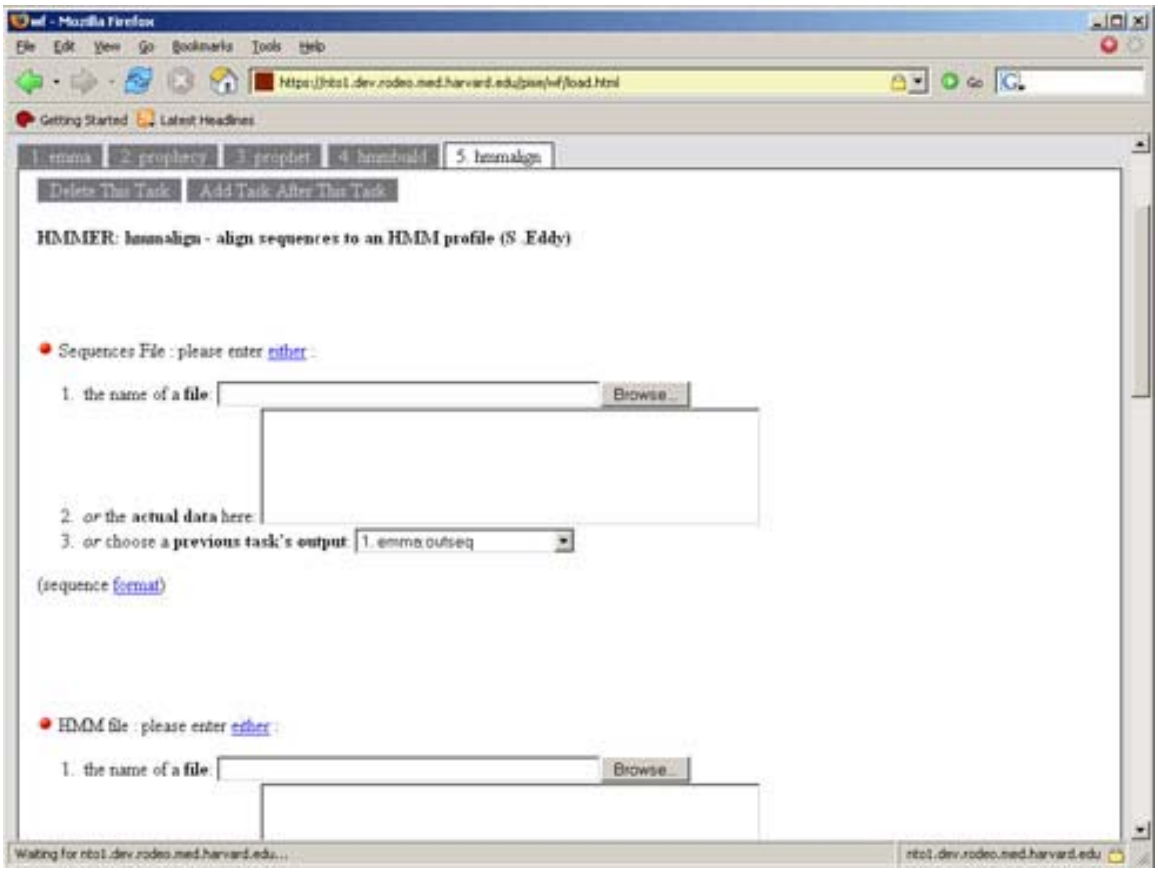

Figure 16. A workflow consisting of emma, prophecy, prophet, hmmbuild, and hmmalign.

Here the output from Emma is sent to the programs Prophecy and Hmmbuild. Prophecy still sends its results to Prophet, and Hmmbuild still sends its profile to Hmmalign. In this manner, the workflows from the previous two examples can

accomplished in one execution, and their results will be returned simultaneously for comparison (provided they are checking the same family of sequences).

These are just a few examples of how a workflow system can improve and speed up data analyzing for scientists.

5. CONCLUSION

With the ever growing number of programs available, and the continuously increasing complexity of operations being performed, workflows are finding more use within the bioinformatics community. Metaprogrammer reduces this complexity by combining some of the strongest features of Pise and Freefluo. First, Metaprogrammer uses Pise's GUI generating abilities to create common and useful interfaces to each individual program. Second, Metaprogrammer's unique tab interface allows for a simple web-based display of each program's highly descriptive interface within the workflow in a serial and straightforward manner. The use of Freefluo provides a robust and well maintained workflow engine with the possibility of including additional features into Metaprogrammer, such as pausing workflows. It also allows the workflows to be saved in a standard and easily retrievable format. Finally, the XML generator allows new programs to quickly be added along with the same common and descriptive web interface.

The current setup of Metaprogrammer allows access to over a hundred and seventy bioinformatics programs. The programs are all run on a cluster of 88 nodes, providing a fast and distributed processing environment for the local scientists. Many more programs already supported by Pise could quickly be added just by installing the programs on the cluster and rerunning the altered version of Pise.

There are a few highly sought after additional features for Metaprogrammer, most notably batch jobs and a sandbox or "scripting area". Batch jobs, such as the ability to run

a thousand different sequences on the same workflow, would be very useful for high-throughput data analyzing. Such automation could be achieved with the addition of accepting zip files as input, where each file within the zip file would be a unique input.

