Scott Christensen

03/10/15

C.S. 312

Project 4 Report

Time and Space Complexity:

Let n = length of the first subsequence and m = length of the second subsequence to align.

a) Unrestricted algorithm:
- Initialize the left most column in the costs matrix and the previous matrix.
  - Time Complexity: O(n) at worst because it could have n cells.
  - Space Complexity: O(nm) is part of the total storage of entire 2D matrix that stores the costs.
- Initialize the top most rows in the cost matrix and the previous matrix.
  - Time Complexity: O(m) at worst because it could have m cells.
  - Space Complexity: O(nm) is part of the total storage of entire 2D matrix that stores the costs.
- Finding the smallest cost for each remaining cell based off it's neighbors and storing the previous pointers.
  - Time Complexity: O(nm) because we iterate over every point
  - Space Complexity: O(nm) is part of the total storage of entire 2D matrix that stores the costs.
- Put together the resulting alignments
  - Time Complexity: O(n+m) because you can only iterate left n times and up m times from the bottom right cell to the top left. This gives you the alignment for each sequence.
  - Space complexity : O(n + m) because you are storing n characters from the first alignment and m characters from the second alignment.
  -

b) Banded algorithm:
- Initialize the left most column in the costs matrix and the previous matrix up to 4 cells.
  - Time Complexity: O(n) at worst because it could have n cells which could be less than 4.
  - Space Complexity: O(nm) is part of the total storage of entire 2D matrix that stores the costs.
- Initialize the top most rows in the cost matrix and the previous matrix.
  - Time Complexity: O(m) at worst because it could have m cells which could be less than 4.

    - Space Complexity: O(nm) is part of the total storage of entire 2D matrix that stores the costs.
- Starting at the top left, we assign each cell a cost based on its neighbor's. This value = Min(Top, left, diag). We are only calculating 7 consecutive cells in a row. This makes the operation O(7n + 7m) or O(n + m) when constants are dropped.
    - Time Complexity: O(nm) because with the banding we compare each character in the first sequence against no more than 7 characters in the second sequence and vise versa.
    - Space Complexity: O(nm) is part of the total storage of entire 2D matrix that stores the costs.
- Put together the resulting alignments
    - Time Complexity: O(n+m) because you can only iterate left n times and up m times from the bottom right cell to the top left. This gives you the alignment for each sequence.
    - Space complexity : O(n + m) because you are storing n characters from the first alignment and m characters from the second alignment.

c) Conclusion
- Unrestricted algorithm:
    i. Time Complexity: O(nm + n + m) = O(nm) after the constants are dropped.
    ii. Space Complexity: O(nm + n + m) = O(nm) after the constants are dropped.
- Banded algorithm:
    i. Time Complexity: O(2n+2m) = O(nm) after the constants are dropped.
    ii. Space Complexity: O(nm + n + m) = O(nm) after the constants are dropped.

