

Ausarbeitung:

Divide & Conquer

(im Rahmen des Informatik-Seminars „Algorithmen“)

Janno Rothfos

Themensteller: Prof. Dr. Iwanowski

Fachhochschule Wedel

Studiengang: Wirtschaftsinformatik

Inhaltsverzeichnis

Abbildungsverzeichnis.....	3
1 Motivation.....	4
2 Grundlagen.....	5
2.1 Die Idee von Divide-and-Conquer.....	5
2.2 Das Master Theorem.....	7
3 Die Matrixmultiplikation.....	8
3.1 Der Standard-Algorithmus.....	8
3.2 Strassens Algorithmus.....	9
3.3 Analyse von Strassens Algorithmus.....	11
3.4 Strassens Algorithmus in der Praxis.....	13
4 Das Closest-Pair-Problem.....	14
4.1 Problembeschreibung.....	14
4.2 Der Bruteforce Ansatz.....	14
4.3 Closest-Pair mit Divide-and-Conquer.....	14
4.4 Analyse des Divide-and-Conquer Algorithmus.....	17
5 Das Convex-Hull-Problem.....	18
5.1 Problembeschreibung.....	18
5.2 Ein Bruteforce-Ansatz.....	18
5.3 Convex-Hull mit Divide-and-Conquer.....	19
5.4 Implementierung der geometrischen Funktionen.....	21
5.5 Analyse des Divide-and-Conquer Algorithmus.....	22
6 Zusammenfassung.....	23
7 Literaturverzeichnis.....	24

Abbildungsverzeichnis

Abbildung 1: Divide-and-Conquer (typischer Fall) [an1, S.124].....	6
Abbildung 2: Die Idee des Divide-and-Conquer Algo. für das Closest-Pair-Problem[an07 S.151].....	15
Abbildung 3: Die 6 möglichen Kandidaten (worst case).....	16
Abbildung 4: Nicht konvexes Polygon.....	18
Abbildung 5: Konvexes Polygon.....	18
Abbildung 6: Aufteilung der Menge in S1 und S2.....	19
Abbildung 7: Aufteilung der Menge S1 in S1,1 und S1,2.....	20

1 Motivation

Divide-and-Conquer ist wahrscheinlich die bekannteste Designtechnik für Algorithmen. Auch wenn ein Teil dieser Berühmtheit vielleicht etwas mit dem eingängigen Namen zu tun hat, so ist sie dennoch wohlverdient. Ein paar sehr effiziente Algorithmen sind das Ergebnis dieser Designtechnik. [an07 S.122] Zu eben diesen, gehören u.a. auch der in dieser Arbeit beschriebene Algorithmus von Strassen (zur Multiplikation von Matrizen), eine Implementierung des Closest-Pair- und des Convex-Hull-Problems.

Divide-and-Conquer Algorithmen sind in den meisten Fällen rekursiv definiert. Diese rekursiv definierten Algorithmen haben oft den Vorteil, dass sie auf Grund der wenigen Schleifen schneller zu verstehen sind, als ihr Iteratives Pendant. Diesem Vorteil steht der gegenüber, dass viele Menschen eine Abschätzungen über das Laufzeitverhalten von rekursiven Algorithmen schwerer fällt. Für die meisten Divide-and-Conquer Algorithmen kann dieser Nachteil mit Hilfe des Master-Theorems aufgehoben werden.

Ziel dieser Arbeit soll es sein, die Idee von Divide-and-Conquer näher zu erläutern und die Auswirkungen auf die Algorithmen, die dieser Idee zu Grunde liegen zu beschreiben. Anschließend soll das Master-Theorem vorgestellt werden und dessen Anwendung an den später beschriebenen Algorithmen gezeigt werden. Im Hauptteil soll ein kurzer Überblick über die drei Divide-and-Conquer Algorithmen gegeben und diese mit den Bruteforce Implementierungen verglichen werden. In Kapitel zwei werden die Grundlagen für Divide-and-Conquer vorgestellt, damit in Kapitel drei die Divide-and-Conquer Algorithmen besser erklärt und verglichen werden können.

