Gregory Knapp

CS 312

Dr. Grimsman

11/4/2021

Project 4 Gene Alignment Project Report

Time Complexity Analysis of Project 4

The time complexity for each part of the code is pulled from the source code comments, with any additional notes added here for things that aren't explained within the code.

Summary of unrestricted algorithm time/space complexity.

Analysis of Unrestricted Algorithm Time/Space Complexity

GeneSequencing.py Lines 441-446. Space complexity when initializing array.

```
# Initialize 2d matrix for calculating edit distance of sequences
# Time of np.zeros - O(nm) where n is length of align_length_seq1 and m is
length of align_length_seq2
# Time is O(nm) because each value is iterated over in order to initialize
it.
#
# Space is O(nm) because the 2d array has values equal to n * m or the length
of the two sequences after
# taking the align_length variable into account.
```

Genesequencing.py Lines 457-468. Time complexity of while loop in unrestricted algorithm.

```
# Solve for remaining values in matrix starting from second row/column to end
of table.
# Time O((n-1)(m-1)) which is equal to O(nm).
#
# Time complexity is O(nm) because the two nested for loops iterate over each
value in the sequences.
# The way the code is set up, all of the actual genetic sequences are longer
than the expected values of
# 1000 or 3000 for align_length that we were supposed to test for, so in most
of the test cases, n and m
# are equal, so the time could be written as O(n^2). However, the algorithm
does not force the two
# sequences to be the same length. The test examples using polynomial or
exponential against one of the
# gene sequences have vastly different lengths, resulting in significant
differences between n and m.
```

```
# On top of that, the value for align length is subjective, meaning that
while our test cases did not
# cause it to happen, it is very possible that with higher values of
align_length more cases would have
# different values for n and m, so time complexity is best stated as O(nm)
instead of O(n^2) or O(m^2)
```

GeneSequencing.py Lines 26-43. Time/space complexity of traceback function for unrestricted algorithm.

```
# Function that takes in the matrix solved by the unrestricted algorithm and
returns an array containing the two
# alignments for the sequences. The backtrace alignment is found by starting
at the cell containing the score and
# traversing the O(nm) matrix backwards. The next cell is found by doing the
same checks used to calculate the edit
# distance (left insert, top insert, diagonal sub/match) in the same order to
ensure the correct alignment is
# produced.
#
# The time complexity of this algorithm has a worst case of O(nm) which would
happen if every single backtrace was
# an insert, from the score cell to the very left of the matrix then up to
the top left cell. This is an extremely
# unlikely, if impossible case, so it isn't a good time analysis of the
function.
#
# In reality, the average case is approximately O(n) where n is the length of
the longer sequence. In our cases,
# since we limit the sequences with the align_length parameter to our
program, so in most cases, the two
# sequences have the same sliced length. In cases where it doesn't, the
algorithm will need at least as many
# iterations as there are characters in the longer sequence so the complete
alignment can be found for both. If only
# subs/matches are made, then we get the best case O(n), where n =
align_length/longest sequences. Realistically,
# there are almost always inserts made, which add 1 extra loop to the
algorithm. These additional loops are,
# except in the worst of cases, insignificant to the overall complexity of
the function, so the average case
# should still be O(n) when n = align_length/len of the longest sequence.
```

Summary of Unrestricted algorithm

Time Complexity = O(nm)

Space Complexity = O(nm)

Time complexity of Traceback O(n) n = length of longer sequence

As described in the above comments, the time and space complexity of the unrestricted algorithm are both O(nm). The space complexity is straightforward. The array is initialized with the parameters of the

length of the two sequences for rows and columns, giving us a clear array of size O(nm). No other arrays are initialized in this function, so there is no other space complexity to consider here.

Time complexity is dictated by the way the function progresses. Since it is the standard edit distance function, it loops through every value in the array and calculates the edit distance in the array, unlike the banded algorithm. Since we know the space complexity of the array is O(nm), and the algorithm loops through each value to calculate it, we know the time complexity is also O(nm), the same as the size of the array. The time is not greater than this, thanks to the dynamic programming approach of using min and checking previous values in the array. That way, instead of constantly repeating the same problems and having a time of O(nm^2) or something worse, we only perform each calculation once, giving us the time that is the same as the array space.

Of note, the time complexity is not changed by the traceback function, because the traceback is only of O(n) time, which does not impact the overall time performance. The reasons for this time are detailed in the comment, but generally speaking, the average case loops through just a few times more than the length of the longer string, depending on the number of inserts, but its not enough to make the complexity get higher than O(n), except in the very worst cases.

Analysis of Banded Algorithm Time/Space Complexity

GeneSequencing.py Lines 492-508. Time complexity summary of Banded algorithm.

```
# Use banded approach to solve edit distance of two sequences. Bandwidth is
caluclated from globally defined
# variable MAXINDELS, which for testing purposes is defaulted to 3. A matrix
separate from the one used in the
# unrestricted algorithm is created and used. This array of size O(kn), where
k is 2 *MAXINDELS + 1 and n is
# the length of the shorter string, is used for storing and calculating score
values.
#
# The algorithm works by initializing the first band of values since there
are no previous values to reference,
# so to avoid out of bounds errors, the first 7 values are done separately.
Then inside of the main while loop,
# the value diagonal to the first value is calculated. From there, values for
3 inserts/deletes both down and
# right of that value are calculate, referencing values previously calculated
in the matrix. This gives us the
# 2d+1 (k) size band that makes up the array, resulting in the space
complexity of the algorithm being O(kn),
# since only those 2d+1 values are stored in the initialized array.
#
# Time complexity of the algorithm is straightforward. The while loop
iterates until j = length of the shorter
# sequence. For each iteration of j, 2d+1 (in our cases since MAXINDELS = 3,
2d+1 = 7) values, which are stored
# in the array. If the two strings have very different lengths, less values
than k may be calculated on a
# single iteration, but this is not significant enough to change that the
```

```
time complexity of this algorithm is
# also O(kn), where k = 2d + 1 and n = length of the shorter sequence.
```

Genesequencing.py Line 513. Space complexity of initializing array O(kn)

```
# Initialize O(kn) size matrix - Time O(kn) (each value initialized to 0) and
Space - O(kn)
```

Genesequencing.py Lines 95-119. Time/Space complexity of traceback function for banded algorithm.

```
# A function to extract the alignments of the two sequences after the score
is calculated for the banded
# implementation of the alignment algorithm. Siimilarly to the unrestricted
version of the traceback function,
# the function traverses the O(kn) matrix in order to determine which
combination of insert/deletes, subs, and
# matches were used to reach the final score. Due to the restricted size of
the O(kn) function, the actual matrix
# traversal is alot messier, since the way the variables are indexed is not
as easy to follow as just [i-1][j-1]
# like we could for the O(nm) array. Special consideration is made for cases
such as the first value in each row j
# that does not find the value for the previous left insert score the same
place as the cell to the right. Because
# of this, these cases are pulled into their own if branches to handle
appropriately, but the core functionality is
# the same for each case. Check in order left, top, diagonal for the next
value, and update necessary iterators and
# alignments to construct the strings. Despite there being cases that are
treated differently, the conditions are
# set up so that if MAXINDELS changes, resulting in a larger band, the
function still runs the same and identifies
# the cases based on MAXINDELS, not hard coded values.
#
# It is also worth noting that the main function checks if the matrix
# solving algorithm even found a valid alignment, before running the
traceback by checking if the score is infinity.
# If so, the banded traceback algorithm is not run, saving the work. This
only happens when there is a difference
# in the two sequences to where the banded algorithm does not reach the final
cell because it is further away
# than the bandwidth.
#
# The time complexity of the banded traceback is still the same as the
unrestricted version of the function;
# worst case O(nm), but with an average/best case of O(n) where n is the
length of the longer sequence. This is for
# the same reasons that the previous function had the same time complexity.
Each character in each string needs to
# be handled at least once before the algorithm finishes, making the best
cases limited to O(n), if every case was
# a substitution or match. However, there are likely inserts/deletions which
add loops to the algorithm, just not
# enough to where the average case would deviate significantly from the best
case of O(n).
```

Summary of Banded algorithm

Time Complexity = O(kn)

Space Complexity = O(kn)

Time complexity of Traceback O(n), n = length of longer sequence

The analysis of the banded algorithm is very similar to that of the unrestricted one, just with different values. The space complexity is clearly O(kn), since I initialize the matrix array with the value k calculated from 2*d+1. I know this for sure also because I originally implemented this with time O(kn) but space O(nm) by using the full array, since I was unaware of the requirement, so I had to learn the hard way what the difference was.