<u>Alignment Extraction algorithm explanation:</u>

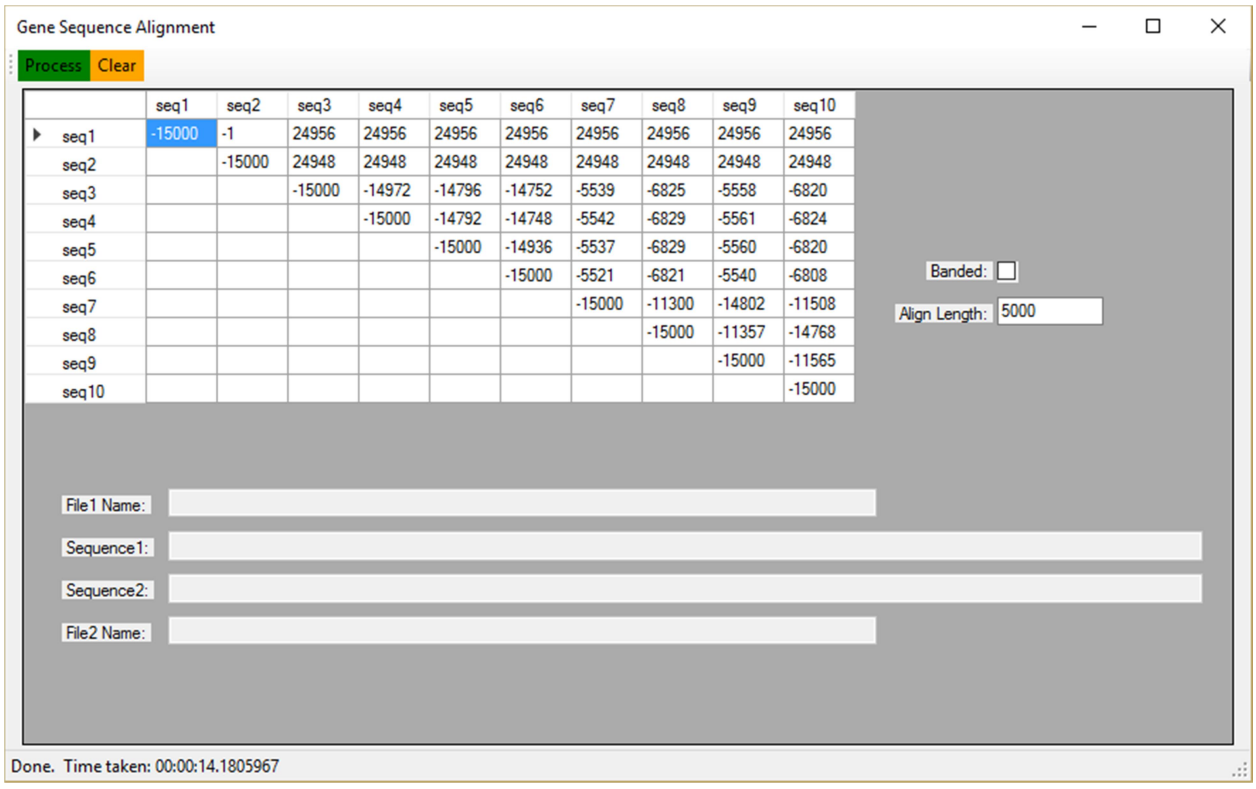
I made a 2D matrix that stores chars for each cell in the 2D cost matrix called Previous. ‘U’ means that we got to that cell via a delete operation, or to move up. “I” means we got there from an add operation or to move left. “D” means we got to that cell from a match or substitution action, or to move diagonal.

Starting from the bottom right cell in the cost matrix, while referring to the previous matrix for the cell we are at, we checked where we came from. If we came from the diagonal, we add both of the chars at that position of the sequence of the gene sequence to the alignment. If we came from the left, we would add the char from the second sequence to its resulting alignment and add a – to the first. If we came from the top, we add the char from the first sequence to its resulting alignment and add a – to the second. Each time we do and operation, we decrement our current position accordingly.

