Extending Context Free Grammars to the Application of Bioinformatics

Carlos Brenneisen and Ben Nelson

CS3100 Final Project

12/10/11

The application of computational theory to biological thought and processes has been invaluable to both fields in recent years. Biological processes that normally would take immense amounts of time to code and analyze can now be formalized with software in a significantly smaller amount of time. The results from the field of bioinformatics have revolutionized the way people think about genetics and the field of human genome research. This is in addition to pharmaceuticals that are influenced by differing genes, gene therapy, cancer research and many other fields. One of the most commonly referenced bioinformatics projects is the Human Genome Project, a research project done to map the nucleotides making up the DNA of a human being. This 'map' refers to the functionality of the nucleotides in terms of the genes that they make up and the use of computational theory to complete this project, which has made it possible for many new drugs to be developed. The continuing study of the genome assists in many aspects of modern medicine. These, along with the nod at BioPython and its applications to computational theory suggested early in the semester by Professor Gopalakrishnan, are the reasons why we chose to work on this project.

Our main intention for the project was to model biological phenomena such as DNA in terms of finite state automata and to better understand both biological and computational theory and practice. The project required a fair amount of understanding biological processes and as such took some research from both Carlos and Ben to garner the level of expertise to model biological phenomena. We took to researching using biology textbooks, online resources and guides as well as asking fellow students who were majoring in majoring in biology and biomedical engineering. Carlos' background in biology, from beginning his career at the University of Utah as a biomedical engineering student, as well as Ben's from his freshman biology course also helped in learning the material necessary to complete the project.

Much of what was needed to be known was about DNA. DNA, or deoxyribonucleic acid, is a (as the name suggests) nucleic acid that contains the genetic information necessary for life. It manages the functions and development of the cells of all living organisms. The other major macromolecules necessary for life are proteins, which provide many functions for organism, and ribonucleic acid, which provides many functions including assisting in protein synthesis. DNA is composed of several nucleotides: thymine, adenine, cytosine and guanine. These nucleotides are combined in several different ways to form what is known as 'base pairs', these base pairs form the basis of genetic information as they instruct a cell on how to function properly. The base pairs are adenine to thymine and cytosine to guanine. Genetic information encoded in DNA (and sometimes RNA) is used by the cell to function via translation into proteins that are then used in various parts of the cell. Amino acids make up the basis of proteins: when genetic information is being 'viewed' by the cell, the cell has to decipher what amino acid groupings in order to obtain the correct protein. Each amino acid formed in protein synthesis is codified by three nucleotides known as 'codons'. The start of an amino acid sequence is known by a 'start codon' and the ending of the sequence is known by a 'stop codon' and the middle sequence of codons is additional information to be used to generate amino acids that change the structure and type of protein. A 'gene' therefore can be viewed as the nucleotides that are sequenced between a start and stop codon that possesses a specific function within the cell. Any given organism may possess tens of thousands of genes and within them hundreds of thousands of codons to define the processes therein, thus the organization and analysis of these codons is critical to understanding further about how genes function.

Using this knowledge base as a guide we took to planning our implementation of the project in Python. The first task we aimed at getting done was the planning of the projects

implementation. Since DNA was the most complex system we had learned about in our research we decided to first model it in terms of computational theory and then if time permitted we would move on to other complex phenomena. We had decided initially to start by plotting a context free grammar to represent valid nucleotide sequences that could be encoded into genetic information for the use of cells of organisms. After the completion of this context free grammar we would use it to build a push down automata to model the grammar and use it in building our software. The software we were going to build would be a generator in Python that would build random sequences of DNA and then check those sequences for validity and further check them across the BLAST database of genetic information to see what they could potentially be representing the genes of. The software would also be able to generate a deterministic finite automata, nondeterministic finite automata and regular expression to represent the amino acid sequence that we had generated previously.

For the actual implementation of the project we made some major design changes and decisions both in the interest of completing the most amount of the project with the time allotted as well as for simplicity of design. After further analysis of our coding strategy (which was to encode the context free grammar in terms of a push-down automata and then proceed) we determined that the push-down automata would be unnecessarily time consuming for encoding valid sequences of nucleotides. Instead, we opted to make a design based on the context free grammar which we had created which would determine the validity of the sequences with a deterministic finite automaton (DFA). We also learned that differing organisms can potentially have differing start and stop codons which we were unaware of to begin with. From this knowledge we determined that we would need to generate different types of DFA's for each different organism type that we encoded. We thus encoded a DFA generator which would create a DFA for a sequence of DNA given the parameters of all the start codons that were necessary along with the stop codons that were required. We also encoded a random generator for DNA sequences of differing types of organisms and a function to check a strand of DNA against the BLAST database to see what it encoded. These functions also had several helper functions which were included as well. The full functionality of our software is detailed in Appendix A.

