CPSC 3620 Project - Closest Pairs Problem

Ricky Bueckert, Lorenzo Conrad, Mathew Richards
CPSC 3620 - Dr. Zhang
University of Lethbridge

December 4, 2019

2.) Charts are included in Appendix A

3.) Based on the two charts, argue and discuss whether your implementations match the theoretical analysis on the time complexity of the two solutions, as discussed in class.

Overall the implementations of our algorithms match the expected behaviour from the theoretical runtime of brute-force and divide and conquer. The only input where brute-force outperformed divide and conquer was at n=100. The results at this data point are not consistent with the complexity of $O(n^2)$ for brute-force and O(n*logn) for divide and conquer. This ratio did not apply to our values at this n input, but this is because a proper n_o has clearly not been reached. As the input size increases the disparity in performance between the two methods begins to grow, with divide and conquer greatly outperforming the brute-force method at larger input sizes. This is expected behaviour of a O(n*logn) algorithm compared to $O(n^2)$. For our brute-force implementation there is an improvement upon the $O(n^2)$ time, however this is not true for our divide and conquer method. In making a proper divide and conquer comparison we need to account for constants that will have an effect on the runtime, especially at smaller inputs. The asymptotic limits eventually make constants irrelevant but especially at input sizes of < 9000 our runtime is greater than the simple O(n*logn) as shown in Figure 2. If we obtained the full equation of the divide and conquer implementation it would surely be in the runtime of O(n*logn) after properly dropping any constants and finding the proper n_o value.

In analyzing our graphs the data sets in Figure 1 are quite consistent. The data greatly conform to the average, and when viewing the data points they are well under the worst case $O(n^2)$. One possible reason for this is in implementing our brute-force algorithm the nested for-loops are iterating through with integers i and j, however the values in j only ever begin at i+1, as starting the iteration at 0 would perform redundant comparisons from when i was toward the beginning of the vector. The data sets for divide and conquer are less consistent but still follow a general pattern. This is because the overall runtime is shorter and more vulnerable to interruptions and sharing resources with other processes on the system. Any aberration will have a greater effect in a small window versus a larger runtime such as in brute-force implementation. We also charted the runtime of each implementation for points <= 9000 together in Figure 3. This comparison helps to see the efficiency of divide and conquer algorithms. Finally in Figure 4

we again compare the two implementations but include the input of 100000 randomly generated points. This is only in Figure 4 because the scale makes viewing any other data points difficult. It is clear at larger inputs that the runtime of divide and conquer is far shorter than for brute-force and has serious efficiency advantages.

4) Discuss the main variables, data structures and algorithms, including the brute-force solution and divide-and-conquer solution, in your implementations.

The Bruteforce Algorithm:

In our brute-force algorithm we create a class (Point) that has two public int variables called xaxis and yaxis to store the coordinates of each point in the same data structure. In order to store all the points, that are used for the calculation after they are loaded in from the file where they are stored indefinitely, we used vector of Point* (Point pointers) called points where each pointer represents a single point that was read in. We chose to make points a global variable for ease of access in both the loadData and bruteforce functions. The loadData function first checks to make sure the ifstream is good, meaning the file exists and contains information, then loads all the data into the points vector using a simple while loop controlled by a bool variable that checks for the end of the file. Once the load is complete the bruteforce algorithm takes affect and does the calculation where finding the closest pair simply involves checking every point against all other points. However we discussed in class there was redundancy in the algorithm because calculations between the same points were done multiple times. We decided to remove the wasted calculations. Instead of running 2 nested for-loops together both running from 0 to n-1 we used two nested for-loops with i and j as indices, but j is only comparing values from i+1 to n-1, thus removing the redundancy. The initial minimum distance is set to the numeric_limits<double>::max() constant that is then updated as soon as a shorter distance is found. When timing the calculation we used std::chrono::high_resolution_clock::now() in order to take a snapshot of the time before and after the calculation, each of which was stored in two auto variable, t1 and t2. The difference was then taken between the two time differences using

std::chrono::duration_cast<std::chrono::microseconds>(t2 - t1).count() and stored in an auto duration variable before being output to the user. The closest pair points and time information are printed to the output stream in the bruteforce algorithm then used to generate the chart.

