Introducing Bioinformatics Concepts in CS1

Stuart Hansen
Computer Science Department
University of Wisconsin - Parkside
hansen@cs.uwp.edu

Erica Eddy
Computer Science Department
University of Wisconsin - Parkside
eddy@cs.uwp.edu

Abstract

Bioinformatics is the application of computer science techniques to problems in molecular biology. It is generally studied at the graduate level, because most real world bioinformatics problems require an in-depth knowledge of both biological processes and computer algorithms. Modeling some biological processes, however, is well within the reach of undergraduate computer science students. We have introduced CS1 students to basic bioinformatics concepts via a closed lab exercise and a programming project. The students found the activities engaging and fun. This paper discusses the implementation of these activities and some of the results/lessons we learned.

Introduction

Molecular biology is a rapidly changing field. Vast amounts of new data are being generated. In the Human Genome Project alone, scientists have sequenced the three billion nucleotides of our genetic structure. Other new technologies are making it possible to explore the molecular structures and processes that are fundamental to life. For example, micro-array analysis systems enable scientists to study how groups of genes work together to accomplish some task.

Computer scientists are working with molecular biologists to address some fundamental problems. The intersection of molecular biology with computer science is known as computational molecular biology or bioinformatics. Organizing and mining data, implementing complex algorithms and developing visualization tools are just a few of the areas where computer scientists are contributing [3, 4].

Huge databases of DNA, RNA and protein structures have been established. Much of this data is raw and needs to be mined to find pertinent information. For example, after isolating a gene in a sequence of DNA, a molecular biologist might want to find “similar” genes in related species. “Similar” is the important word, however. Classic computer science approaches to substring searching fail, because the search needs to find a best fit rather than a perfect fit. The problem is compounded by the fact that biological data is almost always "dirty", i.e., containing introduced mistakes or inconsistencies. DNA samples may be contaminated from a myriad of sources. Even if the source data is not contaminated, current technology frequently introduces sequencing errors. If neither of the above is a problem, genetic mutation might have introduced changes in the gene. Computer scientists are applying advanced database and algorithm techniques like dynamic programming and heuristic search to solve this problem.

Another bioinformatics problem where computer science is contributing is fragment assembly of DNA. DNA strands may contain hundreds of thousands or even millions of nucleotides. Current technology is only capable of sequencing strands of several hundred nucleotides. Molecular biologists solve this problem by breaking up the long strands into fragments, sequencing the fragments and then re-assembling the fragments to create the entire sequence. Computers are very useful during fragment assembly because of the number of fragments,. Current algorithmic techniques for fragment assembly rely heavily on graph theoretic algorithms and heuristic approaches.

A final example of a bioinformatics problem is protein structure prediction. Proteins are built from amino acids, based on a template supplied by RNA. Proteins take on complex three-dimensional shapes that determine their function in the cell. There is no complete theory explaining why proteins fold as they do, but most work suggests that their shape is based on minimizing the energy needed to maintain their chemical bonds. Computer scientists are helping to study the protein-folding problem by designing tools to predict and visualize the shapes of protein molecules.

It is important to realize that bioinformatics is not just an academic discipline. Understanding these basic biological processes holds great potential to improve the human condition. Many diseases have known genetic causes or links. Pharmaceutical companies are interested in designing new treatments for these and other diseases.

Bioinformatics at UW-Parkside

In the past, bioinformatics has been generally studied at the graduate level, because most bioinformatics problems require an in-depth knowledge of both biological processes and computer algorithms. Undergraduates, in general, have not yet obtained a sufficient depth of background to understand the subtleties of both fields. Recently there has been significant interest in establishing undergraduate bioinformatics programs.

Our campus has implemented an undergraduate major in "Molecular Biology and Bioinformatics" (MBBS). The major is housed in the Department of Biological Sciences. The curriculum is rigorous. Students choosing this major must follow a strict plan that locks them into specific courses every semester if they hope to finish their degree in four years. Any deviation from the plan will postpone their graduation. While the program is young, its students are some of the brightest attending our institution.

MBBS students take a limited number of Computer Science (CS) courses. The current curriculum requires them to take only two courses: Computer Science I (CS1) and Discrete Mathematics. Several MBBS majors are choosing to pursue a minor in CS. While these students will have a more complete set of computing skills when they graduate, it is understood that their program of study will take at least five years.

The CS Department is continually seeking ways to enhance the computing skills of the MBBS majors. We are doing this in several ways. We are working with MBBS instructors to introduce more computing skills into MBBS courses. We are also modifying our CS1 and Discrete Mathematics courses to include more bioinformatics examples. This paper reports on one of our efforts in the latter category.

CS1 at UW–Parkside

CS1 at UW-Parkside serves multiple student populations. CS majors take it as their first required course in the major. MBBS majors must take it as their only programming course. Management Information Systems majors may select it as an option. Finally, CS1 can be used to meet a general education requirement. The prerequisite for the course is stated in the catalog as "exposure to programming", so students arrive with a wide range of knowledge and experience. CS1 is a four credit hour course with three hours of lecture and two hours of lab each week. Students work in the lab in teams of 2 or 3. Students program a significant number of individual projects outside of lab, as well. We are currently using the Java programming language in this course.

