알고리즘 15주차 과제 보고서

충남대 컴퓨터공학과
학번: 201701975
이름: 구건모

과제의 목표와 해야할 일

이번 과제의 목표는 지난 과제에서 구현했었던 ClosestPair를 찾는 알고리즘인 ComparingAllPairs와 DivideAndConquer, 그리고 두 방식의 장점을 취합한 Hybrid 알고리즘에 대한 성능 측정을 해보는 것이 목표였습니다. 따라서 이전에 성능측정시 구현하였던 것처럼 측정을 위한 Experiment 클래스들과 parameterSet 클래스를 구현 후 성능을 측정해보는 것이 이번 과제의 주요 내용입니다.

```java
    private PointSet generatePointSet(int size) {
        PointSet pointSet = new PointSet(size);
        Random random = new Random();
        for (int count = 0; count < size; count++) {
            int x = random.nextInt(MAX_COORDINATE_VALUE);
            int y = random.nextInt(MAX_COORDINATE_VALUE);
            Point point = new Point(x, y);
            pointSet.add(point);
        }
        return pointSet;
    }

    public long[][] measureDurationOfSingleSolve() {
        long[][] measurement = new long[NUMBER_OF_KINDS_OF_EXPERIMENTS][this.parameterSet().numberOfSteps()];

        int sizeForStep = this.parameterSet().startingSize();
        for (int step = 0; step < this.parameterSet().numberOfSteps(); step++) {
            PointSet pointSet = this.generatePointSet(sizeForStep);
            measurement[0][step] = this.experimentForComparingAllPairs().durationOfSingleSolve(pointSet);
            measurement[1][step] = this.experimentForDivideAndConquer().durationOfSingleSolve(pointSet);
            sizeForStep += this.parameterSet().incrementSize();

        }
        return measurement;

    }

    public long[][] measureAverageDurationOfSingleSolves() {
        long[][] measurement = new long[NUMBER_OF_KINDS_OF_EXPERIMENTS][this.parameterSet().numberOfSteps()];
        int sizeForStep = this.parameterSet().startingSize();
        for (int step = 0; step < this.parameterSet().numberOfSteps(); step++) {
            PointSet pointSet = this.generatePointSet(sizeForStep);
            measurement[0][step] = this.experimentForComparingAllPairs().averageDurationOfSingleSolves(pointSet);
            measurement[1][step] = this.experimentForDivideAndConquer().averageDurationOfSingleSolves(pointSet);
            sizeForStep += this.parameterSet().incrementSize();
        }
        return measurement;
    }

    public long[][] measurementMinDurationAmongSingleSolves() {
        long[][] measurement = new long[NUMBER_OF_KINDS_OF_EXPERIMENTS][this.parameterSet().numberOfSteps()];

        int sizeForStep = this.parameterSet().startingSize();
        for (int step = 0; step < this.parameterSet().numberOfSteps(); step++) {
            PointSet pointSet = this.generatePointSet(sizeForStep);
            measurement[0][step] = this.experimentForComparingAllPairs().minDurationAmongSingleAmongSolves(pointSet);
            measurement[1][step] = this.experimentForDivideAndConquer().minDurationAmongSingleAmongSolves(pointSet);
            sizeForStep += this.parameterSet().incrementSize();
        }
        return measurement;
    }

    public boolean closestPairAlgorithmAreCorrect() {
        PointSet pointSet = this.generatePointSet(ExperimentManager.SIZE_FOR_CORRECTNESS_TEST);
        PairOfPoints closestPairByComparingAllPairs = this.findClosestPair().solveByComparingAllPairs(pointSet);
        PairOfPoints closestPairByDivdeAndConquer = this.findClosestPair().solveByDivideAndConquer(pointSet);

        return ((closestPairByComparingAllPairs.distance() == closestPairByDivdeAndConquer.distance()));

    }
```