Like the unrestricted algorithm, the function then calculates a distance for each value in the array (with the exception of a few cells at the end when there is a difference in sequence length, but those are not enough to change the time complexity). Since we are using dynamic programming and only iterating over each value one time, we have the time complexity O(kn), instead of something worse.

Again, the traceback function does not change the time complexity since it does not dominate the main time complexity of the main algorithm. The function for the banded one works the same as the unrestricted one, just with a different pattern for tracing back through the array since the set up is completely different.

Explanation of Extraction Algorithms

The extraction algorithms work by starting from the “score” cell (the cell containing the final score of the ideal alignments) and working backwards until it reaches the first cell of the array. The functions start with two blank strings and an iterator for each string that each start at the final index of the string. Next, the function starts from the final square and essentially does what the alignment function does in reverse order; it checks the previous array values and sees if one of them equals the current value with either the indel penalty, substitution penalty, or match reward values added. Importantly, it searches for these in the same left, top, diagonal tie breaking order so that the alignments are the correct ones that the code originally solved for. When it finds the previous value, it updates the alignment strings. If a substitution was made, the current letter the sequence index points to is appended. If an insert/deletion is made, the string keeps the value gets the letter and the iterator is decremented. The other string adds a “-“ to it and does not decrement the counter. This pattern continues until both string iterators reach 0, and all the characters from the sequence are appended to the strings. The result is packaged into an array which main extracts, checks if there is a valid alignment and then slices down to the 100 characters displayed on the screen.

I made two functions, one for the unrestricted algorithm and one for the banded one, since the different array sizes and order the variables are stored in made it hard to write one that worked for either one. (If

I had used the array of pointers or indices/tuples approach, it probably would have been possible to do). However, there are no core differences between the two functions, except for the pattern which previous values are checked for. The O(kn) array has some special cases since sometimes values that are used to calculate the score are not part of the banded array, so considerations for those sorts of cases are made in there. Otherwise, the pattern is the same, start from the bottom and work up to the first cell while appending letters to either or both strings depending on the result found.

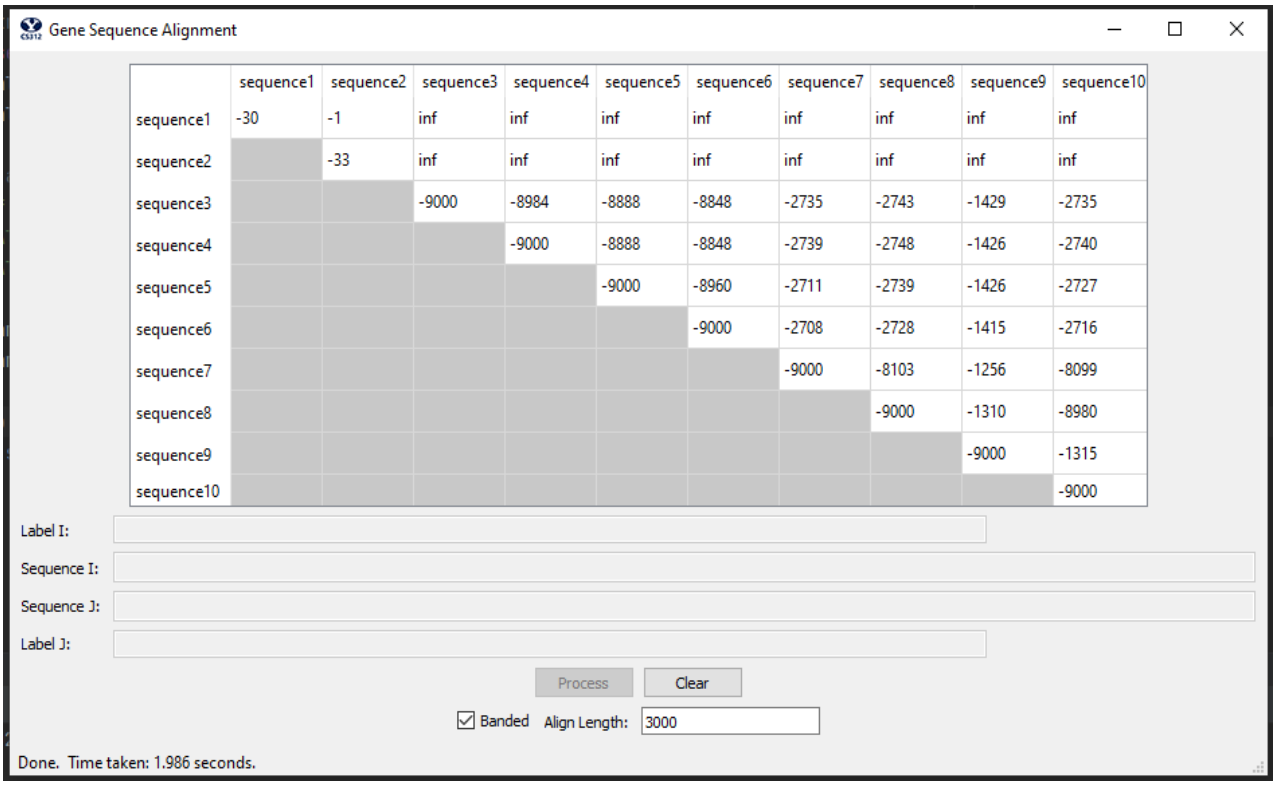
Results

1. 10x10 matrix using unrestricted algorithm with align length n = 1000

	sequence1	sequence2	sequence3	sequence4	sequence5	sequence6	sequence7	sequence8	sequence9	sequence10
sequence1	-30	-1	4956	4956	4956	4956	4956	4956	4956	4956
sequence2		-33	4948	4948	4948	4948	4948	4948	4948	4948
sequence3			-3000	-2996	-2956	-2944	-1431	-1448	-1399	-1448
sequence4				-3000	-2960	-2948	-1431	-1448	-1399	-1448
sequence5					-3000	-2988	-1423	-1452	-1391	-1448
sequence6						-3000	-1426	-1452	-1394	-1448
sequence7							-3000	-2771	-2814	-2767
sequence8								-3000	-2731	-2996
sequence9									-3000	-2727
sequence10										-3000

Time taken: 1 minute and 6.764 seconds. 66.6764 < 120 seconds, passes performance requirement.

2.10x10 matrix using banded algorithm with align length n = 3000.



Time taken: 1.986 seconds. 1.986 < 10, passes performance requirement.