2 Grundlagen

2.1 Die Idee von Divide-and-Conquer

Das Divide-and-Conquer Verfahren beruht auf der Idee, ein Problem solange in Unterprobleme aufzuteilen bis diese elementar gelöst werden können. Divide-and-Conquer Algorithmen arbeiten daher im allgemeinen nach dem folgenden Plan [an07, S. 123]:

1. Die Instanz eines Problems wird in viele kleinere Instanzen des gleichen Problems aufgeteilt.
2. Die kleineren Instanzen werden gelöst (typischerweise rekursiv, wobei manchmal auch der Algorithmus gewechselt wird, wenn die Instanzen klein genug sind).
3. Wenn nötig, werden die Ergebnisse der kleineren Instanzen kombiniert und zu einer Lösung des Ausgangsproblem vereint.

Zu den bekanntesten Algorithmen dieser Art gehören die Sortieralgorithmen: Merge- und Quicksort. Beide sortieren eine n -elementige Menge in der, für Divide & Conquer Algorithmen typischen Laufzeit, von $O(n \log n)$.

Die allgemeine Rekursionsgleichung von Divide-and-Conquer Algorithmen

Wie oben beschrieben, wird in typischen Divide-and-Conquer Algorithmen die Instanz eines Problems in kleinere Instanzen des gleichen Problems zerlegt, gelöst und wenn nötig zusammengesetzt. Noch allgemeiner, kann eine Instanz der Größe n , in b Instanzen der Größe n/b geteilt werden, mit a von diesen Instanzen die gelöst werden müssen [an07, S.124]. Dies führt zu der allgemeinen Rekursionsgleichung von Divide-and-Conquer Algorithmen:

$$T(n) = aT(n/b) + f(n) \quad 1.0$$

$f(n)$ ist dabei eine einfache Funktionen, die die Zeit berücksichtigt, die es dauert das Problem zu zerlegen und wenn nötig, dessen Teillösungen zu kombinieren.

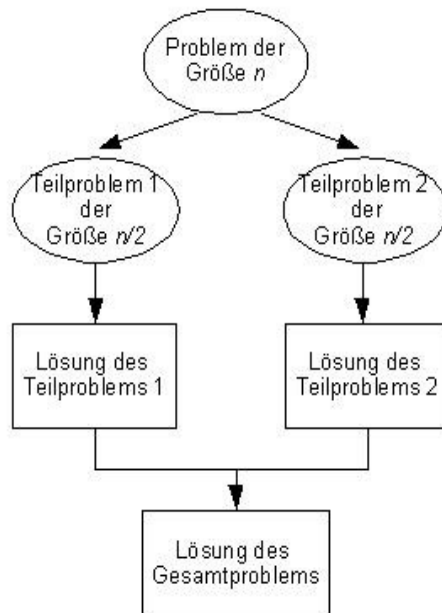


Abbildung 1: Divide-and-Conquer (typischer Fall) [an1, S.124]

2.2 Das Master Theorem

Die Effizienz-Analyse von vielen Divide-and-Conquer Algorithmen wird erheblich durch die Verwendung des Master-Theorems vereinfacht. Ausgangspunkt des Master-Theorems ist die allgemeine Rekursionsgleichung von Divide-and-Conquer Algorithmen. Wenn $f(n) \in \Theta(n^d)$ mit $d \geq 0$ in Gleichung 1.0, dann kann folgende Aussage über $T(n)$ getroffen werden [an07, S.125]:

$$T(n) \in \begin{cases} \Theta(n^d) & \text{wenn } a < b^d \\ \Theta(n^d \log n) & \text{wenn } a = b^d \\ \Theta(n^{\log_b a}) & \text{wenn } a > b^d \end{cases}$$

(Analoge Resultate ergeben sich auch für die Ω - und O -Notation)

Beispiel:

Ein beliebiger Divide-and-Conquer Algorithmus, für den die folgende Rekursiongleichung gilt:

$$T(n) = 2T(n/2) + O(n)$$

In diesem Beispiel wäre $a = 2$, $b = 2$ und $d = 1$. Da $a = b^d$, bzw. $2 = 2^1$ besitzt dieser Algorithmus nach dem Master-Theorem eine Laufzeitkomplexität von $O(n^1 \log n)$. Ein konkretes Beispiel für einen Algorithmus, der nach dieser Formel arbeitet ist Quicksort.