Class ExperimentManager

ExperimentManager 클래스에서는 ClosestPairf를 찾는 여러 방식에 따른 Measurement 기능이 구현되어 있습니다.
각 측정에 대해서 MinDuration, AverageDuration, DurationOfSingleSolve 세가지 방식으로 측정한 측정치를 리턴하도록 구현됩니다.

```java
package experiment;

public class ParameterSet {
    private static final int DEFAULT_NUMBER_OF_STEPS = 10;
    private static final int DEFAULT_STARTING_SIZE = 1000;
    private static final int DEFAULT_INCREMENT_SIZE = DEFAULT_NUMBER_OF_STEPS;
    private static final int DEFAULT_NUMBER_OF_REPETITIONS_OF_SAME_EXECUTION = 10;

    private int _numberOfSteps;
    private int _startingSize;
    private int _incrementSize;
    private int _numberOfRepetitionsOfSameExecution;

    private static final int DEFAULT_MIN_RECURSION_SIZE = 150;

    private int _minRecursiveSize;

    private int _minRecursionSize() {
        return this._minRecursiveSize;
    }

    private void setMinRecursionSize(int newMinRecursionSize) {
        this._minRecursiveSize = newMinRecursionSize;
    }

    public ParameterSet() {
        this(ParameterSet.DEFAULT_NUMBER_OF_STEPS, ParameterSet.DEFAULT_STARTING_SIZE,
                ParameterSet.DEFAULT_INCREMENT_SIZE, ParameterSet.DEFAULT_NUMBER_OF_REPETITIONS_OF_SAME_EXECUTION
        /* ,DEFAULT_MIN_RECURSION_SIZE */);
    }

    public ParameterSet(int givenNumberOfSteps, int givenStartingSize, int givenIncrementSize,
            int givenNumberOfRepetitionsOfSameExecution) {
        this._numberOfSteps = givenNumberOfSteps;
        this._startingSize = givenStartingSize;
        this._incrementSize = givenIncrementSize;
        this._numberOfRepetitionsOfSameExecution = givenNumberOfRepetitionsOfSameExecution;
    }

    public int numberOfSteps() {
        return this._numberOfSteps;
    }

    public void setNumberOfSteps(int newNumberOfSteps) {
        this._numberOfSteps = newNumberOfSteps;
    }

    public int startingSize() {
        return this._startingSize;
    }

    public void setStartingSize(int newStartingSize) {
        this._startingSize = newStartingSize;
    }

    public int incrementSize() {
        return this._incrementSize;
    }

    public void setIncrementSize(int newIncrementSize) {
        this._incrementSize = newIncrementSize;
    }
```

Class ParameterSet

ParameterSet 은 측정을 진행할 때 데이터를
점차 늘려가는 식으로 측정하는데,
이때 시작 데이터 수와 step 당 얼마나
데이터를 증가시킬지 등에 대한
Parameter 값을 다루기 위한 클래스입니다.

```java
package experiment;

import model.FindClosestPair;

public class ExperimentForComparingAllPairs extends Experiment {
    public ExperimentForComparingAllPairs(FindClosestPair givenFindClosestPair, ParameterSet givenParameterSet) {
        super(givenFindClosestPair, givenParameterSet);
    }

    @Override
    public long durationOfSingleSolve(PointSet pointSet) {
        // TODO Auto-generated method stub
        Timer.start();
        this.findClosestPair().solveByComparingAllPairs(pointSet);
        Timer.stop();
        return Timer.duration();

    }
}
```

Class ExperimentForComparingAllPairs

모든 Point들의 distance를 모두 비교하여 closestPair를 찾았을 때에 대한 측정이 구현되어 있는 클래스 입니다.
저번 과제에서 구현하였던 FindClosestPair 클래스의 solveByComparingAllPairs 메서드와 Timer를 이용하여 측정합니다.

```java
package experiment;

import model.FindClosestPair;

public class ExperientForDivideAndConquer extends Experiment {

    protected ExperientForDivideAndConquer(FindClosestPair givenFindClosestPair,
            ParameterSet givenParameterSet) {
        super(givenFindClosestPair, givenParameterSet);
        // TODO Auto-generated constructor stub
    }

    public long durationOfSingleSolve(PointSet pointSet) {
        Timer.start();
        this.findClosestPair().solveByDivideAndConquer(pointSet);
        Timer.stop();
        return Timer.duration();
    }

}
```