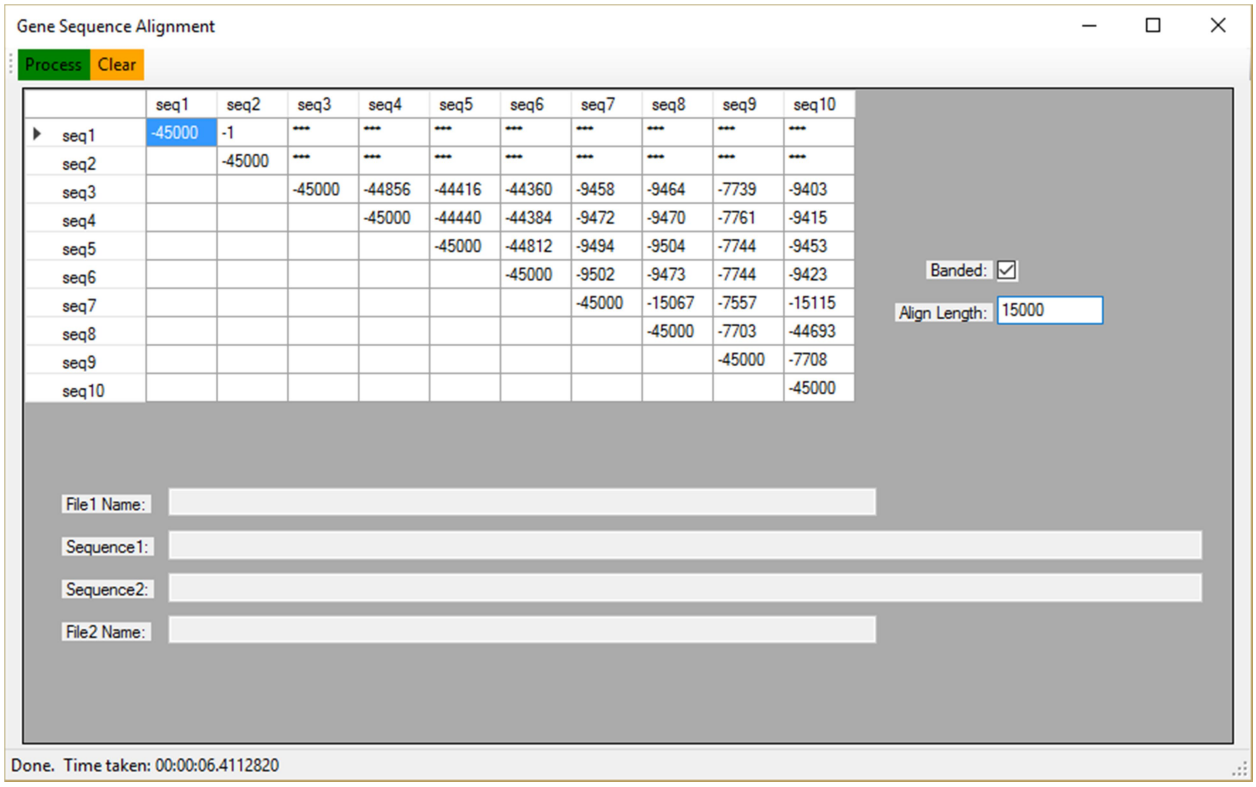
Since both of the resulting alignments are in backwards order, we simply reverse both alignments and return them.

Screenshots:

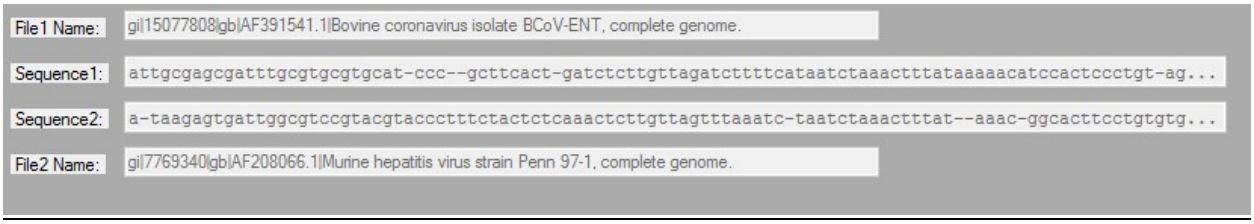
Unrestricted:



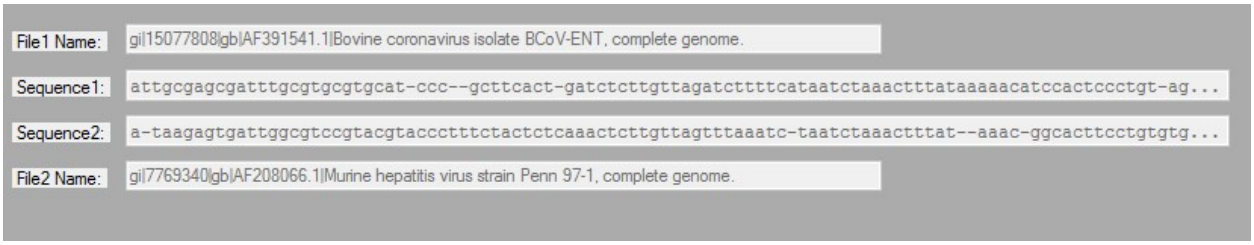
Banded:



Extracted alignment for the first 100 characters of sequences #3 and #10, computed using the unrestricted algorithm with k = 5000.



