Penentuan Daerah Evakuasi Minimum Akibat Kebocoran Radiasi Nuklir dengan Memanfaatkan *Convex Hull*

Penerapan Strategi *Divide and Conquer* dalam *Quickhull*

Paskahlis Anjas Prabowo (13515108)
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung 40132, Indonesia
13515108@std.stei.itb.ac.id

***Abstrak*—Energi nuklir memiliki manfaat yang sangat luas dalam kehidupan manusia. Di balik manfaatnya yang luas itu, terdapat risiko yang besar. Salah satunya risiko tersebut adalah kebocoran radiasi karena adanya gangguan pada reaktor. Terjadinya kecelakaan tersebut memiliki beberapa dampak negatif pada penduduk di sekitar reaktor. Untuk itu perlu dilakukan evakuasi untuk menghindari hal-hal yang tidak diinginkan terjadi pada penduduk di sekitar lokasi reaktor. Evakuasi memerlukan banyak biaya. Semakin sempit daerah evakuasi, semakin sedikit biaya yang dibutuhkan. Untuk itu, perlu ditentukan daerah evakuasi minimum untuk menekan biaya evakuasi. Penentuan daerah evakuasi dapat memanfaatkan prinsip-prinsip yang digunakan dalam penentuan *convex hull* dari suatu himpunan titik. Adapun titik-titik tersebut bersesuaian dengan lokasi sensor yang mendeteksi adanya kebocoran radiasi. Untuk menentukan *convex hull*, dapat digunakan metode *quickhull* yang memanfaatkan stategi *divide and conquer*. Strategi ini merupakan strategi umum yang dapat diterapkan secara luas dan cukup efisien.**

Kata kunci—nuklir, evakuasi, minimum, convex hull, divide and conquer, quickhulls

I. PENDAHULUAN

Energi nuklir merupakan energi yang tersimpan pada inti atom. Energi ini adalah energi yang menjaga atom-atom tetap pada ikatannya. Untuk itu, energi ini harus dilepaskan agar bisa dimanfaatkan. Energi nuklir dikenal memiliki manfaat yang luas bagi berbagai urusan manusia di dunia. Manfaat tersebut diantaranya dapat dijumpai dalam dunia medis, pembangkit tenaga listrik, persenjataan militer, dan lain sebagainya.

Di samping manfaatnya yang luar biasa, tidak jarang didapati kecelakaan yang berakibat fatal akibat adanya gangguan pada instalasi energi nuklir. Setidaknya terdapat 27 kecelakaan akibat adanya gangguan pada pembangkit listrik tenaga nuklir sejak tahun 1952 sampai 2011. Kecelakaan-kecelakaan tersebut memakan banyak korban jiwa dan ditaksir memiliki kerugian lebih dari USD 100 juta. Jika ditinjau secara lebih spesifik, kebocoran radiasi nuklir merupakan salah satu kecelakaan yang sering terjadi pada instalasi reaktor nuklir. Dampak akibat kecelakaan jenis ini memiliki dampak buruk bagi masyarakat pada radius tertentu dari pusat reaktor. Untuk itu, perlu adanya evakuasi untuk menghindari hal-hal yang mengancam keselamatan warga sipil di sekitar reaktor.

Untuk mengevakuasi warga-warga sipil di sekitar reaktor nuklir yang mengalami kebocoran, perlu ditentukan area evakuasinya terlebih dahulu. Di samping untuk memastikan penduduk pada area tersebut aman, hal ini bertujuan untuk menghemat biaya evakuasi, karena evakuasi hanya perlu dilakukan pada daerah yang sudah ditentukan saja. Daerah evakuasi ini dapat disebut sebagai daerah evakuasi minimum. Penentuan daerah evakuasi mininum ini dapat memanfaatkan *convex hull.*

Convex hull merupakan bidang minimum yang dapat menyelimuti seluruh titik dalam suatu himpunan titik yang ditentukan. Diasumsikan bahwa detektor kebocoran yang dapat merespons adanya kebocoran radiasi telah dipasang pada beberapa titik di sekitar reaktor nuklir. Dengan demikian, dapat diketahui titik mana saja yang terkena imbas bocornya radiasi nuklir. Dengan mengetahui titik-titik tersebut, dapat ditentukan daerah evakuasi minimumnya sesuai dengan permasalahaan *convex hull*. Permasalahan *convex hull* dapat diselesai dengan berbagai metode, salah satunya adalah dengan strategi *divide and conquer* (D&C). Algoritma yang menerapkan D&C untuk memperoleh *convex hull* dari sebuah himpunan titik dikenal dengan nama *quickhull*.