The concept of a sandbox has been partially met by Metaprogrammer. A sandbox allows users a place to write their own scripts, generally for simple manipulation of data. Sandboxes raise many security concerns, since any user with access is capable of executing any program of their choice. To meet this request halfway, the XML generator was created to allow users to easily submit their scripts for inclusion into the general database of programs available within the workflow. Ideally, however, a subset of 'safe' execution commands would be available for manipulating data between programs within the workflow, without the addition of an entire new program.

In regards to the alignment selection problem, the development of the alignment selection filter program provides another step forward to the goal of protein structure prediction. Combined with the alignment prediction program EPIMAP, the two programs will be used experimentally by chemists to attempt to predict structural features of proteins from antibody binding information.

In conclusion, as the number of programs available within the bioinformatics community and the complexity of operations increases, additional steps must be taken to provide the end user with the simplest interface possible. Standardizing program availability through web services is just one step. Providing common and useful visual interfaces to the individual programs can allow access to these programs and workflows to all levels of scientists.

REFERENCES CITED

Bleasby A, Prophecy Manual, Emboss Website
http://emboss.bioinformatics.nl/cgi-bin/emboss/help/prophecy (2005)

Bleasby A, Prophet Manual, Emboss Website
http://emboss.bioinformatics.nl/cgi-bin/emboss/help/prophet (2005)

Cerami E, Web Services Essentials: Distributed applications with XML-RPC, SOAP, UDDI & WSDL O'Reilly (2002)

Chua CL, Tang F, Issac P, Krishnan A, GEL: Grid Execution Language. J. Parallel and Distributed Computing Vol. 65 No. 7 (2005)

Eddy S, HMMER User's Guide, Washington University School of Medicine
ftp://ftp.genetics.wustl.edu/pub/eddy/hmmer/CURRENT/Userguide.pdf (2003)

Faller M, Emma Manual, Emboss Website
http://emboss.bioinformatics.nl/cgi-bin/emboss/help/emma (2005)

Garey MR, Johnson DS, Computers and Intractability: A guide to the theory of NP-completeness. W.H. Freeman and Company (1979)

Garrett JJ, Ajax: A new approach to web applications. Adaptive Path Website
http://www.adaptivepath.com/publications/essays/archives/000385.php (2005)

Hashmi N, Lee S, and Cummings MP, Abstracting Workflows: Unifying bioinformatics task conceptualization and specification through semantic web services. W3C Workshop on Semantic Web for Life Sciences Oct 27-28 (2004)

Krishnan S, Wagstrom P, von Laszewski G, GSFL: A workflow framework for grid services, Argonne National Laboratory, Preprint ANL/MCS-P980-0802 (2002)

Letondal C, A web interface generator for molecular biology programs in Unix, Bioinformatics Vol 17 No 1 (2001)

Letondal C, Pise Website,
http://www.pasteur.fr/recherche/unites/sis/Pise/ (2005)

Mumey, B, Bailey B, Kirkpatrick B, Jesaitis A, and Dratz E, Revealing Protein Structure: A new method for mapping discontinuous antibody epitopes to reveal structural features of proteins. J. of Computational Biology Vol. 10 No. 3-4 (2003)

Oinn T, Freefluo Website
http://freefluo.sourceforge.net/ (2005)

Oinn T, Addis M, Ferris J, Marvin D, Senger M, Greenwood M, Carver T, Glover K, Pocock MR, Wipat A, and Li P, Taverna: a tool for the composition and enactment of bioinformatics workflows. Bioinformatics Vol. 20 No. 17 (2004)

Oinn T, Burdett T, Down T, Prlic A, Kahari A, Graf S, DALEC Website,
http://taverna.sourceforge.net/projects/dalec/index.html (2005)

Oinn T, Greenwood M, Addis M, Alpdemir MN, Ferris J, Glover K, Goble C, Goderis A, Hull D, Marvin D, Li P, Lord P, Pocock MR, Senger M, Stevens R, Wipat A, Wroe C, Taverna: Lessons in creating a workflow environment for the life sciences. Concurrency and Computation: Practice and Experience (2000)

Paulson LD, Building Rich Web Applications with Ajax, Computer Vol 38 No 10 (2005)

Reich M, GenePattern Website,
http://www.broad.mit.edu/cancer/software/genepattern/index.html (2005)

Shah SP, He DYM, Sawkins JN, Druce JC, Quon G, Lett D, Zheng GXY, Xu T, and Ouellette BFF, Pegasys: software for executing and integrating analyses of biological sequences. BMC Bioinformatics Vol. 5 No. 40 (2004)

Stevens R, McEntire R, Goble C, Greenwood M, Zhao J, Wipat A, and Li P, Drug Discovery Today: BIOSILICO Vol 2 No 4 (2004)

Tang F, Chua CL, Ho L, Lim YP, Issac P, and Krishnan A, Wildfire: distributed, Grid-enabled workflow construction and execution. BMC Bioinformatics Vol. 6 No. 69 (2005)

Thain D, Distributed Computing in Practice: The condor experience. Concurrency and Computation: Practice and Experience Vol 17 No 2-4 (2005)

Van der Aalst WMP Don’t go with the flow: Web services composition standards exposed. IEEE Intelligent Systems (2003)

Van der Aalst WMP, and ter Hofstede AHM, YAWL: Yet Another Workflow Language. Information Systems, Vol 30 No 4 (2005)