Extracted alignment for the first 100 characters of sequences #3 and #10, computed using the banded algorithm with k = 15000.



My Code:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace GeneticsLab
{
    class PairWiseAlign
    {
        int MaxCharactersToAlign;
        //The cost values
        int indel_val = 5;
        int sub_val = 1;
        int match_val = -3;

        //strings we use for comparing
        char diag = 'd';
        char up = 'u';
        char left = 'l';
        char slash = '-';

        public PairWiseAlign()
        {
            // Default is to align only 5000 characters in each sequence.
            this.MaxCharactersToAlign = 5000;
        }

        public PairWiseAlign(int len)
        {
            // Alternatively, we can use an different length; typically used with the
banded option checked.
            this.MaxCharactersToAlign = len;
        }

        /// <summary>
        /// this is the function you implement.
        /// </summary>
        /// <param name="sequenceA">the first sequence</param>
        /// <param name="sequenceB">the second sequence, may have length not equal to the
length of the first seq.</param>
        /// <param name="banded">true if alignment should be band limited.</param>
        /// <returns>the alignment score and the alignment (in a Result object) for
sequenceA and sequenceB.  The calling function places the result in the dispay
appropriately.
        ///
        public ResultTable.Result Align_And_Extract(GeneSequence sequenceA, GeneSequence
sequenceB, bool banded)
        {
            ResultTable.Result result = new ResultTable.Result();
            int score;                                                      // place
your computed alignment score here
            string[] alignment = new string[2];                             // place
your two computed alignments here

            // ********* these are placeholder assignments that you'll replace with your
code  *******
            score = 0;
            alignment[0] = "";
            alignment[1] = "";
```

```
            int row_size = 0;
            int col_size = 0;
            int i_loop = 0;
            int j_loop = 0;
            char[] a = sequenceA.Sequence.ToCharArray();
            char[] b = sequenceB.Sequence.ToCharArray();
            i_loop = Math.Min(sequenceA.Sequence.Length, MaxCharactersToAlign);
            j_loop = Math.Min(sequenceB.Sequence.Length, MaxCharactersToAlign);

            //if the case it itself vs itself, jsut return all diagnols
            if (sequenceA.Sequence.Substring(0,
i_loop).Equals(sequenceB.Sequence.Substring(0, j_loop)))
            {
                result.Update(MaxCharactersToAlign * -3, sequenceA.Sequence.Substring(0,
i_loop), sequenceB.Sequence.Substring(0, j_loop));
                return (result);
            }



            if (!banded)//if the banded checkbox is false
            {
                //store the length of the first and seond sequence, which is used in the
for loops
                //i_loop, row_size is n
                //j-loop, col_size is m
                //gives us the length of the genom or the max number we put in, like 5000
or 15000.
                row_size = Math.Min(a.Length, MaxCharactersToAlign) + 1;
                col_size = Math.Min(b.Length, MaxCharactersToAlign) + 1;
                //the 2D array we'll use to store the costs.
                int[,] e = new int[row_size, col_size];
                //the array to help us recreate the alignment strings with previous
points.
                char[,] previous = new char[row_size, col_size];