3 Die Matrixmultiplikation

3.1 Der Standard-Algorithmus

Nach dem Standard-Algorithmus zur Multiplikation zweier $n \times n$ Matrizen A und B werden n^2 einzelne Multiplikationen und $n^2 - n$ Additionen benötigt. Dabei wird jedes Element der Ergebnismatrix C nach folgender Formel bestimmt [ti02, S. 9-14].

$$c_{i_j} = \sum_{k=1}^n a_{i_k} b_{k_j}$$

Betrachtet man die Anzahl der Multiplikationen als Basis-Operation des Algorithmus, so kann die Laufzeit nach folgender Formel bestimmt werden:

$$T(n) = n^2 * c_{i_j} = n^2 * n = n^3$$

Da die Ergebnismatrix C von dem Produkt zweier $n \times n$ Matrizen $n * n = n^2$ Elemente enthält, müssen n^2 Ergebnisse berechnet werden. Da jedes Ergebnis durch c_{i_j} berechnet wird und für c_{i_j} n Multiplikationen benötigt werden, ergibt sich eine Laufzeit von n^3 , wie oben angegeben.

3.2 Strassens Algorithmus

1969 konnte Volker Strassen zeigen, dass es möglich ist das Produkt von den Matrizen A und B mit 7 Multiplikationen zu berechnen. Diese Verbesserung erkaufte sich Strassen mit 14 zusätzlichen Additionen (im Vergleich zu dem Standard-Algorithmus). Bevor der Frage nachgegangen werden kann, ob dieser Trade-off einen Vorteil bringt, soll der Algorithmus im folgenden kurz beschrieben werden. Im ersten Schritt werden die 3 Matrizen A , B und C in Blöcke aufgeteilt. Jeder Block entspricht bei einer 2×2 Matrix einer Zahl.

$$\begin{pmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{pmatrix} = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix}$$

Anschließend werden 7 neue Matrizen unter Verwendung der definierten Blöcke erstellt. Jeder Block kann unabhängig berechnet werden. Dies ist im Hinblick auf eine spätere Implementierung mit dem Divide-and-Conquer Ansatz wichtig. Konstruktion der neuen Matrizen m_i :

$$m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11})$$

$$m_2 = (a_{10} + a_{11}) * b_{00}$$

$$m_3 = a_{00} * (b_{01} - b_{11})$$

$$m_4 = a_{11} * (b_{10} - b_{00})$$

$$m_5 = (a_{00} + a_{01}) * b_{11}$$

$$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01})$$

$$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11})$$

Durch geschicktes Kombinieren der konstruierten Matrizen können nun die Ergebnisblöcke c_i berechnet werden. Berechnung der Ergebnisblöcke:

$$\begin{pmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{pmatrix} = \begin{pmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{pmatrix}$$

Um zu überprüfen ob Strassens Algorithmus die richtigen Ergebnisse liefert, wird im folgenden exemplarisch die Berechnung von c_{00} nach Strassen und des Standard-Algorithmus verglichen.

Nach dem Standard-Algorithmus ist: $c_{00} = a_{00}b_{00} + a_{01}b_{10}$

Nach Strassen ist:

$$\begin{aligned} c_{00} &= m_1 + m_4 - m_5 + m_7 = (a_{00} + a_{11}) * (b_{00} + b_{11}) \\ &\quad + a_{11} * (b_{10} - b_{00}) - (a_{00} + a_{01}) * b_{11} + (a_{01} - a_{11}) * (b_{10} + b_{11}) \end{aligned}$$

$$\begin{aligned} c_{00} &= a_{00}b_{00} + \cancel{a_{00}b_{11}} + \cancel{a_{11}b_{00}} + \cancel{a_{11}b_{11}} + \cancel{a_{11}b_{10}} - \cancel{a_{11}b_{00}} - \cancel{a_{00}b_{11}} + \cancel{a_{01}b_{11}} \\ &\quad + a_{01}b_{10} + \cancel{a_{01}b_{11}} - \cancel{a_{11}b_{10}} - \cancel{a_{11}b_{11}} \end{aligned}$$

Für c_{00} wäre also gezeigt, dass Strassens Algorithmus identische Resultate liefert.

