UNIVERSITÀ DEGLI STUDI
DI PERUGIA

Tesina finale di

Algoritmi e Strutture Dati

Corso di Laurea in Ingegneria Informatica ed Elettronica

DIPARTIMENTO DI INGEGNERIA

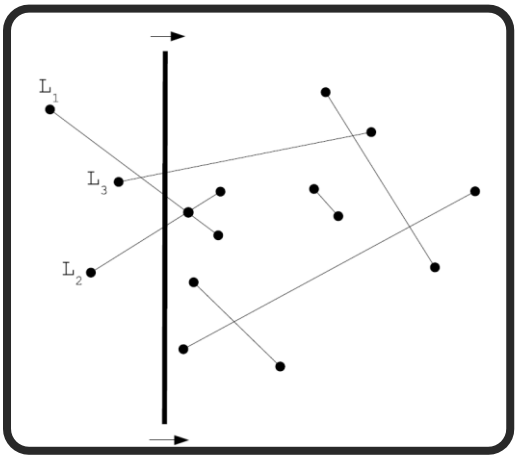
A.A. 2021-2022

Docente

Prof. Emilio Di Giacomo

Segment Intersection

Implementazione algoritmi per il calcolo di intersezioni tra segmenti nel piano



Studente

321306 Leonardo Ignazio Pagliochini leonardoignazio.pagliochini@studenti.unipg.it

325971 Andrea Bobò andrea.bobo@studenti.unipg.it

Sommario

1)Descrizione del problema

Il progetto sviluppato si propone lo scopo di analizzare il problema della rilevazione di intersezioni tra segmenti di un piano e di realizzare un'implementazione di una soluzione attraverso il linguaggio di programmazione Java.

Il problema della rilevazione delle intersezioni appartiene alla categoria dei problemi di Geometria Computazionale: la branca dell'informatica che studia algoritmi per risolvere problemi geometrici. L'input di questi problemi è una descrizione di un insieme di oggetti geometrici (es: insieme di punti o di segmenti) e come output una risposta ad una domanda sugli oggetti.

Vedremo algoritmi di geometria computazionale su due dimensioni, ovvero nel piano. Ogni oggetto in input è rappresentato da un insieme di punti.

Abbiamo studiato due soluzioni: la prima prevede lo studio di tutte le possibili coppie di segmenti per verificare, a due a due, la loro intersezione, ma tale metodo risulta inefficiente a livello di complessità rispetto al metodo basato su un approccio *Sweep Line*, il secondo algoritmo che andremo a studiare.

1.1)Intersezione

Prima di tutto riportiamo le condizioni che si devono verificare nel caso in cui due segmenti si intersechino in quanto questa sarà alla base dell'implementazione di entrambi gli algoritmi presenti nel progetto.

L'analisi si articola in 2 passaggi:

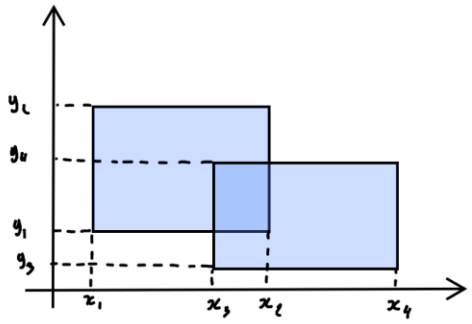
- Rifiuto immediato (i segmenti non possono intersecarsi se i loro rettangoli delimitanti non si intersecano)
- Analisi dell'"inforcazione".

<u>Rifiuto immediato</u>

Il rettangolo delimitante di un segmento è il più piccolo rettangolo che contiene il segmento e i cui lati sono paralleli agli assi *x* ed *y*, quindi nel caso di un segmento $\overline{p_1 p_2}$ è rappresentato dal rettangolo $< \widehat{p_1}, \widehat{p_2} >$ con vertice in basso a sinistra $\widehat{p_1} = (\widehat{x_1}, \widehat{y_1})$ e vertice in alto a destra $\widehat{p_2} = (\widehat{x_2}, \widehat{y_2})$ con $\widehat{x_1} = \min(x_1, x_2)$, $\widehat{y_1} = \min(y_1, y_2)$, $\widehat{x_2} = \max(x_1, x_2)$ e $\widehat{y_2} = \max(y_1, y_2)$.

