University of Connecticut

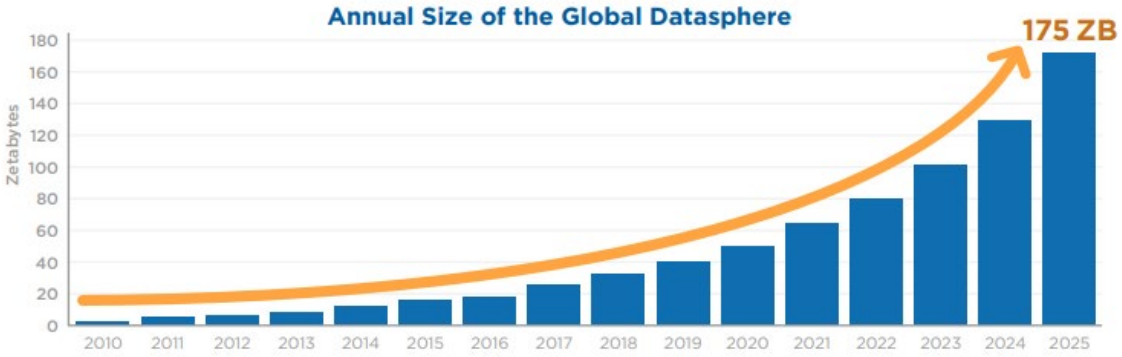
Analysis of Closest Pair Problem

CSE 5717: Big Data Analytics
Professor Sanguthevar Rajasekaran

Ramon Chavarro; Kirk Heilman
12-14-2020

Problem Statement

We live in an era of big data, where the amount of data being accumulated is growing at an increasing rate. In 2018, David Reinsel, John Grantz, and John Rydning from IDC published a paper emphasizing the rapid expansion of data collection as the world becomes more and more digitalized. The authors predict that by 2025 the Global Datasphere is projected to grow to 175 zettabytes. To put things into perspective, one zettabyte is equivalent to one trillion gigabytes.



As data collection and storage capacities increase at this rapid pace, we need ways to process and analyze data in an efficient manner. In this paper, an efficient algorithm is defined in terms of time complexity and amount of work done. We can analyze data and solve problems using algorithms. However, not all algorithms are created equal. Depending on how the algorithm is designed, it can greatly impact the time complexity to solve a given problem, ceteris paribus. It is vital to develop algorithms that are efficient so that problems can be solved in a reasonable amount of time; especially when given a large dataset.

To illustrate why algorithm design is important, this paper showcases several algorithms designed to solve a Closest Pair-of-Points problem. The data utilized to demonstrate algorithm performance is a roster of Starbucks coffeeshop locations in operation in February 2017. The dataset has 17,774 unique locations distinguished by a Store Number (primary key), along with Latitude and Longitude (Meller, 2017). To determine which two Starbucks locations are closest to each other, a two-dimensional closest pair problem can be solved using the Latitude and Longitude values provided. Multiple algorithms will be tested to show how performance will vary.

Algorithm #1: Divide & Conquer

The Divide & Conquer algorithm is the standard to solve the Closest Pair-of-Points problem. The core logic of algorithm is in the title; it divides the dataset into two parts to save time and ultimately make the algorithm more efficient. Making efficient algorithms is an essential part of data science, as the goal is to minimize the time of analysis so more analysis can be done. To start, sort the longitude points (x-axis values). The algorithm will need to utilize the simple distance formula to find the distance from each store to another. This simple formula is as follows:

$$Distance\ between\ points\ A\ and\ B = \sqrt{(A_x - B_x)^2 + \left(A_y - B_y\right)^2}$$

Let $A(x, y)$ be the first point with the x-axis value x and the y-axis value y. Let $B(x, y)$ be the second point that will be compared to the first point. This formula is derived from the Pythagorean theorem since the distance from one point to another can simply be looked at as the hypotenuse to a triangle whose sides are the difference in the horizontal and vertical directions. Here is the step-by-step breakdown of the algorithm.

Step 1: Divide the dataset into two equal parts

This is a straightforward step. The dataset contains 17,774 unique Starbucks locations. These data points are divided into equal parts of 8,887 datapoints in each half. The first part will be called as P1 and contains points whose longitude is less than or equal to the 8,887th smallest point. The second half, identified as P2, will contain any points whose longitude is greater than or equal to the 8,888th point.

Time Complexity

Since this part is a simple split, it can be done in constant $O(1)$ time.

Step 2: Find the shortest distance between any two points in each part

This is the first major step of the algorithm. Basically, each point is compared to all other points in its part to find the two points in each part with the shortest distance. The shortest distance between points in P1 is labelled SD1 and the shortest distance in P2 is labeled SD2. Finding the shortest distances gives a minimum bound to work when all the points are analyzed against each other. This narrows the view on points to compare since if any two points have a

longitude difference greater than either SD1 or SD2, then the pair of points cannot be the shortest distance.