Analoge Resultate erhält man auch für c_{01} , c_{10} und c_{11} . Es ist nun leicht zu zeigen, dass auch größere Matrizen gemäß der Blockstruktur in Submatrizen eingeteilt und durch den gleichen Algorithmus rekursiv gelöst werden können [an07 S.147].

3.3 Analyse von Strassens Algorithmus

Betrachten man wieder die Multiplikationen als Basis-Operation des Algorithmus, kann das Laufzeitverhalten nach folgender rekursiver Gleichung bestimmt werden:

$$T(n) = 7T(n/2)$$

Pro Rekursionsschritt müssen 7 neue Matrizen berechnet werden. Für jede neue Matrix die berechnet wird, benötigen wir 1 neuen rekursiven Aufruf. Mit jedem rekursiven Aufruf halbiert sich dabei die Problemgröße.

Unter Verwendung des Master-Theorems ($a=7$, $b=2$, $d=0$) ergibt sich eine Laufzeit von:

$$O(n^{\log_2 7}) = O(n^{2.807})$$

Da die Ersparnis bei den Multiplikationen aber nur dadurch erkaufte wird, das 14 zusätzliche Additionen durchgeführt werden, ist es sinnvoll, die gesamten Operationen des Algorithmus zu betrachten. Die Rekursionsgleichung für den gesamten Algorithmus lautet:

$$T(n) = 7T(n/2) + 18(n/2)^2$$

Der erste Teil der Gleichung entspricht den Multiplikationen die wir schon betrachtet haben.

Der zweite Teil entspricht den Additionen. Für jeden Rekursiven Aufruf benötigen wir 18 Additionen. Die Addition von Matrizen hat eine Komplexität von $O(n^2)$. Die Größe der zu addierenden Matrizen hängt - natürlich - auch von der Problemgröße ab.

Unter Verwendung des Master-Theorems ($a=7, b=2, d=2$) ergibt sich ebenfalls eine Laufzeit von:

$$O(n^{\log_2 7}) = O(n^{2.807})$$

Die zusätzlichen Additionen haben also keine Auswirkung auf die Komplexität des Algorithmus.

3.4 Strassens Algorithmus in der Praxis

Wird der Algorithmus von Strassen ohne Veränderungen implementiert, kann er im Vergleich zu dem Standard-Algorithmus kaum einen Nutzen aus seiner überlegenen Effizienzklasse ziehen. Als Grund ist die Anzahl der Operationen für kleine Matrizen bzw. für kleine Werte von n zu nennen. Um dies zu verdeutlichen, vergleichen wir die Gesamtanzahl der Operationen bei beiden Algorithmen. Der Standard-Algorithmus benötigt: $M(n) = 2n^3 - n^2$ Operationen (n^3 Multiplikationen und $(n^3 - n^2)$ Additionen) [we00]. Der Algorithmus von Strassen benötigt im Vergleich: $S(n) = 7 * 7^{\log_2 n} - 6 * 4^{\log_2 n}$ Operationen [we00]. Wenn $n \in 2^k$, kann $S(n)$ zu $S(2^k) = 7n^{\log_2 7} - 6n^2$ vereinfacht werden [we00]. Daraus können wir berechnen das der Algorithmus von Strassen bis $n < 654$ eine größere Anzahl von Operationen benötigt.

Es scheint also unabdingbar zu sein, für eine effiziente Implementierung des Algorithmus, bei geeigneten Werten von n , auf den Standard-Algorithmus zu wechseln. Da Multiplikationen mehr Rechenzeit benötigen als Additionen, ist diese Schwelle nicht bei $n=654$ erreicht. Im Internet kursieren mehrere Empfehlungen für die richtige Wahl der Grenze. Diese liegen alle im Intervall von $16 \leq n \leq 256$. Jedoch ist die Wahl der Grenze auch von der Güte der Implementierung abhängig.

4 Das Closest-Pair-Problem

4.1 Problembeschreibung

Bei dem Closest-Pair-Problem wird die kürzeste Distanz zwischen zwei Punkten in einer Menge von Punkten gesucht. In der Ebene wird somit die Distanz zwischen 2 Punkten gesucht, die die euklidische Distanz ($d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$) minimiert. Anwendung findet der Algorithmus z.B. bei der Radarüberwachung oder in graphischen Anwendungen wie Computerspielen. Der Algorithmus kann mit einigen kleineren Anpassungen auch im N-dimensionalen Raum verwendet werden. Die folgende Beschreibung beschränkt sich aus Gründen der Übersichtlichkeit auf den 2-dimensionalen Fall.

