Sam Rapp
UCSC Computer Science & Bioinformatics undergraduate
Nov 29, 2018
Working under Professor Nyguen of USF, and TidePool

# Overview and Analysis:
# Diagonal Traversals to Find Duplicate Sets in Time Series Data

**Introduction:**

My java code for the Matrix Profile currently runs based on two files. Everything runs inside the main method of the child class, Distance Matrix, which implements the class SquareGrid. Different versions of the code either run based on my test arrays or based on arrays created via .txt files full of the raw integer data points Tide Pool provides. Given two arrays of equal or unequal length, the code creates a distance matrix. When two values being compared are exactly equal the Distance Matrix will produce a zero. We are looking for continuous strings of zeros, and total zeros per comparison, to find duplicate sets of data.

**Explanation and Comparison of traversals options:**

This section of the paper will cover the differences between horizontal and diagonal traversals to create of distance matrices and why this implementation is focused around diagonal traversals.

The coordinate grid values are only positive and are formatted (x,y). Y being the vertical axis(up/down) X being the horizontal axis(left/right). See diagram 1.

Diagram 1:

| | | | |
|---|---|---|---|
| (0,0) | (1,0) | (2,0) | (3,0) |
| (0,1) | (1,1) | (2,1) | (3,1) |
| (0,2) | (1,2) | (2,2) | (3,2) |
| (0,3) | (1,3) | (2,3) | (3,3) |

*Grid coordinates for a comparison of 2 data sets each 4 values long.

The computation to make a Distance Matrix is rather simple, especially when the only focus is on exact matches (zeros). My code makes the grid via diagonal traversals rather than going row by row(horizontally). Making the grid row by row is simpler to code (all that is required is two nested for loops), but is limited because it only compares one value from a set against every value from the other set. So given two sets of data called X and Y. Row by row comparison is comparing Xi against Yi, Yi+1, Yi+2...Yfinal. The same process is then repeated for Xi+1...Xfinal. With horizontal traversals, there are as many traversals as there are rows. (Diagram 3) Horizontal traversals can easily create a Distance Matrix, however simply creating the Distance Matrix data structure provides no information. The diagonal traversal approach allows information to be gathered, during the process of creating the matrix.

**Horizontal Traversals:**

Diagram 2:

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | 16 |

*The numbers(1-16) represent the order of comparisons in a row by row(horizontal) Distance Matrix creation.

Diagram 3:

| | | | |
|---|---|---|---|
| 1 | x | x | x |
| 2 | x | x | x |
| 3 | x | x | x |
| 4 | x | x | x |

*The numbers 1-4, each represent a horizontal traversals and the order in which they occur.

---

Building the Distance Matrix diagonally, allows for consecutive comparisons. So if you start by comparing Xi with Yi you then compare Xi+1 with Y(i+1), then X3 with Y(i+2) until the program reaches the end of one of the data sets. Creating an distance matrix with diagonal traversal means incrementally advantances the comparison data points in both sets, rather than one at a time. Thus, diagonal traversals allows the program to make every possible comparison, and record structured data about matching values and their locations, simultaneously. One simple thing it can find that a horizontal based traversal cannot is strings of matching data points rather than individual matches.

**Diagonal Traversals:**

Diagram 4:

| | | | |
|---|---|---|---|
| 1 | 11 | 14 | 16 |
| 5 | 2 | 12 | 15 |
| 8 | 6 | 3 | 13 |
| 10 | 9 | 7 | 4 |

*The numbers(1-16) represent the order of comparisons in a diagonal Distance Matrix creation.

Diagram 5:

| | | | |
|---|---|---|---|
| 1 | 5 | 6 | 7 |
| 2 | x | x | x |
| 3 | x | x | x |
| 4 | x | x | x |

*The numbers 1-7, each represent a diagonal traversals and the order in which they occur. With diagonal traversals, there are as many traversals as there are edge positions on the grid or (SideLengthX+ SideLengthY-1).

