Algorithmische Anwendungen

Schnittpunkte von Liniensegmenten

Closest Pair of Points

Teilnehmer:

Martin Zimmermann	11043834
Martin Shad-Manfaat	11042014
Stefan Barndt	11041000

25.01.07

Inhaltsverzeichnis

Einleitung

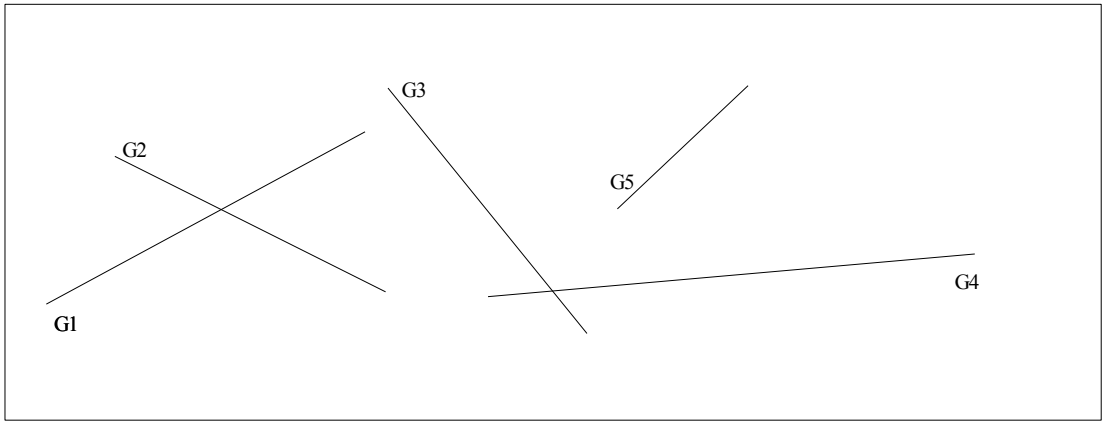
In dieser Ausarbeitung wollen wir zwei Themen behandeln, die in der algorithmischen Geometrie eine fundamentale Rolle spielen. Das erste Thema befasst sich mit dem Auffinden der Schnittpunkte von Liniensegmenten in der Ebene (intersection detection) mittels Plane-Sweep Technik, das zweite mit der Suche nach dem geringsten Abstand zwischen zwei Punkten, wenn eine Menge von Punkten gegeben ist. Hierbei kommt der Closest Pair of Points Algorithmus zum Einsatz. Als Algorithmus für das Liniensegment reporting problem haben wir uns für die Arbeit von Bentley und Ottmann entschieden **[BO79]**.

Das Finden der Schnittpunkte von Linien in der Ebene

Das Auffinden von Schnittpunkten zweier geometrischer Objekte und die Berechnung der von ihnen eingeschlossenen Regionen, wird in einer Vielzahl von Anwendungen benötigt. Ob es nun um computergestützte Hardwareentwicklung wie VLSI (very-large-scale Integration), Kartenüberlagerungsalgorithmen (map-overlay) für GIS (geographic information systems) oder Kollisionsdetektion geht, die komplexen Probleme können oftmals in kleinere Probleme aufgeteilt werden und sukzessiv bearbeitet werden. Wir werden das Schnittpunktproblem anhand eines geometrischen Objektes, nämlich der Linie, hier diskutieren. Wir lehnen unsere Ausarbeitung primär an der Arbeit von Bentley und Ottmann **[BO79]** an, die wiederum die Ideen von Shamos und Hoey **[SH75]** erweiterten. Die Plane-Sweep Technik ist hierbei Grundlage für viele Anwendungen der algorithmischen Geometrie. Wir wollen sowohl die Algorithmen selbst, als auch die Datenstrukturen die dafür benötigt werden hier kurz vorstellen.

Die Problemstellung

Wir haben eine Menge M = { G1, G2, ..., Gn } von n Geraden in der Ebene.
Die Geraden liegen in einer Form vor, wie G((x1, y1), (x2,y2)). Gesucht sind alle Schnittpunkte von Geradenpaaren.



Der naive Ansatz

Vergleicht man jede Gerade mit jeder anderen (Brute- Force- Methode), ergeben sich daraus $(\frac{n*(n-1)}{2})$ Vergleiche, da wir im ersten Durchlauf n Vergleiche haben, im zweiten n-1, im dritten dann n-2 usw. Die Laufzeit beträgt also $O(n^2)$. Diese Laufzeit ist natürlich nur für kleine n akzeptabel, gerade wenn viele Linien mit wenig Schnittpunkten vorliegen ist dies natürlich sehr ineffizient.