The software development process for the project was fairly straightforward. Using pair programming the team developed each of the functions piece by piece as was needed starting with the DNA DFA generator and working from that process forward. Carlos sat as the 'driver' performing the main code implementation while Ben performed navigation and debugging assistance. The pair worked in tandem developing the software of the course of the project. The majority of the project was completed without many software bugs. The problems encountered were due to Python problems (such as syntax errors, array out of bounds exceptions and the like) or due to a lack of understanding of the theory behind the code and why it was generating what it was. One of the major debugging issues was an issue that arose when working on the random DNA generator. The problem ended up being that the random sequences generated were causing problems since they were having nucleotides that were not codons being generated (ie you would have a start codon, stop codon, a middle noncoding sequence and some additional extra nucleotides causing problems). One further issue that arose that gave a major headache for the team was an issue with the DNA DFA generator and how it had problems with multiple codons having a difference of one nucleotide and generating an incorrect DFA for these organism types. The main underlying issue with this problem was Python's dictionaries and how they only accepted having one key for each codon. This would cause a problem for example with the DFA generator since it would generate a transition line for the codon to the correct state, then delete

this line when another codon having the same beginning nucleotide would be seen. It would then add this new codon but the correct DFA sequence would be lost. After many hours of debugging the team was able to resolve this and other issues. The tutorial for BioPython (listed in the appendix) was an invaluable tool for the team in its completion of the project. The tutorial provided many helpful hints and solutions to problems encountered along the way such as how to parse information from BLAST queries, how to perform said queries and how Sequence objects and strings could be related. One major thing we would change about how the team completed the project would be the implementation of a software version control system (such as git or SubVersion) to aid in debugging and implementation.

Theory of computation was the key backbone to the entirety of the project. Without the intense working knowledge of finite state automata, context free grammars, randomization, and string generation (among other topics) this project would have failed. For this project we needed an intimate working knowledge of how to build deterministic finite state automata. We we're required to know exactly how state transitions worked, how to determine which states to add based on new parameters, how to determine the correctness of a DFA, minimization and verification of correctness for DFA's, how to fix an incorrect DFA, and the translation of this knowledge to code. We were also required to know how context free grammars work, the building of context free grammars, translation of context free grammars to strings and the verification of correctness and completeness, the translation of context free grammars to push down automata and to deterministic finite state automata and verifying the correctness of both. We were also required to know the fundamentals on randomized string generation and verifying correctness of the strings using DFA's.

For representing strands of DNA we had decided to use strings, which would later be converted to the Biopython Seq object, with each character representing a nucleotide. We started out by building a context free grammar that would be correct for any strand of DNA. After gaining our background biology knowledge we can up with a context free grammar that we deemed would be appropriate. The grammar would start with the start codons that could be generated by this organism's genetic makeup along with a terminal which would continue the building of the string. The next terminal would make sure that the characters generated would make a noncoding sequence or that it would make a stop codon. We decided to allow start codons to be immediately followed by a stop codon, even though this results in an uninteresting DNA sequence simply because it is still technically valid. We additionally got exposed to several other types of organisms that had differing start and stop codon sequences so we were required to create similar context free grammars for each of them. Verifying the correctness of this grammar was a task done with the use of many parse trees for correct and incorrect strings.

After the completion of the grammar we had initially thought to build a push down automata for which to generate and verify random segments of DNA. This mode of thought was due to the fact that we were initially going to verify strands of nucleotides against other strands of nucleotides, as well as the fact that converting a CFG to a PDA is very straightforward. We had thought that since DNA was constructed in base pairs that we would need to verify both strands were matched correctly in addition to being constructed correctly (ie. the coding and noncoding sequences for the strand were correct). After much discussion and thought we decided that this thought process was wrong. Since we were going to be checking sequences to BLAST to see what they represented we were only going to derive something to verify one side of a

sequence was correct. Thus the added functionality from a push-down automaton such as a stack would be unnecessary and expensive in terms of time to build.

