REPORT

Programming Assignment 1

Abstract

The following document details Programming Assignment 1

John Herrmann

Jherrma6@jh.edu | 770-519-9435

Table of Contents

Problem #1 – Closest Pairs

Problem 1A:

See Attachment 1 – Closest Pairs Brute Force Algorithm Logic

Problem 1B:

/Source/Assignment1/src/closestPairs/CartesianPairs.java

```java
/***************************************************************************
* itSearchCP (Iterative-Search-Closest Pair)Method iterative search
*                                            ...for closest pair
* @return closest pair distance
***************************************************************************/
public static double itSearchCP(PointList points, ArrayList<PointList> pairs) {
                                                /*Iterate through list to find
                                                 *... smallest distance     */
    double smallest = 0;                        /*Init smallest with blank  */

    /* Nested For-Loop to Calculate Pair Permutations                     */
    for(int pivot = 0; pivot < points.size(); pivot++) {
        for(int i = 0; i < points.size(); i++) {
            /* ************************************************************
             * Only do calculation if I > Pivot.. Otherwise calculation:
             *   (1) Is Not necessary  (I = Pivot)
             *   (2) Already been done (I < Pivot)
             *
             * [A] [B] [C] [D]
             *   ^       ^
             *   |       |
             * Pivot     I
             ***************************************************************/
            if(i > pivot) {
                Point A = points.list.get(pivot);
                Point B = points.list.get(i);

                                            /*Create a structure to store
                                             *... a single pair of points */
                PointList single_pair = new PointList();

                                            /*Store A & B in single pair
                                             *... structure
                                             */
                single_pair.createPair(A, B);

                pairs.add(single_pair);     /*Add single pair structure to the
                                            /*list of all pairs

                                            /*If we're looking at beginning
                                             *... of array, initialize
                                             *... smallest variable     */
                if(pivot == 0 && i == 1) {
                    smallest = single_pair.distance;
                }
                else if (single_pair.distance < smallest) {
                    smallest = single_pair.distance;
                }
            }
        }
    }
 return smallest;
}
```

Figure 1. Brute force search algorithm

Problem 1C:

See Attachment 2 – Closest Pairs Better-than-Brute-Force Algorithm Logic

Problem 1D:

/Source/Assignment1/src/closestPairs/CartesianPairs.java

```java
    /***************************************************************************
     * MinDist Method uses an efficient algorithm to find closest pairs
     * @return closest pair distance
     ***************************************************************************/
    public static double MinDist(PointList points, ArrayList<PointList> pairs) {
        double minDist;
        int numPoints = points.size();
                                            /* If |P| < 3; Try All Pairs */
        if(numPoints < 4 && numPoints > 1){
            return itSearchCP(points, pairs);
          }
        else {

            int median = (numPoints) / 2;       /*Calculate median Index        */
                                            /*Calculate median value           */
            double medianValue = points.list.get(median).x;
                                            /*Create a sublist from [0-Median)*/

            List<Point> head = points.list.subList(0,  median);
                                            /*Create a sublist from [Median-End)*/
            List<Point> tail = points.list.subList(median, numPoints);

                                            /* Create A Sub PointList for Head*/
            PointList headPoints = new PointList();
            headPoints.list = new ArrayList<Point>(head);

                                            /* Create A Sub PointList for Tail*/
            PointList tailPoints = new PointList();
            tailPoints.list = new ArrayList<Point>(tail);

                                            /* Calculate upper bounds of min */
            double minTail = MinDist(tailPoints, pairs);
            double minHead = MinDist(headPoints, pairs);
            minDist = (minTail < minHead) ? minTail : minHead;

            PointList BL = new PointList();
            PointList BR = new PointList();

                                            /* Find points in head and tail within a minimum
                                             * distance from the median
                                             */
            BL.list = headPoints.withinRange(medianValue - minDist, median, Cartesian.xSort);
            BR.list = tailPoints.withinRange(medianValue, median + minDist, Cartesian.xSort);

                                            /* Combine BL and BR Lists          */
            PointList combined = new PointList();
            if(BL.list == null && BR.list == null) { return minDist; }
            if(BL.list != null) { combined.list.addAll(BL.list); }
            if(BR.list != null) { combined.list.addAll(BR.list); }

            double innerMin = itSearchCP(combined, pairs);
                                            /*Return the minimum distance found */
            return (innerMin < minDist) ? innerMin : minDist;
            }
```

Figure 2a. Better-than-Brute force search algorithm

```java
    /*************************************************************************
     * findSmart Method uses an efficient algorithm to find closest pairs
     * @return closest pair distance
     *************************************************************************/
    public double findSmart() {
                                                /*Stores pair calculations */
        ArrayList<PointList> temp_pairs = new ArrayList<PointList>();
        this.pairList = temp_pairs;

        this.list.sortXAxis();                  /*Sort along x before start*/
        this.analyzed = true;                   /*Enables pair analysis    */

        return MinDist(this.list, this.pairList);   /*Start of recursive algorithm*/
    }
    /*************************************************************************
```