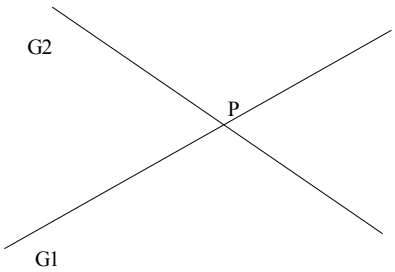
Sweeplinetechnik

Im Jahre 1975 haben Shamos und Hoey in ihrer Arbeit "Closest-Point Problems" **[SH75]** effiziente Algorithmen für das Auffinden zweier am nächsten zusammenliegender Punkte implementiert, die das Gebiet der computational geometry revolutionieren sollten. Die Lösung des Problems, auf die wir an anderer Stelle genauer eingehen werden, brachte eine neue Technik mit sich um bestimmte geometrische Fragestellungen in der Ebene schneller zu beantworten als es der Brute- Force Ansatz kann, die Scanline- oder auch wie sie heute genannt wird Sweeplinetechnik.
Die Sweeplinetechnik allgemein lässt sich recht schnell und einfach beschreiben. Nehmen wir gleich das Beispiel der Liniensegmentschnittpunkte. Wenn man sich das Problem nochmal genau anschaut, geht es ja um n Linien dessen Schnittpunkte uns interessieren. Entweder möchte man nur wissen, ob es überhaupt einen Schnittpunkt gibt (Existenzproblem) oder evtl. möchte man auch wissen welche Linien sich schneiden, bzw. wo sie sich schneiden (intersection reporting). Tatsache ist ja, dass die Linien lediglich geistig so vorzustellen sind, als lägen sie in der Ebene. Und genauso müssen wir uns die Sweepline vorstellen wie sie über eine imaginäre Ebene läuft.
Um das ganze zu simulieren, nimmt man normalerweise Sortierungen nach den X- Koordinaten der Linien vor. Dann können wir die sortierte Liste einfach der Reihe nach abwandern und erreichen so das geistige Konstrukt der Anordnung der Linien in einer Ebene.
Diese Technik haben sie, ein Jahr später, in ihrer Arbeit ‘‘Geometric intersection problems’’ **[SH76]** auch zum Auffinden von Schnittpunkten benutzt. Mit ihrer Arbeit schafften sie es das Existenzproblem innerhalb $\Theta(n*\log(n))$ Zeit zu lösen, wobei n die Anzahl der Linien sind.

Bentley- Ottmann Algorithmus

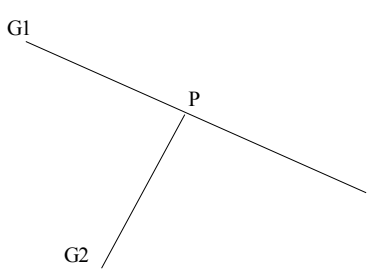
Etwas später haben Bentley und Ottmann **[BO79]** den Algorithmus noch verändert um auch die Anzahl und Position der Schnittpunkte zu ermitteln. Die Laufzeit ist dadurch etwas verlangsamt worden, und zwar braucht der Algorithmus jetzt soviel mehr Zeit wie Schnittpunkte vorhanden sind $\Theta(n*\log(n)+k*\log(n))$, wobei n die Anzahl der Linien und k die Anzahl an Schnittpunkten ist. In dieser Abhandlung wollen wir uns darauf beschränken nur den Bentley- Ottmann Algorithmus zu besprechen und nicht auf den ursprünglichen eingehen, da sich beide kaum unterscheiden. Dieser Algorithmus ist aber asymptotisch nicht optimal. Da er die Schnittpunkte aufgrund ihrer x – Koordinate findet, und somit sortieren muss kann er die theoretisch optimale Laufzeit von $\Theta(n*\log(n)+k)$ nicht erreichen. Erst 12 Jahre später konnte dieses Problem durch Neuauflage der Sweeprichtung durch Chazelle und Edelsbrunner **[CE88]** beseitigt werden, worauf wir aber hier nicht eingehen werden.

Definition eines Schnittpunkts:
Zu allererst muss definiert werden was alles als Schnittpunkt zählt und gefunden werden soll.
Als gültiger Schnittpunkt sei nur ein echter Schnitt zweier Geraden definiert.