4.2 Der Bruteforce Ansatz

Bei der Bruteforce Implementierung des Problems, wird jeder Punkt mit jedem anderen verglichen. Somit müssen $C(n) = n \cdot (n-1) / 2$ Vergleiche durchgeführt werden. Damit fällt die Bruteforce Variante des Closest-Pair Algorithmus in die Effizienzklasse $O(n^2)$.

4.3 Closest-Pair mit Divide-and-Conquer

Für den Divide-and-Conquer Ansatz wird eine nach ihren X-Koordinaten vorsortierte Liste aller betrachteten Punkte benötigt. Sind die Punkte nicht

sortiert, können Sie z.B. mit Mergesort in $O(n \log n)$ sortiert werden. Ziel der Sortierung ist es einerseits, die Menge der Punkte in konstanter Zeit bei jedem Rekursionsschritt in ca. 2 gleich große Mengen aufzuteilen. Andererseits, durch die Sortierung mehr Information über die Lage der Punkte zu gewinnen. So können bei der späteren Suche, Punkte ausgeschlossen und die Anzahl der Vergleiche reduziert werden. Auf jeder Seite, bzw. in jeder Untermenge werden nun rekursiv die Punkte mit der kürzesten Distanz gesucht. Da die kürzeste Distanz aber auch zwischen 2 Punkten in verschiedenen Mengen liegen kann, müssen die Punkte in der Nähe der Trennlinie mit den Punkten auf der anderen Seite verglichen werden. Bei diesen Vergleichen helfen 2 wichtige Einsichten. Zum Einen müssen nur Punkte in der Nähe der Trennlinie betrachtet werden, die einen maximalen Abstand d_{min} zur Trennlinie besitzen (wobei d_{min} das Minimum der kürzesten Distanz der rechten und linken Teilmengen ist).

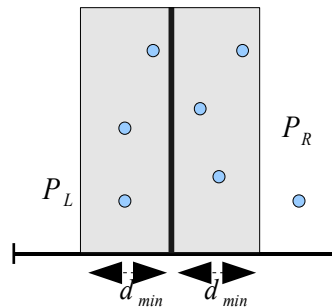


Abbildung 2: Die Idee des Divide-and-Conquer Algo. für das Closest-Pair-Problem[an07 S.151]

Zum Anderen können nur jene Punkte erfolgreich mit einem Punkt p gepaart werden, deren Y-Koordinaten in einem Intervall von $[y-d, y+d]$ um den Punkt p liegen. Während durch die erste Beobachtung die Kandidaten für das Closest-Pair vertikal eingegrenzt werden (siehe Abbildung 2), können sie durch die zweite Beobachtung horizontal eingegrenzt werden. Da jeder Punkt einen

minimalen Abstand d_{min} zu jedem anderen Punkt in der gleichen Teilmenge haben muss, kann aus den beiden Beobachtungen auch abgeleitet werden, das maximal 6 Punkte als Kandidaten für das Closest-Pair mit dem Punkt p in Frage kommen. Die folgende Abbildung soll dies noch einmal verdeutlichen.