Time Complexity

Let the total time complexity of the algorithm be identified as $T(n)$. When the dataset is divided into two parts, the algorithm runs as if it is running on half the data. Therefore, the time it takes is $T\left(\frac{n}{2}\right)$. However, since both parts are being run, the run time is $2 * T\left(\frac{n}{2}\right)$

Step 3: Determine the upper and lower bounds of points to compare.

First, compare SD1 and SD2 and take the lower value of the two. This value will be called SD. Compare the 8,887th smallest point (identify as x_m) to the 8,886th smallest point (identify as x_{m-1}). If the difference between the longitudinal values is less than SD, then move to the next smallest point, (x_{m-2}). Continue this until the point whose longitudinal distance from x_m is greater than SD. The point in the array before this point will the lower bound, which will be called x_l. Now do the same put in the opposite direction to find the upper bound which will be called x_u.

Time Complexity

To find the time complexity, the maximum number of comparisons made must be considered. Assume theoretically that all the data points must be analyzed to find the new upper and lower bounds. This would take n time. Therefore, the run time for this step is $O(n)$.

Step 4: Find the shortest distance in the new array

Now that bounds have been calculated, a new smaller dataset can be analyzed to find the shortest distance. Since all the points in this dataset have an x-axis value that is lower than previously found distance, only the points in this array need to be tested. This will guarantee that the closest points in the whole dataset will be found.

Time Complexity

Like the previous step, the theoretical max number of comparisons needs to be considered. Therefore, the run-time for this step is also $O(n)$.

Divide and Conquer Total Time Complexity

In total, the algorithm takes four steps with each step taking

$$O(1), 2 * T\left(\frac{n}{2}\right), O(n), and\ O(n)$$

Therefore, the total runtime can be written as

$$T(n) = O(1) + 2 * T\left(\frac{n}{2}\right) + O(n) + O(n)$$

Since $O(1)$ and $O(n)$ are both smaller values than $O(n^2)$, these values can be removed from the equation. The equation is now simplified to

$$T(n) = 2 * T\left(\frac{n}{2}\right) + 2 * O(n)$$

Using a key principle known as the "Master Theorem", this theorem is used to solve recurrence relations and can be used to further simplify the time complexity. The Master Theorem states the following:

$$T(n) = a * T(n/2) + f(n)) = O(f(n) * log(n))$$

By applying the Master Theorem in the above Time Complexity equation, the total time complexity simplifies to

$$O(n * \log(n))$$

Algorithm #2: Brute Force

A direct way to solve the Closest Pair-of-Points problem is by calculating every Euclidean distance between every point and taking the minimum value. This approach solves the problem using "brute force" rather than the finesse of Divide and Conquer. The same formula between two points applies here:

$$Distance\ between\ points\ A\ and\ B = \sqrt{(A_x - B_x)^2 + \left(A_y - B_y\right)^2}$$

Step 1: Data Preparation and cleanup

The dataset contains 17,774 unique Starbucks locations. To calculate the Euclidean Distance between two unique locations, some data manipulation must be done prior to calculating the distances. First, the Longitude values (x-coordinates) are sorted in ascending order. Then, the points A and B must be distinct coordinates, otherwise the minimum distance will be calculated as 0. Therefore, A and B must be distinct store locations.

Time Complexity

The sorting method used by the Python compiler is one named TimSort, which is a hybrid between a merge sort and insertion sort (Auger et al.,2018). The worst-case time complexity for sorting this data is $O(n * \log(n))$.

Step 2: Calculate the Euclidean Distance between every point

Once the ordered pairs have been determined, the next step is to apply the distance formula to the ordered pairs to determine all the distances. Every store location is compared to another.

Time Complexity

Since there are 17,774 datapoints, there are 17,774 comparisons to be made which is simply a runtime of $O(n^2 - n)$. Removing the n data points where A = B is negligible when it comes to calculating time complexity and therefore leaves the time complexity as $O(n^2)$.

Step 3: Find the minimum distance

Once all the distances are calculated, the next step is to find the minimum. This is a straightforward step which is done in $O(n)$ time.

Time Complexity

Finding the minimum of an array takes $O(n)$ time.

Brute Force Total Time Complexity

$$T(n) = O(n * \log(n)) + O(n^2) + O(n)$$

Since Big Oh only cares about the dominant factor in the asymptotic behavior of the function, the time complexity simplifies to:

$$T(n) = O(n^2)$$

Algorithm #2a: Brute Force Alternative

When calculating the Euclidean Distance between A and B, a good to note is that the distance between $(A, B)\ and\ (B, A)$ is obviously equal and therefore redundant to calculate the distance of both points. Therefore, the time complexity of step 2 can be cut in half.

$$T(n) = O\left(\frac{n(n-1)}{2}\right)$$

However, since Big Oh only cares about the dominant factor in the asymptotic behavior of the function, the time complexity simplifies to:

$$T(n) = O(n^2)$$

Relative Comparison of Algorithms & Detailed Code Analysis

The following table shows a relative comparison of the different algorithms tested, their Big Oh values and the time the algorithms took when the sample dataset was tested. The Divide and Conquer Algorithm was tested using both Python and VBA. The Brute Force and Brute Force Alternative were tested in Python. In Python the three algorithms used similar functions for calculating the Euclidean Distance. The main differences between the three was the organization of the data structure…in other words, how the data was organized and processed. Here are the performance results for the algorithms built and tested using the Starbucks data.