3.Extracted alignment for first 100 characters of sequences #3 and #10 for both unrestricted and banded algorithms.

Unrestricted sequences for #3 and #10 (#3 is top, #10 is below)

ataa-gagtgattggcgtccgtacgtacccttttctactctcaaactcttgttagtttaaatc-taatctaaactttataaa--cggc-acttcctgtgtg

attgcgagcgatttgcgtgcgtgcatcccgcttc-actg--at-ctcttgttagatcttttcataatctaaactttataaaaacatccactccctgta-g

Banded sequences for #3 and #10 (#3 is top, #10 is below)

ataa-gagtgattggcgtccgtacgtacccttttctactctcaaactcttgttagtttaaatc-taatctaaactttataaa--cggc-acttcctgtgtg

attgcgagcgatttgcgtgcgtgcatcccgcttc-actg--at-ctcttgttagatcttttcataatctaaactttataaaaacatccactccctgta-g

Source Code Files

GeneSequencing.py

```
#!/usr/bin/python3

from which_pyqt import PYQT_VER

if PYQT_VER == 'PYQT5':
    from PyQt5.QtCore import QLineF, QPointF
else:
    raise Exception('Unsupported Version of PyQt: {}'.format(PYQT_VER))

import numpy as np

# Used to compute the bandwidth for banded version
MAXINDELS = 3

# Used to implement Needleman-Wunsch scoring
MATCH = -3
INDEL = 5
SUB = 1


class GeneSequencing:

    def __init__(self):
        pass

    # Function that takes in the matrix solved by the unrestricted algorithm
and returns an array containing the two
    # alignments for the sequences. The backtrace alignment is found by
starting at the cell containing the score and
    # traversing the O(nm) matrix backwards. The next cell is found by doing
the same checks used to calculate the edit
    # distance (left insert, top insert, diagonal sub/match) in the same
order to ensure the correct alignment is
    # produced.
    #
    # The time complexity of this algorithm has a worst case of O(nm) which
would happen if every single backtrace was
    # an insert, from the score cell to the very left of the matrix then up
to the top left cell. This is an extremely
    # unlikely, if impossible case, so it isn't a good time analysis of the
function.
    #
    # In reality, the average case is approximately O(n) where n is the
length of the longer sequence. In our cases,
    # since we limit the sequences with the align_length parameter to our
program, so in most cases, the two
    # sequences have the same sliced length. In cases where it doesn't, the
algorithm will need at least as many
    # iterations as there are characters in the longer sequence so the
complete alignment can be found for both. If only
    # subs/matches are made, then we get the best case O(n), where n =
```

```
align_length/longest sequences. Realistically,
    # there are almost always inserts made, which add 1 extra loop to the
algorithm. These additional loops are,
    # except in the worst of cases, insignificant to the overall complexity
of the function, so the average case
    # should still be O(n) when n = align_length/len of the longest sequence.
    def traceback_unrestricted(self, matrix, align_length_seq1,
align_length_seq2):
        # Initialize penalty variables used in backtrace calculation - Time
O(1)
        indel_penalty = 5

        # Set traceback counter variables for each sequence - Time O(1)
        tracebacki = len(align_length_seq1)
        tracebackj = len(align_length_seq2)

        # Initialize alignment strings to store results - Time O(1)
        alignment1 = ""
        alignment2 = ""

        # Get traceback of strings as long as there are characters yet to be
appended for both sequences. As described
        # in the comments above the function, the while loop runs at least
align_length/len of the longest sequence
        # times, with additional loops happening for each insert/deletion
made by the algorithm. Except in the worst
        # cases, the insert/delete count is not high enough to affect the
time complexity of the loop, so the time
        # of the core while loop is O(n) when n = align_length/len of longest
sequence.
        while tracebacki > 0 and tracebackj > 0:
            # Check if previous result was an indel from the left first.
            if tracebackj > 0 and matrix[tracebacki][tracebackj] ==
matrix[tracebacki][
                tracebackj - 1] + indel_penalty:  # If statement evaluation -
Time O(1)

                # Append values to strings - Time O(1)
                alignment1 = "-" + alignment1
                alignment2 = align_length_seq2[tracebackj - 1] + alignment2

                # Update traceback variable for sequence 2 - Time O(1)
                tracebackj = tracebackj - 1
            # Check if previous result was an indel from the top next.
            elif tracebacki > 0 and matrix[tracebacki][tracebackj] ==
matrix[tracebacki - 1][
                tracebackj] + indel_penalty:  # If statement evaluation -
Time O(1)

                # Append value to strings - Time O(1)
                alignment1 = align_length_seq1[tracebacki - 1] + alignment1
                alignment2 = "-" + alignment2

                # Update traceback variable for sequence 1 - Time O(1)
                tracebacki = tracebacki - 1
            # Otherwise, previous step was either a substitution or a match
            else:
```

```
                # Append value to both strings - Time O(1)
                alignment1 = align_length_seq1[tracebacki - 1] + alignment1
                alignment2 = align_length_seq2[tracebackj - 1] + alignment2

                # Update traceback variable for both strings - Time O(1)
                tracebacki = tracebacki - 1
                tracebackj = tracebackj - 1

        # Create and return array of two alignments to extract in main
function - Time O(1)
        return [alignment1, alignment2]

    # A function to extract the alignments of the two sequences after the
score is calculated for the banded
    # implementation of the alignment algorithm. Siimilarly to the
unrestricted version of the traceback function,
    # the function traverses the O(kn) matrix in order to determine which
combination of insert/deletes, subs, and
    # matches were used to reach the final score. Due to the restricted size
of the O(kn) function, the actual matrix
    # traversal is alot messier, since the way the variables are indexed is
not as easy to follow as just [i-1][j-1]
    # like we could for the O(nm) array. Special consideration is made for
cases such as the first value in each row j
    # that does not find the value for the previous left insert score the
same place as the cell to the right. Because
    # of this, these cases are pulled into their own if branches to handle
appropriately, but the core functionality is
    # the same for each case. Check in order left, top, diagonal for the next
value, and update necessary iterators and
    # alignments to construct the strings. Despite there being cases that are
treated differently, the conditions are
    # set up so that if MAXINDELS changes, resulting in a larger band, the
function still runs the same and identifies
    # the cases based on MAXINDELS, not hard coded values.
    #
    # It is also worth noting that the main function checks if the matrix
    # solving algorithm even found a valid alignment, before running the
traceback by checking if the score is infinity.
    # If so, the banded traceback algorithm is not run, saving the work. This
only happens when there is a difference
    # in the two sequences to where the banded algorithm does not reach the
final cell because it is further away
    # than the bandwidth.
    #
    # The time complexity of the banded traceback is still the same as the
unrestricted version of the function;
    # worst case O(nm), but with an average/best case of O(n) where n is the
length of the longer sequence. This is for
    # the same reasons that the previous function had the same time
complexity. Each character in each string needs to
    # be handled at least once before the algorithm finishes, making the best
cases limited to O(n), if every case was
    # a substitution or match. However, there are likely inserts/deletions
which add loops to the algorithm, just not
    # enough to where the average case would deviate significantly from the
best case of O(n).
```

```
    def traceback_banded(self, matrix, align_length_seq1, align_length_seq2,
diff, j):

        # Initialize penalty value used in calculations - Time O(1)
        indel_penalty = 5

        # Initialize alignment variables used to store results - Time O(1)
        alignment1 = ""
        alignment2 = ""