The Divide and Conquer Algorithm:

All of the above data structures were used again for the Divide and Conquer algortihm. For readability and better coding practices we split up the workload across multiple functions which will be discussed in depth below as well as some of the most important data structures we used to implement the closest pair problem using the divide and conquer method.

Data Structures:

Pairs:

We used pair<double, pair<Point*, Point*>> where the first element of the pair was a double representing the distance between the 2 points represented in the nested pair containing two Point*, which were the two points used.

Vectors:

We used vector<Point*> points to store all the points that were read in from the file. We also used vectors to store copies of the left and right sides of the points vector, throughout the recursive calls to the divide and conquer method.

Extra Features:

Sort:

We used the C++ sort function for all our sorting instead of writing our own sort. This however required us to write a custom comparator because we were passing Point* to the sort function.

Custom Functions:

loadData(std::string filename):

The loadData function first checks to make sure the ifstream is good, meaning the file exists and contains information, then loads all the data into the points vector using a simple while loop controlled by a bool variable that checks for the end of the file.

Xcomp(Point* a, Point* b):

The xcomp function is a very simple comparator that when given two Point* returns true if Point* a->xaxis < Point* b->xaxis otherwise returns false. This function is used in the C++ sort function when sorting by the x-axis.

Ycomp(Point* a, Point* b):

The ycomp function is a very simple comparator that when given two Point* returns true if Point* a->yaxis < Point* b->yaxis otherwise returns false. This function is used in the C++ sort function when sorting by the y-axis.

findD(int psize, std::vector<Point*> temppoints):

The findD function uses the bruteforce algorithm to calculate the distance between two points. The function however is only ever run between 2 or 3 points to minimize the calculation time. The function is passed in a vector of points and the size of that vector of points. It then calculates the shortest distance between the points. After the shortest distance is determined, the distance and the two points that have that distance are stored in a pair with the format pair<double, pair<Point*, Point*>>. This pair is then returned to the calling function.

copy(std::vector<Point*> p, int begin, int end):

The copy function is used to copy parts of the original vector to other vectors that will then be worked on. The copy function is passed the original vector that is being split and 2 iterators that determine the beginning and ending indices of what needs to be copied. After the vector copy is complete the smaller vector is returned to the calling function.

middlePair(double min, std::vector<Point*> temppoints, int mid):

The middlePair function is used to find the closest distance of any points across the middle “c-line” within dmin distance of the middle line where the y-distance from p1 to p2 is less than or equal to dmin. First the C++ sort is called with the custom ycomp comparator sorting the points within the range by the y-axis position. For each point in dmin distance from the midddle line a check is run for the closest pair where the other point is across the middle line and

the difference in y is within dmin distance. For each point in the middle section the check is run at most 6 times. If a pair of points is found that is closer then dmin then the dmin is changed to that distance and those points are stored as the new dmin points. After completing the calculation is complete the dmin and the corresponding points are returned as a pair with the format std::pair<int, std::pair<Point*, Point*>>.

closestPair(int numpoints, std::vector<Point*> temppoints):

The closest Pair function is the recursive function that does the majority of the work. The function is passed a vector of Point* and the size of that vector. First off a dmin pair is initialized with the format std::pair<double, std::pair<Point*, Point*>>. If the size of the vector less than or equal to 3 then it simply calls findD() on the vector of points otherwise it splits the vector of points in half by copying the left half to the vector<Point*> leftpoints and the right half to the vector<Point*> rightpoints. The function then calls closestPair recursively on leftpoints and rightpoints which return a std::pair<double, std::pair<Point*, Point*>> pair1 and pair2 respectively where double represents the distance between the two points. The distance from pair1 is then compared to dmin pair distance, if it is less than dmin then pair1 distance becomes dmin and the dmin points are set to the points in pair1. If it is not less than dmin then pair2 distance becomes dmin and the dmin points are set to the points in pair2. The closestPair function calls middlePair() with the new dmin, the midpoint of vector<Point*> before it was split, and the vector itself. The middlePair() then returns the dmin of all the middle points. If the min distance from the middle pair is smaller than that from pair1 or pair2 then the midpair becomes the min pair. If it is not, then nothing changes. The closestPair() function then returns the dmin pair with the format std::pair<double, std::pair<Point*, Point*>>.

divideconquer():