Due rettangoli si intersecano se e solo se la condizione:

$$(\widehat{x_2} \geq \widehat{x_3}) \wedge (\widehat{x_4} \geq \widehat{x_1}) \wedge (\widehat{y_2} \geq \widehat{y_3}) \wedge (\widehat{y_4} \geq \widehat{y_1})$$



Analisi dell' "inforcazione"

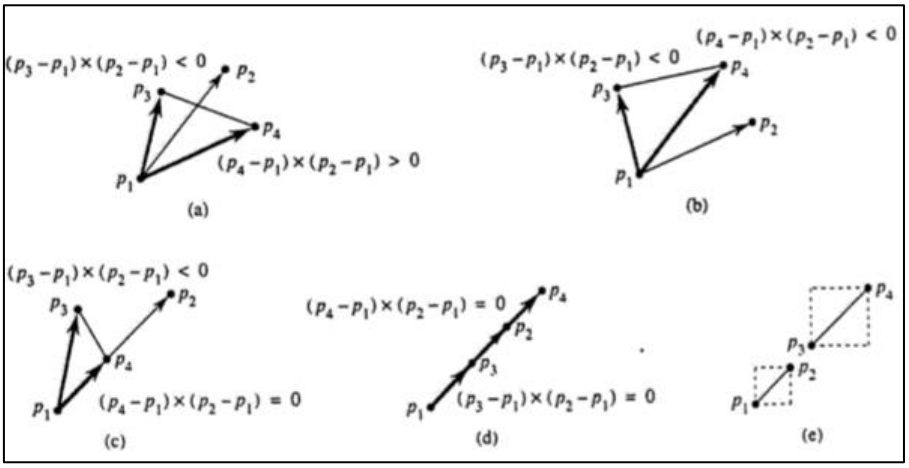
Questo passaggio per determinare se due segmenti si intersecano rileva se ogni segmento "inforca" la retta che contiene l'altro. Un segmento $\overline{p_1p_2}$ "inforca" una retta se il punto p_1 si trova da un lato della retta e il punto $p_2$si trova dall'altro lato. Se p_1 o p_2 si trovano sulla stessa retta, allora si dice che il segmento inforca la retta. Si può utilizzare il metodo dei prodotti incrociati per verificare se il segmento $\overline{p_3p_4}$ inforca la retta contenente i punti p_1e p_2.

L'idea è di determinare se i segmenti orientati $\overrightarrow{p_1p_3}$ e $\overrightarrow{p_1p_4}$ hanno direzioni opposte rispetto a $\overline{p_1p_2}$.

Per determinare le direzioni basta controllare se i segni dei prodotti $(p_3 - p_1)x(p_2 - p_1)$ e $(p_4 - p_1)$x$(p_2 - p_1)$ sono diversi.

Si verifica una condizione limite se:

- uno dei due prodotti è uguale a zero(in questo caso uno tra $p_3 e\ p_4$ si trova sulla retta che contiene il segmento $\overline{p_1p_2}$);
- uno o entrambi i segmenti hanno lunghezza nulla(se solo uno, per esempio $\overline{p_3p_4}$, i segmenti si intersecano solo se $(p_3 - p_1)$x$(p_2 - p_1)$ è zero)



Il seguente pseudocodice implementa l'idea dell'algoritmo:

SEGMENT-INTERSECT: stampa TRUE se i segmenti $\overline{p_1p_2}$ e $\overline{p_3p_4}$ si intersecano e FALSE in caso contrario;

```
SEGMENTS-INTERSECT(p1, p2, p3, p4)
 1  d1 = DIRECTION(p3, p4, p1)
 2  d2 = DIRECTION(p3, p4, p2)
 3  d3 = DIRECTION(p1, p2, p3)
 4  d4 = DIRECTION(p1, p2, p4)
 5  if ((d1 > 0 and d2 < 0) or (d1 < 0 and d2 > 0)) and
        ((d3 > 0 and d4 < 0) or (d3 < 0 and d4 > 0))
 6      return TRUE
 7  elseif d1 == 0 and ON-SEGMENT(p3, p4, p1)
 8      return TRUE
 9  elseif d2 == 0 and ON-SEGMENT(p3, p4, p2)
10      return TRUE
11  elseif d3 == 0 and ON-SEGMENT(p1, p2, p3)
12      return TRUE
13  elseif d4 == 0 and ON-SEGMENT(p1, p2, p4)
14      return TRUE
15  else return FALSE
```