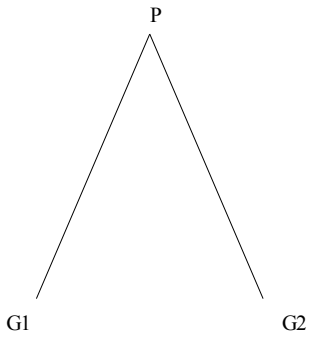


Echter Schnittpunkt in P

Wenn allerdings ein unechter Schnitt vorliegt, wie in dem Fall, dass eine Gerade beispielsweise senkrecht auf einer anderen Geraden liegt, oder sich beide Geraden in einem Punkt treffen, dann soll dies nicht als Schnittpunkt zählen.



unecht

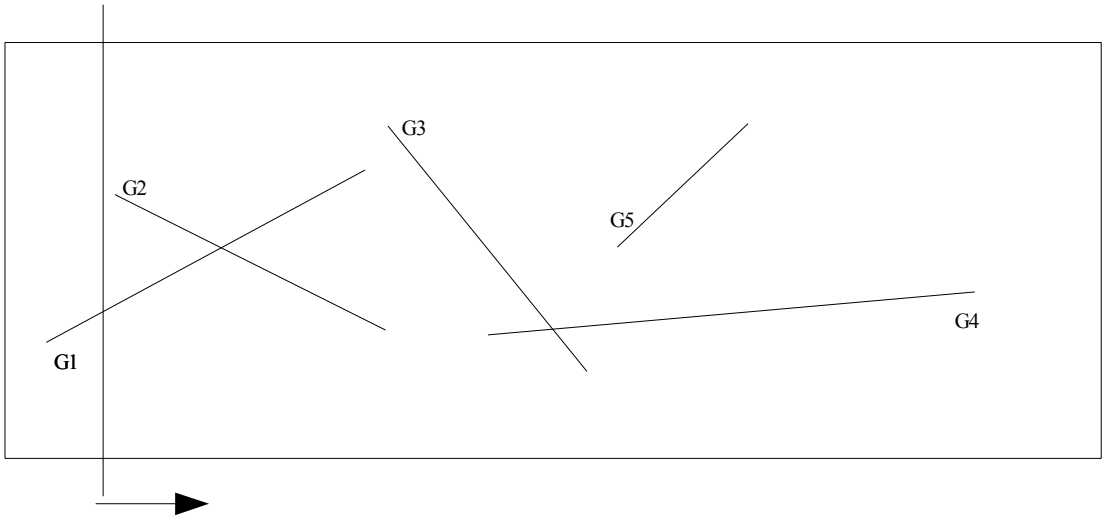


unecht

Ferner legen wir der Einfachheit halber folgende Voraussetzungen fest:

1. Es gibt keine vertikalen Linien
2. Linien schneiden sich nicht an ihren Endpunkten
3. Nicht mehr als drei Linien schneiden sich an dem gleichen Punkt
4. Alle Endpunkte der Linien und alle Schnittpunkte haben unterschiedliche x-Koordinaten
5. Keine zwei Linien überlappen sich

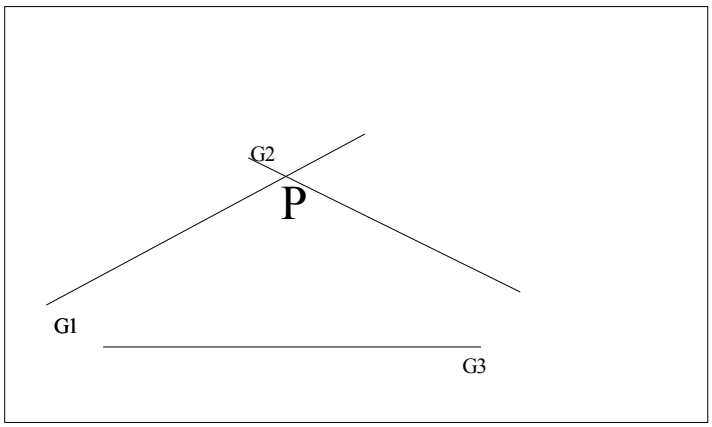
Der Algorithmus funktioniert nach folgendem Prinzip:
Wie bereits erwähnt wollen wir die Schnittpunkte mittels Sweeplinetechnik finden und müssen dafür die Geraden aufsteigend nach ihrer X- Koordinate sortieren. Als zugrunde liegende Datenstruktur sollte ein balancierter Baum benutzt werden, da er uns logarithmische Zugriffszeit garantiert. Mit der Sortierung erreichen wir das „Abwandern“ der Sweepline über die imaginäre Ebene.



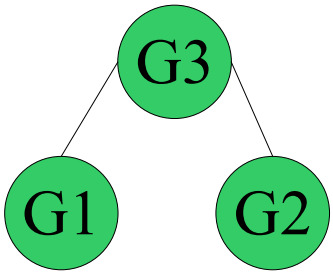
Wenn wir also davon sprechen, dass die Sweepline von links nach rechts über die Ebene geht, meinen wir eigentlich das Traversieren des Baums.
Während dieser Aktion können drei für uns interessante Ereignisse (Events) auftreten, die wir verschieden interpretieren müssen.

