Yazan Halawa

3/13/16

CS 312 Section 1

Lab 4 Report-Gene Sequencing

***Lab Code:***

////////////////////////////////////////////////////////////////////////////////////////////////////

///////////////////////////////////////// Helper Functions //////////////////////////////////////// ////////////////////////////////////////////////////////////////////////////////////////////////////

/\*\*

\* This function fills the first row and column with the cost of insert/delete for each one

\* Time Complexity: O(n+m) where n is the length of the first sequence and m is the length of the second sequence.

\* This is because it iterates over all letters in each sequence once

\* Space Complexity: O(1) because it passes the values by reference meaning it does not create a copy and

\* it does not create any variables that depend on the size of the input.

\*/

void fillStartCells(ref int[,] values, ref directions[,] prev, int lengthA, int lengthB, bool banded)

{

for (int column = 0; column < lengthB + 1; column++)

{

if (banded == true && (column > distance))

{

break;

}

values[0, column] = column \* 5;

prev[0, column] = directions.LEFT;

}

for (int row = 0; row < lengthA + 1; row++)

{

if (banded == true && (row > distance))

{

break;

}

values[row, 0] = row \* 5;

prev[row, 0] = directions.TOP;

}

}

/\*\*

\* This function creates the alignments for both sequences using the previous pointers array

\* Time Complexity: O(n) where n is the length of the larger sequence because it the best alignment

\* is as long as the length of the longest sequence

\* Space Complexity: O(n) where n is the length of the larger sequence as it creates a string as long as it

\*/

void createAlignments(ref string[] alignment, ref directions[,] prev, ref GeneSequence sequenceA, ref GeneSequence sequenceB,

ref int lengthOfSequenceA, ref int lengthOfSequenceB)

{

int rowIterator = lengthOfSequenceA, columnIterator = lengthOfSequenceB;

StringBuilder first = new StringBuilder(), second = new StringBuilder();

while (rowIterator != 0 || columnIterator != 0)

{

if (prev[rowIterator, columnIterator] == directions.DIAGONAL) // match/sub

{

first.Insert(0, sequenceA.Sequence[rowIterator - 1]);

second.Insert(0, sequenceB.Sequence[columnIterator - 1]);

rowIterator--;

columnIterator--;

}

else if (prev[rowIterator, columnIterator] == directions.LEFT) //insert

{

first.Insert(0, '-');

second.Insert(0, sequenceB.Sequence[columnIterator - 1]);

columnIterator--;

}

else // delete

{

first.Insert(0, sequenceA.Sequence[rowIterator - 1]);

second.Insert(0, '-');

rowIterator--;

}

}

// Limiting the length of the string to 100 if it exceeds it

alignment[0] = first.ToString().Substring(0, Math.Min(first.Length, 100));

alignment[1] = second.ToString().Substring(0, Math.Min(second.Length, 100));

}

//////////////////////////////////////////////////////////////////////////////////////////////////// /////////////////////////////////// Unrestricted Algorithm ///////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////////////////////////

/\*\*

\* This function performs the unrestricted algorithm on the two sequences using dynamic programming to come up with

\* the best alignment for both.

\* Time Complexity: O(nm) where n is the length of the first sequence and m is the length of the second sequence. This

\* is because the algorithm iterates over all cells in the array of n x m

\* Space Complexity: O(nm) where n is the length of the first sequence and m is the length of the second sequence. This

\* is because the algorithm creates an array of n x m

\*/

void unrestrictedAlgorithm (ref int score, ref string[] alignment, ref GeneSequence sequenceA, ref GeneSequence sequenceB)

{

// Limiting the lengths of the sequences to the max characters to align

int lengthOfSequenceA = Math.Min(sequenceA.Sequence.Length, MaxCharactersToAlign);

int lengthOfSequenceB = Math.Min(sequenceB.Sequence.Length, MaxCharactersToAlign);

// Create two arrays to hold the intermediate values and the alignment details

int[,] values = new int[lengthOfSequenceA + 1, lengthOfSequenceB + 1];

directions[,] prev = new directions[lengthOfSequenceA + 1, lengthOfSequenceB + 1];