Nella sua implementazione lo pseudocodice si avvale di altre 2 funzioni:

- DIRECTION
- ON-SEGMENT

Le quali possono essere implementate come segue:

```
DIRECTION(pi, pj, pk)
1  return (pk − pi) × (pj − pi)

ON-SEGMENT(pi, pj, pk)
1  if min(xi, xj) ≤ xk ≤ max(xi, xj) and min(yi, yj) ≤ yk ≤ max(yi, yj)
2      return TRUE
3  else return FALSE
```

Le righe 1-4 calcolano le direzioni per ogni punto caratterizzante dei due segmenti (punto di inizio e punto di fine) in modo da poterle confrontare successivamente.

Nel caso in cui tutti i valori calcolati siano diversi da zero, si procede al confronto (riga 5) e, nel caso in cui i segni di *d1* e *d2* e quelli di *d3* e *d4* siano opposti l'algoritmo ritorna *true* alla riga 6.

Nel caso in cui uno o più di questi valori sia nullo ci si trova nel caso limite per cui il punto che stiamo valutando è collineare con il segmento. (Se d_1 è pari a zero, allora il punto p1 è collineare con il segmento $\overline{p_3 p_4}$)

Allora si utilizza il metodo ON SEGMENT per valutare se questo sia o meno contenuto all'interno del segmento.

Nel caso in cui questo si verifichi, alle righe 8,10,12,14 l'algoritmo ritorna *true*.

Se durante tutta l'esecuzione l'algoritmo non è "entrato" in nessuna istruzione *if*, alla riga 15 termina l'esecuzione dando come risultato *false*.

2)Algoritmi utilizzati

Nella nostra analisi abbiamo utilizzato due diversi algoritmi in modo da poterne confrontare la complessità.

Entrambi gli algoritmi studiano la presenza dell'intersezione avvalendosi del criterio definito al capitolo precedente, ma la loro differenza risiede nel fatto che:

- *CoupleIntersection* valuta tutte le possibili combinazioni di coppie di segmenti per studiarne l'intersezione
- *Sweepline* valuta solo segmenti appartenenti alla stessa "regione di piano" per cui è quindi sensato supporne l'intersezione.

2.1) CoupleIntersection

L'algoritmo *coupleIntersection* prevede l'utilizzo di 2 cicli for per considerare tutte le possibili coppie di segmenti e valutarne l'intersezione.

Inserendo come input un pool di segmenti, l'algoritmo restituisce *true* se trova un'intersezione e *false* nel caso contrario.

Di seguito troviamo lo pseudocodice:

```
1     sa = array di segmenti
2     for i = 0 to i = numerosegmenti
3           for j = i+1 to j = numerosegmenti
4                 if sa[i] interseca sa[j]
5                       return true
6     return false
```

Alla riga 1 abbiamo un'assegnazione mentre alle righe 1-2 vediamo che il codice è formato da due cicli for annidati che scandiscono tutti gli elementi dell'array, a coppie di segmenti. Alla riga 4 la condizione if permette di controllare se i segmenti sono intersecati o no; se lo sono il codice restituisce true (riga 5), al contrario restituisce false.

2.2) Sweepline

L'algoritmo in questione utilizza una tecnica conosciuta come "rastrellamento" che è comune a molti problemi di geometria computazionale.

Nel rastrellamento un'immaginaria retta verticale, detta "retta di rastrellamento"("SweepLine" in inglese), passa attraverso l'insieme degli oggetti geometrici ordinati da sinistra a destra spostandosi lungo la dimensione x.

Il rastrellamento prevede un metodo per riordinare oggetti geometrici inserendoli in una struttura dati dinamica e sfruttando le relazioni tra di essi.

Gli algoritmi "Sweepline" sfruttano il concetto di *punti di arresto* per definire gli stati della retta di rastrellamento per cui l'algoritmo entra in funzione.

L'algoritmo, per rilevare le intersezioni tra segmenti, considera tutti gli estremi dei segmenti come punti di arresto e, quindi, punti in cui viene svolta l'analisi delle intersezioni.

Per semplificare l'algoritmo vengono fatte due assunzioni:

1. nessun segmento in input è verticale;
2. non esistono tre segmenti che si intersecano in uno solo punto;

Dato che non vi sono segmenti verticali, qualunque segmento in input interseca al più una sola volta la "sweepline" in un certo stato.