Algoritma *divide and conquer* secara umum merupakan pendekatan yang dapat digunakan dalam menyelesaikan suatu masalah dengan membagi dan menggabungkannya kembali. Pertama, instansiasi dari permasalah tersebut dibagi menjadi bagian-bagian yang lebih kecil dari sebelumnya. Setelah itu, dilakukan penyelesaian terhadap bagian-bagian kecil tersebut. Penyelesaian dari bagian-bagian kecil tersebut kemudian digabungkan untuk mendapatkan solusi permasalahan secara utuh.

II. DASAR TEORI

A. *Convex*

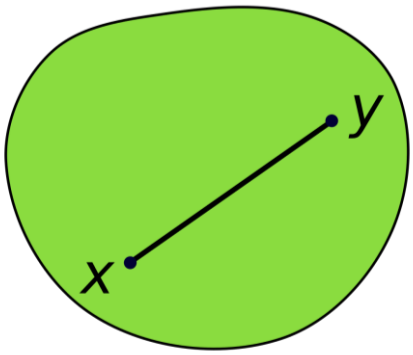
Convex merupakan salah satu istilah matematis yang termasuk predikat dari suatu himpunan yang beranggotakan titik. Diberikan himpunan titik-titik H dalam suatu bidang dengan

$$H = \{p_1, p_2, p_3, \dots, p_n\}.$$

Himpunan H dikatakan *convex* jika untuk setiap $p_i, p_j \in H$ dan $0 \leq k \leq 1, k \in \mathbb{R}$ berlaku

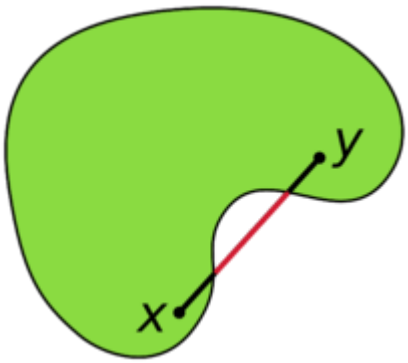
$$kp_i + (1-k)p_j \in H.$$

Hal ini dengan kata lain menyatakan bahwa suatu himpunan dari titik (H) pada suatu bidang dikatakan *convex* jika setiap garis yang menghubungkan 2 titik sembarang dalam H terdapat pada bidang bidang tersebut.



Sumber: https://en.wikipedia.org/wiki/Convex_set

Gambar 2.1.a. Ilustrasi himpunan c*onvex*

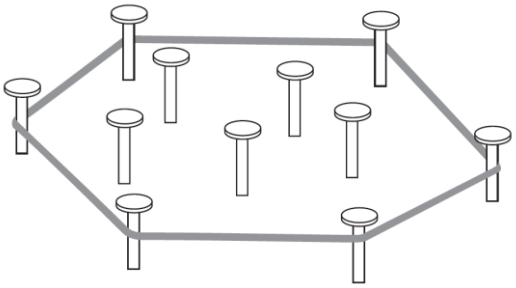


Sumber: https://en.wikipedia.org/wiki/Convex_set

Gambar 2.1.b. Ilustrasi himpunan *non-convex*

B. *Convex Hull*

Convex hull merupakan konsep lanjutan dari *convex*. *Convex hull* dari sebuah himpunan titik H adalah himpunan *convex* minimum yang mengandung H. Dengan kata lain, *convex hull* dari H adalah himpunan titik-titik P dalam bidang dengan luas minimum sedemikian hingga $H \subseteq P$.



Sumber: *Introduction to: The Design and Analysis of Algorithms*

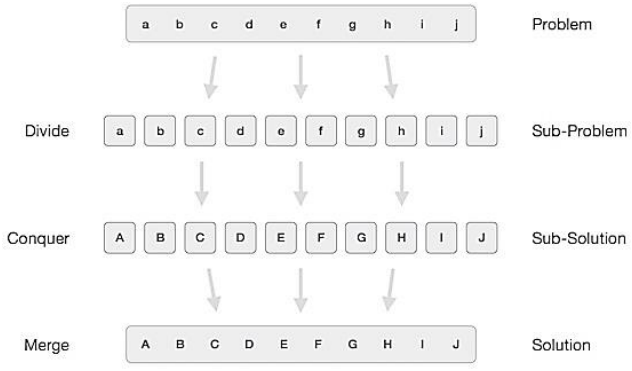
Gambar 2.2. Ilustrasi *convex hull* sebagai area yang diliputi oleh karet gelang oleh D. Harel

