CS341 – Final Project (WSU-R)

Design decisions for an MHC peptides project

May 7, 2019

Ajalon Corcoran and Doug Plager

In brief, this project sought to identify and partially assess potential peptides derived from a given gene. Such peptides represent a set of molecules that might be bound by a major histocompatibility complex (MHC) molecular complex and presented to complementary white blood cells to induce an immune response.

The three grading milestones for this project included (1) translating a DNA nucleotide sequence into an amino acid sequence, (2) generating and collecting the translated amino acid sequence as peptides (i.e., relatively short chains of amino acids), and (3) assessing each peptide for its molecular weight and a “priority score” when compared to a user-designated consensus peptide.

An abstract data type (ADT) is defined as a collection of data values together with a set of well-specified operations on that data. Thus, we defined a “PeptideSet” ADT that included an interface of relevant operations (i.e., abstract methods), namely PeptideSetInterface, and an associated implementing “constructor” class, PeptideSet. The abstract methods of PeptideSetInterface included appropriate getters and setters, a toString method, and several other methods relevant to identifying and assessing a set of peptides from a DNA gene sequence.

Because the most frequent operation was likely to be conversion from a three-nucleotide DNA codon to its corresponding amino acid, we chose to establish a LinkedHashMap<String, AminoAcid> dnaCodonTable variable as a private field of our PeptideSet class. This “hash table” functioned as a ‘key’ – ‘value’ map pair to facilitate rapid lookup of a particular codon and retrieval of its corresponding AminoAcid object. We continue to wonder whether it may be advantageous to establish this codon hash table as a non-instance, “final static” constant of PeptideSet rather than as a private, instance field variable; however, perhaps this would complicate reconstitution of a given PeptideSet object if saved. Regardless, this codon hash table approach assisted in the rapid conversion of a DNA nucleotide sequence into its corresponding amino acid sequence (milestone 1). Along with the preceding, we considered using a stack- or queue-type arrangement to pop/dequeue three nucleotide (Nt) characters at a time to provide each consecutive codon but settled on a substring( firstNt, fourthNt) approach to grab each codon, instead.

Thinking about what exactly should constitute an “amino acid” for this particular project and where to store each amino acid translated from a given DNA codon, we chose to develop an AminoAcid class with instances capable of storing relevant name, molecular weight, and electrostatic property information. Each translated AminoAcid object could then be added to a (generic) Java API-available ArrayList<AminoAcid>-type array. We chose ArrayList<E> because of its ease of use and because the length of the array would be known from the user-designated peptide length. Because a given ArrayList<AminoAcid> would ultimately correspond to the sequence of a given peptide, we chose to also establish a Peptide class with an “aggregated” private field variable referencing its associated ArrayList<AminoAcid> (milestone 2). The Peptide class also was designed to include several other private fields for storing relevant sequence position, peptide molecular weight, and peptide priority score information.

In a similar relationship, each Peptide object generated from a gene’s DNA nucleotide sequence would need to be stored as part of the generated peptide set (i.e., as part of its PeptideSet object). Because calculating the number of expected peptides from user-input parameters and a given gene was somewhat more complicated and given the conveniently similar method signatures to that of ArrayList<E>, we chose a (generic) Java API-available LinkedList<Peptide>-type of collection for storage of generated peptides. Thus, this aggregated LinkedList<Peptide> collection was designated as a private field of the PeptideSet class. Because thousands of peptides might be generated from a given DNA nucleotide sequence, we were concerned about whether the .get(i) method of LinkedList<E> would prove problematically slow if each ‘get’ within a ‘for’ – ‘each’ Loop would start from the LinkedList ‘head’ reference rather than step through the LinkedList like a traversal. Although we did not precisely time this operation (e.g., the listPeptides() method), no particularly noticeable output delay was observed in practice.

At this point of the process, each stored peptide is yet to have a calculated molecular weight (MW) or a calculated priority score (PS). These calculations are currently performed in a relatively unremarkable manner (milestone 3). However, we did wonder about the feasibility of applying some sort of 20 amino acid by 20 amino acid “weighted graph adjacency matrix” as a means of providing more accurate priority scores, e.g., considering amino acid size as well as electorstatic property (nonpolar, polar, positive, or negative) to determine a weighted score for each amino acid when compared to each of the other 20 amino acids.

One final notable decision involved the application of the Java Collections Framework (JCF) Collection.sort() method to sort our LinkedList of generated peptides based on their MW, PS, or starting Nt position.

Besides the (a) 20 by 20 weighted adjacency matrix mentioned above, (b) a recursive version of our generatePeptides( \_ ) method, (c) an enumeration set of “np”, “p”, “+”, and “–“ to verify our current consensus sequence input, (d) Comparator usage (e.g., “…Peptide implements Comparator”) for peptide MW, PS, and starting Nt comparisions, and (e) object serialization (“…implements Serializable” by ‘PeptideSet’, ‘Peptide’, and ‘AminoAcid’) for output and future retrieval of a generated PeptideSet object are additional programming concepts which seem applicable to our program but which we currently have insufficient experience with to apply in a timely manner.