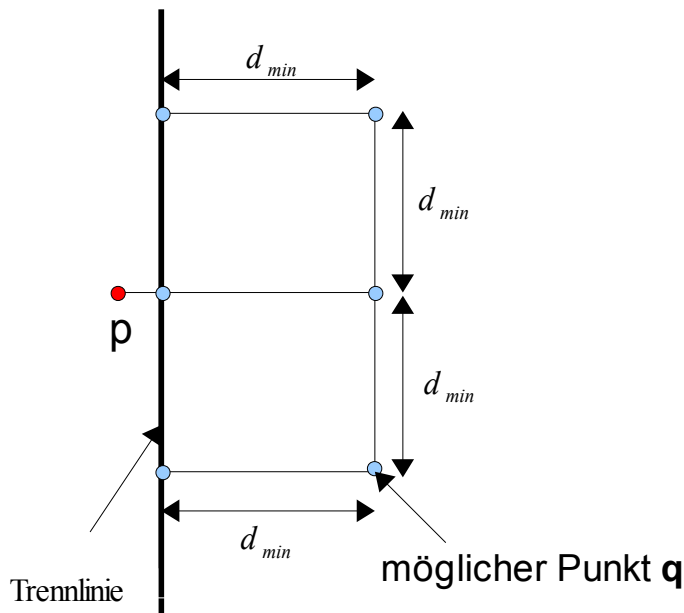


Abbildung 3: Die 6 möglichen Kandidaten (worst case)

Um die Vergleiche in der Nähe der Trennlinie möglichst zeitsparend durchzuführen ist es sinnvoll, die zu vergleichenden Punkte nach ihren Y-Koordinaten zu sortieren. Durch diese Sortierung sinkt die Anzahl der Vergleiche soweit, dass diese mit linearem Zeitaufwand realisiert werden können.

Wird die Sortierung mit Mergesort durchgeführt und am Ende der Rekursion begonnen, ergibt sich der Fall, dass nur einzelne Elemente in eine sortierte Liste eingefügt werden müssen.

In diesem Best-Case läuft Mergesort in $O(n)$.

4.4 Analyse des Divide-and-Conquer Algorithmus

Der Divide-Anteil des Algorithmus, das aufteilen der Menge bzw. das finden der Trennlinie, kann in einer sortierten Liste in konstanter Zeit realisiert werden. Da die Problemgröße bei jedem Schritt halbiert und in beiden Hälften weiter gesucht werden muss, erfordert der Conquer-Anteil $2T(n/2)$ Operationen. Der Combine-Anteil, das Suchen des Closest-Pair in der Nähe der Trennlinie, hat in einer nach Y-Koordinaten sortierten Liste der Kandidaten eine Laufzeitkomplexität von $O(n)$.

Die Gesamtlaufzeit des Algorithmus kann also nach folgender Formel bestimmt werden:

$$T(n) = 2T(n/2) + O(n)$$

Unter Verwendung des Master-Theorems (mit $a=2$, $b=2$, $d=1$) ergibt sich eine Laufzeitkomplexität von $O(n \log n)$.

Besonderheit:

An dieser Stelle sei noch kurz erwähnt, dass der Algorithmus auch in höheren Dimensionen ebenfalls eine Laufzeitkomplexität von $O(n \log n)$ besitzt. Damit unterliegt der Algorithmus nicht dem *curse of Dimensionality*, nachdem die Komplexität eines Algorithmus exponentiell, mit jeder zusätzlichen Dimension wächst.

5 Das Convex-Hull-Problem

5.1 Problembeschreibung

Bei dem Convex-Hull-Problem wird das kleinste konvexe Polygon gesucht, das eine Menge von n Punkten enthält. Ein Polygon wird als konvex bezeichnet, wenn alle Verbindungen bzw. Geraden, zwischen den Punkten des Polygons, innerhalb des Polygons liegen.

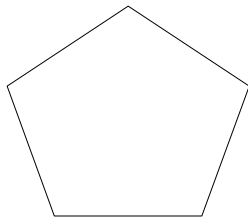


Abbildung 5: Konvexes Polygon

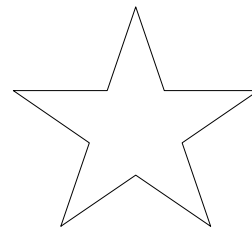


Abbildung 4: Nicht konvexes Polygon

5.2 Ein Brute-force-Ansatz

Ein naiver Brute-force-Ansatz könnte für eine n -elementige Menge alle möglichen n^3 Dreiecke berechnen und für jeden der n Punkte testen ob dieser Punkt innerhalb eines der berechneten Dreiecke liegt. Jeder Punkt der nicht innerhalb eines dieser Dreiecke liegt, wäre ein Punkt der konvexen Hülle. Dieser Brute-force Algorithmus hätte eine Laufzeitkomplexität von $O(n^4)$.