C. *Divide and Conquer*

Divide and conquer (D&C) merupakan salah satu algoritma yang populer. Hal ini dikarenakan D&C merupakan algoritma yang cukup efisien dengan strategi umum yang dapat diterapkan secara luas. Berikut merupakan strategi umum dalam D&C:

1. Instansiasi masalah dibagi menjadi beberapa submasalah dengan macam yang sama. Idealnya, pembagian masalah dilakukan sehingga ukuran submasalah yang sama.
2. Masing-masing submasalah diselesaikan sehingga didapatkan solusi dari setiap submasalah.
3. Solusi dari setiap submasalah digabungkan untuk mendapatkan solusi masalah sebenarnya secara utuh.

Algoritma *divide and conquer* pada umumnya diimplementasi secara rekursif. Alasan dilakukannya pemrosesan masalah secara rekursif adalah karena submasalah pada dasarnya memiliki jenis yang sama dengan masalah yang sebenarnya. Hal yang membedakan submasalah dengan masalah yang sebenarnya adalah ukuran instansiasinya.



Sumber: https://www.tutorialspoint.com/data_structures_algorithms

Gambar 2.3. Alur penyelesaian masalah dalam algoritma *divide and conquer*

Secara umum, kompleksitas waktu algoritma D&C untuk menyelesaikan suatu masalah dapat dinyatakan sebagai fungsi menaik $T(n)$ yang memenuhi relasi rekurens sebagai berikut

$$T(n) = aT\left(\frac{n}{b}\right) + cn^d$$

dimana $n = b^k, k \in \mathbb{Z}^+, a \geq 1, b \geq 2, c \geq 0, d \geq 0$ dan $a, b, c, d \in \mathbb{R}$.

Berdasarkan teorema master, dapat dibuktikan bahwa untuk berntuk di atas berlaku bahwa

$$T(n) \text{ adalah } \begin{cases} O(n^d) \text{ jika } a < b^d \\ O(n^d \log n) \text{ jika } a = b^d \\ O\left(n^{\log_b a}\right) \text{ jika } a > b^d \end{cases}.$$

D. Quickhull

Quickhull merupakan salah satu metode yang dapat digunakan untuk menyelesaikan permasalahan *convex hull*. *Quickhull* berpedoman pada strategi *divide and conquer* dalam implementasinya. Adapun langkah-langkah penyelesaian permasalahan *convex hull* dalam algoritma ini adalah sebagai berikut:

1. Dinyatakan H adalah himpunan titik dengan

   $$H = \{p_1, p_2, p_3, \ldots, p_n\}$$

   dimana indeks setiap titik bersesuaian dengan urutan masing-masing titik secara menaik berdasarkan absisnya. Diberikan pula himpunan titik S yang disiapkan untuk menampung solusi, dan diinisiasi sebagai himpunan kosong.

2. Garis pembagi dibentuk dari dua titik terluar, yaitu titik yang berada di posisi paling kiri, dan titik yang berada di posisi paling kanan. Dalam hal ini, titik terkiri adalah p_1 dan titik terkanan adalah p_n. Garis ini dinamai sebagai $\overrightarrow{p_1 p_n}$. p_1 dan p_n ditambahkan ke S, karena titik terluar pasti merupakan anggota dari himpunan solusi *convex hull*. Dengan demikian, garis $\overrightarrow{p_1 p_n}$ akan membagi H menjadi 2 subhimpunan titik, yaitu H_1 (titik-titik pada H yang dapat ditemukan melalui penelusuran p_n ke p_1 secara berlawanan arah jarum jam) dan H_2 (titik-titik pada H yang dapat ditemukan melalui penelusuran p_n ke p_1 secara searah jarum jam).

3. Untuk H_1, dicari titik yang memiliki jarak terjauh dengan garis $\overrightarrow{p_1 p_n}$. Misalkan titik terjauh tersebut adalah C, makan tambahkan C ke S. Dengan demikian, terdapat dua subhimpunan baru, yaitu himpunan titik-titik yang ada di dalam segitiga $p_1 C p_n$, dan titik-titik yang berada di luar segitiga tersebut. Dengan mengabaikan titik-titik yang ada di dalam segitiga, dilakukan hal yang sama dengan sebelumnya terhadap titik-titik yang ada di luar segitiga untuk garis $\overrightarrow{p_1 C}$ maupun $\overrightarrow{p_n C}$.