Algorithm	Work Done	Actual Time (tested in minutes)
Divide and Conquer - Python	$O(n * \log(n))$	2.45 minutes
Brute Force	$O(n^2 - n)$	4.27 minutes
Brute Force Alternative	$O\left(\frac{n^2 - n}{2}\right)$	9.15 minutes
Divide and Conquer - VBA	$O(n * \log(n))$	9 minutes

For the Python analysis, the following steps were identical for all three algorithms:

Step 1: Read the data in from an Excel file

Read data from file

```
allStarbucksLocations = pd.read_excel(open('./data/Data.xlsx','rb'),sheet_name='Sheet1')
```

The data is read in from an Excel file and placed into a Pandas DataFrame named "allStarbucksLocations".

Step 2: Prepare data for analysis

Data Cleanup

Use to get list of columns from data frame

```
list(allStarbucksLocations.columns)
```

```
allStarbucksLocations = allStarbucksLocations[allStarbucksLocations["Brand"] == "Starbucks"]
coordinates = allStarbucksLocations[["Store Number","Longitude","Latitude"]]
coordinates.shape
```

```
(17774, 3)
```

Next, the data is filtered to contain only Starbucks locations. Furthermore, a new DataFrame named "coordinates" is created, keeping only the columns deemed essential for the analysis: Store Number, Longitude and Latitude. The shape of the DataFrame verifies that there are 17,774 stores being analyzed and that only three columns are retained.

Step 3: Sort the data

Sort DataFrame on Longitude

```
sortedCoordinates = coordinates.sort_values(by=['Longitude'])
```

Next, the data is sorted on the longitude value. Longitude is a measurement of location east or west of the prime meridian. The prime meridian runs vertically from the North Pole to the South Pole and through Greenwich London. This is the equivalent of a line y = x where x = 0. Every longitude value that moves away from the prime meridian east or west is a point moving along the x-axis. Hence why the longitude value is can be treated like an x-axis. To verify this in the data, below is a screenshot of some longitude values close to 0. There is in fact, a Starbucks in Greenwich London!

Store_Number	Store_Name	Street_Address	City	State_Province	Country	Longitude	Latitude
12891-137726	West Wickham - High Street	11-73, HIGH STREET, WEST WICKHAM - HIGH ST...	LONDON	ENG	GB	-0.02	51.38
12310-237068	East Grinstead Sainsburys	Brooklands Way, Starbucks c/o Sainsbury's	East Grinstead	ENG	GB	-0.02	51.13
12081-11216	Canary Wharf - Canada Square	Unit R:P:490, One Canada Square	London	ENG	GB	-0.02	51.5
12625-224605	Canary Wharf - Churchill Place	Churchill Place	London	ENG	GB	-0.01	51.5
16594-165226	Aramark Canary Wharf	25 Bank Street, Canary Wharf, 7th Floor	London	ENG	GB	-0.01	51.51
12157-22022	Greenwich - Cutty Sark	54-56 Greenwich Church St	London	ENG	GB	-0.01	51.48
16025-157744	Stratford - Westfield Lower Ground	Unit SU0038, Westfield Stratford	London	ENG	GB	-0.01	51.54
12737-140844	Stratford - The Broadway	46 The Broadway	Stratford	ENG	GB	0	51.54
12671-104117	Greenwich - The O2	The O2, Unit 101	London	ENG	GB	0	51.5
24321-237126	London - Greenwich Shopping Park	Unit D, 1C Greenwich Shopping Park, Bugsbys Way, ...	London	ENG	GB	0.02	51.49
12088-11316	Bromley - Market Square	25 Market Square	Bromley	ENG	GB	0.02	51.41
12904-137732	Bromley - Glades Shopping Centre	Bromley - Glades Shopping Centre, unit 220	Bromley	ENG	GB	0.02	51.4
25409-240951	Fox Excel	Warehouse K, ExCel Centre, 2 Western Gateway	London	ENG	GB	0.03	51.51
47774-259781	Wanstead High Street	56A - 58 High Street	Wanstead	ENG	GB	0.03	51.58

Step 4: Convert data into a tuple

Convert dataframe to tuple

```
coordinatesTuple = [tuple(x) for x in sortedCoordinates.values]
```

Once the data is sorted, the data is converted into a list of tuples. Tuples are a collection of objects which are ordered and immutable. Tuples cannot be modified and are a safe way to perform analysis outside of a structured dataframe.

Step 5a: Divide & Conquer analysis

Once the data has been prepared for the analysis, the three algorithms solve the problem differently. Starting with Divide and Conquer, the list of tuples is split in two.