                /*
                 * Assign all of the cells on the leftmost column to go 'up' with the
cost of INDEL, or 5 .
                 * Time complexity: O(n) at worst because it could have n cells.
                 * Space complexity: O(nm) at worst for the entire 2D matrix the stores
the costs ant the matrix that stores the previous pointers.
                 */
                for (int i = 0; i <= i_loop; i++)
                {
                    e[i, 0] = i * indel_val;
                    previous[i, 0] = up;
                }
                /*
                 * Assign all of the cells on the top row to go 'left' with the cost of
INDEL, or 5 .
                 * Time complexity: O(m) at worst because it could have m cells.
                 * Space complexity: O(nm) at worst for the entire 2D matrix the stores
the costs ant the matrix that stores the previous pointers.
                 */
                for (int j = 1; j <= j_loop; j++)
                {
                    e[0, j] = j * indel_val;
```

```
                    previous[0, j] = left;
                }

                /*
                 * Starting at the top left, we assign each cell a cost based on its
neighbor's. This value = Min( Top, left, diag). Then we store the direction that we came
from into the previous matrix.
                 * Time complexity: O(nm) because we iterate over every cell.
                 * Space complexity : O(nm)  because we store every cell.
                 */
                for (int i = 1; i <= i_loop; i++)
                {
                    for (int j = 1; j <= j_loop; j++)
                    {
                        int UP = e[i - 1, j] + indel_val;
                        int LEFT = e[i, j - 1] + indel_val;
                        int DIAG = e[i - 1, j - 1] + matcher(a[i - 1], b[j - 1]);
                        int min = Math.Min(Math.Min(UP, LEFT), DIAG);

                        e[i, j] = min;

                        if (DIAG == min) previous[i, j] = diag;
                        else if (UP == min) previous[i, j] = up;
                        else previous[i, j] = left;
                    }
                }
                //assign the total min cost
                score = e[i_loop, j_loop];

                //make the alignment strings
                alignment = makeAligments(i_loop, j_loop, previous, a, b);

                result.Update(score, alignment[0], alignment[1]);
                return result;
            }
            else//banded ----------------------------------------------------------------
----------------------------------------------------------------------
            {
                //initialize some values.
                row_size = Math.Min(a.Length, MaxCharactersToAlign) + 1;
                col_size = Math.Min(b.Length, MaxCharactersToAlign) + 1;
                i_loop = Math.Min(a.Length, MaxCharactersToAlign);//length1
                j_loop = Math.Min(b.Length, MaxCharactersToAlign);//length2

                //check to see if we can actually do this operation because the banding
will not calculate strings that are too different in size.
                if (Math.Abs(i_loop - j_loop) > 3)
                {
                    score = int.MaxValue;
                    alignment[0] = "No Alignment Possible";
                    alignment[1] = "No Alignment Possible";
                    result.Update(score, alignment[0], alignment[1]);
                    return result;
                }


                int[,] e = new int[row_size, col_size];//the 2D array we'll use to solve
our problem.
```

```
                char[,] previous = new char[row_size, col_size];//the array to help us
recreate the alignment strings.

                /*
                 * Assign 4 cells on the leftmost column to go 'up' with the cost of
INDEL, or 5 .
                 * Time complexity: O(n) at worst because it could have n cells which
happens to be less that 4.
                 * Space complexity: O(nm) at worst for the entire 2D matrix the stores
the costs ant the matrix that stores the previous pointers.
                 */
                for (int i = 0; i <= Math.Min(i_loop, 3); i++)
                {
                    e[i, 0] = i * indel_val;
                    previous[i, 0] = up;
                }
                /*
                 * Assign 4 cells on the top row to go 'left' with the cost of INDEL, or
5 .
                 * Time complexity: O(m) at worst because it could have n cells which
happens to be less that 4.
                 * Space complexity: O(nm) at worst for the entire 2D matrix the stores
the costs ant the matrix that stores the previous pointers.
                 */
                for (int j = 1; j <= Math.Min(j_loop, 3); j++)
                {
                    e[0, j] = j * indel_val;
                    previous[0, j] = left;
                }


