Emre Yeniay 20/04/2018

BLG 336E – Project 2

Problem: In a three-dimensional space (x, y, z), there are lots of identical balls which are hanging in free space. Suppose that balls start to enlarging at the same time and with same speed. Determine the initial distance between first pair of balls which are going to touch to each-other by using the data file you are given. All your code must be written in C++, and we must be able to compile and run on it on ITU's Linux Server (you can access it through SSH) using g++.

Solution: 3 cpp files and 2 header files in C++ programming language are created for this project. C++11 version of C++ is required to compile the files. It can be compiled in ITU SSH servers by typing "g++ -std=c++11 main.cpp ballmanager.cpp ball.cpp". There are lots of comments inside the source code, hence the details about the code can be achieved by looking inside the files.

1) Explain the master theorem with your own words briefly. What do a and b mean in divide-and-conquer approach?

Master theorem is one of three methods we have learned from the previous algorithm course to find complexity of recursive algorithms. It is a more direct approach and has 3 cases to solve recurrences. Note that, not all recurrences can be solved by master theorem. "a" value means number of sub-problems in the recursive algorithm. "b" value means dividing factor of data to create sub-datas which sent to sub-problems.

2) Present your problem formulation in detail.

This growing ball problem is actually a closest pair problem in three-dimensional space. Because of constant growing speed; we need to find which ball pairs are the closest to each other in this problem to find the distance between first touching ball pairs. Using naive method, we can pick all balls one by one and find distance from picked ball to all remaining other balls. However, this approach has $O(n-1) + O(n-2) +...+ O(1) = O(n^2)$ time complexity which means not efficient with big n numbers. Instead, we can apply divide and conquer algorithm design to solve this problem. With this approach, we can solve it in O(nlogn) time complexity. It can be achieved by dividing our data into left and right sets recursively. After elements in a set are small enough, we can find minimum distance in a set with naive method. The tricking part is a pair of balls with minimum distance may have a ball which is on a side, and the other ball which is on the other side. We have to consider that tricking part in merging process to get the right result. To solve this problem, we should create a new set called strip including elements which are close enough to middle line. Luckily, we do not have to use naive method for all elements inside strip. If we observe the elements inside strip, we will notice that there is a sparsity. The elements inside same left or right

side of strip cannot have a distance less than half of strip width (strip distance) with same side elements. Because, if such distance would occur, the strip width would change as well.

In my solution, after the program generates left and right sets, it divides Z axis into pieces which I call columns at strip part. They have xyz dimensions of 2d, inf, 1d and numbered from increasing z order. Then program puts each element inside strip into the its proper column. My first key observation is if an element is inside a column numbered i, the other element with minimum distance should be inside in i-1, i or i+1 numbered columns (Left column of the main column, main column itself and right column of main column). Therefore, program only needs to check 3 columns at most for each ball in strip.

The strip set which the program creates is already sorted in increasing Y order. In first phase of strip process, we put values from strip to correct columns. This process makes elements inside columns sorted in Y order too without calling another sorting process. The program also saves its position in the middle(main) column as well as position of last element inside left and right columns. That's because, the program needs to remember its position in increasing Y order inside column for all three columns. In seconds phase of strip process, program starts to calculate distances between elements. It starts with picking the first element of strip again. It checks the elements at upper position of this element by checking columns. If the Y distance is in range, then it calculates distance. If it is not in Y range, it means program does not need to check remaining values in the column anymore and simply breaks the loop then check another column. That means we only check a fixed volume for each ball, and since balls can be inside that area is limited, real complexity of that process is actually constant. The program does not need to check elements under (related to Y axis) the current element, since we are investigating from minimum Y valued element to maximum Y valued one progressively.

Strip process returns if there is such minimum distance inside strip, otherwise return the already known minimum value which is half of strip width.

3) How does your algorithms work?