Split into two sets

```
P1 = round(len(coordinatesTuple)/2)
coordinatesTuple_Set1 = coordinatesTuple[:P1]
```

```
P2 = len(coordinatesTuple)
coordinatesTuple_Set2 = coordinatesTuple[P1:P2]
```

The data is divided into equal haves and each half is analyzed independently from one another. Figure 1 shows the functions that are defined to determine the pairs of points being compared, the function that calculates the Euclidean Distance, and then combines these two functions in a function that determines which pair of Starbucks stores are the closest.

Figure 1 – Python Divide & Conquer

Define a function that returns a set of store pairs

```
def findUniqueStores(pairs):
    listOfPairs = []

    for c1 in range(len(pairs)):
        for c2 in range(len(pairs)):
            if ( c1 == c2 ):
                pass
            else:
                orderedPair = ( c1 , c2 )
                listOfPairs.append(orderedPair)
    return listOfPairs
```

Define function: euclidean_distance

```
def euclidean_distance(value1, value2):
    distLongitude = (( value1[1] - value2[1] )**2)
    distLatitude = (( value1[2] - value2[2] )**2)

    return ( value1[0] , value2[0] , distLongitude , distLatitude )
```

Combine the two functions to get euclidean distance of unique stores

```
def findMinDistance(data):
    global minDistance
    global newTuple
    newTuple = []
    iteration = 0
    for s in findUniqueStores(data):

        d = euclidean_distance( data[s[0]] , data[s[1]] )

        if iteration == 0:
            minDist = math.sqrt( d[2] + d[3] )
            minDistance = d

            newTuple.append((iteration,d[0],d[1],d[2],d[3],minDist))

        else:
            dist = math.sqrt( d[2] + d[3] )

            newTuple.append((iteration,d[0],d[1],d[2],d[3],minDist))

            if ( dist <= minDist ):
                minDist = dist
                minDistance = d
            else:
                pass

        iteration += 1

    print("Number of Inputs: {0}".format(len(data)))
    print("Number of Comparisons: {0}".format(iteration))
    print("Closest Starbucks Stores: Store Numbers {0} and {1}".format(minDistance[0],minDistance[1]))
    print("Longitude Distance: {:.5f}, Latitude Distance: {:.5f}".format(minDistance[2],minDistance[3]))
    return minDistance
```

Both sets of tuples are run through the function findMinDistance. This function cycles through each set of tuples and calculates the minimum Euclidean Distance between all the points in each set. Like in the analysis, the first 8,887 stores are analyzed in Set 1 and the other half of the

stores are analyzed in Set 2. Each set takes 1.24 minutes to run. Since these sets are run sequentially, the total run time is 2.49 minutes. Each Set of data performs a total of 78,969,882-point comparisons (shown in Figure 2) .

Figure 2 – Python Divide & Conquer

Set 1 Data

```
start_time = time.time()
minDistance_Set1 = findMinDistance(coordinatesTuple_Set1)
distance_Set1 = math.sqrt( minDistance_Set1[2] + minDistance_Set1[3] )
newTuple_Set1 = newTuple
end_time = time.time()
timeToComputeSet1 = datetime.timedelta(seconds=(end_time - start_time))
print("Time taken:{0}".format(timeToComputeSet1))
```

```
Number of Inputs: 8887
Number of Comparisons: 78969882
Closest Starbucks Stores: Store Numbers 8485-57681 and 8428-28454
Longitude Distance: 0.00010, Latitude Distance: 0.00000
Time taken:0:01:25.064252
```

Set 2 Data

```
start_time = time.time()
minDistance_Set2 = findMinDistance(coordinatesTuple_Set2)
distance_Set2 = math.sqrt( minDistance_Set2[2] + minDistance_Set2[3] )
newTuple_Set2 = newTuple
end_time = time.time()
timeToComputeSet2 = datetime.timedelta(seconds=(end_time - start_time))
print("Time taken:{0}".format(timeToComputeSet2))
```

```
Number of Inputs: 8887
Number of Comparisons: 78969882
Closest Starbucks Stores: Store Numbers 25282-240406 and 25278-240393
Longitude Distance: 0.00010, Latitude Distance: 0.00000
Time taken:0:01:24.473777
```

Each set has determined a pair of Starbucks stores that have the smallest Euclidean Distance. Now that each set has a minimum distance, now the results must be compared to each other.