- Event A: Die Sweep-Linie erreicht den linken Endpunkt einer Linie
- Event B: Die Sweep-Linie erreicht den rechten Endpunkt einer Linie
- Event C: Die Sweep-Linie trifft auf einen Schnittpunkt zweier Linien L1 und L2

Schauen wir uns mal dazu ein Fallbeispiel an (gegeben drei Linien G1-G3, ein Schnittpunkt zwischen G1 und G2 in P):

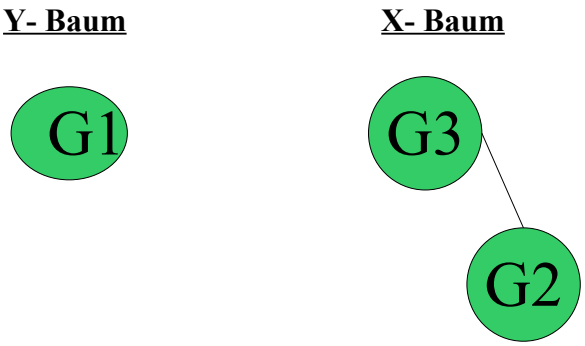


Den Baum, nach der X- Koordinate der Geraden sortiert und daher auch im Folgenden X- Baum genannt, könnte man sich dann folgendermaßen vorstellen:



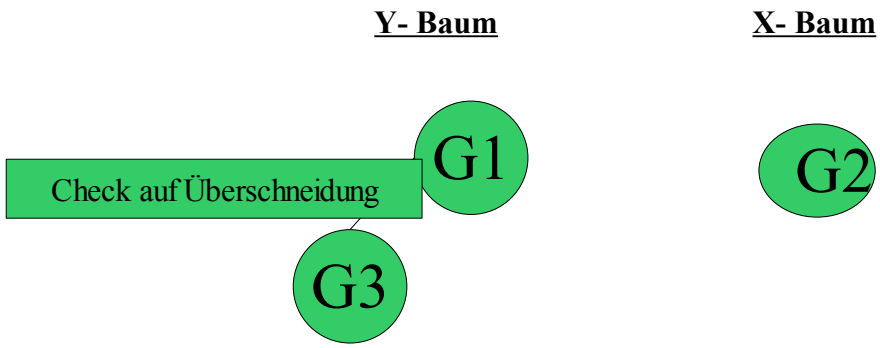
Während wir nun den X-Baum der kleinsten X- Koordinate nach traversieren, entfernen wir die Gerade aus dem Baum und stecken sie in eine weitere, ebenso balancierte, Baumstruktur, die allerdings nach der Y- Koordinate sortiert wird und daher auch treffenderweise Y- Baum genannt wird. Der Y- Baum dient zur Überprüfung auf Überschneidung. Sobald ein weiteres Element mit in den Baum eingefügt wird, wird überprüft ob es Überschneidungen gibt. Die entsprechenden Baumeigenschaften und die damit verbundenen Operationen sind bei den Einfüge- und Löschoperationen natürlich zu berücksichtigen.
Bei der ersten Operation, dies entspricht dem Event A (wir treffen auf den linken Endpunkt von G1), wird die Gerade G1 vom X- Baum in den Y- Baum übertragen. Da dies das erste Element in dem Y- Baum ist, bedarf es keiner weiteren Operationen um zu überprüfen ob es einen Schnittpunkt gibt.

Nach der ersten Operation:



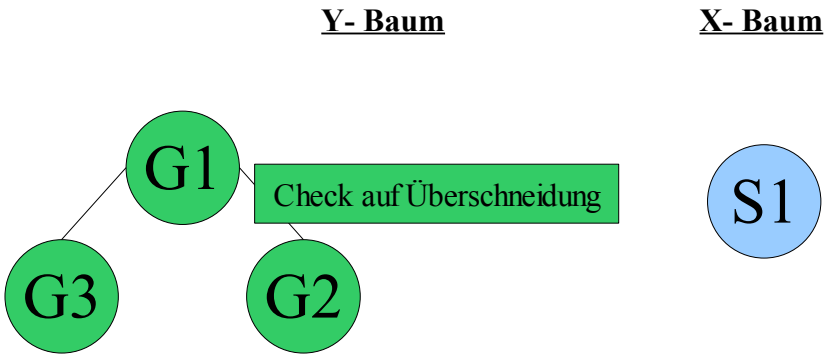
Als nächstes treffen wir auf den linken Endpunkt von G3. Die Gerade G3 wird also ebenfalls aus dem X-Baum gelöscht und in den Y-Baum eingefügt. Da G3 von der Y-Koordinate unterhalb von G1 liegt, wird es entsprechend auch unter G1 in den Y-Baum einsortiert.