4. Untuk H_2, dilakukan hal yang sama dengan langkah 3.

5. Diperoleh bahwa S merupakan himpunan titik-titik yang merupakan himpunan bagian dari H sedemikian sehingga elemen S merupakan titik-titik terluar dalam himpunan H yang dapat melingkupi seluruh elemen H. Dengan kata lain, *convex hull* sudah ditemukan.

Adapun *pseudo-code* untuk metode *quickhull* adalah sebagai berikut.

```
procedure quickHull(input H : himpunan_titik, output
S : himpunan_titik)
{ Menerima H sebagai masukan, dan mengembalikan S
sebagai solusi convex hull }

deklarasi
   pkiri, pkanan : titik
   H1, H2 : himpunan_titik

algoritma
   S ← {}
   pkiri ← titik terkiri dalam H
   pkanan ← titik terkanan dalam H
   bagi H menjadi H1 dan H2 berdasarkan
        garis yang dibentuk pkiri dan pkanan
   S ← S + findHull(H1, pkiri, pkanan)
        +findHull(H2, pkanan, pkiri)

function findHull(input H : himpunan_titik, input p1:
titik, input p2 : titik) → himpunan_titik
{ Menerima H, p1, dan p2 sebagai masukan, dan
mengembalikan himpunan_titik }

deklarasi
   farthest : titik
   H1, H2 : himpunan_titik

algoritma
   if H kosong then
        return {}
   else
        farthest ← titik terjauh dari garis p1 dan p2
        H1 ← himpunan titik berlawanan arah jarum jam
                (dari farthest ke p1)
        H2 ← himpunan titik searah jarum jam (dari
                farthest ke p2)
        return fartherst + findHull(H1, farthest, p1)
                + findHull(H2, farthest, p2)
   endif
```

III. Analisis dan Implementasi

A. Pemodelan

Diasumsikan bahwa sensor yang dapat mendeteksi radiasi nuklir telah dipasang pada beberapa titik secara merata di sekitar reaktor. Letak masing-masing sensor sudah tercatat pada sistem. Sensor akan segera mengirim sinyal ke sistem jika dideteksi adanya kebocoran radiasi nuklir yang sampai ke posisi sensor tersebut. Dengan demikian, dapat diketahui titik-titik mana saja yang terkena imbas kebocoran radiasi dari reaktor nuklir.

Misalkan himpunan titik H merupakan titik-titik ysng merepresentasi letak sensor-sensor yang diketahui terkena imbas kebocoran radiasi dari reaktor nuklir. Dengan demikian, daerah evakuasi minimum dari kecelakaan tersebut dapat diketahui dengan mencari *convex hull* dari himpunan H. Setelah itu, akan diperoleh titik-titik terluar dalam himpunan H yang dapat dipandang sebagai perimeter daerah evakuasi.