A primary goal of CS1 is to teach students to solve problems by writing computer programs. It is also important that students leave the course with an understanding of the breadth of problem domains addressable by computer programs [2, 5]. Our current curriculum introduces students to problems in such diverse domains as: physics, statistics, data encryption, business data processing, graphics and bioinformatics.

We introduce breadth through closed laboratory exercises and programming projects that solve problems over a wide range of application areas. Before coming to each lab, students read a three to five page pre-lab paper that introduces them to the problem domain. They take a quiz at the beginning of the lab that covers both the previous week's lecture material and their pre-lab reading. Students take the reading seriously, partly because their grade depends on it and partly because they find it engaging. It allows us to pose much more interesting problems during lab. While questions from the pre-lab reading are fair game for the quizzes, this is only to assure that students come to the lab prepared.

An added benefit of this approach is the informal experience students gain in program design. In most of our lab exercises, students work from a pre-existing code base. They add or modify methods in classes supplied by the instructors. In this way, they see and work with well-designed, partially written software. While the scope of the labs and projects in CS1 is seldom large enough to require formal designs, we want students to use good design techniques when approaching a problem. Students report, and their work shows, that the experiences they gain in the lab exercises helps overall performance.

The CS1 Bioinformatics Activities

During the Fall 2002 semester CS1 students participated in two different bioinformatics activities. The first was a closed lab exercise. This was followed immediately by a one-week programming project where they extended the lab results.

When preparing these activities, the instructors sought problems that could be addressed by CS1 students. We chose to work in the domain of translating DNA to RNA and translating RNA to protein sequences. These problems have well understood answers and the processes can be readily explained. While the problems addressed are not considered to be research problems, it did introduce the students to the flavor of problems addressed in bioinformatics.

The Bioinformatics Lab Exercise

The goal of the closed lab exercise was to have students model a DNA molecule within a Java class. When we design a closed lab exercise, we have three objectives in mind. We want students to:

1. Gain some understanding of a problem domain. In this lab, students learn about the field of bioinformatics and the molecular structures of DNA and RNA.

2. Learn something about object-oriented approaches to solving a problem. In this lab, students implement a method guaranteeing the validity of the data.
3. Gain experience using syntactic features of a programming language. In this lab, the students write
    a. a method that returns an object belonging to a class that the student has implemented,
    b. Java code using several methods and operations from the Java String class, including extracting characters and appending additional characters, and
    c. Java code using while loops and if statements.

The instructors provided students with a shell of a DNA class to complete. The shell class contained one instance data member, a String named `sequence`. It contained 3 method stubs, one for a constructor and two for instance methods named `validate()` and `toRNA()`. Within the 2 hour lab period, students needed to complete each of the stubbed-in methods and write a additional method named `reverseComplement()`. Each of these methods is described below:

1. `DNA()`, the constructor. DNA consists of a sequence of *nucleotides*. The nucleotides contain a representative *base* that may be any of four: adenine (A), cytosine (C), guanine (G), or thymine (T). The constructor takes the given String parameter, translates it into all uppercase characters, assigns it to `sequence` and makes sure that it contains only valid characters. To check its validity the constructor calls a helper method, `validate()`.
2. `validate()` walks through the nucleotide sequence and checks each character to guarantee that it is A, C, G or T. If it is some other character, the method prints an appropriate error message.
3. `toRNA()` translates the DNA sequence to its corresponding RNA sequence. DNA and RNA are closely related molecules. The major difference is that RNA contains Uracil (U) rather than Thymine (T). `toRNA()` created a new String object containing the same data as the DNA sequence, but with T replaced by U.
4. `reverseComplement()`. DNA exists most of the time as a double helix. There are two strands of DNA joined together in a long spiral structure. Given one of the strands of DNA it is possible to completely determine the other strand. To do this, the method manipulates the `sequence` by swapping all A's with T's and C's with G's, and then reverses its order.

The Programming Project

The programming project extended the work the students did in the lab. Students built an RNA class and then constructed RNA objects. RNA objects contained a `String` which represented the nucleotide sequence of the RNA. The primary methods in the RNA class were

1. `validate()`, which, like the identically-named DNA class method, checked the String to guarantee all the characters were valid RNA nucleotides, and

2. `toProtein(int frame)`, which translated the RNA String into a String representing a protein.

Students first completed the RNA class, then returned to the DNA class and modified their `toRNA()` method so that it returned an RNA object rather than a String.

The `toProtein(int frame)` method proved challenging. The building blocks of proteins are twenty different *amino acids*. A single amino acid is coded as a sequence of three nucleotides in the RNA string, called a *codon*. Sixty-four distinct codons exist. One or more codons represent each amino acid. Two of the sixty-four codons are "stop" codons, representing the end of the translation process. To complicate matters further, it is not exactly clear where the translation process should start. Since we are processing the RNA three nucleotides at a time, the first grouping of three may start at the zeroth, first or second nucleotide. In biological terms these starting locations are referred to as *frames*. `toProtein()`'s parameter is an integer (0, 1 or 2) representing the desired translation frame.