Nach der zweiten Operation:



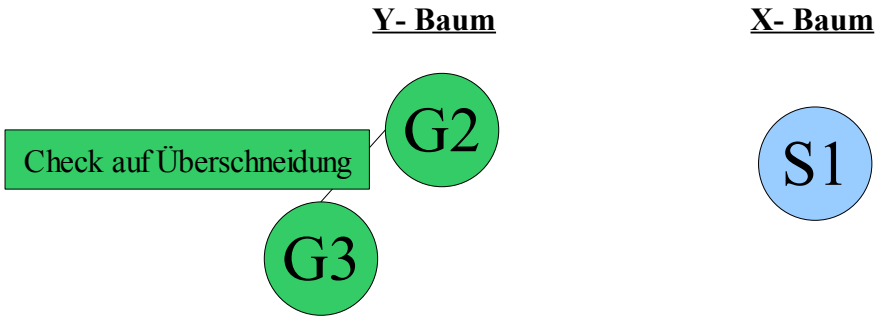
Hier findet jetzt die erste Überprüfung auf eine Überschneidung statt, da sowohl G1 und G3 gerade „aktiv“ (also von beiden der linke Endpunkt aber noch nicht der rechte Endpunkt gefunden wurde), als auch benachbart sind. Es wird jedoch keine Überschneidung festgestellt. Durch Errechnen des Anstiegs beider Linien kann man die entsprechenden Funktionen bilden. Indem man diese gleichsetzt, kann man herausfinden, ob die Linien sich überschneiden.

Nach der dritten Operation:



Nun treffen wir auf den linken Endpunkt der Linie G2. Diese wird also wiederum aus dem X-Baum gelöscht und als Linie mit höchster Y-Koordinate entsprechend rechts von G1 einsortiert. Anschließend wird wieder überprüft, ob eine Überschneidung vorliegt. In diesem Fall liegt tatsächlich eine vor und diese wird als Punkt im X-Baum abgelegt.

Nach der vierten und fünften Operation:



Wir treffen jetzt auf den rechten Endpunkt der Linie G1. Diese wird also aus dem Y-Baum entfernt und der Baum entsprechend umgebaut. Durch diesen Umbau sind jetzt die Linien G2 und G3 benachbart. Eine Überprüfung auf Überschneidung ergibt jedoch ein negatives Ergebnis.

Nach der sechsten Operation:

Es folgt noch der rechte Endpunkt der Linie G3, die ebenfalls aus dem Y-Baum entfernt wird,

Nach der siebten Operation:

sowie der rechte Endpunkt der Linie G2, die als letztes aus dem Y-Baum gelöscht wird. Danach ist der Y-Baum leer und der Algorithmus beendet. Im X-Baum befinden sich nun alle gefundenen Überschneidungen.

Pseudocode

```
Initialize event queue x = all segment endpoints;
    Sort x by increasing x and y;
    Initialize sweep line SL to be empty;
    Initialize output intersection list L to be empty;

    While (x is nonempty) {
        Let E = the next event from x;
        If (E is a left endpoint) {
            Let segE = E's segment;
            Add segE to SL;
            Let segA = the segment above segE in SL;
            Let segB = the segment below segE in SL;
            If (I = Intersect( segE with segA) exists)
                Insert I into x;
            If (I = Intersect( segE with segB) exists)
                Insert I into x;
        }
        Else If (E is a right endpoint) {
            Let segE = E's segment;
            Let segA = the segment above segE in SL;
            Let segB = the segment below segE in SL;
            Remove segE from SL;
            If (I = Intersect( segA with segB) exists)
                If (I is not in x already) Insert I into x;
        }
        Else {  // E is an intersection event
            Add E to the output list L;
            Let segE1 above segE2 be E's intersecting segments in SL;
            Swap their positions so that segE2 is now above segE1;
            Let segA = the segment above segE2 in SL;
            Let segB = the segment below segE1 in SL;
            If (I = Intersect(segE2 with segA) exists)
                If (I is not in x already) Insert I into x;
            If (I = Intersect(segE1 with segB) exists)
                If (I is not in x already) Insert I into x;
        }
        remove E from x;
    }
    return L;
}
```

Laufzeitbetrachtung

Die Laufzeit des Bentley- Ottmann Algorithmus beträgt $\Theta(n*\log(n)+k*\log(n))$, wobei n die Anzahl der Geraden ist und k die Anzahl der Schnittpunkte die gefunden werden.
Die Laufzeit lässt sich auch recht schnell und simpel erklären. Da wir die ganze Zeit mit balancierten Bäumen arbeiten, wissen wir, dass jede Zugriffsoperation logarithmisch zu erreichen ist. Da wir n Geraden haben, müssen wir also mindestens n * log(n) Operationen haben um die Elemente aus dem Baum herauszuholen, sobald wir auf einen Schnittpunkt treffen, müssen wir ein außerplanmäßiges Mal auf einen Baum zugreifen, daher kommen bei k Schnittpunkten noch k*log(n) Schritte hinzu. Diese Laufzeit ist wie bereits oben erwähnt nicht die asymptotisch optimale Laufzeit.