In the end we settled on crafting deterministic finite state automata for this purpose. The DFA that we would be creating would be a direct minimized representation of the context free grammar that we had created. The purpose of this DFA would be to verify that a string of DNA that was passed into it was a valid string of DNA: that it included start and stop codons along with a noncoding sequence of nucleotides. It would be minimal since no other states other than the ones needed directly for encoding correct sequences of DNA would be included and all the states would be unique. No black holes would be generated since all nucleotides would eventually lead to either another codon or would fail to generate a valid sequence but the potential for another codon would always be there. The DFA would take into account three states from any string sequence that was passed into it. There might be the possibility of a noncoding sequence before the coding sequence of DNA (ie. there might be gibberish before the proteins amino acid is encoded), the possibility of the actual coding sequence and the possibility of nucleotides afterwards (ie. the DNA sequence continues on after the coding sequence is completed). For each of these possibilities the program performs the same operations. The main operation for the DFA is to create a path from all the possible codons to an end state. Thus each time you get a new codon for the DFA to generate a path it is necessary to check it against what has already been generated to see if it can use the same path or use some of the same path (if a codon has a difference of one or two it can still use some of a previous path as long as the difference is not in the first and last nucleotide). If the codons differ by more than two it is necessary to create a new state to represent this codon. After going through all the possible codon possibilities for the organism the DFA will be created. The initial state will be a state represented by no characters parsed yet and the final state will be states represented by all valid DNA strands (ie. has at least one valid coding sequence).

The strings that we generated would be randomized strings of DNA built with the context free grammar we had developed. The strings would be built from the ground up starting with a randomized coding sequence. The randomized coding sequence would contain one start codon and one stop codon along with a random assortment of codons that would buffer the space of the string until it reached an allotted size. After generating this string depending on the remaining length of the string the remaining two parts of the sequence for generation would either be a coding sequence or noncoding sequence.

The things that we were able to do while combing this and the Biopython library appeared to be endless. For example, we could were able to use the random coding sequence generator to sequences of DNA that corresponded to existing genes. We could then combine these with noncoding sequences to create a SeqFeature object. Here we were able to say which sections corresponded to coding sequences and what genes they were by using the location object in Biopython. We were also able to check DNA strands returned by blast and run them through the corresponding DFA to check that they were valid according to our own definition. We were also able to analyze those strands and see how one might be able to derive any given sequence by using a CFG and writing out parse trees. There were also many other things that we saw along the way that could be related to theory. For example, the translation methods used in the library could also be mimicked using a PDA, by pushing and popping complement pairs. We could have also applied these concepts directly to RNA and protein sequences, although these are different projects for another time.

This project taught us a lot. We learned a lot of information about genetics that we had either forgotten or was new to us. We also reviewed a lot of the material from the beginning of the semester about finite state automata. This information will hopefully prove invaluable for the final exam and for having a much better understanding and appreciation of computational theory. Cross discipline projects such as this are intense and challenging in many ways. Learning to relate vastly different subjects such as computer science and biology and to make models that are valid for both was an interesting experience to say the least. By far the most rewarding part of the project was searching through the BLAST database with information that we had generated at random for probably the thousandth time and actually receiving results which had descriptions of genes that came from actual organisms in real life. While not world changing, our project was fun and enlightening and ended up giving a perspective to projects in the future that not all computer science has to deal with theoretical subjects and that all scientific subjects can be related.

Works Cited

"About the Human Genome Project." *Oak Ridge National Laboratory*. Web. 11 Dec. 2011. <http://www.ornl.gov/sci/techresources/Human_Genome/project/about.shtml>.

"Amino Acid." *Wikipedia, the Free Encyclopedia*. Web. 11 Dec. 2011. <http://en.wikipedia.org/wiki/Amino_acid>.

"Bioinformatics: Human Genome Research in Progress." *Oak Ridge National Laboratory*. Web. 11 Dec. 2011. <http://www.ornl.gov/sci/techresources/Human_Genome/research/informatics.shtml>.

"DNA." *Wikipedia, the Free Encyclopedia*. Web. 11 Dec. 2011. <http://en.wikipedia.org/wiki/DNA>.

"Genetic Code." *Wikipedia, the Free Encyclopedia*. Web. 11 Dec. 2011. <http://en.wikipedia.org/wiki/Codon>.