Class ExperimentForDivideAndConquer

Divide and Conquer 방식으로 ClosestPair를 찾는 경우에 대한 측정이 구현되어 있는 클래스입니다. 기존에 구현했었던 findClosestPair 클래스의 SolveByDivideAndConquer 메서드와 Timer를 이용하여 시간을 측정합니다.

```java
package experiment;

import model.FindClosestPair;
import model.PointSet;

public abstract class Experiment {
    private ParameterSet _parameterSet;
    private FindClosestPair _findClosestPair;

    private ParameterSet ParameterSet() {
        return this._parameterSet;
    }

    private void setParameterSet(ParameterSet newParamenterSet) {
        this._parameterSet = newParamenterSet;
    }

    protected FindClosestPair findClosestPair() {
        return this._findClosestPair;
    }

    private void setFindClosetPair(FindClosestPair newFindClosestPair) {
        this._findClosestPair = newFindClosestPair;
    }

    protected Experiment(FindClosestPair givenFindClosestPair, ParameterSet givenParameterSet) {
        this.setFindClosetPair(givenFindClosestPair);
        this.setParameterSet(givenParameterSet);
    }

    public abstract long durationOfSingleSolve(PointSet pointSet);

    public long averageDurationOfSingleSolves(PointSet pointSet) {
        long sum = 0;
        for (int count = 0; count < this.ParameterSet().numberOfRepetitionOfSameExecution(); count++) {
            sum += this.durationOfSingleSolve(pointSet);
        }
        long average = sum / this.ParameterSet().numberOfRepetitionsForAverage();
        return average;
    }

    public long minDurationAmongSingleAmongSolves(PointSet pointSet) {
        long minDuration = this.durationOfSingleSolve(pointSet);
        for (int count = 1; count < this.ParameterSet().numberOfRepetitionOfSameExecution(); count++) {
            long duration = this.durationOfSingleSolve(pointSet);
            if (duration < minDuration)
                minDuration = duration;
        }
        return minDuration;
    }

}
```

Class Experiment

Experiment의 기능들이 정의되어 있는 abstract 클래스입니다. DivideAndConquer와 ComparingAllPairs에 대한 각각의 Experiment를 구현할 때 extends 하여 사용됩니다.

결과화면S1

```
<terminated> _Main_AL15_201701975_구건모 [Java Application] C:₩Users₩gmk

<<< 최단거리 쌍 찾기 성능 측정을 시작합니다 >>>
<한번 실행측정> (단위: 마이크로 초)
   Size     Compare-All-Pairs   Divide-And-Conquer
[  1000]                   622                  662
[  2000]                  2591                 1434
[  3000]                  5700                 2170
[  4000]                 10271                 3241
[  5000]                 16390                 4061
[  6000]                 23438                 4931
[  7000]                 31909                 5952
[  8000]                 41401                 7002
[  9000]                 52753                 8028
[ 10000]                 65231                 9061
<반복 실행의 평균측정> (단위: 마이크로 초)
   Size     Compare-All-Pairs   Divide-And-Conquer
[  1000]                   657                  655
[  2000]                  2618                 1413
[  3000]                  5975                 2162
[  4000]                 10610                 3679
[  5000]                 17317                 3876
[  6000]                 24093                 4886
[  7000]                 32810                 5596
[  8000]                 38725                 7283
[  9000]                 56644                 9262
[ 10000]                 67393                10030
<반복 실행 중 최소 시간측정> (단위: 마이크로 초)
   Size     Compare-All-Pairs   Divide-And-Conquer
[  1000]                   673                  621
[  2000]                  2452                 1410
[  3000]                  5760                 1906
[  4000]                  9244                 2739
[  5000]                 14396                 3535
[  6000]                 20826                 4209
[  7000]                 28669                 5828
[  8000]                 37368                 6293
[  9000]                 48119                 7189
[ 10000]                 59019                 7708

<<< 최단거리 쌍 찾기 성능 측정을 종료합니다 >>>
```

크기가 증가함에 따라, 측정 값이 알고리즘의 분석 결과와 어느정도 일치하는지?

크기가 증가함에 따라서 이론적인 복잡도에 수렴해가는 것을 볼 수 있습니다.

Java 환경으로 인해 발생하는 측정 요인을 최소화 하려면?

- 한번만 측정할 경우의 문제점

한번만 측정할 경우, GarbageCollector 등의 자바 환경으로 인해서 영향을 받게 되면 부정확한 결과가 도출될 수 있습니다. 따라서 여러번 반복한 후 평균값을 취하는 것이 안전하다고 생각합니다.

- 평균 측정에서는 어떻게 최소화 되고 있나?

평균측정에서는 여러번의 측정값의 평균을 사용하기 때문에 Java 환경으로 인한 측정요인의 영향이 기존보다 최소화 되었다고 생각합니다.

- 최솟값 측정에서는 어떻게 최소화 되고 있나?

최솟값 측정에서는 모든 측정치 중 최솟값을 택하여 자바의 환경으로 발생할 수 있는 잘못된 측정을 최소화 하고 있습니다.

- 최솟값 측정이 평균값 측정보다 더 좋은 방법일까?