5.3 Convex-Hull mit Divide-and-Conquer

Für den Divide-and-Conquer-Ansatz muss die Menge der Punkte P vorher nach X-Koordinaten sortiert werden. Ist die Menge nicht sortiert, kann sie z.B. mit Quicksort in $O(n \log n)$ sortiert werden. Ebenfalls als ein Schritt der Vorbereitung, können nun die ersten beiden Punkte der konvexen Hülle bestimmt werden. Diese Punkte sind der erste Punkt P_1 und der letzte Punkt P_n , in der sortierten Menge P [an07 S.152]. Alle Punkte die links, bzw. oberhalb der Geraden von $\overrightarrow{P_1 P_n}$ liegen, werden nun der Menge S_1 zugeordnet. Alle Punkte die auf der Geraden $\overrightarrow{P_1 P_n}$ liegen, können keine Punkte der konvexen Hülle sein. Sie können daher aus der Betrachtung herausgenommen werden. Alle anderen Punkte werden der Menge S_2 zugeordnet. Mit P_1, S_1, P_n kann nun die obere Hülle konstruiert werden, mit P_1, S_2, P_n entsprechend die untere Hülle.

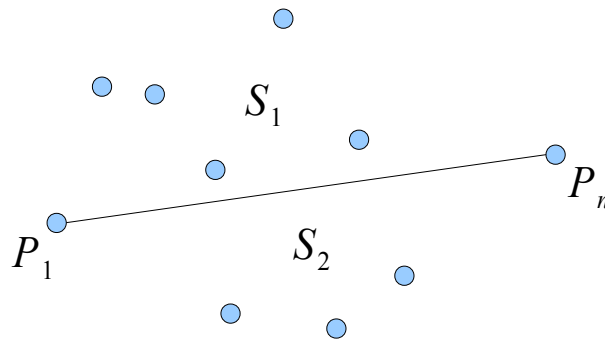


Abbildung 6: Aufteilung der Menge in S_1 und S_2

Nach diesen vorbereitenden Maßnahmen kann der eigentliche Algorithmus aufgerufen werden. Exemplarisch wird das Vorgehen für S_1 beschrieben. Der

erste Schritt besteht darin, in der Menge S_1 den Punkt P_{max} zu finden. P_{max} ist dabei der Punkt in der Menge S_1 , der den größten Abstand zu der Geraden $\overline{P_1 P_n}$ besitzt. Haben 2 oder mehrere Punkte den gleichen Abstand zu der Geraden $\overline{P_1 P_n}$, wird P_{max} so gewählt, das der Winkel zwischen P_{max} und $\overline{P_1 P_n}$ maximiert wird [an07 S.152]. Diese Wahl von P_{max} in Abhängigkeit des Winkels dient nicht dem Finden eines Punktes der konvexen Hülle (alle Punkte mit dem gleichen Abstand wie P_{max} zu der Geraden $\overline{P_1 P_n}$ sind Punkte der konvexen Hülle), sondern der besseren Partitionierung der verbliebenen Menge, und damit, der Reduzierung der Laufzeit. Da P_{max} den größten Abstand zur Geraden $\overline{P_1 P_n}$ besitzt, muss P_{max} ein Punkt der konvexen Hülle sein. Außerdem können alle Punkte von der weiteren Betrachtung ausgeschlossen werden, die innerhalb des Dreiecks das durch P_1, P_{max} und P_n aufgespannt wird liegen. Alle Punkte die links von der Geraden $\overline{P_1 P_{max}}$ liegen, werden der neuen Menge $S_{1,1}$ zugeordnet. Alle Punkte die links von der Geraden $\overline{P_{max} P_n}$ liegen, werden der Menge $S_{1,2}$ zugeordnet. Für beide Mengen wird wie eben beschrieben analog verfahren. Das Verfahren bricht ab, wenn keine neue Menge mehr gebildet werden kann, weil keine Punkte links von den Geraden mehr gefunden werden.