        # Create traceback iterator variables for each sequence - Time O(1)
        tracebacki = len(align_length_seq1)
        tracebackj = len(align_length_seq2)

        # Get the position of the cell containing the final score variable to
start calculating from - Time O(1)
        startingj = j - 1
        startingk = 0 + diff

        # Iterate until both sequences have been processed and saved into the
alignment variables. Time - Detailed
        # reasoning is explained in comments above function, worst case =
O(nm), average/best case is O(n) as it takes
        # about as many iterations as the length of the longer sequence plus
a few extra for any insert/deletions made
        # in order to finish processing the alignment.
        while tracebacki > 0 and tracebackj > 0:

            # Check special case, value in first column.
            if startingk == 0:  # If statement evaluation - Time O(1)

                # Check if previous step was left insert
                if tracebackj > 0 and matrix[startingj - 1][startingk + 1] +
indel_penalty == \
                        matrix[startingj][
                            startingk]:  # If statement evaluation - Time
O(1)

                    # Update alignment variables - Time O(1)
                    alignment1 = "-" + alignment1
                    alignment2 = align_length_seq2[tracebackj - 1] +
alignment2

                    # Update traceback variable for sequence 2 only - Time
O(1)
                    tracebackj = tracebackj - 1

                    # Update position of current score to compare - Time O(1)
                    startingj = startingj - 1
                    startingk = startingk + 1
                # Check if previous step was top insert.
                elif tracebacki > 0 and matrix[startingj - 1][startingk + 1 +
MAXINDELS] + indel_penalty == \
                        matrix[startingj][
                            startingk]:  # If statement evaluation - Time
O(1)
```

```
                    # Update alignment variables - Time O(1)
                    alignment1 = align_length_seq1[tracebacki - 1] +
alignment1
                    alignment2 = "-" + alignment2

                    # Update traceback variable for sequence 1 only - Time
O(1)
                    tracebacki = tracebacki - 1

                    # Update position of current score to compare - Time O(1)
                    startingj = startingj - 1
                    startingk = startingk + 1 + MAXINDELS
                # Otherwise previous step was a sub/match
                else:
                    # Update alignment variables - Time O(1)
                    alignment1 = align_length_seq1[tracebacki - 1] +
alignment1
                    alignment2 = align_length_seq2[tracebackj - 1] +
alignment2

                    # Update both traceback variables - Time O(1)
                    tracebacki = tracebacki - 1
                    tracebackj = tracebackj - 1

                    # Update position of current score to compare - Time O(1)
                    startingj = startingj - 1

            # Check general cases for any value from bottom part of band for
this iteration.
            elif startingk < MAXINDELS:  # If statment evaluation - Time O(1)

                # Check if previous step was left insert
                if tracebackj > 0 and matrix[startingj - 1][startingk + 1] +
indel_penalty == \
                        matrix[startingj][
                            startingk]:  # If statement evaluation - Time
O(1)

                    # Update alignment variables - Time O(1)
                    alignment1 = "-" + alignment1
                    alignment2 = align_length_seq2[tracebackj - 1] +
alignment2

                    # Update traceback variable for sequence 2 only - Time
O(1)
                    tracebackj = tracebackj - 1

                    # Update position of current score to compare - Time O(1)
                    startingj = startingj - 1
                    startingk = startingk + 1
                # Check if previous step was top insert
                elif tracebacki > 0 and matrix[startingj][startingk - 1] +
indel_penalty == matrix[startingj][
                    startingk]:  # If statement evaluation - Time O(1)

                    # Update alignment variables - Time O(1)
                    alignment1 = align_length_seq1[tracebacki - 1] +
```

```
alignment1
                    alignment2 = "-" + alignment2

                    # Update traceback variable for sequence 1 only - Time
O(1)
                    tracebacki = tracebacki - 1

                    # Update position of current score to compare - Time O(1)
                    startingk = startingk - 1
                # Otherwise, previous step was a sub/match.
                else:
                    # Update alignment variables - Time O(1)
                    alignment1 = align_length_seq1[tracebacki - 1] +
alignment1
                    alignment2 = align_length_seq2[tracebackj - 1] +
alignment2

                    # Update both traceback variables - Time O(1)
                    tracebacki = tracebacki - 1
                    tracebackj = tracebackj - 1

                    # Update position of current score to compare - Time O(1)
                    startingj = startingj - 1

            # Check special case of variable at bottom edge of band. The
value used to calculate if a left insert
            # was made is not part of the O(kn) array, so we ignore that
check since it is not possible to exist in
            # our array since we are only using the values in the band.
            elif startingk == MAXINDELS:  # If statement evaluation - Time
O(1)

                # No need to check if previous step was left insert since it
is impossible at this position.

                # Check if previous step was top insert.
                if tracebacki > 0 and matrix[startingj][startingk - 1] +
indel_penalty == matrix[startingj][
                    startingk]:  # If statement evaluation - Time O(1)

                    # Update alignment variables - Time O(1)
                    alignment1 = align_length_seq1[tracebacki - 1] +
alignment1
                    alignment2 = "-" + alignment2

                    # Update traceback varaible for sequence 1 only - Time
O(1)
                    tracebacki = tracebacki - 1

                    # Update position of current score to compare - Time O(1)
                    startingk = startingk - 1

                # Otherwise, previous step was a sub/match
                else:
                    # Update alignment variables - Time O(1)
                    alignment1 = align_length_seq1[tracebacki - 1] +
alignment1
```

```
                    alignment2 = align_length_seq2[tracebackj - 1] +
alignment2

                    # Update both traceback variables - Time O(1)
                    tracebacki = tracebacki - 1
                    tracebackj = tracebackj - 1

                    # Update position of current score to compare - Time O(1)
                    startingj = startingj - 1

            # Treat special case of first value after band switches
direction. Because of the way the values of the band
            # are saved in the array, in most cases, the value used in
determining a left shift can be found at
            # [startingk - 1] except for the first value when the band
changes direction. That is because it uses the
            # value at startingk = 0 in the same row j to evaluate a left
insert/deletion. While this could be
            # calculated and set up in a different branch, to ensure accurate
calculations and less debugging hasssle
            # I split it into its own case to make simple.
            elif startingk == MAXINDELS + 1:  # If statement evaluation -
Time O(1)
                # Check if previous step was left insert using startingk = 0
- Time O(1)
                if tracebackj > 0 and matrix[startingj][0] + indel_penalty ==
matrix[startingj][startingk]:

                    # Update alignment variables - Time O(1)
                    alignment1 = "-" + alignment1
                    alignment2 = align_length_seq2[tracebackj - 1] +
alignment2

                    # Update traceback variable for sequence 2 only - Time
O(1)
                    tracebackj = tracebackj - 1

                    # Update position of current score to compare - Time O(1)
                    startingk = 0
                # Check if previous step was top insert - Time O(1)