**Explanation of My Codes Diagonal Traversal Method**

The first traversal is always from the top left corner of the grid, coordinate (0,0), to the bottom right position, (SideLengthX-1, SideLengthY-1), and represents the longest traversal. This is worth noting because it has the greatest potential for matches. While traversals numbers 4/7 in diagram 5 represent the minimum comparisons possible, 1. To optimize the time complexity of the code we set limits on how small a given traversal could be. If it fell below a certain threshold the traversal and all remaining traversals of smaller lengths were not computed, thus saving computation time. The threshold could represent a percentage of the longest traversal or a static number. This adaptation "cuts" off the corners of the matrix.

The code is run primarily by two main functions, DisMatrixFillingDiagonally(), and MatrixHelper(). DisMatrixFillingDiagonally() takes in two arrays. It determines the number of diagonal traversals that will begin on the X and Y axises, based on their lengths. It then determines the shorter side length between the two sets, called Q. Q is the longest possible diagonal traversal and thus is used to set our threshold in this cut off senario. There are then two separate for loops. One iterates through the # of diagonal traversals on the X and the other does the same for Y axis traversals. In each loop there is a check if the traversal will be above or below the threshold. If it is above that threshold and we want to compute the Distance Matrix value, the loop calls MatrixHelper(), which completes one diagonal traversal then returns the values needed for the starting position of the next traversal.

MatrixHelper(), takes in two arrays, the starting X, Y coordinates for the traversal, a counter for how far from the center diagonal this traversal is, and a counter of the number of traversals that have occured. It returns how far it is from the center diagonal. Within MatrixHelper there is a for loop that iterates from the starting position to the final position of the traversal. In a given loop it takes the coordinates, compares the values, records weather it was a match or not, records how many matches have occured consecutively if any, then jumps to the next iteration of the loop by incrementing the X and Y values. It then checks to see if we have stepped out of the bounds of either of our lists aka if the X/Y index is greater than the length of X/Y array, if so it jumps out knowing it has completed a diagonal traversal. If not then it repeats the previous steps.

Within this method are various information gathering tools(static members of the Distance Matrix class). Two arrays, and matrix(see Visualized Data Structures x,z,y in the results section below). The different structures record: each zeros(match), the longest string of consecutive zeros, and its starting

position  in a traversal. Each index corresponds to a specific a traversal. Index zero refers to the longest traversal starting at (0,0). The shortest traversals, each 1 comparison long are at the middle index and the last index in the arrays and matrix

**Time Complexity Analysis DisMatrixFillingDiagonally():**

The time complexity of calling the DisMatrixFillingDiagonally() method given two arrays of length N and M, where N>= M, is O(N^2), Ω(M^2). Keep in mind calling DisMatrixFillingDiagonally() means calling MatrixHelper(). The helper function,  MatrixHelper(), acts as a nested for loop, because it is called within DisMatrixFillingDiagonally()'s for loops.

int TotalTraversals = DY.length+DX.length-1;
int TraversalsStartingonYAxis = TotalTraversals-DX.length+1;
int TraversalsStartingonXAxis = TotalTraversals-DY.length;

A = number of diagonal traversals on X axis
B = number of diagonal traversal on the Y axis
Q = maximum possible length of a diagonal, shorter side length value between A and B.
T = Q*(.x) = Threshold minimum for traversal length, where  0 =< x < 1.

Given two for loops in DisMatrixFillingDiagonally(), the first one is called A times(minus T times) , the second one is called B times(minus T times). Minus T because after the traversals reach a given length T, there are T remaining traversals to reach a corner. Inside MatrixHelper() the matrix is traversed till either (Xpos >= (SidelengthX-1) || Ypos >= (SidelengthY-1). So the maximum amount of spaces you can visit is Q, because no matter the position or location, if the coordinate(X or Y) associated with shorter side length exceeds, Q, we have left the matrix coordinates.

(A-T)*Q represents the first for loop in DisMatrixFillingDiagonally() while (B-T)*Q represents the second for loop in DisMatrixFillingDiagonally().

Assuming A =< B → Q = A.

////(is the run time worse for a square or a rectangle? //does it change anything)

In the algorithms worst case running time(Upper bound), T = 0,
DisMatrixFillingDiagonally()  =  O(A^2+B*A) for a rectangle, O(2(A^2)) for a square.

