John Herrmann

Jherrma6@jh.edu | 770-519-9435

Abstract

The following document details Programming Assignment 1

Report

Programming Assignment 1

Table of Contents

[Problem #1 – Closest Pairs 2](#_Toc34266311)

[Problem 1A: 2](#_Toc34266312)

[Problem 1B: 2](#_Toc34266313)

[Problem 1C: 3](#_Toc34266314)

[Problem 1D: 3](#_Toc34266315)

[Use Cases 6](#_Toc34266316)

[Use Case 1: Brute force algorithm finds closest pair of points in a two-dimensional plane 6](#_Toc34266317)

[Problem #2 – Deterministic Turing Machine 7](#_Toc34266318)

[Use Cases 7](#_Toc34266319)

[Use Case 1: DTM takes user input and calculates a decision 7](#_Toc34266320)

[Software Requirement: Validate Tape Input 8](#_Toc34266321)

[Data Structure Requirement: Tape 8](#_Toc34266322)

[Data Structure Requirement: DTM 9](#_Toc34266323)

[Data Structure Requirement: response 9](#_Toc34266324)

[Attachment 1: Closest Pairs Brute Force Algorithm Logic 10](#_Toc34266325)

[Attachment 2 – Closest Pairs Better-than-Brute-Force Algorithm Logic 12](#_Toc34266326)

# Problem #1 – Closest Pairs

## Problem 1A:

See Attachment 1 – Closest Pairs Brute Force Algorithm Logic

## Problem 1B:

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

A screenshot of a social media post

Description automatically generated

**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

A screenshot of a social media post

Description automatically generated

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

A screenshot of a cell phone

Description automatically generated

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

A screenshot of a social media post

Description automatically generated

**Figure 3.** ClosestNPairs method

A screenshot of a social media post

Description automatically generated

**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 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 Γ = {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 |

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

|  |  |
| --- | --- |
| **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 |

### Software Requirement: Validate Tape Input

### 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 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…]

1. Since a team never plays itself, if there are N teams, the maximum amount of games a team can play are (N-1) games:
2. If there are N teams, each of which plays (N-1) games with another team, logically, the total number of games payed is then:

# 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
   1. **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: //Assuming N are divided into

Step 7: //Worst-case all points are equal

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

**Average-Case Analysis**

Steps 1-6: //Assuming N are divided into

Step 7: //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.**