Prendendo due segmenti $s_1 s_2$, si dice che questi sono confrontabili in x se la retta di rastrellamento con ascissa x li interseca entrambi. Si dice che s_1è sopra s_2 e si scrive $s_1 >_x s_2$ se s_1 e s_2 sono confrontabili in x e l'intersezione di $s_1$con la retta di ascissa x è più alta dell'intersezione di s_2.

Data un'ascissa qualsiasi a la relazione "$>_a$" impone un ordinamento totale[1] sui segmenti che intersecano la "sweepline" in x.

L'algoritmo in questione gestisce 2 insiemi di dati:

- La struttura dati contenente i punti di arresto, la quale deve essere ordinata.
- La struttura dati contenente i segmenti (T)

Ogni segmento viene inserito nella base dati quando la sweepline "incontra" l'estremo sinistro e viene cancellato quando quest'ultima "incontra" il suo estremo destro.

Una volta definito un ordinamento totale per i segmenti, si costruisce la base dati T in modo che questa possa implementare le seguenti operazioni:

- INSERT (T,s) che permette di inserire il segmento s nella base dati
- DELETE (T,s) che permette di eliminare il segmento s dalla base dati
- ABOVE (T,s) che permette di ottenere l'elemento immediatamente superiore ad s nell'ordinamento
- BELOW (T,s) che permette di ottenere l'elemento immediatamente inferiore ad s nell'ordinamento

Di seguito troviamo l'implementazione dell'algoritmo che prende in input n segmenti e restituisce *true* se trova un'intersezione e *false* in caso contrario.

[1] $ordinamento\ totale$: è una relazione binaria su un insieme X che è riflessiva, antisimmetrica, transitiva e totale.

```
ANY-SEGMENTS-INTERSECT(S)
 1  T = ∅
 2  sort the endpoints of the segments in S from left to right,
        breaking ties by putting left endpoints before right endpoints
        and breaking further ties by putting points with lower
        y-coordinates first
 3  for each point p in the sorted list of endpoints
 4      if p is the left endpoint of a segment s
 5          INSERT(T, s)
 6          if (ABOVE(T, s) exists and intersects s)
                or (BELOW(T, s) exists and intersects s)
 7              return TRUE
 8      if p is the right endpoint of a segment s
 9          if both ABOVE(T, s) and BELOW(T, s) exist
                and ABOVE(T, s) intersects BELOW(T, s)
10              return TRUE
11          DELETE(T, s)
12  return FALSE
```

Alla riga 1 inizializziamo l'ordinamento totale a vuoto. Alla linea 2 determiniamo i 2n punti di arresto attraverso l'algoritmo di Quick Sort, che vedremo in seguito.

Ogni iterazione del ciclo alle linee 3-11 considera un punto di arresto p: se p è l'estremo sinistro di un segmento s, alla linea 5 si aggiunge s all'ordinamento totale e alle linee 6-7 è restituito true se s interseca uno dei due segmenti ad esso consecutivi rispetto all'ordinamento totale T, che passa per il punto p.

Se p è l'estremo destro di un segmento x, allora s deve essere cancellato dall'ordinamento totale. Alle linee 9-10 è restituito *true* se vi è un'intersezione tra i segmenti che sono accanto a s nell'ordinamento totale in quanto questi segmenti diventeranno consecutivi in T quando s sarà cancellato. Se non si intersecano la linea 11 cancella il segmento s da T. Infine se non si trova nessun'intersezione la linea 12 restituisce false.

QuickSort

All'interno dell'algoritmo "sweepline" viene utilizzato l'algoritmo QuickSort per l'ordinamento dei punti d'arresto, il cui pseudocodice è riportato di seguito:

```
QUICKSORT(A,p,r)
1  if p<r                    //se ci sono almeno 2 elem
2      q=PARTITION(A,p,r)    // divide
3      QUICKSORT (A,p,q-1)   // impera
4      QUICKSORT (A,q+1,r)   // impera
```

Quick Sort è un algoritmo basato sulla tecnica *divide et impera.* Dalla linea 1 vediamo subito che gli elementi dentro l'array A devono essere almeno due per far partire l'algoritmo. In seguito, se la condizione viene verificata, Quick Sort prevede la scelta di un elemento dell'array, detto PIVOT, che ci permette di dividere l'array in input in due sotto-array: uno con elementi maggiori del pivot e uno con elementi minori.