"Nucleotide." *Wikipedia, the Free Encyclopedia*. Web. 11 Dec. 2011. <http://en.wikipedia.org/wiki/Nucleotide>.

"RNA." *Wikipedia, the Free Encyclopedia*. Web. 11 Dec. 2011. <http://en.wikipedia.org/wiki/RNA>.

"What Is DNA? - Genetics Home Reference." *Genetics Home Reference - Your Guide to Understanding Genetic Conditions*. Web. 11 Dec. 2011. <http://ghr.nlm.nih.gov/handbook/basics/dna>.

Helping Friends

-These friends helped us understand the biology needed to complete this project-

Renee Merchel
Maggie Reid
Jake McDougall

APPENDIX A
(Software functionality, context free grammars, DFAs, & other figures)

Here is the sample context free grammar that we developed, used for coding sequences for standard organisms: (Must use the common start codon, must end with a stop codon – TAA, TAG or TGA)

S -> atgD
D -> XD | tag | taa | tga
X -> cY | aY | gY | tT
Y -> cZ | aZ | gZ | tZ
Z -> c | a | g | t
T -> cZ | aA | gG | tZ
A -> c | t
G -> g | c | t

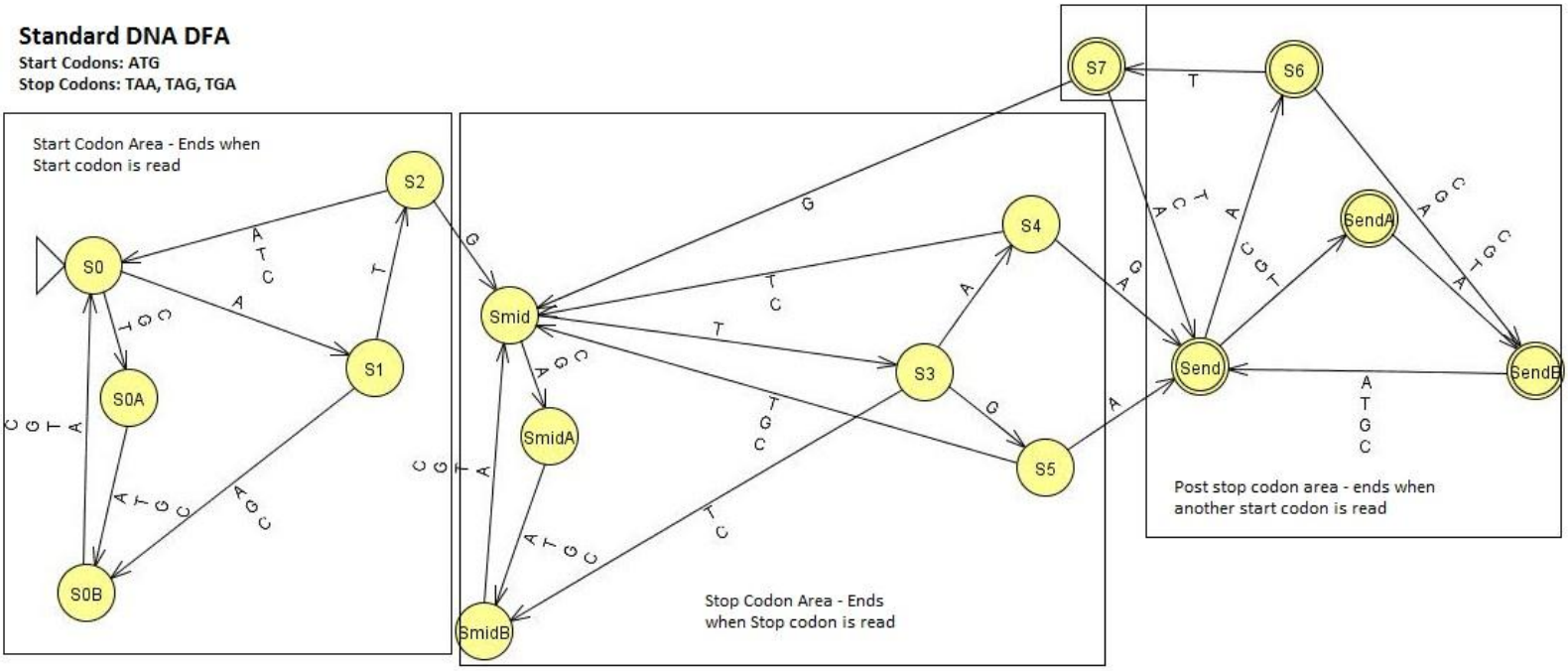
Below is a similar one, created for noncoding sequences using the standard translation table: (the start codon, ATG, is not allowed)