                elif tracebacki > 0 and matrix[startingj - 1][startingk + 1]
+ indel_penalty == \
                        matrix[startingj][startingk]:

                    # Update alignment variables - Time O(1)
                    alignment1 = align_length_seq1[tracebacki - 1] +
alignment1
                    alignment2 = "-" + alignment2

                    # Update traceback variable for sequence 1 only - Time
O(1)
                    tracebacki = tracebacki - 1

                    # Update position of current score to compare - Time O(1)
                    startingj = startingj - 1
                    startingk = startingk + 1
                # Otherwise, previous step was sub/match
```

```
                else:

                    # Update alignment variables - Time O(1)
                    alignment1 = align_length_seq1[tracebacki - 1] +
alignment1
                    alignment2 = align_length_seq2[tracebackj - 1] +
alignment2

                    # Update both traceback variables - Time O(1)
                    tracebacki = tracebacki - 1
                    tracebackj = tracebackj - 1

                    # Update position of current score to compare - Time O(1)
                    startingj = startingj - 1

            # Check regular cases for all values on horizontal arm of band,
excluding the end value of the band.
            elif startingk < 2 * MAXINDELS:  # If statement evaluation - Time
O(1)

                # Check is previous step was left insert.
                if tracebackj > 0 and matrix[startingj][startingk - 1] +
indel_penalty == matrix[startingj][
                    startingk]:  # If statement evaluation - Time O(1)

                    # Update alignment variables - Time O(1)
                    alignment1 = "-" + alignment1
                    alignment2 = align_length_seq2[tracebackj - 1] +
alignment2

                    # Update traceback variable for sequence 2 only - Time
O(1)
                    tracebackj = tracebackj - 1

                    # Update position of current score to compare - Time O(1)
                    startingk = startingk - 1

                elif tracebacki > 0 and matrix[startingj - 1][startingk + 1]
+ indel_penalty == \
                        matrix[startingj][startingk]:  # If statement
evaluation - Time O(1)

                    # Update alignment variables - Time O(1)
                    alignment1 = align_length_seq1[tracebacki - 1] +
alignment1
                    alignment2 = "-" + alignment2

                    # Update traceback variable for sequence 1 only - Time
O(1)
                    tracebacki = tracebacki - 1

                    # Update position of current score to compare - Time O(1)
                    startingj = startingj - 1
                    startingk = startingk + 1

                # Otherwise, previous step was a sub/match.
                else:
```

```
                    # Update alignment variables - Time O(1)
                    alignment1 = align_length_seq1[tracebacki - 1] +
alignment1
                    alignment2 = align_length_seq2[tracebackj - 1] +
alignment2

                    # Update both traceback variables - Time O(1)
                    tracebacki = tracebacki - 1
                    tracebackj = tracebackj - 1

                    # Update position of current score to compare - Time O(1)
                    startingj = startingj - 1

            # Check final special case if value is the end of the horizontal
arm of the band/final value in current
            # row k of O(kn) array. Same with the value at startingk = 0, the
value at the end of the row of the array
            # does not have a reference for the previous value of a top
insert, so that is not calculated in this
            # special case since it will never exist given the banded
algorithm set up.
            elif startingk == 2*MAXINDELS:  # Evaluate if statement - O(1)
                # Check if previous step was left insert
                if tracebackj > 0 and matrix[startingj][startingk - 1] +
indel_penalty == matrix[startingj][
                    startingk]:  # If statement evaluation - Time O(1)

                    # Update alignment variables - Time O(1)
                    alignment1 = "-" + alignment1
                    alignment2 = align_length_seq2[tracebackj - 1] +
alignment2

                    # Update traceback variable for sequence 2 only - Time
O(1)
                    tracebackj = tracebackj - 1

                    # Update position of current score to compare - Time O(1)
                    startingk = startingk - 1

                # No need to check for possible top insert/delete since it
cannot exist in the band for this value.

                # Otherwise, previous step was sub/match
                else:

                    # Update alignment variables - Time O(1)
                    alignment1 = align_length_seq1[tracebacki - 1] +
alignment1
                    alignment2 = align_length_seq2[tracebackj - 1] +
alignment2

                    # Update both traceback variables. - Time O(1)
                    tracebacki = tracebacki - 1
                    tracebackj = tracebackj - 1

                    # Update position of current score to compare - Time O(1)
```

```
                    startingj = startingj - 1

        # Create and return array of alignments to extract in main - Time
O(1)
        return [alignment1, alignment2]

    # This is the method called by the GUI.  _seq1_ and _seq2_ are two
sequences to be aligned, _banded_ is a boolean that tells
    # you whether you should compute a banded alignment or full alignment,
and _align_length_ tells you
    # how many base pairs to use in computing the alignment

    def align(self, seq1, seq2, banded, align_length):
        self.banded = banded
        self.MaxCharactersToAlign = align_length

        # Initialize penalty/reward values - Time O(1)
        match_reward = -3
        indel_penalty = 5
        sub_penalty = 1

        # Initialize return values
        alignment1 = ""
        alignment2 = ""
        score = 0

        # Initialize Strings for shortened sequences used in computation -
Time O(n)
        align_length_seq1 = ""
        align_length_seq2 = ""

        # If the entire sequence is shorter than align_length, use sequence
length instead
        if len(seq1) < align_length:  # If statement evaluation - Time O(1)
            align_length_seq1 = seq1  # Variable assignment - Time O(1)
        else:
            # Slicing portion of string - Time O(s) where s is the size of
the slice (s = align_length)
            align_length_seq1 = seq1[:align_length]

        if len(seq2) < align_length:  # If statement evaluation - Time O(1)
            align_length_seq2 = seq2  # Variable Assignment - Time O(1)
        else:
            # Slicing portion of string - Time O(s) where s is the size of
the slice (s = align_length)
            align_length_seq2 = seq2[:align_length]

        # If sequence 1 is the shorter sequence, make it the sequence for
columns
        if len(seq1) < len(seq2):  # If statement comparison - Time O(1)
            temp = align_length_seq1  # Variable assignment - Time O(1)
            align_length_seq1 = align_length_seq2  # Variable assignment -
Time O(1)
            align_length_seq2 = temp  # Variable assignment - Time O(1)

        # Solve matrix depending on banded is true or false.
        if banded is False:  # If statement evaluation - Time O(1)
```

```
            # Initialize 2d matrix for calculating edit distance of sequences
            # Time of np.zeros - O(nm) where n is length of align_length_seq1
and m is length of align_length_seq2
            # Time is O(nm) because each value is iterated over in order to
initialize it.
            #
            # Space is O(nm) because the 2d array has values equal to n * m
or the length of the two sequences after
            # taking the align_length variable into account.
            matrix = np.zeros((len(align_length_seq1) + 1,
len(align_length_seq2) + 1))

            # Initialize values of left hand column of array - Time O(n)
where n = length of align_length_seq1
            for i in range(len(align_length_seq1) + 1):
                matrix[i][0] = i * indel_penalty  # Assign variable with
multiplication - Time O(1)

            # Initialize values of top row of matrix graph - Time O(m) where