Solitamente un algoritmo basato sulla tecnica del *divide et impera* si articola in tre fasi:
Divide, Impera e Combina

• *Divide*: l'array A[p,…,r] viene suddiviso in due sotto Array A[p…q-1] e A[q+1…r](eventualmente vuoti), in modo che ogni elemento del primo array sia minore di A[q](PIVOT) e ogni elemento del secondo array sia maggiore di A[q];

Questa fase viene eseguita attraverso la chiamata di PARTITION di cui analizzeremo in seguito lo pseudocodice.

• *Impera*: due sotto Array A[p…q-1] e A[q+1…r] sono ordinati, ricorsivamente;

In questo caso non ci sarà la fase di Combina dato che gli elementi sono già ordinati.

	PARTITION(*A*,*p*,*r*)
1	*x*=*A*[*r*]
2	*i*=*p*-1
3	**for** *j*=*p* **to** *r*-1
4	**if** *A*[*j*]≤*x*
5	*i*=*i*+1
6	scambia *A*[*i*] e *A*[*j*]
7	scambia *A*[*i*+1] e *A*[*r*]
8	**return** *i*+1

La classe PARTITION è composta da due assegnazioni alle righe 1-2, mentre alla riga 3 troviamo un ciclo for. Esso scansiona gli elementi dell'array A e attraverso la condizione alla riga 4 i valori vengono spostati nei due sottoArray sopra citati, ricorsivamente.

La complessità di Partition è $\theta(n)$, dato che il ciclo for è ripetuto n=r-p+1 volte.

3)Analisi della Complessità

Si riportano di seguito gli pseudocodici dei due algoritmi, Couple Intersection & Sweep Line, di cui studieremo la complessità asintotica.

3.1) CoupleIntersection

Il primo ha il seguente pseudocodice:

```
1     sa = array di segmenti
2     for i = 0 to i = numerosegmenti
3           for j = i+1 to j = numerosegmenti
4                 if sa[i] interseca sa[j]
5                       return true
6     return false
```

L'algoritmo si compone di due cicli for principali annidati. Il primo ciclo, alla riga 2, scansiona tutti i segmenti dell'array così come il secondo ciclo for, riga 3. Questo ci permette di controllare l'intersezione dei segmenti analizzando tutte le coppie possibili. L'assegnazione alla riga 1 ha costo costante O(1), mentre l'istruzione dominante alla riga 4 ha costo O(n^2). La complessità totale sarà O(n^2).

3.2) Sweepline

La SweepLine, come abbiamo visto prima, ha il seguente pseudocodice:

```
ANY-SEGMENTS-INTERSECT(S)
 1  T = ∅
 2  sort the endpoints of the segments in S from left to right,
        breaking ties by putting left endpoints before right endpoints
        and breaking further ties by putting points with lower
        y-coordinates first
 3  for each point p in the sorted list of endpoints
 4      if p is the left endpoint of a segment s
 5          INSERT(T, s)
 6          if (ABOVE(T, s) exists and intersects s)
                or (BELOW(T, s) exists and intersects s)
 7              return TRUE
 8      if p is the right endpoint of a segment s
 9          if both ABOVE(T, s) and BELOW(T, s) exist
                and ABOVE(T, s) intersects BELOW(T, s)
10              return TRUE
11          DELETE(T, s)
12  return FALSE
```

Se vi sono n segmenti nell'insieme S, allora ANY-SEGMENT-INTERSECT impiega tempo O(n*lg n). La riga 1 richiede tempo O(1), la riga 2 richiede tempo O(n*lg n) utilizzando il Quick Sort (in quanto si procede all'analisi del caso medio essendo improbabile che venga scelto come pivot il valore minimo o massimo, essendo l'array generato randomicamente). Poiché vi sono 2n punti di arresto il for alle righe 3-11 si ripete al più 2n volte. Ogni iterazione richiede tempo O(n*lg n), poiché ogni operazione su un RB-Tree richiede tempo O(lg n) e, ogni controllo di intersezione richiederà tempi O(1). Il tempo totale è O(n*lg n).

4)Strutture Dati

Nella realizzazione del codice è stato necessario impiegare due basi dati:

- Un array per contenere i dati relativi ai punti di arresto
- Un albero rossonero per contenere i dati relativi ai segmenti

Entrambe le basi di dati sono state utilizzate nell'algoritmo "sweepline".