Figure 3– Python Divide & Conquer

Compare the two sets of data

Set the smallest euclidean distance as the mininum distance

```
start_time = time.time()
if distance_Set1 < distance_Set2:
    dataSummary = newTuple_Set1
    mindis = distance_Set1
    closestStores = minDistance_Set1

    print("Store Numbers {0} and {1}".format(minDistance_Set1[0],minDistance_Set1[1]))
    print("Longitude Distance: {:.5f}".format(minDistance_Set1[2]))
    print("Latitude Distance: {:.5f}".format(minDistance_Set1[3]))

else:
    dataSummary = newTuple_Set2
    mindis = distance_Set2
    closestStores = minDistance_Set2

    print("Store Numbers {0} and {1}".format(minDistance_Set2[0],minDistance_Set2[1]))
    print("Longitude Distance:{:.5f}".format(minDistance_Set2[2]))
    print("Latitude Distance: {:.5f}".format(minDistance_Set2[3]))

end_time = time.time()
timeToCompareDatasets = datetime.timedelta(seconds=(end_time - start_time))
cumulativeTime = timeToComputeSet1 + timeToComputeSet2 + timeToCompareDatasets
print("Total Time:{0}".format(cumulativeTime))
```

```
Store Numbers 25282-240406 and 25278-240393
Longitude Distance:0.00010
Latitude Distance: 0.00000
Total Time:0:02:44.264298
```

Figure 3 shows how the minimum distance from each set is compared to one another and how the minimum distance is set. The time to compute Divide and Conquer up to this point is 2.49 minutes.

The next step defines bounds around the midpoint of the dataset to determine if points around the median can potentially be closer and therefore have a smaller Euclidean Distance.

Determine the upper and lower bounds

```
start_time = time.time()
LongitudeMid = coordinatesTuple[P1][1]
print("The minimum distance recorded is: {0}".format(mindis))
print("The longitudnal midpoint is: {:.7f}".format(LongitudeMid))
```

```
The minimum distance recorded is: 0.009999999999990905
The longitudnal midpoint is: -82.4900000
```

```
for i in range(P1):
    j = P1 - i
    if LongitudeMid - coordinatesTuple[j][1] < mindis:
        pass
    else:
        Lowerbound = j
        return_i = P1 - 1
        break
```

```
for i in range(P1,P2):
    if coordinatesTuple[i][1] - LongitudeMid < mindis:
        pass
    else:
        Upperbound = i
        break
```

```
end_time = time.time()
timeToDetermineUpperLowerBound = datetime.timedelta(seconds=(end_time - start_time))
cumulativeTime = timeToComputeSet1 + timeToComputeSet2 + timeToCompareDatasets + timeToDetermineUpperLowerBound
print("Lowerbound value is: {0}".format(Lowerbound))
print("Upperbound value is: {0}".format(Upperbound))
print("Total Time:{0}".format(cumulativeTime))
```

```
Lowerbound value is: 8886
Upperbound value is: 8892
Total Time:0:02:44.311335
```

These bounds indicate that the Euclidean distance from the median to these points have a minimum distance greater than the recorded minimum distance. Figure 4 shows the closest pair of points found along with some simple logic to determine where the smallest pair of points are relative to the collection of points. This pair exists close to the end of the dataset, index numbers 17,744 and 17,745. Later, it was uncovered by the VBA algorithm that there were some very tight tolerances in calculating the min distance. The bounds here did not work as expected and some additional code was introduced to address this.

Figure 4 – Python Divide & Conquer

```
closestStores
```

```
('25282-240406', '25278-240393', 9.99999999998181e-05, 0.0)
```

```
index = 0
for t in coordinatesTuple:
    index += 1
    if t[0] == closestStores[0]:
        print(index, t)
```

```
17745 ('25282-240406', 153.03, -27.47)
```

```
index = 0
for t in coordinatesTuple:
    index += 1
    if t[0] == closestStores[1]:
        print(index, t)
```

```
17744 ('25278-240393', 153.02, -27.47)
```

Figure 5 – Python Divide & Conquer

Find shortest distance in new array

```
start_time = time.time()
for i in range(Lowerbound,Upperbound+1):
    Lng1 = coordinatesTuple[i][1]
    Lat1 = coordinatesTuple[i][2]

    for j in range(i + 1, Upperbound + 1):
        Lng2 = coordinatesTuple[j][1]
        Lat2 = coordinatesTuple[j][2]

        dis = math.sqrt(((Lng1 - Lng2)**2) + ((Lat1 - Lat2)**2))

        if dis <= mindis:
            mindis = dis
            Store1 = coordinatesTuple[i]
            Store2 = coordinatesTuple[j]
        else:
            pass
end_time = time.time()
timeToDetermineShortestDistance = datetime.timedelta(seconds=(end_time - start_time))
cumulativeTime = timeToComputeSet1 + \
                 timeToComputeSet2 + \
                 timeToCompareDatasets +\
                 timeToDetermineUpperLowerBound + \
                 timeToDetermineShortestDistance

try: Store1
except NameError: Store1 = closestStores[0]

try: Store2
except NameError: Store2 = closestStores[1]

print("Total Time:{0}".format(cumulativeTime))
print("The Closest Starbucks stores are {0} and {1}".format(Store1,Store2))
```

```
Total Time:0:02:44.311335
The Closest Starbucks stores are 25282-240406 and 25278-240393
```