m = length of align_length_seq2
            for j in range(len(align_length_seq2) + 1):
                matrix[0][j] = j * indel_penalty  # Assign variable with
multiplication - Time O(1)

            # Solve for remaining values in matrix starting from second
row/column to end of table.
            # Time O((n-1)(m-1)) which is equal to O(nm).
            #
            # Time complexity is O(nm) because the two nested for loops
iterate over each value in the sequences.
            # The way the code is set up, all of the actual genetic sequences
are longer than the expected values of
            # 1000 or 3000 for align_length that we were supposed to test
for, so in most of the test cases, n and m
            # are equal, so the time could be written as O(n^2). However, the
algorithm does not force the two
            # sequences to be the same length. The test examples using
polynomial or exponential against one of the
            # gene sequences have vastly different lengths, resulting in
significant differences between n and m.
            # On top of that, the value for align length is subjective,
meaning that while our test cases did not
            # cause it to happen, it is very possible that with higher values
of align_length more cases would have
            # different values for n and m, so time complexity is best stated
as O(nm) instead of O(n^2) or O(m^2)
            for i in range(1, len(align_length_seq1) + 1):
                for j in range(1, len(align_length_seq2) + 1):

                    # Check if diagonal value is a match or not, if so,
calculate min using match value
                    if align_length_seq1[i - 1] == align_length_seq2[j - 1]:
# If statement comparison - Time O(1)

                        # Python min()/max() break ties by selected the first
value found, so search for values in order
```

```
                        # of left, top, diagonal in order to have consistent
tie breaking.
                        matrix[i][j] = min((matrix[i][j - 1] +
indel_penalty), (matrix[i - 1][j]) + indel_penalty,
                                           matrix[i - 1][j - 1] +
match_reward)
                        # Python min - Time O(1) because it is comparing 3
values, not an entire array or list.

                    else:  # Diagonal value is not a match, use sub_penalty
for calculating min.
                        matrix[i][j] = min((matrix[i][j - 1] +
indel_penalty), (matrix[i - 1][j]) + indel_penalty,
                                           matrix[i - 1][j - 1] +
sub_penalty)
                        # Python min - Time O(1) because it is comparing 3
values, not an entire array or list.

            score = matrix[len(align_length_seq1)][len(align_length_seq2)]

            alignments = self.traceback_unrestricted(matrix,
align_length_seq1, align_length_seq2)
            alignment1 = alignments[0]
            alignment2 = alignments[1]

        # Use banded approach to solve edit distance of two sequences.
Bandwidth is caluclated from globally defined
        # variable MAXINDELS, which for testing purposes is defaulted to 3. A
matrix separate from the one used in the
        # unrestricted algorithm is created and used. This array of size
O(kn), where k is 2 *MAXINDELS + 1 and n is
        # the length of the shorter string, is used for storing and
calculating score values.
        #
        # The algorithm works by initializing the first band of values since
there are no previous values to reference,
        # so to avoid out of bounds errors, the first 7 values are done
separately. Then inside of the main while loop,
        # the value diagonal to the first value is calculated. From there,
values for 3 inserts/deletes both down and
        # right of that value are calculate, referencing values previously
calculated in the matrix. This gives us the
        # 2d+1 (k) size band that makes up the array, resulting in the space
complexity of the algorithm being O(kn),
        # since only those 2d+1 values are stored in the initialized array.
        #
        # Time complexity of the algorithm is straightforward. The while loop
iterates until j = length of the shorter
        # sequence. For each iteration of j, 2d+1 (in our cases since
MAXINDELS = 3, 2d+1 = 7) values, which are stored
        # in the array. If the two strings have very different lengths, less
values than k may be calculated on a
        # single iteration, but this is not significant enough to change that
the time complexity of this algorithm is
        # also O(kn), where k = 2d + 1 and n = length of the shorter
sequence.
        else:
```

```
            # Solve k using 2d + 1 - Time O(1)
            k = 2 * MAXINDELS + 1

            # Initialize O(kn) size matrix - Time O(kn) (each value
initialized to 0) and Space - O(kn)
            matrix = np.zeros((len(align_length_seq2) + 1, k))

            # Initialize first iteration of i/j values for bandwidth - Time
O(d) d = MAXINDELS + 1
            for i in range(0, MAXINDELS + 1):
                matrix[0][i] = i * indel_penalty

            for j in range(0, MAXINDELS + 1):
                if j > 0:
                    matrix[0][j + 3] = j * indel_penalty

            # Initialize iterator j to track current position in shorter
sequence - Time O(1)
            j = 1

            # While loop that iterates until every character in the shorter
string has been processed. The while loop
            # alone only has a time complexity of O(n). The for loop below
has the time complexity O(k), so the total
            # time complexity of the nested loops is O(kn).
            while j < len(align_length_seq2) + 1:

                # Initialize iterator values used for getting correct
position in row k - Time O(1)
                i_iterator = 1
                j_iterator = 1

                # Loop through each column k of matrix - Time O(k)
                for iterator_k in range(0, k):

                    # Handle special case of value in position 0 of current
row j.
                    if iterator_k == 0:  # If statement evaluation - Time
O(1)
                        # Search for min value from previous values in
matrix, check in order left, top, diagonal.

                        if align_length_seq1[j - 1] == align_length_seq2[j -
1]:  # If statement evaluation - Time O(1)

                            # Check for min value if diagonal is a match -
Time O(1)
                            matrix[j][iterator_k] = min((matrix[j -
1][iterator_k + 1] + indel_penalty),
                                                        matrix[j -
1][iterator_k + 1 + MAXINDELS] + indel_penalty,
                                                        matrix[j -
1][iterator_k] + match_reward)
                        else:

                            # Check for min value if diagonal is a sub - Time
O(1)
```

```
                            matrix[j][iterator_k] = min((matrix[j -
1][iterator_k + 1] + indel_penalty),
                                                        matrix[j -
1][iterator_k + 1 + MAXINDELS] + indel_penalty,
                                                        matrix[j -
1][iterator_k] + sub_penalty)

                    # Check normal case where current value to solve is in
vertical arm of band.
                    elif iterator_k < MAXINDELS:  # If statement evaluation -
Time O(1)

                        # Skip current value if band would extend beyond
bounds of array/string
                        if j + i_iterator > len(align_length_seq1):  # If
statement evaluation - Time O(1)
                            continue

                        # Search for min value from previous values in
matrix, check in order left, top, diagonal.
                        if align_length_seq1[(j - 1) + i_iterator] ==
align_length_seq2[
                            j - 1]:  # If statement evaluation - Time O(1)

                            # Check for min value if diagonal is a match -
Time O(1)
                            matrix[j][iterator_k] = min((matrix[j -
1][iterator_k + 1] + indel_penalty),
                                                        matrix[j][iterator_k
- 1] + indel_penalty,
                                                        matrix[j -
1][iterator_k] + match_reward)
                        else:

                            # Check for min value if diagonal is a sub - Time
O(1)
                            matrix[j][iterator_k] = min((matrix[j -