최솟값 측정은 모든 측정중 최솟값을 측정하는 것이기 때문에 여러번 측정한 후 평균을 낸 평균값 측정이 더 일반성을 가진 결과라고 생각합니다.

크기가 작은 경우에 "모든 쌍의 거리를 비교" 하는 방식의 성능이 더 좋게 나온다.

데이터 크기가 작은 경우에는 Divide and Conquer, combine 하는 방식으로 진행하는 것 보다 일일히 비교하는게 더 빠른 것으로 보입니다. 재귀적으로 분할하게 되면 양분된 Partition 과 Separation Line 에서의 ClosestPair 간의 거리를 계속 비교해주어야 하는데 데이터가 그렇게 크지 않은 상황에서 분할을 하게되면 분할정복의 이점보다는, 그 과정이 더 효율을 떨어뜨릴 수 있다고 생각합니다.

결과화면(S2)

```
<terminated> _Main_AL15_201701975_구건모 (1) [Java Application] C:₩Users₩gmku1₩.p2₩pool₩plugins₩org.eclip

<<< 최단거리 쌍 찾기 성능 측정을 시작합니다 >>>
<한번 실행측정> (단위: 마이크로 초)
    Size     Compare-All-Pairs   Divide-And-Conquer   Hybrid(MinRS: 200)   Saving (%)
[  1000]                   628                  694                  550           21
[  2000]                  2700                 1547                 1149           26
[  3000]                  6021                 2218                 2111            5
[  4000]                 10945                 3255                 2787           14
[  5000]                 17029                 4153                 3786            9
[  6000]                 24288                 4984                 4810            3
[  7000]                 33485                 6177                 5455           12
[  8000]                 42999                 7334                 6550           11
[  9000]                 55032                 8071                 7607            6
[ 10000]                 68332                 9055                 8658            4
<반복 실행의 평균측정> (단위: 마이크로 초)
    Size     Compare-All-Pairs   Divide-And-Conquer   Hybrid(MinRS: 200)   Saving (%)
[  1000]                   621                  665                  522           22
[  2000]                  2492                 1390                 1178           15
[  3000]                  5873                 2292                 2129            7
[  4000]                 11059                 3486                 2831           19
[  5000]                 17484                 4343                 4009            8
[  6000]                 24938                 5207                 5180            1
[  7000]                 33101                 5604                 4771           15
[  8000]                 44834                 9172                 6476           29
[  9000]                 57761                 8581                 7292           15
[ 10000]                 69680                 9556                 8702            9
<반복 실행 중 최소 시간측정> (단위: 마이크로 초)
    Size     Compare-All-Pairs   Divide-And-Conquer   Hybrid(MinRS: 200)   Saving (%)
[  1000]                   673                  610                  484           21
[  2000]                  2699                 1261                 1112           12
[  3000]                  5332                 1936                 1769            9
[  4000]                 10661                 2707                 2294           15
[  5000]                 14349                 3545                 3317            6
[  6000]                 21106                 4473                 4264            5
[  7000]                 31627                 5019                 4455           11
[  8000]                 38268                 6240                 5516           12
[  9000]                 50039                 6976                 6385            8
[ 10000]                 62205                 8349                 7396           11

<<< 최단거리 쌍 찾기 성능 측정을 종료합니다 >>>
```

최소 재귀 크기를 1000으로한 Hybrid 방식이 DivideAndConquer 방식보다 성능이 더 나쁘게 나오는 이유

- Hybrid 방식은 최소재귀크기가 되면 ComparingAllPairs 방식으로 ClosestPair를 찾는데, 이때 DivideAndConquer 를 수행하면서 발생한 모든 Partition 들에 대해서 ComparingAllPairs를 수행하다보니 성능이 더 나쁘게 나온것이라고 생각합니다.

Hybrid에서 크기의 증가에 따른 성능이득은?

크기가 늘어남에 따라, 성능의 이득의 비율이 줄어드는 것을 볼 수 있다. 그이유는 무엇일까?

- 모든 포인트를 비교하는 ComparingAllPair 방식은 작은 구간에서 유리하고 Divide And Conquer 방식은 데이터가 많을수록 유리해지는데 Hybrid를 사용하게 되면 데이터가 커졌을때 Recursive하게 Divide 하다가 최소재귀크기가 되면 ComparingAllPairs를 적용합니다. 데이터가 커지면 커질수록 Divide 되는 Partition 수도 많아지므로 ComparingAllPairs를 더 많은 데이터에 대해서 적용하게 되는데, 이때 성능 이득의 비율이 줄어들게 되는 것이라고 생각합니다.