// first fill first row and column with cost of inserts/deletes

fillStartCells(ref values, ref prev, lengthOfSequenceA, lengthOfSequenceB, false);

// Now iterate through the rest of the cells filling out the min value for each

for (int row = 1; row < lengthOfSequenceA + 1; row++)

{

for (int column = 1; column < lengthOfSequenceB + 1; column++)

{

// Compute values for each direction

int costOfTop\_Delete = values[row - 1, column] + 5;

int costOfLeft\_Insert = values[row, column - 1] + 5;

// Compute cost of moving from diagonal depending on whether the letters match

int costOfMovingFromDiagonal = (sequenceA.Sequence[row - 1] == sequenceB.Sequence[column - 1]) ? -3 : 1;

int costOfDiagonal = values[row - 1, column - 1] + costOfMovingFromDiagonal;

// value of cell would be the minimum cost out of the three directions

int costOfMin = Math.Min(costOfTop\_Delete, Math.Min(costOfLeft\_Insert, costOfDiagonal));

values[row, column] = costOfMin;

// Store the direction

if (costOfMin == costOfDiagonal)

{

prev[row, column] = directions.DIAGONAL;

}

else if (costOfMin == costOfLeft\_Insert)

{

prev[row, column] = directions.LEFT;

}

else

{

prev[row, column] = directions.TOP;

}

}

}

// score would be value of the last cell

score = values[lengthOfSequenceA, lengthOfSequenceB];

// Create the alignments

createAlignments(ref alignment, ref prev, ref sequenceA, ref sequenceB, ref lengthOfSequenceA, ref lengthOfSequenceB);

}

////////////////////////////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////// Banded Algorithm ///////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////////////////////////

/\*\*

\* This function performs the banded algorithm on the two sequences using dynamic programming to come up with

\* the best alignment for both. The band is set to whatever the distance is. Currently it is d = 3 which makes the

\* bandwidth equals 2d+1 = 7.

\* Time Complexity: O(n+m) where n is the length of the first sequence and m is the length of the second sequence. This

\* is because the algorithm iterates over a specific number of cells for each row and column. As we don't

\* care about constants, the time would depend on the length of sequence A and B. Meaning each time

\* the array size is increased by a row or a column, we have to compute those bandwidth number of cells

\* again, so it is O(n+m).

\* Space Complexity: O(nm) where n is the length of the first sequence and m is the length of the second sequence. This

\* is because the algorithm creates an array of n x m

\*/

void bandedAlgorithm(ref int score, ref string[] alignment, ref GeneSequence sequenceA, ref GeneSequence sequenceB)

{

// Limiting the lengths of the sequences to the max characters to align

int lengthOfSequenceA = Math.Min(sequenceA.Sequence.Length, MaxCharactersToAlign);

int lengthOfSequenceB = Math.Min(sequenceB.Sequence.Length, MaxCharactersToAlign);

// Create two arrays to hold the intermediate values and the alignment details

int[,] values = new int[lengthOfSequenceA + 1, lengthOfSequenceB + 1];

directions[,] prev = new directions[lengthOfSequenceA + 1, lengthOfSequenceB + 1];

// first fill first row and column with cost of inserts/deletes

fillStartCells(ref values, ref prev, lengthOfSequenceA, lengthOfSequenceB, true);

int columnStart = 1;

bool alignmentFound = false;

int row = 1;

int column = columnStart;

// Now iterate through the rest of the cells filling out the min value for each

for (row = 1; row < lengthOfSequenceA + 1; row++)

{

for (column = columnStart; column < lengthOfSequenceB + 1; column++)

{

if ((distance + row) < column)

{

break;

}

// Compute values for each direction

int costOfTop\_Delete = values[row - 1, column] + 5;

if ((distance + row) == column)

{

costOfTop\_Delete = int.MaxValue;

}

int costOfLeft\_Insert = values[row, column - 1] + 5;

if ((distance + column) == row)

{

costOfLeft\_Insert = int.MaxValue;