Students exercised several programming skills while writing the `toProtein()` method. The driving control structure of the method must be a loop that continues until either a stop codon is encountered or the end of the RNA string is reached. Only one of the three translation frames reaches the exact end of the RNA string, however. Many students struggled with writing a condition that checked if they were within three characters of the end of the string.

The loop condition also illustrated the importance of short circuit evaluation. Pseudocode for the loop condition is:

```
while (there is a next codon
         and the next codon is not a stop codon)
```

The sequence of conditions is important, since the second condition assumes the existence of a next codon.

Inside the loop, students needed a fairly large if-then-else block. It contained twenty different Boolean expressions, most with multiple conditions. The following code illustrates just one of the expressions:

```
//Checks for Isoleucine codon
else if(codon.equals("AUA") || codon.equals("AUC") ||
        codon.equals("AUU"))
{
    aminoAcid = "I";
}
```

Once a student began the if block, the coding proved more tedious than difficult. Still, missing or incorrect conditions were the most frequently encountered problems when grading student code.

We, the instructors, had fun creating the datasets for this programming project. Each amino acid is represented by a different letter of the alphabet. Twenty of the twenty-six letters are used. We created messages using just those twenty letters and then translated them backwards into an RNA sequence. Students knew they had a correct answer when their protein sequence spelled a recognizable phrase such as “ANAPPLEADAY”.

Lessons learned

In this section we give some initial reflections on the results of the closed lab and programming project.

Student Reactions

The bioinformatics students were visibly excited to see an application of their major field of study meshed with CS1 assignments. Other students, already familiar with the human genome project from the popular press, recognized a "real world" application dealing with a hot topic. Students continued to discuss the lab and project for weeks afterwards.

An Exam Question

On the first exam following the programming project we set a substring search question couched in bioinformatics terminology. Our purpose was to test the student’s knowledge of Java String methods and control structures, not to test their knowledge of bioinformatics. The question read:

> Chromosomes are long strings of DNA that contain multiple genes. Each gene encodes one protein. One of the things that bioinformaticists want to do is identify where a gene lies on a chromosome. Write a Java method name `findGene()` that takes two Strings as parameters, the chromosome and the gene, and returns the index where the gene starts. If the gene is not found the method should return -1.

The problem is a direct application substring search. There are several easy ways to approach this problem in Java. However, we had not taught the students the `indexOf()` or `startsWith()` methods from the Java String class, so we expected them to implement a naïve matching algorithm. The programming project had given them significant practice with substrings and testing String equality. Our ideal answer used a for loop which contained an equality test for the substring starting at each location.

Surprisingly, only one student received full marks for their answer. We observed and categorized the following errors:

- The most frequently encountered error was incorrect loop termination. Of the 30 students who took the exam, 27 had incorrect looping conditions in their code. This happened even though the looping condition closely paralleled the condition from their programming project.
- The second most common problem was incorrect use of String methods. Our approach in class is never to emphasize memorization, but rather to show students where to look up information about classes and methods. Therefore, it didn't surprise us to see minor errors in method names or parameters, and these errors were not harshly penalized.

The most disappointing error we encountered during grading was the number of students who confused the "gene" in the problem with "codon" from the programming assignment. 8 students out of 30 assumed the gene was of length 3. While "codon" might have been a new term for the students during the programming project, the test questions asked about "chromosomes" and "genes", both of which have appeared many, many times in the popular press. Also, we always encourage students to ask for clarification if they are unclear about the meaning of a problem.

Conclusions

Our approach to closed lab exercises is very successful. Pre-lab readings and pre-lab quizzes work. Students are living up to our expectations. Once it is clear to them that success in the labs depends on their coming prepared, they do. It allows us to develop much more interesting and engaging lab exercises.

Overall, we feel our bioinformatics exercises were a big success. We are looking forward to refining our activities in future semesters. Bioinformatics is a very rich problem domain. We hope to introduce additional bioinformatics exercises in our CS2, Data Structures and Algorithms, and Files and Databases courses.

The prelab and lab are available for download from:
http://www.cs.uwp.edu/Cs241/Labs/prelab9.html and
http://www.cs.uwp.edu/Cs241/Labs/lab9.html

References

1. Benfey, P. (2001) *Gene Discovery Lab,* California, Brooks/Cole.
2. Roberts, E. Editor and Co-Chair, (2001) *Computing Curricula 2001: Computer Science*, IEEE and ACM, available on-line at: http://www.acm.org/sigcse/cc2001/cc2001.pdf
3. Setubal, J. and Meidanis, J., (1997) *Introduction to Computational Molecular Biology,* California, Brooks/Cole.
4. Tisdall, J. (2001). *Beginning Perl for Bioinformatics.* California, O'Reilly.
5. Tucker, A. Editor and Co-Chair, (1991) *Computing Curricula 1991*, IEEE and ACM, available on-line at: http://www.acm.org/education/curr91/homepage.html