Figure 5 shows the calculation of determining the minimum Euclidean Distance in the range defined by the upper and lower bound. In this case, there were no Starbucks stores closer to the ones found originally in Set 2, and therefore were found to be the closest pair of points. ■

Step 5b: Brute Force

The abovementioned steps 1-4 are the same. Figure 6 shows the entire brute force algorithm. The functions “findUniqueStores” and “euclidean_distance” are the same as in Step 5a. The function “findMinDistance” is similar but is more straightforward in this algorithm. There is not as much tracking needed for the brute force algorithm. This algorithm performs 315,897,302 comparisons, which is equal to:

$$Number\ of\ Comparisons: n * (n-1) = 17{,}774 * (17{,}774 - 1) = 315{,}897{,}302$$

Based on the number of comparisons done, it was surprising to see this algorithm run in 4.27 minutes.

Figure 6 – Brute Force Algorithm

Define a function that returns a set of store pairs

```
def findUniqueStores(pairs):
    listOfPairs = []

    for c1 in range(len(pairs)):
        for c2 in range(len(pairs)):
            if ( c1 == c2 ):
                pass
            else:
                orderedPair = ( c1 , c2 )
                listOfPairs.append(orderedPair)
    return listOfPairs
```

Define function: euclidean_distance

```
def euclidean_distance(value1, value2):
    distLongitude = (( value1[1] - value2[1] )**2)
    distLatitude = (( value1[2] - value2[2] )**2)

    return ( value1[0] , value2[0] , distLongitude , distLatitude )
```

Combine the two functions to get euclidean distance of unique stores

```
def findMinDistance(data):
    iteration = 0
    for s in findUniqueStores(data):

        d = euclidean_distance( data[s[0]] , data[s[1]] )

        if iteration == 0:
            minDist = math.sqrt( d[2] + d[3] )
            minDistance = d
        else:
            dist = math.sqrt( d[2] + d[3] )
            if ( dist <= minDist ):
                minDist = dist
                minDistance = d
            else:
                pass
        iteration += 1

    print("Number of Inputs: {0}".format(len(data)))
    print("Number of Comparisons: {0}".format(iteration))
    print("Closest Starbucks Stores: Store Numbers {0} and {1}".format(minDistance[0],minDistance[1]))
    print("Longitude Distance: {:.5f}, Latitude Distance: {:.5f}".format(minDistance[2],minDistance[3]))
```

Result

```
start_time = time.time()
findMinDistance(coordinatesTuple)
end_time = time.time()
print("Time taken:{0}".format(datetime.timedelta(seconds=(end_time - start_time))))
```

```
Number of Inputs: 17774
Number of Comparisons: 315897302
Closest Starbucks Stores: Store Numbers 25282-240406 and 25278-240393
Longitude Distance: 0.00010, Latitude Distance: 0.00000
Time taken:0:04:27.894947
```

■

Step 5c: Brute Force Alternative

The most surprising result from the Python tests was the Alternative Brute Force Algorithm. Despite having the same Big-Oh, it was hypothesized that this algorithm would run in less time than the brute force algorithm because it has less comparisons. However, this was the longest running algorithm (see figure 7). The key differences are highlighted in yellow.

$$Number\ of\ Comparisons: n * \left(\frac{n-1}{2}\right) = 17{,}774 * \left(\frac{17{,}774-1}{2}\right) = 157{,}948{,}651$$

Figure 7 – Brute Force Alternative

Define a function that returns a set of store pairs

```
def findUniqueStores(pairs):
    listOfPairs = []

    for c1 in range(len(pairs)):
        for c2 in range(len(pairs)):
            if ( c1 == c2 ):
                pass
            else:
                orderedPair = ( c1 , c2 )
                listOfPairs.append(orderedPair)
    return set([tuple(sorted(i)) for i in listOfPairs])
```

Define function: euclidean_distance

```
def euclidean_distance(value1, value2):
    distLongitude = (( value1[1] - value2[1] )**2)
    distLatitude = (( value1[2] - value2[2] )**2)

    return ( value1[0] , value2[0] , distLongitude , distLatitude )
```

Combine the two functions to get euclidean distance of unique stores

```
def findMinDistance(data):
    iteration = 0
    for s in findUniqueStores(data):

        d = euclidean_distance( data[s[0]] , data[s[1]] )

        if iteration == 0:
            minDist = math.sqrt( d[2] + d[3] )
            minDistance = d
        else:
            dist = math.sqrt( d[2] + d[3] )
            if ( dist <= minDist ):
                minDist = dist
                minDistance = d
            else:
                pass
        iteration += 1

    print("Number of Inputs: {0}".format(len(data)))
    print("Number of Comparisons: {0}".format(iteration))
    print("Closest Starbucks Stores: Store Numbers {0} and {1}".format(minDistance[0],minDistance[1]))
    print("Longitude Distance: {:.5f}, Latitude Distance: {:.5f}".format(minDistance[2],minDistance[3]))
```