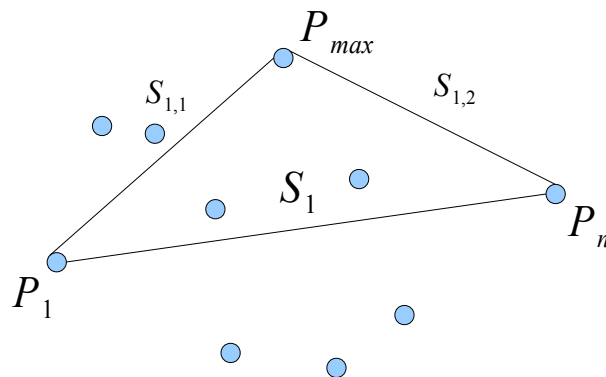


Abbildung 7: Aufteilung der Menge S_1 in $S_{1,1}$ und $S_{1,2}$

5.4 Implementierung der geometrischen Funktionen

Die benötigten geometrischen Funktionen können mit Hilfe einer einfachen Determinanten implementiert werden. Wenn $P_1=(x_1,y_1)$, $P_2=(x_2,y_2)$ und $P_3=(x_3,y_3)$, dann ist das Ergebnis der folgenden Determinanten positiv, unter der Voraussetzung das P_3 links von $\overline{P_1P_2}$ liegt. Das Ergebnis ist 0 falls die Punkte alle auf einer Geraden liegen, andernfalls negativ[an07, S.153].

$$\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} = x_1 y_2 + x_3 y_1 + x_2 y_3 - x_3 y_2 - x_2 y_1 - x_1 y_3$$

Darüber hinaus entspricht der Betrag der Determinanten der Hälfte der Fläche die durch P_1, P_2 und P_3 aufgespannt wird[an07, S.153]. Durch die Verwendung der Determinanten ist es in konstanter Zeit möglich, den Abstand und die Position eines Punktes, in Relation zu einer Geraden zu bestimmen. Auf das Beispiel bezogen kann P_{\max} , $S_{1,1}$ und $S_{1,2}$ mit linearem Zeitaufwand bestimmt werden.

5.5 Analyse des Divide-and-Conquer Algorithmus

Der Divide-Anteil des Algorithmus, das Aufteilen der Menge S_i in $S_{i,1}, S_{i,2}$ kann mit linearem Aufwand, also in $O(n)$ realisiert werden. Jeder Punkt in S_i wird mit der Geraden $\overrightarrow{p_i p_{max}}$ bzw. $\overrightarrow{p_{max} p_j}$ verglichen und abhängig davon der Menge $S_{i,1}$ oder $S_{i,2}$ zugeordnet. Der Combine-Anteil, das Suchen von P_{max} hat ebenfalls eine Komplexität von $O(n)$, da für jeden Punkt die Entfernung zu der Geraden berechnet werden muss. Leider können die beiden Schritte nicht zusammengefasst werden, da die Lage der Punkte erst bestimmt werden kann, nachdem P_{max} gefunden wurde. Der Combine-Anteil des Algorithmus, das Sammeln der Punkte der konvexen Hülle in einer Liste, hat ebenfalls eine Komplexität von $O(n)$. Die Laufzeit des Algorithmus kann also nach folgender Formel bestimmt werden:

$$T(n) = 2T(n/2) + O(n)$$

Unter Verwendung des Master-Theorems ergibt sich eine Laufzeitkomplexität von $O(n \log n)$.

6 Zusammenfassung

Divide-and-Conquer ist eine nützliche Designtechnik um komplizierte Probleme zu lösen. Anstatt sich des gesamten Problems anzunehmen, wird es zerlegt und Stück für Stück gelöst. Mit Divide-and-Conquer können wie das Beispiel des Closest-Pair-Problem zeigt, in manchen Problembereichen die effizientesten Algorithmen geschrieben werden. Das Master-Theorem bietet dabei ein einfaches und schnelles Werkzeug, um die Effizienz von Divide-and-Conquer Algorithmen zu analysieren. Im Bereich der Matrixmultiplikation bietet der Algorithmus von Strassen ebenfalls eine herausragende Performance. So bleibt die Erkenntnis, dass die wahre Herausforderung eines guten Divide-and-Conquer Algorithmus darin besteht, das Problem zerlegen und wenn nötig, dessen Lösungen zusammen zu setzen.

7 Literaturverzeichnis

[an07] Anany Levitin: The Design and Analysis of Algorithms.2007

[ti02] Jürgen Tietze: Einführung in die angewandte Wirtschaftsinformatik. 2000

[we00] Erik Weisstein: <http://mathworld.wolfram.com/StrassenFormulas.html>