B. Implementasi

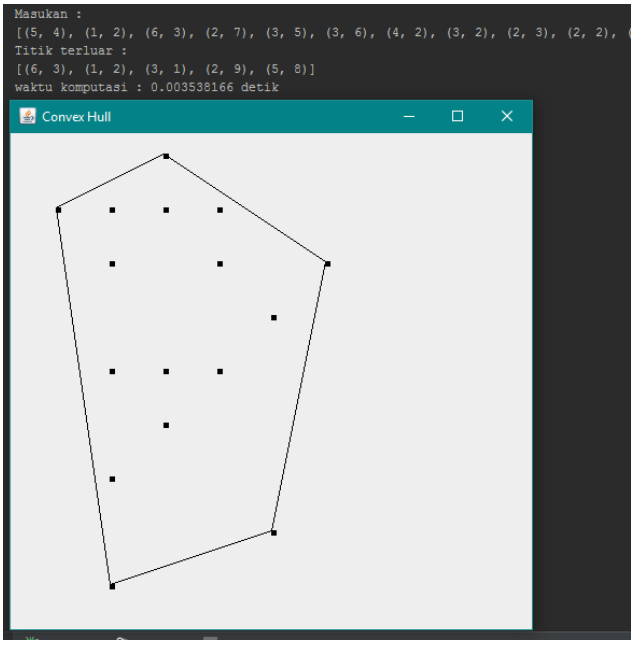
Untuk keperluan simulasi dan pengujian algoritma, dilakukan implementasi metode *quickhull* pada program yang ditulis dengan bahasa Java. Adapun bagian utama yang menunjukan alur metode *quickhull* adalah sebagai berikut.

```
...

public QuickHull(Set<Point> setOfPoint) {
    solution = new HashSet<>();

    Point max =
        new Point(Collections.max(setOfPoint));
    solution.add(max);
    Point min =
        new Point(Collections.min(setOfPoint));
    solution.add(min);

    firstSeparator = new Line(min, max);

    Set<Point> upperArea = new HashSet<>();
    Set<Point> lowerArea = new HashSet<>();
    for (Point point : setOfPoint) {
      if (!max.equalsTo(point)
        && !min.equalsTo(point)
        && !firstSeparator.inLine(point)) {
        if (firstSeparator.moreThan(point))
          upperArea.add(point);
        else
          lowerArea.add(point);
      }
    }

    solution.addAll(findHull(upperArea,
        firstSeparator, true));
    solution.addAll(findHull(lowerArea,
        firstSeparator, false));
  }

  private Set<Point> findHull(Set<Point> input,
        Line separator, boolean up) {
    if (input.isEmpty()) {
      return input;
    }
    else {
      Set<Point> retval = new HashSet<>();
      Set<Point> temp = new HashSet<>();
      double distance = 0;
      for (Point point : input) {
        distance =
        distance < separator.getDistanceFrom(point) ?
        separator.getDistanceFrom(point) : distance;
      }

      Point found = new Point(0,0);
      for (Point point : input) {
        if (distance ==
        separator.getDistanceFrom(point)) {
          found = point;
          break;
        }
      }
      if (!solution.contains(found))
        solution.add(found);

      Line separator1 =
        new Line(separator.getBegin(), found);
      for (Point point : input) {
        if (!separator1.inLine(point)) {
          if (up ? separator1.moreThan(point) :
        !separator1.moreThan(point)) {
            temp.add(point);
          }
        }
      }
      retval.addAll(temp);
      retval.addAll(findHull(temp, separator1, up));

      temp = new HashSet<>();
      Line separator2 = new Line(separator.getEnd(),
        found);
      for (Point point : input) {
        if (!separator2.inLine(point)) {
          if (up ? separator2.moreThan(point) :
        !separator2.moreThan(point)) {
            temp.add(point);
          }
        }
      }
      retval.addAll(temp);
      retval.addAll(findHull(temp, separator2, up));
      return retval;
    }
  }

...
```

Prosedur penentuan daerah evakuasi minimum, yang dalam hal ini adalah *convex hull*, dieksekusi secara rekursif. Prosedur menerima masukan berupa himpunan titik yang akan dicari perimeter terluarnya yang dapat mencakup seluruh titik pada himpunan tersebut. Setelah dilakukan proses sesuai dengan alur metode *quickhull*, keluaran dinyatakan dalam himpunan yang beranggotakan titik-titik terluar pada himpunan masukan. Setelah didapatkan himpunan titik-titik terluar, dapat ditentukan daerah evaluasi minimum.

C. Pengujian

Berikut merupakan tampilan program saat dijalankan. Program dijalankan pada Java IDE IntelliJ IDEA Community Edition 2017.1, di atas sistem operasi Windows 10 Pro 64-bit (10.0, Build 14393), pada laptop ASUS X550Z dengan 4096 MB RAM dan *processor* AMD A 10-7400P. Pada jendela program, titik pusat koordinat $(0,0)$ berada pada ujun kiri atas. Sumbu *X* positif memiliki arah dari pusat koordinat ke kanan merupakan. Sumbu *Y* positif memiliki arah dari pusat koordinat ke bawah. Hal ini berbeda dengan sistem koordinat pada umumnya. Alasan utama penggunaan sistem koordinat tersebut sebagai acuan adalah untuk mempercepat waktu komputasi karena dalam hal ini tidak diperlukan adanya transformasi.