La scelta dell'array per i dati dei punti d'arresto è stata fatta in quanto si tratta di una struttura dati molto facile da implementare e che risulta sconveniente solo nel caso in cui non si sappia fin dall'inizio il numero di elementi che andrà a contenere.

Nel nostro caso, essendo noto il numero dei segmenti, era noto anche il numero dei punti d'arresto pari a due volte il numero di segmenti.

Questo ci ha permesso di utilizzare l'algoritmo di ordinamento *quickSort* senza difficoltà.

Per quanto riguarda la base dati relativa ai segmenti, la scelta è ricaduta sugli alberi rosso neri in quanto erano già presenti le funzioni *insert, delete, above, below* presenti nello pseudocodice rendendo quindi necessario solo definire un criterio di ordinamento tra segmenti.

5)Implementazione

Segue una breve descrizione di come siano stati implementati i due algoritmi.

Prima di tutto sono state definite le 2 classi *Point e Segment.*

All'interno della classe *segment* è stata implementata la funzione *segmentIntersect* che traduce in java lo pseudocodice riportato al capitolo 1.1 avvalendosi delle due funzioni *direction* e *onSegment* presenti all'interno della classe ComputationalGeometryUtils.

```java
public class Point {
    private double x;
    private double y;

    public Point(double x, double y) {...}

    public double getX() { return this.x; }

    public void setX(double x) { this.x = x; }

    public double getY() { return this.y; }

    public void setY(double y) { this.y = y; }
}
```

```java
public class Segment {
    private Point p1;
    private Point p2;


    public Segment(Point p1,Point p2)
    {...}

    public Point getP1() { return this.p1; }

    public Point getP2() { return this.p2; }

    public double getX1() { return this.p1.getX(); }

    public void setX1(double x1) { this.p1.setX(x1);}

    public double getX2() { return this.p2.getX(); }

    public void setX2(double x2) { this.p2.setX(x2);}

    public double getY1() { return this.p1.getY(); }

    public void setY1(double y1) { this.p1.setX(y1);}

    public double getY2() { return this.p2.getY(); }

    public void setY2(double y2) { this.p2.setX(y2);}

    public static boolean segmentIntersect(Segment s1 , Segment s2)
    {...}

    public double lineIntersection(double x)
    {...}

    public static Segment[] segmentArrayFromSequence(Double[][] seq)
    {...}
}
```

```java
public static boolean segmentIntersect(Segment s1 , Segment s2)
{
    Point p1 = s1.getP1();
    Point p2 = s1.getP2();
    Point p3 = s2.getP1();
    Point p4 = s2.getP2();

   double d1 = ComputationalGeometryUtils.direction(p3,p4,p1);
   double d2 = ComputationalGeometryUtils.direction(p3,p4,p2);
   double d3 = ComputationalGeometryUtils.direction(p1,p2,p3);
   double d4 = ComputationalGeometryUtils.direction(p1,p2,p4);

   if((d1>0 && d2<0 || d1<0 && d2>0) && (d3>0 && d4<0 || d3<0 && d4>0))
        return true;
   if (d1 == 0 && ComputationalGeometryUtils.onSegment(s2.getP1(),s2.getP2(),s1.getP1()))
        return true;
   if (d2 == 0 && ComputationalGeometryUtils.onSegment(s2.getP1(),s2.getP2(),s1.getP2()))
        return true;
   if (d3 == 0 && ComputationalGeometryUtils.onSegment(s1.getP1(),s1.getP2(),s2.getP1()))
        return true;
   if (d4 == 0 && ComputationalGeometryUtils.onSegment(s1.getP1(),s1.getP2(),s2.getP2()))
        return true;
   return false;
}
```

Successivamente abbiamo definito la classe *FileUtils* il cui scopo è quello di contenere due metodi funzionali al programma per leggere e per scrivere dati.

Tramite l'ausilio della funzione *FileUtils.readSequences* è stato possibile passare, come riferimento ai due algoritmi per la rilevazione delle intersezioni, un file di testo contenente i dati dei segmenti da analizzare.

Sia nel caso della *coupleIntersection* che nel caso della *sweepLine* viene richiamato, subito dopo la lettura dell'input, un metodo presente nella classe segment (*segmentArrayFromSequence*) che permette di ottenere un array contenente tutti i segmenti presenti nel file.