In the algorithms best case running time(Lower Bound),
DisMatrixFillingDiagonally()  =  Ω( (A-T)*A + (B-T)*A ) for a rectangle, Ω( 2(A-T)*A ) for a square.

**Time Complexity Analysis of Other Methods:**

getIndexOfLargest(Array)) - It iterates through the array and stores/returns the index with the largest value, O(n), n is the length of the given array
Could designate the first/last index in the arrays of interest to hold the index of the largest value in the array. Have it check and update as the values are being placed in. But that assumes that you only care about the single largest possible value. Where as we probably care about finding a variety of matching value sets, above a certain point.

New method idea-
getIndexsOfValuesGreaterThan(Array, x)
{
Iterate through the list
Creates linkedlist which stores the indexes with value greater than X
Return LL
}

Can't improve timecomplexity//Could nlog(n) to merge sort the array return final value in the array would also require a new way to organize the index as they would get switched around. Like a new column or something. nlog(n) +

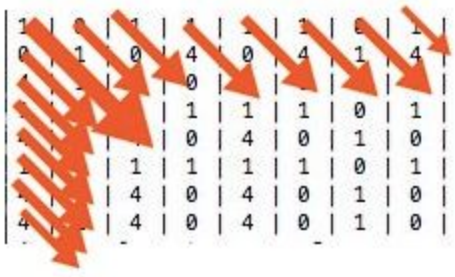
PercentMatch(int[]Array, int[][]Matrix); - O(n) it calls getIndexOfLargest
LZScoordinates();-calls getIndexOfLargest
GridEdgeCoordinates(int[] array, int[][]Matrix);-getIndexOfLargest

---

Here is a **Simple example**:

```
int[] ArrayX = new int[]{3,2,3,1,3,1,2,1};
int[] ArrayY = new int[]{2,3,1,2,1,2,1,1};
```

A Distance Matrix based on these values is created diagonally



Distance Matrix:

*Remember the top left of the grid starts at (0,0)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 |
| 0 | 1 | 0 | 4 | 0 | 4 | 1 | 4 |
| 4 | 1 | 4 | 0 | 4 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 |
| 4 | 1 | 4 | 0 | 4 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 |
| 4 | 1 | 4 | 0 | 4 | 0 | 1 | 0 |
| 4 | 1 | 4 | 0 | 4 | 0 | 1 | 0 |

**Results Section:** (built into the main method using a few helper functions to provide, immediate feedback to whoever runs a comparison)

```
1. Array index of most zeros: 8
2. Array index of longest string of zeros: 10
3. Starting coordinates of the traversal(on the grid edge) with the longest continuous string of zeros: (x,y)_(3,0) value: 4
4. Starting coordinates of the traversal(on the grid edge) with the most total zeros: (x,y)_(1,0) value: 6
5. Exact X & Y grid coordinates of longest String of Zeros (x,y)_(4,1)
6. Percentage Match relative to perfect match_(most total zeros)_75.0%
7. Percentage Match relative to perfect match_(longest continous string of zeros)_50.0%
```

Visualized Data Structures (x,y,z):

ZeroCounter Array

| |
|---|
| 1 |
| 3 |
| 2 |
| 1 |
| 2 |
| 0 |
| 0 |
| 0 |
| 6 |
| 0 |
| 4 |
| 0 |
| 1 |
| 1 |
| 0 |

LongestZeroString Array

| |
|---|
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 0 |
| 0 |
| 0 |
| 3 |
| 0 |
| 4 |
| 0 |
| 1 |
| 1 |
| 0 |

LZScoordinates Points

| x | | y |
|---|---|---|
| 7 | | 7 |
| 1 | | 0 |
| 3 | | 1 |
| 6 | | 3 |
| 5 | | 1 |
| 0 | | 0 |
| 0 | | 0 |
| 0 | | 0 |
| 0 | | 1 |
| 0 | | 0 |
| 1 | | 4 |
| 0 | | 0 |
| 2 | | 7 |
| 0 | | 6 |
| 0 | | 0 |

**Example of how to read the Data Structures:**

Each index in these arrays and matrix are assigned to the same traversal. So for index 0 (the traversal is from top left to bottom right) one zero was found, as seen in the ZeroCounter Array, thus the longest string of zeros found was one long, as seen in the LongestZeroString Array, and the coordinates of the longest string of zeros was at (7,7) the bottom right hand corner on the distance matrix, as seen in LZScoordinates Points. This checks out given that the bottom right corner value is 0.