S -> XS | e
X -> cY | aA | gY | tY | e
Y -> cZ | aZ | gZ | tZ
Z -> c | a | g | t
A -> cZ | aZ | gZ | tT
T -> a | c | t

Finally, we can combine these two to create an infinite amount of strands of DNA consisting of coding and noncoding sequences. We require that there be at least one coding sequence somewhere in the strand. Here C is a coding sequence defined by the first CFG and N is a noncoding sequence defined by the second CFG.

S -> TCT
T -> CT | NT | e

This is what we used to aid us in generating random strands of DNA.



Source Code

DNAdfa.py

```
import re
from dfa import *
from Bio import SeqIO
from Bio.Seq import Seq
from Bio import Alphabet
from Bio.Alphabet import IUPAC
from Bio.Data import CodonTable
from random import randint
from random import randrange

''' mk_dna_dfa():
    Takes a species number and obtains the start codons and a set of stop codons and
       generates a corresponding DFA
    Facts:
    1) Non coding sequences are allowed  - A valid string contains a sequence of non
       coding and coding sequences (coding sequences are designated by start and stop
       codons)
    2) Must contain at least one coding sequence
    3) Does not allow for point or frame shift mutations - they can be added later
'''
def mk_dna_dfa(species):
    Q = {'S0', 'S0A', 'S0B', 'Smid', 'SmidA', 'SmidB', 'Send', 'SendA', 'SendB'}
    Sigma = {'A','T','G','C'}
    Delta = dict()
    F = {'Send', 'SendA', 'SendB'}
    q0 = 'S0'
```

```
table = CodonTable.unambiguous_dna_by_id[species]
start_codons = table.start_codons
stop_codons = table.stop_codons

count = 0

"""There are three loops that we must account for"""
for n in range(3):

    if n==0:
        """First loop - Non Coding loop"""
        origin = 'S0'           #loop begins at 'S0' (start of DFA)
        destination = 'Smid'    #loop ends at 'Smid' (midpoint of DFA)
        sideA = 'S0A'
        sideB = 'S0B'
        codons = start_codons   #loop will end once a start codon is read
    elif n==1:
        """Second loop - Coding Loop"""
        origin = 'Smid'         #loop begins at 'Smid' (midpoint of DFA)
        destination = 'Send'    #loop ends at 'Send' (endpoint of DFA)
        sideA = 'SmidA'
        sideB = 'SmidB'
        codons = stop_codons    #loop will end once a stop codon is read
    elif n==2:
        """Third loop - Non Coding Loop"""
        origin = 'Send'        #loop begins at 'Send' (endpoint of DFA)
        destination = 'Smid'   #goes back 'Smid' (midpoint)
        sideA = 'SendA'
        sideB = 'SendB'
        codons = start_codons   #loop will end once a start codon is read

    paths = list()      #contains the codons that have been taken care of so far
    newstates1 = list() #first tier of new states
    newstates2 = list() #second tier of new states

    for i in codons:
        """Make a path for all of the codons from the origin to the destination"""
        nucleotide1 = i[0]  #get the first nucleotide of the current codon
        nucleotide2 = i[1]  #get the second nucleotide of the current codon
        nucleotide3 = i[2]  #get the third nucleotide of the current codon

        """Check to see if we can merge paths with another codon"""
        solved = False
        for j in paths: #check all of the codons that have been taken care of already
            if not solved :       #dont check the current codon against itself!
                result = get_difference(i, j)   #get the difference in nucleotides
                if result[0]==1:                #if they are off by one nucleotide,
                    if result[1]==1 and not Delta.has_key((origin,nucleotide1)):
                        """If they differ by the first codon"""
                        state_to_join = Delta[(origin, j[0])]  #get the state
                        Delta.update({(origin,nucleotide1): state_to_join}) #add def
                        solved = True
                    elif result[2]==1:
                        """If they differ by the second codon"""
                        stateA = Delta[(origin, j[0])] #get the state
                        state_to_join = Delta[(stateA, j[1])] #get the state
                        Delta.update({(stateA,nucleotide2): state_to_join})
```

```
                    solved = True
                elif result[3]==1:
                    """If they differ by the third codon"""
                    stateA = Delta[(origin, j[0])]    #get the state
                    stateB = Delta[(stateA, j[1])]    #get the second state
                    Delta.update({(stateB,nucleotide3): destination})
                    solved = True
        if not solved:  #if we are not able to merge yet
            for j in paths:
                if i is not j and not solved:
                    result = get_difference(i, j)
                    if result[0]==2:    #if the codons are off by two
                        if result[1]==1 and result[2]==1:
                            """If the first two nucleotides are different"""
                            stateA = Delta[(origin, j[0])]  #get the state
                            state_to_join = Delta[(stateA, j[1])] #get the last state
                            count += 1
                            new_state = 'S' + str(count)   #new state name
                            Delta.update({(origin, nucleotide1): new_state})
                            Q.add(state1)        #must update Q
                            newstates1.append(state1)  #this is in the first tier

                            Delta.update({(origin,nucleotide1): state_to_join})
                            solved = True

                        elif result[2]==1 and result[3]==1:
                            """if the last two nucleotides are different"""
                            state_to_join = Delta[(origin, j[0])]     #get the state

                            count += 1
                            new_state = 'S' + str(count)   #new state name
                            Delta.update({(state_to_join, nucleotide2): new_state})
                            Q.add(state1)           #must update Q
                            newstates2.append(state1)      #in the second tier
                            Delta.update({(new_state,nucleotide3): destination})
                             solved = True
        if not solved:
            """If it is not solved up until this point either:
                  1) It is the first codon being used in this loop or 2) It differs
                from the previous ones in every position (or the 1st and last
                positions)"""
            count += 1
            state1 = 'S' + str(count)                       #new state name
            Delta.update({(origin, nucleotide1): state1})   #path from q0
            Q.add(state1)                                   #must update Q
            newstates1.append(state1)

            count += 1
            state2 = 'S' + str(count)                       #new state name
            Delta.update({(state1, nucleotide2): state2})   #path from previous state
            Q.add(state2)                                   #must update Q
            newstates2.append(state2)

            Delta.update({(state2, nucleotide3): destination})   #path from previous
        paths.append(i)

    for nucleotide in Sigma:
        if not Delta.has_key((origin,nucleotide)):
```

```
                Delta.update({(origin,nucleotide): sideA})

        for j in newstates1:
            for nucleotide in Sigma:
                if not Delta.has_key((j,nucleotide)):
                    Delta.update({(j,nucleotide): sideB})

        for j in newstates2:
            for nucleotide in Sigma:
                if not Delta.has_key((j,nucleotide)):
                    Delta.update({(j,nucleotide): origin})
        if n==2:
            for a in newstates1: F.add(a)
            for b in newstates2: F.add(b)


    for i in Sigma:
        Delta.update({('S0A', i): 'S0B'})
        Delta.update({('S0B', i): 'S0'})
        Delta.update({('SmidA', i): 'SmidB'})
        Delta.update({('SmidB', i): 'Smid'})
        Delta.update({('SendA', i): 'SendB'})
        Delta.update({('SendB', i): 'Send'})

    dfa = mk_dfa(Q,Sigma,Delta,q0,F)
    return dfa




"""Return the difference in nucleotides between the two codons - WORKS
    Return in the form: list(difference, n1, n2, n3) - if they differ in the first
nucleotide n1 = 1. If they do not, n1=0, and so forth """
def get_difference(codon1, codon2):
    count = n1 = n2 = n3 = 0
    if codon1[0] is not codon2[0]:  #compare the first nucleotide
        n1=1                        #if they are different, n1=1
        count += 1
    if codon1[1] is not codon2[1]:  #compare the second nucleotide
        n2=1                        #if they are different, n2=1
        count += 1
    if codon1[2] is not codon2[2]:  #compare the third nucleotide
        n3=1                        #if they are different, n3=1
        count += 1
    if count==2 and n1 == 1 and n3 == 1:  #if they differ in the first and last one
        count = 3
    result = [count,n1,n2,n3]
    return result

""" generate_random_dna():
Creates a random sequence of DNA
Type: Organism (Standard, Vertebrate Mitochondrial...)
Size: The number of nucleotides
"""
def random_dna(type, size):
    assert size >= 9, "There must be at least 9 nucleotides"
    assert size % 3 == 0, "Only codons!"
```

```
    ncodons = size / 3