Speicherbedarf des Algorithmus

Der Speicherbedarf des Bentley- Ottmann Algorithmus beträgt $\Theta(n+k)$, wobei n wieder die Anzahl der Geraden und k die Anzahl der Schnittpunkte ist. Es werden zwar sowohl die Start- als auch die Endpunkte gespeichert, allerdings nie beide einer Linie gleichzeitig. Sobald der Startpunkt erreicht wird, wird dieser vom X-Baum in den Y-Baum übertragen, wird der Endpunkt erreicht, führt dies nur dazu, dass der Punkt aus dem Y-Baum gelöscht wird. Zusätzlich müssen noch k Schnittpunkte im X-Baum gespeichert werden. Das ist der Grund warum der Algorithmus n+k Speicherplatz benötigt.

Closest Pair of Points

Das Verfahren zur Ermittlung des geringsten Abstandes von zwei Punkten in einer Menge von Punkten findet in verschiedensten praktischen Anwendungen statt. Diese sind zum Beispiel die Luft- oder Seeverkehrsüberwachung, zum Berechnen von Abständen zwischen Schiffen, bzw. Flugzeugen. Bei der Luftüberwachung wird man natürlich den Dreidimensionalen Raum betrachten, während wir uns im folgenden auf den Zweidimensionalen Raum konzentrieren werden. Wie also in dem eben genannten Beispiel der Seeüberwachung.
Bei einer Aufgabenstellung wie dieser kann der naive Ansatz einer Brute-Force Methode nicht zum Einsatz kommen, da man zum Berechnen aller Abstände zwischen allen Punkten n^2 Vergleiche benötigen würde. Da die oben genannten Anwendungsbereiche aber beide extrem zeitkritisch sind, liegt es auf der Hand, dass die Problemstellung gerade bei einer großen Anzahl von Punkten nicht mit einem Brute-Force Verfahren gelöst werden kann. Daher wollen wir nun auf den Closest Pair of Points Algorithmus eingehen.

Closest Pair of Points Algorithmus

Bei der rekursiven Variante des Closest Pair of Points Algorithmus, wird zunächst die Punktmenge aufsteigend nach der x Koordinate mittels dem MergeSort Algorithmus sortiert, bevor der eigentliche Algorithmus gestartet wird. Anschließend wir die Liste halbiert. Das wird insgesamt sooft wiederholt, bis in den einzelnen Teilsegmenten nur noch maximal 3 Punkte enthalten sind. Nun wird der minimale Abstand zwischen diesen 2 bzw. 3 Punkten ermittelt. Hierzu wird die Euklidische Distanz berechnet, was wie folgt geschieht:

$$\sqrt{(x2-x1)^2+(y2-y1)^2}$$

Nach der Berechnung wird nun der minimale Abschnitt dieses Teilsegmentes zurückgegeben.

Bei der anschließenden wieder Zusammenführung der Teilergebnisse, werden dann die jeweiligen kürzesten Distanzen verglichen, und der niedrigere Wert wird dann auf den neuen kürzesten Abstand gesetzt.
Bei diesem Verfahren wurde nun aber nicht der Fall berücksichtigt, in dem sich die Punkte in unterschiedlichen Teilsegmenten befinden.

Um nun zu überprüfen ob es Punkte in den zwei zusammenzuführenden Teilsegmenten mit einer geringeren Distanz zu einander gibt, als die bisher ermittelte, geht man die nach der x Koordinate sortieren Punkte von der Grenze aus nacheinander durch, bis man einen Punkt gefunden hat, dessen x Wert eine höhere Differenz zu dem x Wert der Grenze angenommen hat, als die bisher gefundene minimale Distanz. Dies ist dann der erste Punkt der nicht mehr zu der Grenzregion gehört und weder dieser Punkt, noch alle nachfolgenden Punkte, sind in dieser Berechnung weiter zu berücksichtigen.

Wichtig ist, dass man die Grenzregion in beide Richtungen der Grenze ausweitet, so dass man links und rechts von der Grenze die Punkte eingeschränkt hat, die noch zu beachten sind.