Sumber: penulis

Gambar 3.1. Hasil pengujian program penentu *convex hull*

Masukan pada pengujian di atas dibaca dari file eksternal berformat *text*. Adapun isi dari file eksternal tersebut adalah sebagai berikut.

```
1,2
2,3
4,5
6,3
2,2
4,3
2,9
2,7
5,8
3,6
3,2
5,4
3,5
2,5
3,1
4,2
```

D. Analisis

Terdapat beberapa keperluan komputasi dalam metode *quickhull* yang menentukan kompleksitas waktunya. Adapun analisis terhadap komputasi yang dilakukan adalah sebagai berikut:

1. Kompleksitas waktu untuk mencari titik yang memiliki absis minimum dan maksimum adalah $O(n)$, karena dalam hal ini diperlukan iterasi sebanyak n kali untuk menentukan nilai absis maksimum dan minimum dari setiap titik dalam himpunan masukan. Titik dengan absis minimum dan maksimum tersebut dicari untuk membentuk garis pembagi. Setelah itu proses akan berlanjut dengan objek kumpulan titik-titik yang sudah terbagi.
2. Misalkan $T(n)$ adalah kompleksitas waktu untuk menentukan *convex hull* dari suatu himpunan titik. Dengan demikian, kompleksitas waktu untuk memproses setengah dari kumpulan titik awal adalah $T\left(\frac{n}{2}\right)$.
3. Setiap terjadi pembagian, perlu dilakukan 2 kali komputasi karena garis pembagi membagi kumpulan titik pada himpunan menjadi 2 bagian. Dalam hal ini di asumsikan kedua bagian tersebut memiliki ukuran yang sama sehingga komputasi yang diperlukan oleh keduanya sama.

Dengan demikian, kompleksitas waktu metode *quickhull* pada kasus rata-rata dapat dinyatakan dalam persamaan rekursif sebagai berikut:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n).$$

Jika persamaan di atas dianalogikan dengan persamaan

$$T(n) = aT\left(\frac{n}{b}\right) + cn^d,$$

maka akan diperoleh bahwa $a = 2$, $b = 2$, dan $d = 1$, sehingga $a = b^d$. Sesuai dengan teorema master, notasi *Big-O* untuk metode *quickhull* adalah $O(n \log n)$.

IV. Simpulan

Divide and Conquer (D&C) merupakan starategi umum yang dapat diterapkan secara luas dan cukup efisien. Metode *quickhull* merupakan pernerapan dari D&C. *Quickhull* merupakan metode untuk menentukan *convex hull* dari sebuah himpunan titik dengan kompleksitas waktu $O(n \log n)$ pada kasus rata-rata. Prinsip-prinsip yang digunakan dalam menentukan *convex hull* dapat diterapkan untuk menentukan daerah evakuasi minimum di sekitar reaktor nuklir yang mengalami kebocoran. Dengan mengetahui titik mana saja yang terkena imbas kebocoran, dapat ditentukan daerah evakuasi minimumnya dengan metode *quickhull*.

Ucapan Terima Kasih

Penulis bersyukur kepada Tuhan berkat anugerah dan karunia yang dilimpahkan sehingga penulis dapat menyelesaikan makalah ini tanpa hambatan yang berarti. Ucapan terima kasih juga penulis sampaikan kepada Bapak Rinaldi Munir, Ibu Masayu Leylia Khodra, dan Ibu Nur Ulfa

Maulidevi atas bimbingannya dalam matakuliah IF2121 Strategi Algoritma.

REFERENSI

[1] https://www.quora.com/What-are-the-real-life-applications-of-convex-hulls diakses pada tanggal 13 April 2017 pukul 10.14

[2] http://www.csl.mtu.edu/cs4321/www/Lectures/Lecture%209%20-%20Divide%20and%20Conquer%20Closest%20Pair%20and%20Convex.htm diakses pada tanggal 13 April 2017 pukul 10.28

[3] http://dreuarchive.cra.org/2009/Jaramillo/DREU/files/final_report.pdf diakses pada tanggal 13 April 2017 pukul 10.43

[4] http://www.nnr.co.za/what-is-nuclear-energy/ diakses pada tanggal 13 April 2017 pukul 11.05

[5] https://www.theguardian.com/environment/2017/feb/03/fukushima-daiichi-radiation-levels-highest-since-2011-meltdown diakses pada tanggal 13 April 2017 pukul 11.17

[6] http://www.cse.yorku.ca/~aaw/Hang/quick_hull/Algorithm.html diakses pada tanggal 13 April 2017 pukul 23.19

[7] Benjamin K. Sovacool (2009). "*The Accidental Century - Prominent Energy Accidents in the Last 100 Years*"