    table = CodonTable.unambiguous_dna_by_id[type]
    start = table.start_codons
    stop = table.stop_codons
    dna = ''
    T1 = ''
    T2 = ''

    T1length = 1
    T2length = 1

    """S -> TCT means: S -> (T1)(C)(T2)"""

    C = rand_coding_seq(start,stop,(randint(3,ncodons))*3)
    Ccodons = len(C)/3

    T1length = randint(0,ncodons-Ccodons)
    while T1length > 0:
        curlength = randint(0,T1length)*3
        if randrange(0,2,1) == 0 and curlength >= 9: current =
              rand_coding_seq(start,stop,curlength)
        else: current = rand_noncoding_seq(start,curlength,True)
        T1length -= curlength/3
        T1 += current
    T1codons = len(T1)/3

    T2length = ncodons - Ccodons - T1codons
    while T2length > 0:
        curlength = randint(0,T2length)*3
        if randrange(0,2,1) == 0 and curlength >= 9: current =
              rand_coding_seq(start,stop,curlength)
        else: current = rand_noncoding_seq(start,curlength,True)
        T2length -= curlength/3
        T2 += current

    return Seq(str(T1 + C + T2), IUPAC.unambiguous_dna)

""" generate_coding_sequence(): WORKS?
Creates a random coding sequence of DNA
startcodons: do not allow the sequence to contain one of these
size: the number of nucleotides in the return sequence
onlycodons: whether or not the strand should be a multiple of three
"""
def rand_noncoding_seq(startcodons, size, onlycodons=False):
    if onlycodons:  assert size % 3 is 0, "Size must be a multiple of 3"
    assert isinstance(startcodons,list), "startcodons must be a list"
    assert len(startcodons)>0, "There must be at least one start codon"
    if size <= 0: return ''

    nucleotides = ['A','T','G','C'] #all possible nucleotides
    ncodons = size / 3
    remainder = size % 3
    strand = ''

    for n in range(0, ncodons):

        codon = startcodons[0]
```

```
        while startcodons.__contains__(codon):
            n1 = nucleotides[randint(0,3)]
            n2 = nucleotides[randint(0,3)]
            n3 = nucleotides[randint(0,3)]
            codon = n1 + n2 + n3    #create a codon from the three random nucleotides
        strand += codon #add the new codon to the strand

    """add additional nucleotides if necessary:"""
    if remainder == 1:
        strand += nucleotides[randint(0,3)]
    elif remainder == 2:
        strand += nucleotides[randint(0,3)]
        strand += nucleotides[randint(0,3)]

    return strand

""" generate_coding_sequence():
Creates a random coding sequence of DNA
startcodons: must start with one of these codons
stopcodons: must end with one of these codons
size: the number of nucleotides in the return sequence
"""
def rand_coding_seq(startcodons, stopcodons, size):
    assert size % 3 is 0, "Size must be a multiple of 3"
    assert isinstance(startcodons,list) and isinstance(stopcodons,list)
    assert len(startcodons)>0 and len(stopcodons)>0,
    if size <= 8: return ''

    nucleotides = ['A','T','G','C']
    ncodons = (size / 3) - 2

    strand = str(startcodons[randint(0,len(startcodons)-1)]) #begin strand

    for n in range(0, ncodons):
        codon = stopcodons[0]
        while stopcodons.__contains__(codon):
            n1 = nucleotides[randint(0,3)]
            n2 = nucleotides[randint(0,3)]
            n3 = nucleotides[randint(0,3)]
            codon = n1 + n2 + n3    #create a codon from the three random nucleotides
        strand += codon #add the new codon to the strand

    strand += str(stopcodons[randint(0,len(stopcodons)-1)]) #end strand with a stop codon

    return strand

""" validate_dna():
Takes a random strand of DNA, and a number corresponding to the species type
Creates a dna for the specified species and then checks to see if the DFA accepts
"""
def validate_dna(strand, type):
    dfa = mk_dna_dfa(type);
    return accepts(dfa,dfa['q0'],strand)

def mk_dfa(Q, Sigma, Delta, q0, F):

    assert len(Q)>0, "Q must not be empty"
    assert len(Sigma)>0, "Sigma must not be empty"
```

```
    for i in Sigma:
        assert isinstance(i, str), "Sigma must be a set of single character strings"
        assert len(i)==1, "Sigma must be a set of single character strings"

    assert Q.__contains__(q0), "q0 must belong to Q"

    assert isinstance(F, set), "F must be a non-empty set"
    assert len(F&Q)>0 and len(F-Q)==0, "F must be a subset of Q"

    assert isinstance(Delta, dict), "Delta is not a total function”

    for i in Delta:
        assert Q.__contains__(i[0]), "For key pairs (q,c) every q must be in Q"
        assert Sigma.__contains__(i[1]), "For key pairs (q,c) every c must be in Sigma"
        assert Q.__contains__(Delta[i]), "All values must be in Q"

    dfa = dict({"Q":Q, "Sigma":Sigma, "Delta":Delta, "q0":q0, "F":F})
    return dfa
```