Result

```
start_time = time.time()
findMinDistance(coordinatesTuple)
end_time = time.time()
print("Time taken:{0}".format(datetime.timedelta(seconds=(end_time - start_time))))
```

```
Number of Inputs: 17774
Number of Comparisons: 157948651
Closest Starbucks Stores: Store Numbers 13213-100434 and 76362-97182
Longitude Distance: 0.00000, Latitude Distance: 0.00010
Time taken:0:09:07.265600
```

■

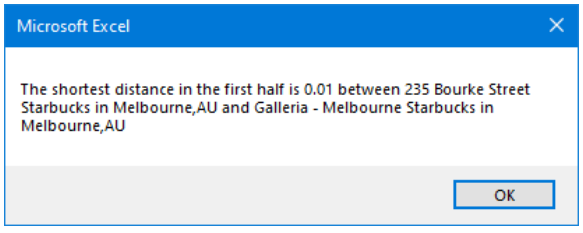
Divide and Conquer using VBA

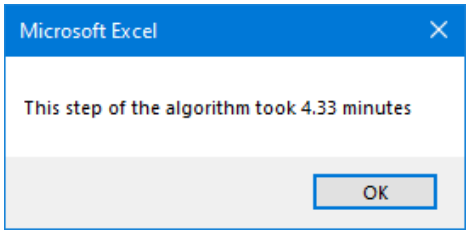
For the divide and conquer algorithm, an alternative approach was taken by coding the algorithm using VBA. Rather than reading in the data from a flat file or an Excel file, the VBA code calculates the Euclidean Distance directly from the Excel file, as can be seen in Step 1 in Figure 8

Figure 8 – Brute Force VBA

```
Sub DivConAlgo()

Sheets("Data").Select

mindis1 = 1000000

mindis2 = 1000000

Time1 = Now

Time1 = Minute(Time1) + (Second(Time1) / 60)

'Step 1

Total = Application.WorksheetFunction.CountA(Range("A:A"))
P1 = Round(Total / 2)
P2 = Total - P1

'Step 2

For i = 2 To P1 - 1

Lat11 = Cells(i, 13)
Long11 = Cells(i, 12)

For j = i + 1 To P1

Lat12 = Cells(j, 13)
Long12 = Cells(j, 12)

dis1 = ((Lat11 - Lat12) ^ 2 + (Long11 - Long12) ^ 2) ^ 0.5

If dis1 < mindis1 Then

mindis1 = dis1
Store11 = Cells(i, 3) & " " & Cells(i, 1) & " in " & Cells(i, 6) & "," & Cells(i, 8)
Store12 = Cells(j, 3) & " " & Cells(j, 1) & " in " & Cells(j, 6) & "," & Cells(j, 8)

End If

Next j

Next i

Time2 = Now

Time2 = Minute(Time2) + (Second(Time2) / 60)

Time3 = Round(Time2 - Time1, 2)

MsgBox ("The shortest distance in the first half is " & Round(mindis1, 4) & " between " & Store11 & " and " & Store12)

MsgBox ("This step of the algorithm took " & Time3 & " minutes")
```

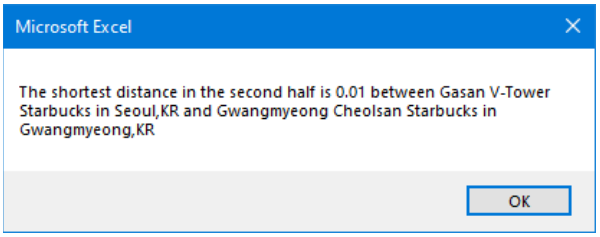
Here, the algorithm immediately begins calculating the Euclidean Distance for the first half of the elements. This segment in VBA takes about 4.3 minutes.

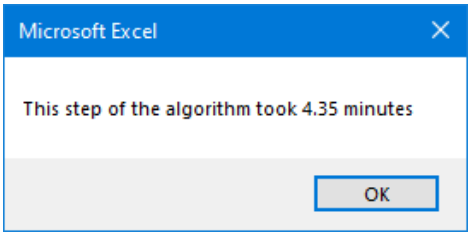




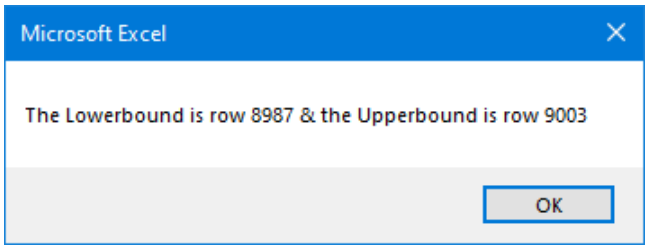
The algorithm runs sequentially for the second half and takes 4.3 minutes.

```
Time4 = Now

Time4 = Minute(Time4) + (Second(Time4) / 60)

For k = P2 To Total - 1

Lat21 = Cells(k, 13)
Long21 = Cells(k, 12)

For m = k + 1 To Total

Lat22 = Cells(m, 13)
Long22 = Cells(m, 12)

dis2 = ((Lat21 - Lat22) ^ 2 + (Long21 - Long22) ^ 2) ^ 0.5

If dis2 < mindis2 Then

mindis2 = dis2
Store21 = Cells(k, 3) & " " & Cells(k, 1) & " in " & Cells(k, 6) & "," & Cells(k, 8)
Store22 = Cells(m, 3) & " " & Cells(m, 1) & " in " & Cells(m, 6) & "," & Cells(m, 8)

End If

Next m

Next k

Time5 = Now

Time5 = Minute(Time5) + (Second(Time5) / 60)

Time6 = Round(Time5 - Time4, 2)

MsgBox ("The shortest distance in the second half is " & Round(mindis2, 4) & " between " & Store21 & " and " & Store22)

MsgBox ("This step of the algorithm took " & Time6 & " minutes")
```