Per l'implementazione della *coupleIntersection* è stato sufficiente riportare i 2 cicli for presenti nello pseudocodice.

```java
public class CoupleIntersection {
    public static boolean intersection(String fileName, int num)
    {
        Double[][] seq = FileUtils.readSequences(fileName,num);
        Segment[] sa = Segment.segmentArrayFromSequence(seq);

        for(int i = 0; i < num; i++)
        {
            for (int j = i+1 ; j < num; j++)
            {
                if (Segment.segmentIntersect(sa[i] ,sa[j])) {
                    return true;
                }
            }
        }
        return false;
    }
}
```

Nel caso della *sweepLine* è stato necessario implementare altre 2 classi:

- RBTree che permette di utilizzare gli alberi rosso-neri per memorizzare i dati relativi ai segmenti
- QuickSorter che ha permesso di ordinare l'array contenente i punti di arresto della sweepLine.

```
public class SweepLine {

    public static boolean intersection(String filename,int num)
    {
        double[] eventPoints = new double[2*num];
        Double[][] seq = FileUtils.readSequences(filename,num);
        Segment[] sa = Segment.segmentArrayFromSequence(seq);
        QuickSorter quickSorter = new QuickSorter();

        int x = 0;
        for (int i = 0 ; i < num ; i++)
        {
            for (int j = 0 ; j < 3;j = j+2)
            {
                eventPoints[x] = seq[i][j];
                x++;
            }
        }
        quickSorter.sort(eventPoints);

        RBTree T = new RBTree();

        for (int i = 0 ; i < eventPoints.length ; i++)
        {
            if (isLeftEndpoint(eventPoints[i], sa))
            {
                Segment s = findSegmentFromLeftEndpoint(eventPoints[i], sa);
                RBTree.RBTElement w = new RBTree.RBTElement(s);
                T.insert(w , eventPoints[i]);
                if ((T.above(w).getKey() != null && Segment.segmentIntersect(T.above(w).getKey(),s)) ||
                        (T.below(w).getKey() != null && Segment.segmentIntersect(T.below(w).getKey(),s)))
                    return true;

            }
            if (isRightEndpoint(eventPoints[i], sa))
            {
                Segment z = findSegmentFromRightEndpoint(eventPoints[i], sa);
                RBTree.RBTElement w = T.search(z,z.getX1());
                if (T.above(w).getKey() != null && T.below(w).getKey() != null && Segment.segmentIntersect(T.above(w).getKey(),T.below(w).getKey()))
                    return true;
                T.delete(w);
            }
        }
        return false;
    }
```

In aggiunta rispetto allo pseudocodice è presente il doppio ciclo for per costruire l'array contenente i punti d'arresto

Ultimo aspetto fondamentale dell'implementazione è stato definire un criterio di ordinamento totale tra segmenti da inserire all'interno della classe RBTree per fare in modo che fosse possibile utilizzare le operazioni richieste dallo pseudocodice.

Il criterio è stato definito sulla base del valore di ordinata dell'intersezione tra il segmento e la retta di rastrellamento.

Questa scelta implementativa potrebbe comportare un errore in quanto, quando si incontra il punto destro del segmento, viene avviata una ricerca sulla base dell'ordinamento attuale e non sulla base di quello rispetto al quale era stato inserito il segmento; ma questo comporta un problema solo se questi 2

ordinamenti sono diversi (ovvero nel caso in cui ad esempio il punto sinistro del segmento s1 si trovi “sopra al segmento s2 e il punto destro si trovi sotto”) ma questo avviene solo nel caso di un intersezione che viene individuata dal primo *if* che interrompe l’esecuzione.

Quindi il programma non si imbatte mai in questo errore.

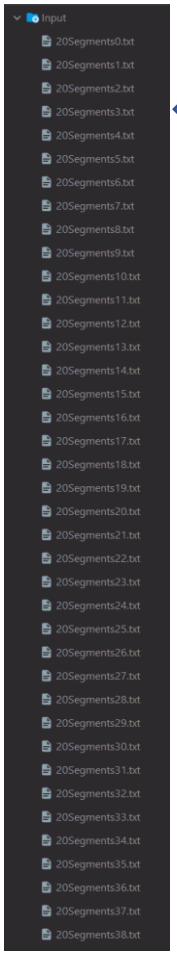
6)Esecuzione e Dati Sperimentali

Lo scopo del programma da noi realizzato non è quello di rilevare effettivamente la presenza di un’intersezione in un pool di segmenti, ma è quello di valutare quale dei due algoritmi è maggiormente efficiente, permettendo a chi avvia il programma di selezionare la quantità di file di input per i quali misurare il tempo impiegato dai due algoritmi.