Die Punkte, die sich in der soeben gefundenen Grenzregion befinden, werden nun anschließend ebenfalls mit dem MergeSort nach der y Koordinate sortiert.

Diese neue Liste wird nun sequenziell durchlaufen, wobei jeder Punkt mit den nachfolgenden Punkten in Verbindung gesetzt und deren Distanz berechnet wird. Tatsächlich müssen allerdings nicht alle nachfolgenden Punkte berücksichtigt werden, da ein näherer Punkt zwangsläufig nicht weiter vom aktuellen Punkt entfernt seien kann, als die Differenz der beiden y Werte. In dem Moment wo die Differenz der beiden y Werte größer ist als die bisher gefundene minimale Distanz kann die Schleife abgebrochen werden und der nächste Vergleichspunkt kann gewählt werden.

Nachdem nun alle Teilsegmente wieder zusammengefügt wurden, erhält man die beiden Punkte die die kürzeste Distanz zueinander haben.

Komplexität

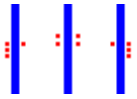
Um nun die Komplexität zu bestimmen gehen wir zunächst einmal auf das Master Theorem ein, mit dessen Hilfe wir die Komplexität des Closest Pair of Points Algorithmus bestimmen wollen. Die Formel lautet folgendermaßen:

$$T(n)=a*T(n/2)+c*n^k$$

Hierbei stehen die einzelnen Buchstaben für folgende Werte:

- n: Größe des zu lösenden Problems
- a: Anzahl der Teilprobleme
- n/b : Größe der Teilprobleme
- c*n^k: Zeit für das Zusammenfügen

Da wir die Unteraufgaben jeweils in 2 Teilprobleme aufteilen haben gilt a = 2. Des weiteren beläuft sich die Größe der jeweiligen Teilprobleme auf n/2, da die komplette Liste auf 2 Teillisten aufgeteilt wird. Das Zusammenfügen der einzelnen Teilprobleme lässt sich insgesamt in einer Zeit c * n ^ 1 lösen, womit wir k = 1 hätten.
Die Komplexität für das Zusammenfügen beträgt n, obwohl in der eigentlichen Implementierung des Algorithmus eine in einander verschachtelte for – Schleife verwendet wird, die ja eigentlich auf eine Komplexität von $O(n^2)$ hindeutet.
Das dies aber nicht der Fall ist, wird an diesem Beispiel deutlich:



Bei diesem Beispiel gibt es insgesamt 12 Punkte, die in Teilsegmente aufgeteilt werden, bis maximal drei Punkte in jedem Teilsegment vorhanden sind. Hierbei handelt es sich um ein Worst-Case Szenario, da alle Punkte so aufgeteilt sind, dass sie sich in der Grenzregion befinden und beim Zusammenfügen der Teillösungen die Entfernung zu allen anderen Punkten jeweils berechnet werden muss. Da sich in jeder Grenzregion vier Punkte befinden, ergeben sich für jede Grenzregion $n^2=4^2=16$ Vergleiche, die getätigt werden müssen. Da es insgesamt 3 Grenzregionen gibt, haben wir 3 * 16 = 48 Vergleiche für das Zusammenfügen aller Teillösungen. Dazu ergibt sich dann folgende Formel für das Zusammenfügen der Teillösungen $c*n^k=3*12^1=48\Rightarrow k=1$.
Wäre die Komplexität für das Zusammenfügen allerdings tatsächlich n^2 gewesen, so müsste es insgesamt $12^2=144$ Vergleiche geben.
Durch diese relativ einfache Rechnung ergibt sich recht schnell, warum k = 1, wie oben behauptet, tatsächlich gilt.

Zur Lösung des Master Theorems gibt es insgesamt folgende drei Fälle, die zu beachten sind:

- $a > b^k$
- $a = b^k$
- $a < b^k$

Für den Closest Pair of Points Algorithmus gilt:

$2 = 2^1 \Rightarrow a = b^k \Rightarrow$ Fall 2

Für Fall 2 besagt das Master Theorem folgende Komplexität:

$O(n^k * \log n)$, woraus sich für unseren Fall eine Komplexität von $O(n * \log n)$ ergibt.

Noch nicht berücksichtigt haben wir hierbei, dass vor der Anwendung des Closest Pair of Points Algorithmus zunächst einmal die Liste der Punkte mit dem MergeSort nach der x Koordinate sortiert werden muss. Da die Laufzeit für den MergeSort ebenfalls $O(n * \log(n))$ beträgt, ergibt sich für den Closest Pair of Points Algorithmus insgesamt eine Komplexität von $n * \log n + n * \log n = 2 * n * \log n$, was allerdings bei der Asymptotischen Betrachtung wieder eine Komplexität von $O(n * \log n)$ ergibt.