}

// Compute cost of moving from diagonal depending on whether the letters match

int costOfMovingFromDiagonal = (sequenceA.Sequence[row - 1] == sequenceB.Sequence[column - 1]) ? -3 : 1;

int costOfDiagonal = values[row - 1, column - 1] + costOfMovingFromDiagonal;

// value of cell would be the minimum cost out of the three directions

int costOfMin = Math.Min(costOfDiagonal, Math.Min(costOfLeft\_Insert, costOfTop\_Delete));

values[row, column] = costOfMin;

// Store the direction

if (costOfMin == costOfDiagonal)

{

prev[row, column] = directions.DIAGONAL;

}

else if (costOfMin == costOfLeft\_Insert)

{

prev[row, column] = directions.LEFT;

}

else

{

prev[row, column] = directions.TOP;

}

if (column == lengthOfSequenceB && row == lengthOfSequenceA)

alignmentFound = true;

}

if (row > distance)

columnStart++;

}

// score would be value of the last cell

if (alignmentFound)

{

score = values[lengthOfSequenceA, lengthOfSequenceB];

// Create the alignments

createAlignments(ref alignment, ref prev, ref sequenceA, ref sequenceB,

ref lengthOfSequenceA, ref lengthOfSequenceB);

}

else {

score = int.MaxValue;

alignment[0] = "No Alignment Possible";

alignment[1] = "No Alignment Possible";

}

}

//////////////////////////////////////////////////////////////////////////////////////////////////// /////////////////////////////////////Main Entry Point/////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////////////////////////

/// <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] = "";

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

if (!banded)

unrestrictedAlgorithm(ref score, ref alignment, ref sequenceA, ref sequenceB);

else

bandedAlgorithm(ref score, ref alignment, ref sequenceA, ref sequenceB);

result.Update(score, alignment[0], alignment[1]); // bundling your results into the right object type

return (result);

}

}

***Time and Space Complexity Analysis:***

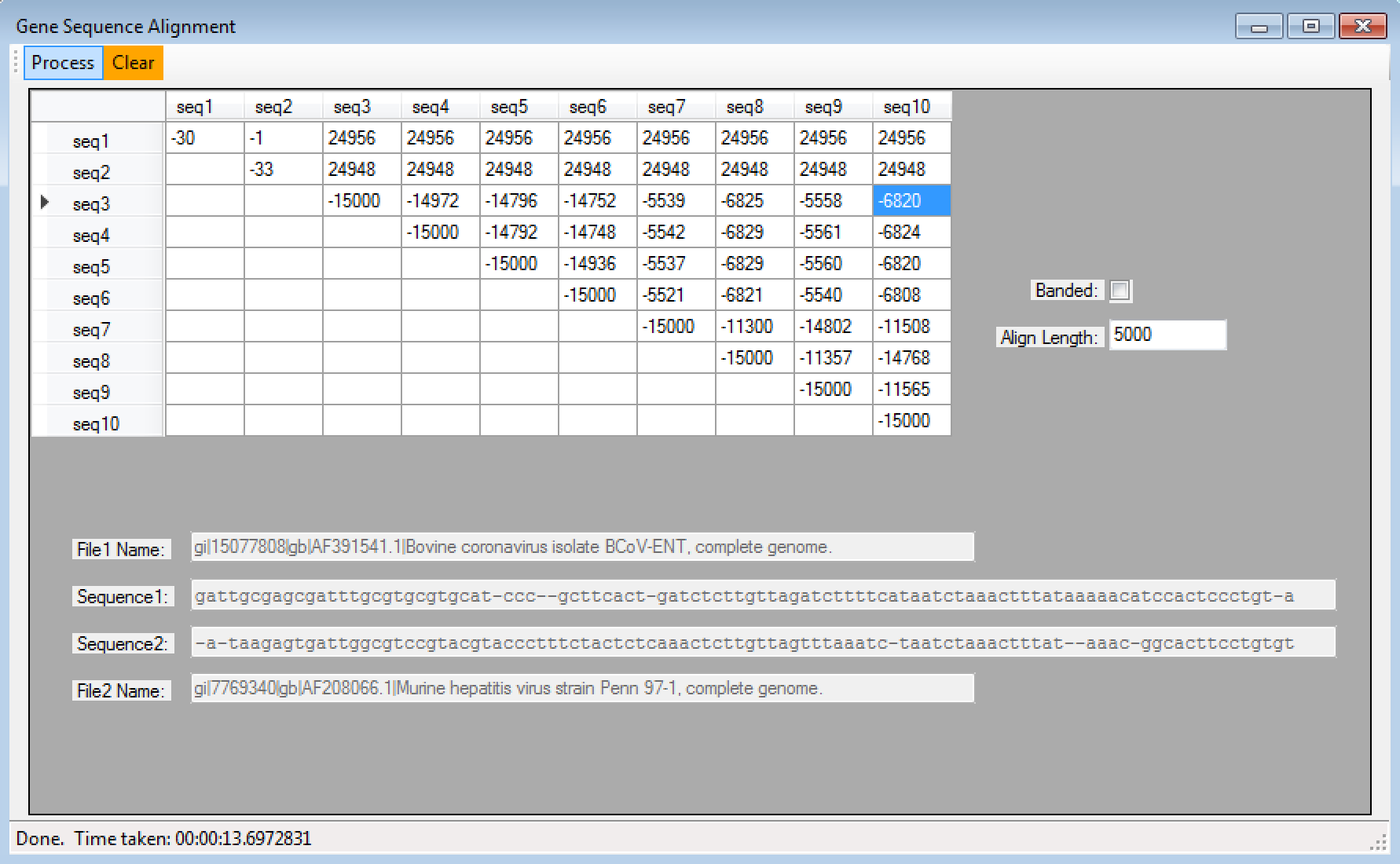
The analysis is presented as comment sections above each function in the lab code. Please refer to the comments above.