Il programma esegue poi un numero di simulazioni pari al numero inserito per ogni dimensione di input (dove per dimensioni di input si intende il numero di segmenti): 20, 50, 100, 200, 500, 1000.

Di seguito riportiamo degli screen del funzionamento del programma seguiti da un’analisi dei risultati ottenuti.

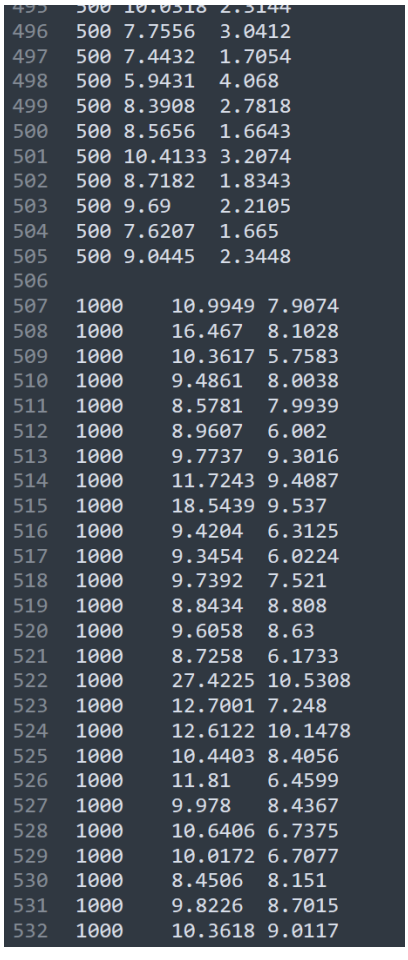
La schermata che permette di scegliere il numero di simulazioni



Una parte dei file di input generati dal programma che in questo caso sono 100×6

Parte del file di output.

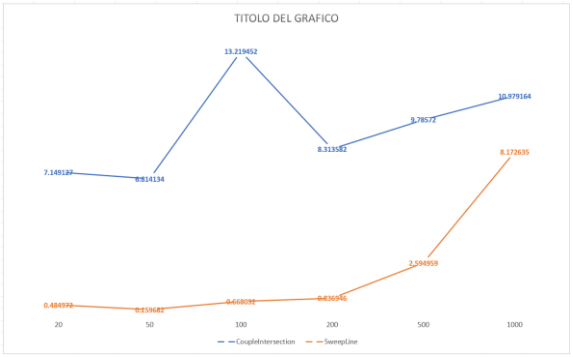
Dall’immagine si vede la struttura del file che presenta il tempo in ms impiegato per l’esecuzione dei 2 algoritmi per ogni input generato



Per proseguire nell'analisi dei dati ottenuti, li abbiamo importati in un foglio di calcolo e, per ciascuna delle dimensioni di input, abbiamo calcolato la media dei tempi di esecuzione di coupleIntersection e di sweepLine.

	CoupleInte	SweepLine
20	7.14913	0.48497
50	6.81413	0.25968
100	13.2195	0.66803
200	8.31358	0.83695
500	9.78572	2.59496
1000	10.9792	8.17264

Inserendo i dati ottenuti all'interno di un grafico, è stato possibile visualizzare i due andamenti al variare dell'input.



Per quanto dal grafico si possa apprezzare come l'algoritmo sweepline risulti più efficiente, i grafici non rispecchiano la diversa velocità di crescita del tempo impiegato dai due algoritmi che nel caso della sweepline dovrebbe essere logaritmica, mentre nel caso della coupleintersection dovrebbe essere quadratica.

La motivazione risiede nel fatto che la probabilità di trovare un'intersezione è molto elevata e quindi questa viene rilevata quasi subito nell'esecuzione.
Questo appiattisce la differenza tra i due algoritmi in quanto non passa abbastanza tempo per fare in modo che il funzionamento più intelligente di sweepline recuperi a pieno il tempo perso per la costruzione della base dati e per l'ordinamento dei punti d'arresto.

Quindi, nonostante un'effettiva convenienza nell'utilizzo di sweepline, questa non è quella teorizzata durante lo studio della complessità dei due algoritmi.