Blast.py

```
from Bio import Blast
from Bio.Blast import NCBIWWW
from Bio.Blast import NCBIXML
from Bio.Seq import Seq

#compares the generated sequence to the nonredundant blast database
def check_blast_dna(strand):
    strand = str(strand)
    "opens the blast database and gets the results for the strand that we have inputted"
    result_handle = NCBIWWW.qblast("blastn", "nr", strand)
    "saves the results to the file blast_output and closes the file"
    save_file = open("blast_output.xml", "w")
    save_file.write(result_handle.read())
    save_file.close()
    result_handle.close()

    result_handle = open("blast_output.xml")
    blast_record = NCBIXML.read(result_handle)
    if blast_record.alignments != None:
        for alignment in blast_record.alignments:
            for hsp in alignment.hsps:
                print('****Alignment****')
                print('sequence:', alignment.title)
                print('length:', alignment.length)
                print(hsp.query[0:75] + '...')
                print(hsp.match[0:75] + '...')
                print(hsp.sbjct[0:75] + '...')
    else:
         print("Blast does not recognize this sequence!")


def check_blast_protein(strand):
    strand = str(strand)
```

```
    "opens the blast database and gets the results for the strand that we have inputted"
    result_handle = NCBIWWW.qblast("blastp", "nr", strand)
    "saves the results to the file blast_output and closes the file"
    save_file = open("blast_output.xml", "w")
    save_file.write(result_handle.read())
    save_file.close()
    result_handle.close()

    result_handle = open("blast_output.xml")
    blast_record = NCBIXML.read(result_handle)
    if blast_record.alignments != None:
        for alignment in blast_record.alignments:
            for hsp in alignment.hsps:
                print('****Alignment****')
                print('sequence:', alignment.title)
                print('length:', alignment.length)
                print(hsp.query[0:75] + '...')
                print(hsp.match[0:75] + '...')
                print(hsp.sbjct[0:75] + '...')
    else:
         print("Blast does not recognize this sequence!")


def test_blast():
    result_handle = open("blast_output.xml")
    blast_record = NCBIXML.read(result_handle)
    if blast_record.alignments != None:
        for alignment in blast_record.alignments:
            for hsp in alignment.hsps:
                print('****Alignment****')
                print('sequence:', alignment.title)
                print('length:', alignment.length)
                print(hsp.query[0:75] + '...')
                print(hsp.match[0:75] + '...')
                print(hsp.sbjct[0:75] + '...')
    else:
         print("Blast does not recognize this sequence!")
```

Program.py

```
from Bio.Seq import *
from Bio.SeqRecord import SeqRecord
from Bio import SeqIO
from Bio.Data import CodonTable
from Bio.Alphabet import generic_protein

from DNAdfa import *
from DotDFA import *
from blast import *

def file_to_list(file,format,alphabet=0):
    recordList = list()

    if(alphabet==0):
        for record in SeqIO.parse("Source/" + file, format):
            print record.id
            print repr(record.seq)
            print len(record)
            recordList.append(record.seq)
    else:
        for record in SeqIO.parse("Source/" + file, format, alphabet):
            print record.id
            print repr(record.seq)
            print len(record)
            recordList.append(record.seq)
    return recordList

def print_annotations(record):
    for i in record.annotations:
        print str(i) + (": ") + str(rec.annotations[i])
        print ("")

if __name__ == "__main__":
    print("Welcome")
```