***Alignment Extraction Algorithm:***

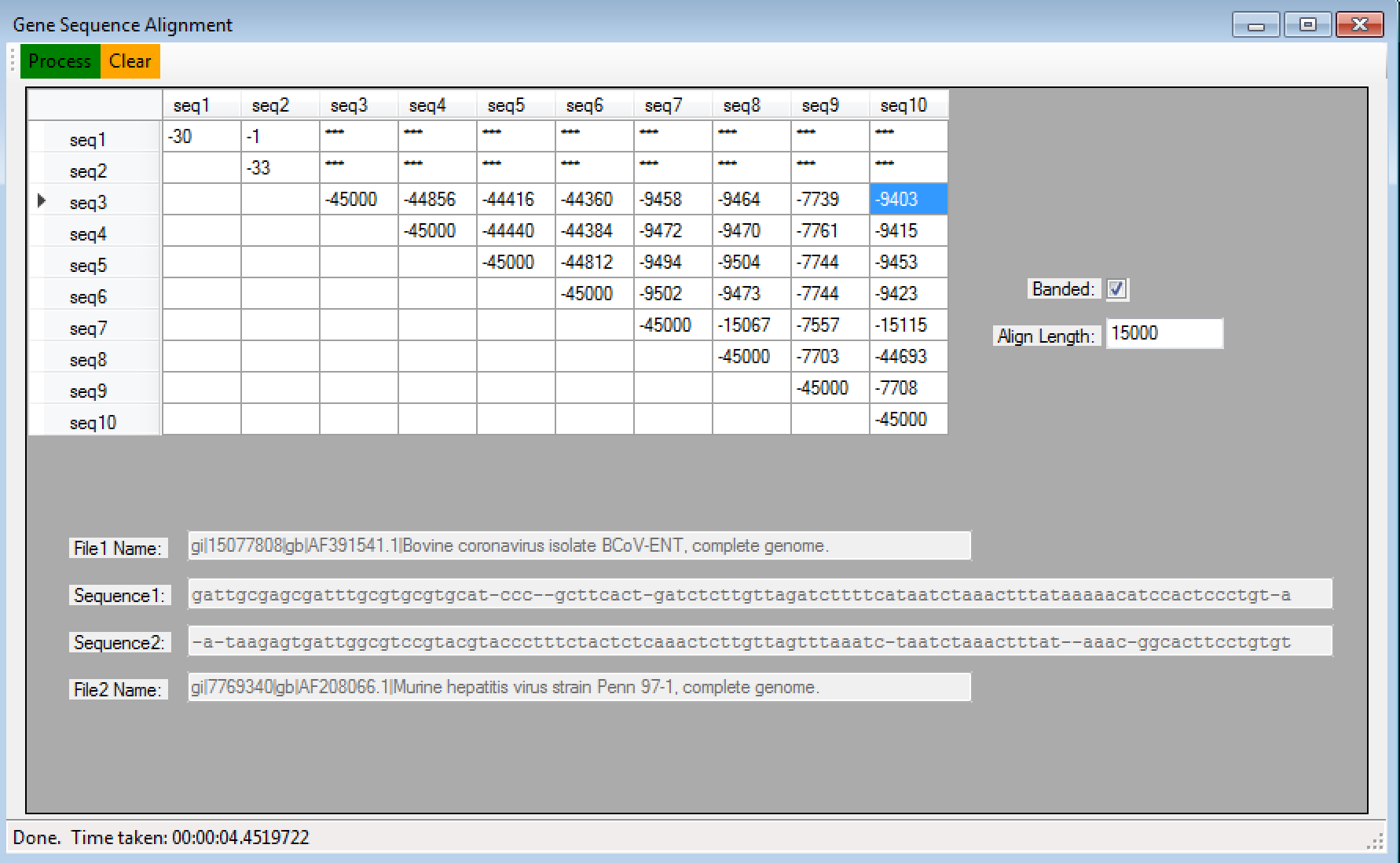
The way this algorithm works is using dynamic programming by dividing the problem into sub problems and using those to compute our results. To go from one string to another, for each character there are three possibilities: inserting a character, deleting a character, and substitution. So for each character in the strings we compare the cost of inserting, deleting, and substitution and pick the minimum one as the way we go. Then to keep track of what we do for each letter, we keep another array that stores the choice we picked from each one. If we substitute, then we are moving diagonally in the list. If we insert, then we are moving across, and if we delete, then we are