[8] Barber, C. Bradford et al (1996). "*The quickhull algorithm for convex hulls*"

[9] Anany Levitin (2012), *Introduction to: The Design and Analysis of Algorithms*

[10] Rinaldi Munir (2017), *Slide* Kuliah

[11] Rinaldi Munir (2017), Diktat Kuliah

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bansung, 13 Mei 2017

Paskahlis Anjas Prabowo
13515108

LAMPIRAN

Kode Program

1. File "Point.java"

```
package entity;

/**
 * Created by Paskahlis Anjas Prabowo on 14/05/2017.
 */
public class Point implements Comparable<Point> {
  private int absis;
  private int ordinat;

  public Point(int absis, int ordinat) {
    this.absis = absis;
    this.ordinat = ordinat;
  }

  public Point(Point point) {
    absis = point.absis;
    ordinat = point.ordinat;
  }

  public int getAbsis() {
    return absis;
  }

  public int getOrdinat() {
    return ordinat;
  }

  @Override
  public int compareTo(Point point) {
    return Double.compare(absis, point.absis);
  }

  @Override
  public String toString() {
    return "(" + absis + ", " + ordinat + ")";
  }

  public boolean equalsTo(Point point) {
    return absis == point.absis && ordinat
        == point.ordinat;
  }

  public static Point stringToPoint(String input) {
    String[] arrstring = input.split(",");
    return new Point(Integer.parseInt(arrstring[0]),
         Integer.parseInt(arrstring[1]));
  }
}
```

2. File "Line.java"

```
package entity;

/**
 * Created by Paskahlis Anjas Prabowo on 14/05/2017.
 */
public class Line {
  private Point begin;
  private Point end;
  private double verticalIntersection;
  private double slop;

  public Line(Point start, Point finish) {
    begin = new Point(start);
    end = new Point(finish);
   slop = (double)(end.getOrdinat()
```

```
        - begin.getOrdinat())/(end.getAbsis()
        - begin.getAbsis());
    verticalIntersection = begin.getOrdinat() -
        (slop * begin.getAbsis());
  }

  public Line(Line line) {
    begin = new Point(line.begin);
    end = new Point(line.end);
    slop = line.slop;
    verticalIntersection =
        line.verticalIntersection;
  }


  public double getSlop() {
    return slop;
  }

  public double getVerticalIntersection() {
    return verticalIntersection;
  }

  public Point getBegin() {
    return begin;
  }

  public Point getEnd() {
    return end;
  }

  public boolean inLine(Point point) {
    return slop * point.getAbsis()
        + verticalIntersection
        - point.getOrdinat() == 0;
  }

  public boolean moreThan(Point point) {
    return slop * point.getAbsis()
        + verticalIntersection
        - point.getOrdinat() < 0;
  }

  public double getDistanceFrom(Point point) {
    return Math.abs(slop * point.getAbsis()
        - point.getOrdinat()
        + verticalIntersection)
        /Math.sqrt(slop * slop + 1);
  }
}
```

3. File “QuickHull.java”

```
package quickhull;

import entity.Line;
import entity.Point;
import java.util.Collections;
import java.util.HashSet;
import java.util.Set;

/**
 * Created by Paskahlis Anjas Prabowo on 14/05/2017.
 */
public class QuickHull {
  private  Set<Point> solution;
  private Line firstSeparator;

  public QuickHull(Set<Point> setOfPoint) {
    solution = new HashSet<>();

    Point max =
         new Point(Collections.max(setOfPoint));
    solution.add(max);
    Point min =
        new Point(Collections.min(setOfPoint));
    solution.add(min);

    firstSeparator = new Line(min, max);

    Set<Point> upperArea = new HashSet<>();
    Set<Point> lowerArea = new HashSet<>();
    for (Point point : setOfPoint) {
      if (!max.equalsTo(point)
        && !min.equalsTo(point)
        && !firstSeparator.inLine(point)) {
        if (firstSeparator.moreThan(point))
          upperArea.add(point);
        else
          lowerArea.add(point);
      }
    }

    solution.addAll(findHull(upperArea,
firstSeparator, true));
    solution.addAll(findHull(lowerArea,
firstSeparator, false));
  }

  private Set<Point> findHull(Set<Point> input, Line
separator, boolean up) {
    if (input.isEmpty()) {
      return input;
    }
    else {
      Set<Point> retval = new HashSet<>();
      Set<Point> temp = new HashSet<>();
      double distance = 0;
      for (Point point : input) {
        distance = distance
        < separator.getDistanceFrom(point)
        ? separator.getDistanceFrom(point)
        : distance;
      }
```

```
      Point found = new Point(0,0);
      for (Point point : input) {
        if (distance
        == separator.getDistanceFrom(point)) {
          found = point;
          break;
        }
      }
      if (!solution.contains(found))
         solution.add(found);

      Line separator1 =
        new Line(separator.getBegin(), found);
      for (Point point : input) {
        if (!separator1.inLine(point)) {
          if (up ? separator1.moreThan(point) :
        !separator1.moreThan(point)) {
            temp.add(point);
          }
        }
      }
      retval.addAll(temp);
      retval.addAll(findHull(temp, separator1, up));

      temp = new HashSet<>();
      Line separator2 = new Line(separator.getEnd(),
        found);
      for (Point point : input) {
        if (!separator2.inLine(point)) {
          if (up ? separator2.moreThan(point) :
        !separator2.moreThan(point)) {
            temp.add(point);
          }
        }
      }
      retval.addAll(temp);
      retval.addAll(findHull(temp, separator2, up));
      return retval;
    }
  }

  public Line getFirstSeparator() {
        return firstSeparator; }

  public Set<Point> getSolution() {
    return solution;
  }
}
```