**Analysis of example results:**

So based on line 3. We determine the traversal with the longest string of continuous zeros starts at the edge position (3,0) with length 4. Based on line 5. the position where the string of zeros truly starts is coordinates (4,1). This means the longest string of zeros starts at ArrayX index 4 and ArrayY index 1, the

2nd comparison in the 11th traversal. According to line 7. The consecutive zero string length was 4 long which represents 50% of what a theoretical perfect match would be, 8 zeros in a row starting at (0,0) ending at (7,7). According to 4. However that are more matches total in the traversal that starts at (1,0), at 6 matches, which according to line 6. Represents a 75% match relative to the perfect match.

**Limitations and next steps:**

In sum, the diagonal traversal method of creating a distance matrix is able to harvest a lot of information during the process of creating an distance matrix for even or uneven data set lengths. This is inherently useful to find out information in the data sets but likely is not the most efficient method of doing so. The Distance Matrix profile is exact, has no risk of false negatives even with missing data. So it could be a potentially slower method to backup check for matching data points, after using a faster algorithm with a risk of inaccuracy. Although the algorithm already makes a assumption to improve its time complexity, it remains inefficient. Given the immense amount of data tidepool is handling the computational complexity of the algorithm's a major flaw. I don't know if any sort of divide and conquer method could be used to improve the creation of the distance matrices because ultimately every value has to be checked against ever value.

One possible method to precheck different traversals is a jumping mechanism. So for example if a diagonal is known to be 100 units in length, and we are only concerning ourselves with consecutive zeros strings of 20 or more. You can jump from index 0 to 20 to 40 to 60 to 80 to 100, and look for zeros. If you don't find any zeros you are move to the next traversal. If you do find zeros then you run the original method on either the whole diagonal or a subset of the diagonal to check for continuous zero strings.

<u>Extra Ideas and exploration:</u>

((Ed also mentioned gaps in the data that have to be accounted for, which again I believe would be best sorted with a unique value that could never be mistaken for a match. (Perhaps if the unique value is found) it doesn't break/add to an ongoing string of matches. )))

The UCR Matrix Profile page states that a Matrix Profile can have constant time complexity. ??https://www.cs.ucr.edu/~eamonn/MatrixProfile.html))

Using the java files DistanceMatrixUpdate2.java and SqaureGridUpdate2.java I did a mock speed test. Given two sets of 11,044 data points to compare, and create a Distance Matrix for, the code takes roughly 4.90 seconds, (I can't call it any closer than, 1/10th of a second, given it was simply me along using my phone timer) on my 2011 MacBook Air to generate only the results section. At roughly 288 data points a day, that is 38.347222 days in 4.90secs. That equates to 46.63secs per one patient's yearly data (assuming daily data donations). It would be interesting to see this done on different computers, because this test far exceeded by speed expectations. Some testruns got as low as 4.4sec. This trial didn't require the program to print anything information with each match, loop break, or to print the matrix itself. However that information is stored in the system.

**Bugs*(9/12/18)-** The coordinates for the longest zero string, found in the Matrix LZScoordinates Points, are sometimes one coordinate position too high or low in their report. The errors tend to occur with strings of zeros that hit the bottom hand corner or begin at the start of the grid. (WILL INVESTIGATE)

Fixed

There are two situations when the coordinates are being logged into the 2 column points matrix. When the final zero is the last position in the traversal, or when it isn't. You have to add +1 to the coordinates no matter what. This is because there is the possibility (X/Ypos-activeZeroCounter) < 0 which is impossible. The strings of zero that end at the final position need +1 added right there mid calculation because the coordinates are being calculated during the same loop as the final zero is found, but in the case above the coordinates are being calculated in the loop after the final zero is found, thus the +1 happened as a result of the loop.