The pseudocode of the algorithm is as follows:

```
Sort the ballArray(initial set) in increasing X order O(nlogn)
Copy addresses of balls inside ballArray to new set named sorted; O(n)
Sort sortedY in increasing Y order O(nlogn)
Call RecursiveFunc

Func RecursiveFunc: Called logn times
If size is less than 4:
    Return distance using NaiveMethod
EndIf
Set midBall to middle ball in x axis
For each balls in sortedY: O(n)
    If X of ball is smaller than X of midBall:
        Put the ball into left set
```

```
    Else:
        Put the ball into right set
    EndIf
EndFor
Set minLeftDist to RecursiveFunc(leftSet) Recursive T(n/2)
Set minRightDist to RecursiveFunc(rightSet) Recursive T(n/2)
Set stripDist to smaller of minLeftDist and minRightDist
Return SearchStripFunc

Func SearchStripFunc: Second part of previous function, has 4O(n)
Set minimum distance to strip distance
If size is less than 2:
    Return strip distance
If size is less than 4:
    Return smaller of strip distance and naiveMethod;
For each balls in strip: O(n)
    If Z of the ball is smaller than smallestZ:
        Set smallestZ to Z of the ball
    EndIf
EndFor
For each balls in strip: O(n)
    If Z of the ball is larger than largestZ:
        Set largestZ to Z of the ball
    EndIf
EndFor
Set rangeZ to largestZ - smallestZ
Set columnCount to (rangeZ/strip distance) + 1
For each balls in strip: O(n)
    Save column ID informations into properties of the ball
    Save column index informations into properties of the ball
EndFor
For each balls in strip: O(n)
    For each three columns: O(1)
        If the column is invalid:
            Skip checking this column
        EndIf
        For upper balls in the column: Seems O(n), but because of sparsity, it is O(1)
            If Y distance between target ball and this ball is more than strip distance:
                Break the loop
            EndIf
            Set newDist to distance between this ball and target ball
            If newDist is smaller than minimum distance:
                Set minimum distance to newDist
            EndIf
        EndFor
    EndFor
EndFor
Return minimum distance
```

```
Func NaiveMethod: Seems O(n^2), but program does not call it for more than 3 balls
Set minDist to maximum value available
For each balls in set: O(n)
    For remaining balls in set: O(n)
        Set newDist to Euclidean-distance of balls
        If newDist is smaller than minDist:
            Set minDist to newDist
        EndIf
    EndFor
Return minDist
```

From the pseudocode, initial part of the algorithm (up to recursive part):

$I(n) = O(n\log n) + O(n) + O(n\log n) = O(n\log n)$

Recursive part of the algorithm:

$R(n) = O(n) + 2R(n/2) + 4O(n) = 2R(n/2) + O(n)$

If we try to solve recurrence and start iterating starting from the base value:

$R(base) = 2R(base/2) + O(base)$

$R(base/2) = 2R(base/4) + O(base/2)$

$R(base/4) = 2R(base/8) + O(base/4)$

…

…

It also looks like there will be logn steps. This algorithm is indeed recursive. This recurrence is in the form of the second case of master theorem with "a" is 2, "b" is 2. Therefore, we can conclude that $R(n) = O(n\log n)$.

Now we can find the final complexity of the whole algorithm:

$T(n) = I(n) + R(n) = O(n\log n) + O(n\log n) = O(n\log n)$

4) Analyze and explain the algorithm results.

The following table is achieved by running the program on ITU SSH servers for each input file:

Input Size	1000	5000	10000	25000
Minimum Distance	16.9115	37.3631	30.2655	39.6737
Total Calculations	2222	11450	22500	54956
Avg Running Time	7ms	43ms	94ms	246ms

It looks like from the table that both calculations and running time increases close to O(n). However, especially between the small size inputs, increase amount of complexity exceeds increase amount of input size. Therefore, the answer cannot be O(n).

If we consider complexity of O(nlogn) instead, we can see that it will fit for all cases. That means we do not need to check $O(n^2)$, since O(nlogn) is a tighter bound. The answer from practical results is the same as we have calculated from pseudocode; which is O(nlogn).