Figure 2b. Better-than-Brute force search algorithm

```java
    /*************************************************************************
     * ClosestNPairs Method prints the closest N pairs.
     *              Note: Method must be called after findSmart() method
     *************************************************************************/
    public void ClosestNPairs(int n) {

        n = Math.abs(n);                                /* Insure N is positive */

        if(this.pairList.size() > 0) {
            Collections.sort(this.pairList, new SortByDistance());

            int nOrAll = (n < this.pairList.size()) ? n : this.pairList.size();

            for(int i = 0; i < nOrAll; i++) {
                                                        /* Acquire a pair */
                PointList tempPair = this.pairList.get(i);
                Point a;
                Point b;
                double distance;
                                                        /*Confirm list is pair */
                if(tempPair.size() == 2) {
                    a = tempPair.list.get(0);           /*Acquire Points A & B */
                    b = tempPair.list.get(1);
                    distance = tempPair.distance;
                    System.out.printf("Point A (%f, %f) and Point B(%f, %f)\n" ,
                            a.x, a.y, b.x, b.y);
                }
            }
        }
    }
```

Figure 3. ClosestNPairs method

```java
    /***************************************************************************
     * ClosestNPairsDistance Method prints the closest N pairs & distances
     *                      Note: Method must be called after findSmart() method
     ***************************************************************************/
    public void ClosestNPairsDistance(int n) {

        n = Math.abs(n);                                 /* Insure N is positive */

        if(this.pairList.size() > 0) {
            Collections.sort(this.pairList, new SortByDistance());

            int nOrAll = (n < this.pairList.size()) ? n : this.pairList.size();

            for(int i = 0; i < nOrAll; i++) {
                                                         /* Acquire a pair */
                PointList tempPair = this.pairList.get(i);
                Point a;
                Point b;
                double distance;
                                                         /*Confirm list is pair */
                if(tempPair.size() == 2) {
                    a = tempPair.list.get(0);            /*Acquire Points A & B */
                    b = tempPair.list.get(1);
                    distance = tempPair.distance;
                    System.out.printf("[%f, %f] & [%f, %f] are distance %f apart \n" ,
                            a.x, a.y, b.x, b.y, distance);
                }
            }
        }
    }
```
Figure 4. ClosestNPairsDistance method

Use Cases

Use Case 1: Brute force algorithm finds closest pair of points in a two-dimensional plane

Use Case Name	Brute force algorithm finds closest pair of points in a two-dimensional plane
Use Case Goal	Algorithm returns the closest pair of points in a two-dimensional plane
Stakeholders	Software Developer – John Herrmann Software Tester – John Herrmann Software User – Dr. Fink
Actors	Command Line Software User
Preconditions	Java is installed on machine Software User provides algorithm with input file containing points Software User provides algorithm with output file
Steps	1. **Algorithm** opens input file 2. **Algorithm** validates the input file 3. **Algorithm** stores the input file contents in data structure (see store2D) 4. **Algorithm** searches the data structure for the closest pair 5. **Algorithm** prints closest pair(s) via the CLI 6. **Algorithm** stores closest pair(s) in output file 7. **Algorithm** terminates execution and exits
Postconditions	Output file contains closest pair analysis
Trigger	Software user executes algorithm via CLI
Alternate Scenario	1a. **Algorithm** fails to load file: 1a-1. **Algorithm** sends error message to user 1a-2. **Algorithm** terminates execution and exists 2a. **Algorithm** file validation fails: 2a-1. **Algorithm** sends error message to user 2a-2. **Algorithm** terminates execution and exists 4a. **Algorithm** finds multiple pairs with equivalent distance 4a-1. Algorithm stores each set of pairs with the smallest distance in data structure 6a. **Algorithm** stores closest pairs in output file validation fails: 6a-1. **Algorithm** sends error message to user 6a-2. **Algorithm** terminates execution and exists

Table 1. Use Case1

Problem #2 – Deterministic Turing Machine

Use Cases

Use Case 1: DTM takes user input and calculates a decision

Use Case Name	DTM takes user input and calculates a decision
Use Case Goal	DTM returns a “yes” or “no” decision based on the user’s input
Stakeholders	Software Developer – John Herrmann Software Tester – John Herrmann Software User – Dr. Fink
Actors	Command Line Software User
Preconditions	Java is installed on machine Software User provides algorithm with input file containing $\Gamma = \{0, 1, b$
Steps	1. **Algorithm** opens input file 2. **Algorithm** validates the input file (see SR: Validate Tape Input) 3. **Algorithm** stores input file contents in an input data structure (see Tape) 4. **Algorithm** computes decision given input data structure (see DTM) 5. **Algorithm** prints decision to the CLI 6. **Algorithm** terminates execution and exits
Postconditions	-
Trigger	Software user executes algorithm via CLI
Alternate Scenario	1a. **Algorithm** fails to load file: 1a-1. **Algorithm** sends error message to user 1a-2. **Algorithm** terminates execution and exists 2a. **Algorithm** file validation fails: 2a-1. **Algorithm** sends error message to user 2a-2. **Algorithm** terminates execution and exists