moving down. That way once we reach the last cell in the table, then we have the minimum cost for aligning the two string sequences, and we have a back-trace stored in a second table that takes us back to the first cell. Now to come up with the alignment, we just follow that back-trace: for each cell, if it we moved diagonally, then we display the two letters, if we are inserting or deleting, then one of the letters will be a dash, and so on.

***Results:***

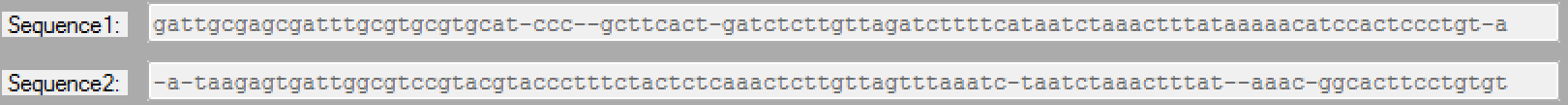
**Unrestricted with K = 5000:**



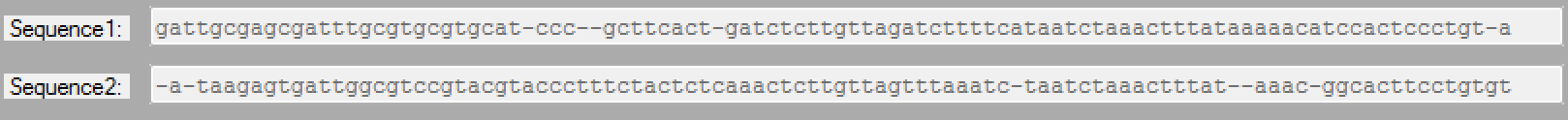
**Banded with K = 15000:**

****

**As shows in the screenshot above, the extracted alignment for row 3, column 10 (unrestricted, K=5000) is:**

****

**(Banded, K=15000) is:**

****