Step 3 (Figure 9) calculates the lower bound and the upper bound values to determine a smaller sample of points to calculate the minimum Euclidian distance. Based on the way the algorithm was written, and because of the tight decimal value tolerances, the VBA code defines good boundaries for the upper and lower bound. The boundaries work as expected and a minimum distance is calculated. Figure 9 shows how the bounds were used in VBA.



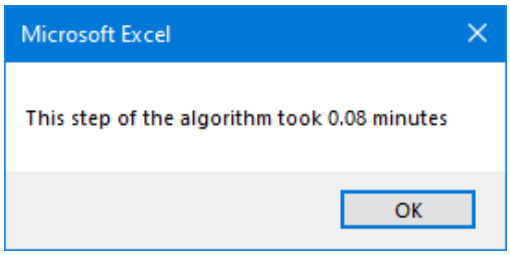
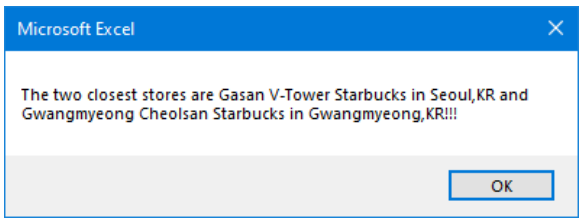


Figure 9 – Brute Force VBA

```
'Step 3
Time1 = Now
Time1 = Minute(Time1) + (Second(Time1) / 60)
If mindis1 >= mindis2 Then
mindis = mindis2
ans = 2
Else
mindis = mindis1
ans = 1
End If
LatMid = Cells(P1, 13)
For i = 1 To P1 - 1
j = P1 - i
x = LatMid - Cells(j, 13)
If LatMid - Cells(j, 13) < mindis Then
resultval = LatMid - Cells(j, 13)
Else
Lowbound = j
i = P1 - 1
End If
Next i

For i = P2 To Total
If Cells(i, 13) - LatMid < mindis Then
Else
Upperbound = i
i = Total
End If
Next i
MsgBox ("The Lowerbound is row " & Lowbound & " & the Upperbound is row " & Upperbound)
```

The final step of the algorithm uses the defined boundaries to calculate the closest pair-of-points. The most time-consuming part of this algorithm is calculating the minimum Euclidian Distance in each half of the partitioned data. The lower and upper bound calculations worked as expected and delivered a solution. ■



Conclusion

After thoroughly testing these three separate algorithms across Python and VBA, the most effective algorithm built was the Divide and Conquer Algorithm using Python. Given more time, the algorithm could be refined further to improve the upper and lower bound calculations as well as applying more business intelligence to get a better result. It was surprising to see how the Divide and Conquer algorithm performed in practice compared to the brute force algorithm and how the alternative brute force algorithm was a disappointment. One major factor that was not evaluated nor considered is how the program used handles a "sort" for example. The main reason the alternative brute force algorithm performed much worse than the other two was that the algorithm used a "set" function which checks if an ordered pair exists in a list. For example, if the ordered pair (1,2) exists in the data, then (2,1) will be ignored as these two pairs would generate the same Euclidian Distance. In practice, this cuts down the number of comparisons performed which was pointed out in the analysis, however, the time cost was far greater than any benefit from reduced calculation iterations.

Another interesting expansion of this project would be to introduce parallelism to the Divide and Conquer algorithm. Due to the complexity of applying parallelism in Python, it was decided that this could be a future enhancement to that algorithm.

References

Reinsel, D., Gantz, J. & Rydning, J. (2018). *The Digitization of the Word From Edge to Core.* [White paper]. IDC. https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf

Auger, N., Jugé, V., Nicaud, C., Pivoteau, C. (2018) *On the Worst-Case Complexity of TimSort.* [White paper]. Creative Commons License CC-BY, 26th Annual European Symposium on Algorithms. https://drops.dagstuhl.de/opus/volltexte/2018/9467/pdf/LIPIcs-ESA-2018-4.pdf

Meller, C. (2017). *Starbucks Locations Worldwide.* [Data set]. Kaggle. https://www.kaggle.com/starbucks/store-locations