                /*
                 * Starting at the top left, we assign each cell a cost based on its
neighbor's. This value = Min( Top, left, diag). We are only caluculating 7 consecutive
cells in a row.
                 *   This makes the operation O(7n + 7m) or O(n + m) when constants are
dropped.
                 * Time complexity: O(n + m) because with the banding we compare each
character in the first sequence against no more than 7 characters in the second sequence
and vise versa.
                 * Space complexity : O(nm)  because we store every cell.
                 */
                for (int i = 1; i <= i_loop; i++)
                {

                    //me must do a calculation for how much of the current row we're on
                    int start = Math.Max(i - 3, 1);
                    int end = Math.Min(j_loop, i + 3);
                    for (int j = start; j <= end; j++)
                    {
                        //We force the operations in banded by setting up and left
directions to it and
                        //we only correct it if there is not 3 of more indels.
                        int UP = int.MaxValue;
                        int LEFT = int.MaxValue;
                        if (j != i + 3) UP = e[i - 1, j] + indel_val;
                        if (j != i - 3) LEFT = e[i, j - 1] + indel_val;
                        int DIAG = e[i - 1, j - 1] + matcher(a[i - 1], b[j - 1]);
```

```
                        int min = Math.Min(Math.Min(UP, LEFT), DIAG);

                        e[i, j] = min;

                        if (DIAG == min) previous[i, j] = diag;

                        else if (UP == min) previous[i, j] = up;

                        else previous[i, j] = left;
                    }
                }
                //set the score from the last value we calculated
                score = e[row_size - 1, col_size - 1];

                //make the alignment strings
                alignment = makeAligments(i_loop, j_loop, previous, a, b);

                result.Update(score, alignment[0], alignment[1]);
                return result;
            }
        }

        public int matcher(char one, char two)
        {
            /*This function tells us if we need to do a equals or swap operation and
assigns MATCH -3 or SWAP 1.
             * Time complexity: O(1) because its a direct comparison.
             * Space complexity : O(1) there is no storage!
             */

            if (one.Equals(two))
            {
                return match_val;
            }
            else
            {
                return sub_val;
            }
        }

        public String[] makeAligments(int i_loop, int j_loop, char[,] previous, char[] a,
char[] b)
        {

            //now we must reconstruct the alignment strings------------------------------
--------------------------------------------------------------------------------
            int row = i_loop;
            int col = j_loop;
            String[] alignment = new String[2];
            alignment[0] = "";
            alignment[1] = "";

            //iterate in reverse
            /*
             * This function in a while loop that follows the back pointers from the
bottom right corner up to the origin and stores the characters along the way.
             * Time complexity: O(n + m) because you can only go left m times and up n
times until you hit the origin.
```

```
             * Space complexity : O(n + m) because you are storeing n characters from the
first alignment and m characters from the second alignment.
             */
            while (row > 0 && col > 0)
            {
                if (previous[row, col].Equals(diag))
                {
                    alignment[0] += a[row - 1];
                    alignment[1] += b[col - 1];
                    row = row - 1;
                    col = col - 1;
                }
                else if (previous[row, col].Equals(up))
                {
                    alignment[0] += a[row - 1];
                    alignment[1] += slash;
                    row = row - 1;
                }
                else //from left//
                {
                    alignment[0] += slash;
                    alignment[1] += b[col - 1];
                    col = col - 1;
                }
            }

            /* Reverse the strings we just made with stringbuilders for the most
efficient time.
             * Time complexity: O(n or m) because we reverse every n or m character.
             * Space complexity : O(n + m)  because we store each of those characters.
             */
            StringBuilder reverse = new StringBuilder(alignment[0].Length);
            for (int i = alignment[0].Length - 1; i >= 0; i--)
            {
                reverse.Append(alignment[0][i]);
            }
            alignment[0] = reverse.ToString();

            reverse.Clear();
            reverse = new StringBuilder(alignment[1].Length);
            for (int i = alignment[1].Length - 1; i >= 0; i--)
            {
                reverse.Append(alignment[1][i]);
            }
            alignment[1] = reverse.ToString();

            return alignment;
        }
    }
}
```