The divideconquer() function is used to time the calculation, sort the initial list of points by the x-axis and call the closest pair function. The function calls the C++ sort method on the vector<Point*> points using the custom comparator xcomp. Then calls closestPair() on the vector of points and its size which will return the closest pair of the list of points with the format std::pair<double, std::pair<Point*, Point*>>. When timing the calculation we used

std::chrono::high_resolution_clock::now() in order to take a snapshot of the time before and after the calculation, each of which was stored in two auto variable, t1 and t2. The difference was then taken between the two time differences using std::chrono::duration_cast<std::chrono::microseconds>(t2 - t1).count() and stored in an auto duration variable before being output to the user. The pair of points and their distance are returned to the user as well as the total time taken to do the calculation.

The algorithm used in the above implementation:

Sort points according to their x-coordinates.

1. Sort the list of points by x-axis.
2. Split the set of points into two equal-sized subsets by a vertical line y = c.
3. Solve the problem recursively in the left and right subsets. This yields the left-side and right-side minimum distances d1 and d2 respectively.
    i. Set dmin to the min(d1,d2)
4. Sort the middle set of points that are within dmin of the middle line (y = c) by the y-axis position
5. Find the minimum distance for the set of pairs of points in which one point lies on the left of the middle line (y = c) and the other point lies to the right, and the points being checked have a difference in y-axis of dmin or less. Each point will have at most 6 other points to check.
    i. If the mid min distance is less than then the dmin the mid min distance becomes dmin
6. Return the dmin, dmin points and the time of calculation to the user.

Creating the Points

For creating the points we used random device along with uniform int distribution to generate two integer between -10000 and 10000, which represented the x-axis and the y-axis. The numbers where then written to a page to store and be read in at a later time.

5) Discuss what you have learned from doing this project and what more you can do to further enhance your implementations.

The main take-away from this project is insight into how real-world variables will affect asymptotic complexity. Our algorithm for divide and conquer did not fit O(nlogn) as we expected it to but this is because we do not have the runtime recorded for extremely large input sizes. Continuing past the largest input would be of interest in further analysis. Another aspect learned from this process is just how slow n^2 performance can be. At the largest input there was quite a wait for the brute-force implementation. When programming the algorithm we were not previously exposed to the time keeping variables needed for this project. Using the "auto" placeholder became a useful way to implement the time variables and we continued to use this throughout the programming. It is a more advanced feature of C++11 we have not been taught about or used up to this point. In the programming portion, the reading and writing to a file was also a great review. We learned to use file operators in a safer and more efficient way. Even when the big O complexity may be n^2, we were able to have a relatively optimized version of our brute-force implementation. Using the start point of i+1 for our inner j iterator helped to improve the runtime performance optimizations such as this can greatly improve the runtime even with a "bad" runtime complexity.

6) Discuss any other specific issues that will help appreciate your efforts in this project.

See Appendix B for additional figures.

To get a better idea of how complexity is affected by real-world variables, a smaller experiment was performed on the home PC of a group member to compare results. This is a rough comparison as some of the involved variables are different. The points created using create.cpp on the home PC are not the same as when the experiment was performed in the CS lab. Only one data set was generated and compared to the average runtime of the four data sets in the CS lab. The CS lab machines run Linux centOS wheras the home computer is using Linux Mint 19.1. On both systems all files were compiled using the makefile with g++. Despite these differences the important difference is in the hardware of the home PC, having a faster processor and core components.

The comparisons are charted in Figures 1 & 2 of Appendix B. It is clear that the home PC outperforms the machines in the CS lab for the majority of data points. One interesting trend is when running divide and conquer on the smaller inputs, the CS machine outperforms until n=500, from which the home PC starts to run faster for each input afterwards. When comparing bruteforce runtime only at n=100 the CS machine outperforms the home PC by a slim margin, then at each point aftewards the home PC is faster. These results reinforce the concept that different constants will affect runtime performance in real world applications. In fact looking at the figures this is a visible trend, that the two graphs nearly differ by a constant. O(nlogn) runtime is included in the divide and conquer graph in Figure 2. This shows that while our CS lab runtime is greater than O(nlogn) the runtime of divide and conquer on the home PC is even closer to being in O(nlogn) but has a constant and other affecting factors. A greater analysis may provide more insight but that is out of the scope of this project. This smaller experiment does however give more insight into the variables that affect asymptotic complexity.