1][iterator_k + 1] + indel_penalty),
                                                        matrix[j][iterator_k
- 1] + indel_penalty,
                                                        matrix[j -
1][iterator_k] + sub_penalty)

                        # Update i_iterator to get next value from vertical
arm of band - Time O(1)
                        i_iterator += 1

                    # Check special case when current value is on lower end
of vertical arm of band. Since the value
                    # found to the left of the current on in the O(nm) array
is not part of the shortened O(kn) array,
                    # left insert is not even calculated in the min function,
since it is not possible in the banded.
                    # Otherwise, the rest of the cases are checked the same
way.
                    elif iterator_k == MAXINDELS:  # If statement evaluation
- Time O(1)
```

```
                        # Skip current value if band would extend beyond
bounds of array/string
                        if j + i_iterator > len(align_length_seq1):  # If
statement evaluation - Time O(1)
                            continue

                        # Search for min value from previous values in
matrix, check in order top, diagonal.
                        # Left is skipped since it does not exist in the
banded array.
                        if align_length_seq1[(j - 1) + i_iterator] ==
align_length_seq2[
                            j - 1]:  # If statement evaluation - Time O(1)

                            # Check for min value if diagonal is a match -
Time O(1)
                            matrix[j][iterator_k] = min(matrix[j][iterator_k
- 1] + indel_penalty,
                                                        matrix[j -
1][iterator_k] + match_reward)
                        else:

                            # Check for min value if diagonal is a sub - Time
O(1)
                            matrix[j][iterator_k] = min(matrix[j][iterator_k
- 1] + indel_penalty,
                                                        matrix[j -
1][iterator_k] + sub_penalty)

                    # Check mostly normal case of values in horizontal arm of
band. The first value in the horizontal
                    # arm is a little tricky, since the value to its left is
found at index 0 of the current row, not
                    # index of MAXINDELS. To counter this, a check is added
to see if the current value is the first
                    # value in the arm, if so, offset is added so the correct
left insert value is used.
                    elif iterator_k < 2*MAXINDELS:  # If statement evaluation
- Time O(1)

                        # Skip current value if band would extend beyond
bounds of array/string
                        if j + j_iterator > len(align_length_seq2):  # If
statement evaluation - Time O(1)
                            continue

                        # Check if current iterator_k is first value of
horizontal arm, set offset accordingly.
                        if iterator_k == MAXINDELS + 1:  # If statement
evaluation - Time O(1)
                            # Set variable value - Time O(1)
                            offset = iterator_k
                        else:
                            # Set variable value - Time O(1)
                            offset = 1
```

```
                        # Search for min value from previous values in
matrix, check in order left, top, diagonal.
                        if align_length_seq1[j - 1] == align_length_seq2[
                            (j - 1) + j_iterator]:  # If statement evaluation
- Time O(1)

                            # Check for min value if diagonal is a match -
Time O(1)
                            matrix[j][iterator_k] = min((matrix[j][iterator_k
- offset] + indel_penalty),
                                                        matrix[j -
1][iterator_k + 1] + indel_penalty,
                                                        matrix[j -
1][iterator_k] + match_reward)
                        else:
                            # Check for min value if diagonal is a sub - Time
O(1)
                            matrix[j][iterator_k] = min((matrix[j][iterator_k
- offset] + indel_penalty),
                                                        matrix[j -
1][iterator_k + 1] + indel_penalty,
                                                        matrix[j -
1][iterator_k] + sub_penalty)

                        # Update iterator variable to get next value in
horizontal arm of band - Time O(1)
                        j_iterator += 1

                    # Check special case of final value in horizontal arm of
band/final value in row of O(kn) array.
                    # Similar to the final value of the vertical arm of the
band, it is impossible to do all the same
                    # calculations for the min insert/sub/match. In this
case, there is no top value to use for the
                    # calculation of min, so it is ignored here.
                    elif iterator_k == 2*MAXINDELS:  # If statement
evaluation - Time O(1)

                        # Skip current value if band would extend beyond
bounds of array/string
                        if j + j_iterator > len(align_length_seq2):  # If
statement evaluation - Time O(1)
                            continue

                        # Search for min value from previous values in
matrix, check in order left, diagonal.
                        # Top is skipped since it does not exist in the
banded array.
                        if align_length_seq1[j - 1] == align_length_seq2[
                            (j - 1) + j_iterator]:  # If statement evaluation
- Time O(1)

                            # Check for min value if diagonal is a match -
Time O(1)
                            matrix[j][iterator_k] = min(matrix[j][iterator_k
- 1] + indel_penalty,
                                                        matrix[j -
```

```
1][iterator_k] + match_reward)
                        else:
                            # Check for min value if diagonal is a sub - Time
O(1)
                            matrix[j][iterator_k] = min(matrix[j][iterator_k
- 1] + indel_penalty,
                                                        matrix[j -
1][iterator_k] + sub_penalty)

                # Update j to get next line of array/next character of string
- Time O(1)
                j = j + 1

            # Calculate difference of two seqence lengths to see if a valid
alignment of the two is possible to be
            # found. If the difference is too great, the score should be
infinity and the alignment variables updated.
            diff = len(align_length_seq1) - len(align_length_seq2)  # Time
O(1)

            # If diff is smaller than k, get the score from the matrix at the
expected location.
            if diff > k:  # Evaluate if statement - Time O(1)
                score = float('inf')  # Set variable - Time O(1)
            else:
                score = matrix[j - 1][0 + diff]  # Set variable from array
value - O(1)

            # Calculate alignment values for each sequence if the score is
not infinity/valid alignment exists.
            if score != float('inf'):  # Evaluate if statement - Time O(1)

                # Use banded traceback function above to solve alignments -
Time O(n) n = length of longer sequence.
                alignments = self.traceback_banded(matrix, align_length_seq1,
align_length_seq2, diff, j)
                alignment1 = alignments[0]  # Extract value from array - Time
O(1)
                alignment2 = alignments[1]  # Extract value from array - Time
O(1)

        # Check for not valid alignments based on score for banded algorithm,
set alignment accordingly.
        if score == float('inf'):  # If statement evaluation - Time O(1)
            alignment1 = "No Alignment Possible"  # Set variable value - Time
O(1)
            alignment2 = "No Alignment Possible"  # Set variable value - Time
O(1)
        else:
            alignment1 = alignment1[:100]  # Slice alignment string O(l)
where l is the length extracted = 100 -> O(1)
            alignment2 = alignment2[:100]  # Slice alignment string O(l)
where l is the length extracted = 100 -> O(1)

        # Return score and two alignment strings to display on GUI - Time
O(1)
```

```
        return {'align_cost': score, 'seqi_first100': alignment1,
'seqj_first100': alignment2}
```