4. File “Main.java”

```
import entity.Line;
import entity.Point;
import quickhull.QuickHull;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.HashSet;
import java.util.Scanner;
import java.util.Set;

/**
 * Created by Paskahlis Anjas Prabowo on 14/05/2017.
 */
public class Main {
  public static void main(String[] args) {
    Set<Point> set = new HashSet<>();

    Scanner input = null;
    try {
      input = new Scanner(new File("input.txt"));
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    }
    while (input.hasNext()) {
      String line = input.nextLine();
      set.add(Point.stringToPoint(line));
    }
    System.out.println("Masukan : \n" + set);

    double start = System.nanoTime();
    QuickHull qh = new QuickHull(set);
    double elapsedTime =
        (System.nanoTime()-start)/1000000000;
    System.out.println("Titik terluar :\n"
        + qh.getSolution());
    System.out.println("waktu komputasi : "
        + elapsedTime + " detik");

    View view = new View(set,
        qh.getSolution(), qh.getFirstSeparator());
  }
}
```

5. File “View.java”

```
import entity.Line;
import entity.Point;

import javax.swing.*;
import java.awt.*;
import java.util.Collections;
import java.util.HashSet;
import java.util.Set;

/**
 * Created by Paskahlis Anjas Prabowo on 15/05/2017.
 */
public class View extends JFrame {
  private Set<Point> inputSet;
  private Set<Point> solutionSet;
  private Line separator;

  public View(Set<Point> inputSet,
        Set<Point> solutionSet, Line separator) {
    this.inputSet = new HashSet<>(inputSet);
```

```
    this.solutionSet = new HashSet<>(solutionSet);
    this.separator = new Line(separator);

    setTitle("Convex Hull");
    setPreferredSize(new Dimension(500,500));
    pack();
    setVisible(true);
    setDefaultCloseOperation(WindowConstants.
        EXIT_ON_CLOSE);
  }

  @Override
  public void paint(Graphics graphics) {
    super.paint(graphics);
    for (Point point : inputSet) {
      int x = point.getAbsis() * 50;
      int y = point.getOrdinat() * 50;
      graphics.fillRect(x, y, 5, 5);
    }
    int[] x = new int[solutionSet.size()];
    int[] y = new int[solutionSet.size()];
    x[0] = separator.getBegin().getAbsis() * 50;
    y[0] = separator.getBegin().getOrdinat() * 50;
    int index = 1;
    for (Point point : solutionSet) {
      if (separator.moreThan(point)) {
        x[index] = point.getAbsis() * 50;
        y[index] = point.getOrdinat() * 50;
        index++;
      }
    }
    x[index] = separator.getEnd().getAbsis() * 50;
    y[index] = separator.getEnd().getOrdinat() * 50;
    index++;
    for (Point point : solutionSet) {
      if (!separator.moreThan(point)
        && !separator.inLine(point)) {
        x[index] = point.getAbsis() * 50;
        y[index] = point.getOrdinat() * 50;
        index++;
      }
    }
   graphics.drawPolygon(x, y, solutionSet.size());
 }
}
```