Software Requirement: Validate Tape Input

Requirement Name	Validate Tape Input
Requirement Goal	Verifies the tape input and provides a success or failure output
Stakeholders	Software Developer – John Herrmann
Actors	DTM Algorithm
Preconditions	Java is installed on machine Software User provides algorithm with input file containing Γ = {0, 1, *b*
Steps	1. **Algorithm** scans the input and confirms each tape element exists in Γ
Postconditions	Input validation is complete
Alternate Scenario	1a. **Algorithm** finds input which is not in Γ 1a-1. **Algorithm** outputs fail and ceases operation 1a-2. **Algorithm** terminates execution and exists

Data Structure Requirement: Tape

Tape	
+ data: Array< Γ >	//An array containing tape input
+headPosition: Int	//Current head position //Initializes to ‘0’
+read()	//Reads the value under tape head
+write(Value)	//Writes a value under tape head
+left()	//Moves tape head to the left
+right()	//Moves tape head to the right

Data Structure Requirement: DTM

Responses
+responses: Array < response>
+nextResponse(input)

Data Structure Requirement: response

response	
+nextState: enum	//Enum representing states
+write: character in Γ	//Character to write to strip
+headMovement: int	//Movement of head left or right

Attachment 1: Closest Pairs Brute Force Algorithm Logic

Algorithm – Calculate Closest Pairs using Brute Force approach

Let point be a data type consisting of two coordinates
Let input be an array consisting of N points
Let pair be a data type consisting of two points
Let result be a data structure which will contain:

- An array **allPairs** which will contain $\frac{N(N-1)}{2}$ pairs
- A variable **closestLength** which will contain the closest pair length
- An array **closestPairs** which will contain all pairs separated by the closet pair length

For each **point** in **input:**
  Create variable **superIndex** which equals the current point's index
  For each **point** in **input:**
    If the **point's index** is less than or equal to **superIndex:**
      Do Nothing
    Else
      Create a **pair** containing **point @ index** and **point @ superIndex**
      Store **pair** in **result's allPairs** data element
      Calculate the **distance** between the points in **pairs**
      If the number of pairs in **allPairs** is equal to one:
        closestLength = **distance**
        closestPairs = pair
      Else if **distance** is less than or equal to **closestLength**
        If **closestLength** == **distance**
          Add **pair** to **closestPairs**
        Else
          Set **closestLength** = **distance**
          Reset **closestPairs** to empty array
          Add **pair** to **closestPairs**

Return the **closestLength** parameter from the **result** variable

END

Example
Given: 4 set of points named A, B, C, D
Find: The maximum number of games assuming each team plays every other team once

Proof

1. From the perspective of each team, the team plays three games.

   Team A's Perspective:
   AB
   AC
   AD

   Team B's Perspective:
   BA
   BC
   BD

   [and so on…]

2. Since a team never plays itself, if there are N teams, the maximum amount of games a team can play are (N-1) games:

$$MaxGames_{SingleTeam} = (N-1)$$

3. If there are N teams, each of which plays (N-1) games with another team, logically, the total number of games payed is then:

$$Total_{Pairs} = \frac{N(N-1)}{2}$$

Attachment 2 – Closest Pairs Better-than-Brute-Force Algorithm Logic

1. **System** sorts the list of points based on the point's x value
2. **System** divides the list in half – creating two smaller lists
3. **System** recursively calls itself
4. If the size of the list is <= three
    a. **System** finds the closest point in the list
5. **System** calculates the upper bound on the minimum distance S= min(min1, min2)
6. **System** creates a third list consisting of points +/- S of the median
7. **System** recursively calls itself on the third list
8. **System** returns the minimum distance found

Worst-Case Analysis:

Steps 1-6: $\frac{N}{3} * \frac{3*(3-1)}{2} = N$ //Assuming N are divided into

Step 7: $\frac{N}{3} * \frac{3*(3-1)}{2} = N$ //Worst-case all points are equal

BigO(N^2) //Note: this is the worst-case

Average-Case Analysis

Steps 1-6: $\frac{N}{3} * \frac{3*(3-1)}{2} = N$ //Assuming N are divided into

Step 7: $\frac{N}{30} * \frac{3*(3-1)}{2} = N/5$ //Assume third list is of size N/10

Average Case: N^2 / 5

Conclusion: The more efficient algorithm, for a large and highly random dataset, will probably run in about 1/5 the amount of time, on an average case.