Proj4GUI.py

```
#!/usr/bin/env python3

import math
import random
import signal
import sys
import time


from which_pyqt import PYQT_VER
if PYQT_VER == 'PYQT5':
    from PyQt5.QtWidgets import *
    from PyQt5.QtGui import *
    from PyQt5.QtCore import *
elif PYQT_VER == 'PYQT4':
    from PyQt4.QtGui import *
    from PyQt4.QtCore import *
else:
    raise Exception('Unsupported Version of PyQt: {}'.format(PYQT_VER))


#TODO: Error checking on txt boxes
#TODO: Color strings


# Import in the code with the actual implementation
from GeneSequencing import *
#from GeneSequencing_complete import *


class Proj4GUI( QMainWindow ):

    def __init__( self ):
        super(Proj4GUI,self).__init__()

        self.RED_STYLE  = "background-color: rgb(255, 220, 220)"
        self.PLAIN_STYLE = "background-color: rgb(255, 255, 255)"

        self.seqs = self.loadSequencesFromFile()
        self.processed_results = []

        self.initUI()
        self.solver = GeneSequencing()

    def processClicked(self):
        sequences = [ self.seqs[i][2] for i in sorted(self.seqs.keys()) ]

        #TODO: validate alignLength
        self.statusBar.showMessage('Processing...')
        app.processEvents()
        start = time.time()

#       self.processed_results = self.solver.align( sequences,
#                                       self.table,
#                                       banded=self.banded.isChecked(),
```

```
#
align_length=int(self.alignLength.text()))
      for i in range(len(sequences)):
         jresults = []
         for j in range(len(sequences)):
            if(j < i):
               s = {}
            else:
               s = self.solver.align(sequences[i], sequences[j],
banded=self.banded.isChecked(),

align_length=int(self.alignLength.text()))
               self.table.item(i,j).setText('{}'.format(int(s['align_cost'])
if s['align_cost'] != math.inf else s['align_cost']))
               #table.repaint()
               app.processEvents()
            jresults.append(s)
         self.processed_results.append(jresults)

      end = time.time()
      ns = end-start
      nm = math.floor(ns/60.)
      ns = ns - 60.*nm
      if nm > 0:
         self.statusBar.showMessage('Done.  Time taken: {} mins and {:3.3f}
seconds.'.format(nm,ns))
      else:
         self.statusBar.showMessage('Done.  Time taken: {:3.3f}
seconds.'.format(ns))
      self.processButton.setEnabled(False)
      self.clearButton.setEnabled(True)
      self.repaint()

   def clearClicked(self):
      self.processed_results = []
      self.resetTable()
      self.processButton.setEnabled(True)
      self.clearButton.setEnabled(False)

      self.seq1n_lbl.setText( 'Label {}: '.format('I') )
      self.seq1c_lbl.setText( 'Sequence {}: '.format('I') )
      self.seq2c_lbl.setText( 'Sequence {}: '.format('J') )
      self.seq2n_lbl.setText( 'Label {}: '.format('J') )

      self.seq1_name.setText( '{}'.format(' ') )
      self.seq2_name.setText( '{}'.format(' ') )
      self.seq1_chars.setText( '{}'.format(' ') )
      self.seq2_chars.setText( '{}'.format(' ') )
      self.statusBar.showMessage('')
      self.repaint()

   def resetTable(self):
      for i in range(self.table.rowCount()):
         for j in range(self.table.columnCount()):
            if j >= i:
               self.table.item(i,j).setText(' ')
```

```
    def cellClicked(self, i, j):
        print('Cell {},{} clicked!'.format(i,j))
        print('lbls: {} and {}'.format(self.seqs[i][1],self.seqs[j][1]))

        if self.processed_results and j >= i:
            print('in if')
            self.seq1n_lbl.setText( 'Label {}: '.format(i+1) )
            self.seq1c_lbl.setText( 'Sequence {}: '.format(i+1) )
            self.seq2c_lbl.setText( 'Sequence {}: '.format(j+1) )
            self.seq2n_lbl.setText( 'Label {}: '.format(j+1) )

            self.seq1_name.setText( '{}'.format(self.seqs[i][1]) )
            self.seq2_name.setText( '{}'.format(self.seqs[j][1]) )
            results = self.processed_results[i][j]
            self.seq1_chars.setText( '{}'.format(results['seqi_first100']) )
            self.seq2_chars.setText( '{}'.format(results['seqj_first100']) )

    def loadSequencesFromFile( self ):
        FILENAME = 'genomes.txt'
        raw = open(FILENAME,'r').readlines()
        sequences = {}

        i = 0
        cur_id = ''
        cur_str = ''
        for liner in raw:
            line = liner.strip()
            if '#' in line:
                if len(cur_id) > 0:
                    sequences[i] = (i,cur_id,cur_str)
                    cur_id = ''
                    cur_str = ''
                    i += 1
                parts = line.split('#')
                cur_id = parts[0]
                cur_str += parts[1]
            else:
                cur_str += line
        if len(cur_str) > 0 or len(cur_id) > 0:
            sequences[i] = (i,cur_id,cur_str)
        return sequences

    def getTableDims( self ):
        w = self.table.columnWidth(self.table.rowCount()-1) - 4
        for i in range(self.table.columnCount()):
            w += self.table.columnWidth(i)
        h = self.table.horizontalHeader().height() + 1
        for i in range(self.table.rowCount()):
            h += self.table.rowHeight(i)
        return (w,h)

    def initUI( self ):
        self.setWindowTitle('Gene Sequence Alignment')
        self.setWindowIcon( QIcon('icon312.png') )

        self.statusBar = QStatusBar()
        self.setStatusBar( self.statusBar )
```

```
        vbox = QVBoxLayout()
        boxwidget = QWidget()
        boxwidget.setLayout(vbox)
        self.setCentralWidget( boxwidget )

        self.table = QTableWidget(self)
        NSEQ = 10
        self.table.setRowCount(NSEQ)
        self.table.setColumnCount(NSEQ)

        headers = [ 'sequence{}'.format(a+1) for a in range(NSEQ) ]
        self.table.setHorizontalHeaderLabels(headers)
        self.table.setVerticalHeaderLabels(headers)
        self.table.setHorizontalScrollBarPolicy(Qt.ScrollBarAlwaysOff)
        self.table.setVerticalScrollBarPolicy(Qt.ScrollBarAlwaysOff)

        for i in range(NSEQ):
           for j in range(NSEQ):
              qitem = QTableWidgetItem(" ")
              #qitem.setEnabled(False)
              qitem.setFlags( Qt.ItemIsSelectable |  Qt.ItemIsEnabled )
              if j < i:
                 qitem.setBackground(QColor(200,200,200))
                 qitem.setFlags( Qt.ItemIsSelectable )
              self.table.setItem(i,j,qitem)
        for i in range(NSEQ):
           self.table.resizeColumnToContents(i)
        for j in range(NSEQ):
           self.table.resizeRowToContents(i)

        w,h = self.getTableDims()
        self.table.setFixedWidth(w)
        self.table.setFixedHeight(h)

        self.processButton = QPushButton('Process')
        self.clearButton   = QPushButton('Clear')

        self.banded       = QCheckBox('Banded')
        self.banded.setChecked(False)
        self.alignLength     = QLineEdit('1000')
        font = QFont()
        font.setFamily("Courier")
        self.seq1_name    = QLineEdit('')
        self.seq1_name.setFixedWidth(650)
        self.seq1_name.setEnabled(False)
        self.seq1_chars       = QLineEdit('')
        self.seq1_chars.setFixedWidth(850)
        self.seq1_chars.setFont(font)
        self.seq1_chars.setEnabled(False)
        self.seq2_chars       = QLineEdit('')
        self.seq2_chars.setFixedWidth(850)
        self.seq2_chars.setFont(font)
        self.seq2_chars.setEnabled(False)
        self.seq2_name    = QLineEdit('')
        self.seq2_name.setFixedWidth(650)
        self.seq2_name.setEnabled(False)
```

```
        h = QHBoxLayout()
        h.addStretch(1)
        h.addWidget( self.table )
        h.addStretch(1)
        vbox.addLayout(h)

        h = QHBoxLayout()
        vleft  = QVBoxLayout()
        vright = QVBoxLayout()
        self.seq1n_lbl = QLabel('Label I: ')
        vleft.addWidget( self.seq1n_lbl )
        vright.addWidget( self.seq1_name )

        self.seq1c_lbl = QLabel('Sequence I: ')
        vleft.addWidget( self.seq1c_lbl )
        vright.addWidget( self.seq1_chars )

        self.seq2c_lbl = QLabel('Sequence J: ')
        vleft.addWidget( self.seq2c_lbl )
        vright.addWidget( self.seq2_chars )

        self.seq2n_lbl = QLabel('Label J: ')
        vleft.addWidget( self.seq2n_lbl )
        vright.addWidget( self.seq2_name )

        h.addLayout(vleft)
        h.addLayout(vright)
        vbox.addLayout(h)

        h = QHBoxLayout()
        h.addStretch(1)
        h.addWidget( self.processButton )
        h.addWidget( self.clearButton )
        h.addStretch(1)
        vbox.addLayout(h)

        h = QHBoxLayout()
        h.addStretch(1)
        h.addWidget( self.banded )
        h.addWidget( QLabel('Align Length: ') )
        h.addWidget( self.alignLength )
        h.addStretch(1)
        vbox.addLayout(h)

        self.processButton.clicked.connect(self.processClicked)
        self.clearButton.clicked.connect(self.clearClicked)
        self.clearButton.setEnabled(False)
        self.table.cellClicked.connect(self.cellClicked)

        self.show()


if __name__ == '__main__':
    # This line allows CNTL-C in the terminal to kill the program
    signal.signal(signal.SIGINT, signal.SIG_DFL)
```

```
    app = QApplication(sys.argv)
    w = Proj4GUI()
    sys.exit(app.exec())
```

which_pyqt.py

```
PYQT_VER = 'PYQT5'
```