Quellcode

```
public Point recClosestPoint(int [][] points, int low, int height)
{
	double minDistance = -1;
	Point point = new Point();

	if (this.log)
	{
		this.xmlWriter.createCanvasXMLFile(points, points[0][low], points[0][height]+10, null);
		this.eventThread.getStatistik().add(new Statistik(null, height - low));
	}

	//Maximal drei Punkte im Teilsegment
	if ((height - low) <= 3)
	{
		double distance;

		for (int i = low; i <= height; i++)
		{
			for (int j = i + 1; j <= height; j++)
			{
				//Euklidischer Abstand wird berechnet
				distance = getDistanz(points[0][i], points[1][i], points[0][j], points[1][j]);


				//Geringerer Abstand wurde gefunden
				if (minDistance == -1 || distance < minDistance)
				{
					minDistance = distance;

					point.setDistance(minDistance);
					point.setX1(points[0][i]);
					point.setY1(points[1][i]);
					point.setX2(points[0][j]);
					point.setY2(points[1][j]);
				}
			}
		}

		if (this.log)
		{
			this.xmlWriter.createCanvasXMLFile(points, points[0][low], points[0][height]+10, point);
			this.eventThread.getStatistik().add(this.xmlWriter.getFilePointer(), new Statistik(point, height - low));
		}

		return point;
	}


	Point left;
	Point right;

	int m = (low + height) / 2;

	//Rekursiver Aufruf der ClosestPoint Methode
	left = recClosestPoint(points, low, m);
	right = recClosestPoint(points, m + 1, height);

	//Vergleich aus welchem Teilsegment die kleinere Distanz stammt
	if (left.getDistance() < right.getDistance())
	{
		point = left;
		minDistance = left.getDistance();
	}
	else
	{
		point = right;
		minDistance = right.getDistance();
	}


	//Grenzabschnitt des linken Teilsegmente wird gesucht
	int borderLow = m;
	while ((borderLow > low + 1) && (points[0][m] - points[0][borderLow - 1] < minDistance))
	{
```

```
                borderLow--;
        }

        //Grenzabschnitt des rechten Teilsegmentes wird gesucht
        int borderHight = m + 1;
        while ((borderHight < height - 1) && (points[0][borderHight + 1] - points[0][m + 1] < minDistance))
        {
                borderHight++;
        }


        MergeSort mergeSort = new MergeSort();

        int [][] tmp = mergeSort.merge(points[1], points[0], borderLow, borderHight);

        int [][] borderPoints;
        borderPoints = new int[2][];
        borderPoints[0] = tmp[1];
        borderPoints[1] = tmp[0];

        double distance;

        for (int i = 0; i < borderPoints[1].length; i++)
        {
                for (int j = i + 1; j < borderPoints[1].length; j++)
                {

                        if (borderPoints[1][i] - borderPoints[1][j] >= minDistance)
                                break;

                        //Euklidischer Abstand wird berechnet
                        distance = getDistanz(borderPoints[0][i], borderPoints[1][i], borderPoints[0][j], borderPoints[1][j]);


                        //Geringerer Abstand wurde gefunden
                        if (distance < minDistance)
                        {
                                minDistance = distance;

                                point.setDistance(minDistance);
                                point.setX1(borderPoints[0][i]);
                                point.setY1(borderPoints[1][i]);
                                point.setX2(borderPoints[0][j]);
                                point.setY2(borderPoints[1][j]);
                        }
                }
        }

        if (this.log)
        {
                this.xmlWriter.createCanvasXMLFile(points, points[0][low], points[0][height]+10, point);
                this.eventThread.getStatistik().add(this.xmlWriter.getFilePointer(), new Statistik(point, height - low));
        }

        return point;
}
```

Quellen

[BO79] - J. Bentley, Th. Ottmann: Algorithms for Reporting and Counting Geometric Intersections. IEEE Trans. Comput. C-28, 9 (Sept. 1979), pp. 643-647

[CE88] - Chazelle, B. and Edelsbrunner, H., ‘‘An optimal algorithm for intersecting line segments in the plane’’, Proceedings of the IEEE Symposium on Foundations of Computer Science, 1988, pp.590-600.

[SH75] - Shamos, M.I. and Hoey, D., ‘‘Closest point problems’’, IEEE Symposium on foundations of computer science, 151-162, 1975.

[SH76] -Shamos, M.I. and Hoey, D., ‘‘Geometric intersection problems’’, IEEESymposium on foundations of computer science, 208-215, 1976.