Rheinische Friedrich Wilhelms
Universität Bonn

Institut für Informatik, Abteilung I

Diplomarbeit

zum Thema

Triangulierungskonstruktion aus Distanzmatrizen

eingereicht von Daniel Krämer
Matrikelnummer 1429652
Bonn, 10. Juni 2006

bei
Prof. Dr. Rolf Klein

Danksagungen

Ich danke Herrn Prof. Dr. Klein für die Möglichkeit, in der Abteilung I des Instituts für Informatik meine Diplomarbeit anzufertigen.
Speziell möchte ich mich bei meinem Betreuer Ansgar Grüne für alle Anregungen und Hilfestellungen bedanken, sowie bei Melanie Elm fürs Korrekturlesen.
Ein besonderer Dank gilt meiner Familie und meinen Freunden für die Unterstützung während meines Studiums.

Eidesstattliche Erklärung

Mit der Abgabe der Diplomarbeit versichere ich, gemäß 19 Absatz 7 der DPO vom 15. August 1998, dass ich die Arbeit selbstständig durchgeführt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

10. Juni 2006

Daniel Krämer

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

In der algorithmischen Geometrie stößt man des öfteren auf Probleme, die sich mithilfe ebener Triangulationen lösen lassen. Beispielsweise können wir zu einer gegebenen Punktmenge in der Ebene das Voronoi-Diagramm berechnen, indem wir das duale Problem, die Delaunay-Triangulation zu finden, lösen (siehe Buch von Rolf Klein, [1], Kapitel 5). Außerdem lassen sich in Triangulationen bestimmte Aussagen über Beschränkung von Kanten-, Knoten- und Flächenanzahl treffen, die bei Laufzeitabschätzungen hilfreich sind, z.B. die Eulersche Formel (Kap. 2.2 und 2.5).
Desweiteren finden sich Triangulationen häufig in der Computergrafik wieder, wenn es z.B. darum geht, ein eingescanntes Modell aus Eingabedaten zu rekonstruieren. Wir erhalten dann ein Relief des Modells, wobei die Oberfläche durch Triangulation der eingescannten Punktmenge nachgebildet wird. Je nachdem wie „dicht“ wir die Scanpunkte setzen (bzw. wie klein die entstehenden Dreiecke werden) erhalten wir eine höhere Genauigkeit, das Bild wird „glatter“. Die Delaunay-Triangulation ist hierbei sehr beliebt, da sie die kleinsten Innenwinkel maximiert, das heißt sie vermeidet lange, spitze Dreiecke, welche zu Rundungsfehlern führen würden.

1.2 Gliederung

In dieser Arbeit beschäftigen wir uns damit, zu einer in Form einer Distanzmatrix D eines metrischen Raumes[1] gegebenen endlichen Punktmenge eine ebene Triangulation zu konstruieren, falls dies möglich ist. Dazu gehen

[1] ein Tupel bestehend aus einer Menge M und einer Distanzfunktion d, die die Entfernungen der Elemente von M beschreibt, siehe Kapitel 2.1

wir zunächst davon aus, daß D von einer ebenen Triangulation T stammt und konstruieren diese, ohne daß wir zusätzlich die enstandenen Kanten auf Schnitt testen, oder darauf, ob alle Kanten die richtige Länge haben. Der Algorithmus stellt nur sicher, daß unter der Annahme und unter der weiteren Vorraussetzung, daß der ersten Kante feste Startkoordinaten zugewiesen werden, alle anderen Punkte notwendig so gesetzt werden, wie wir sie setzen. Außerdem brechen wir an einigen Stellen ab, falls Kantenzahlen zu groß werden, um die Laufzeit nicht zu gefährden, denn wir werden zeigen, daß wir die Anzahl der in der Triangulation vorkommenden Kanten vorher abschätzen können. Am Ende testen wir dann, ob die Kantenlängen stimmen bzw. ob unzulässige Schnittpunkte existieren.
Der Algorithmus beginnt zunächst damit, die direkten Verbindungen zu bestimmen, daß heißt die in T vorkommenden Kanten. Dann stellen wir fest, welche davon zur äußeren Hülle gehören, also nur auf dem Rand eines leeren Dreiecks der endgültigen Triangulation, nicht wie bei inneren Kanten in zwei leeren Dreiecken, vorkommen. Diese müssen einen Zyklus bilden, den wir berechnen können, die äußere Hülle. Diesen Zyklus beginnen wir dann schichtenweise zu triangulieren, bis er vollständig ausgefüllt ist und alle Kanten „verbaut“ wurden. Wir beginnen mit einem leeren Startdreieck, welches mindestens eine Kante der äußeren Hülle enthält, die Startkante (i, j). Anhand dieses Dreiecks legen wir ein Koordinatensystem fest. Jedes leere Dreieck der Triangulation besitzt mindestens eine innere Kante, sofern sie nicht nur aus einem Dreieck besteht. Von der inneren Kante (j, k) des Startdreiecks ausgehend berechnen wir das andere angrenzende leere Dreieck, das (j, k) enthält und nennen die neue Spitze k. Diesen Schritt wiederholen wir solange, bis das berechnete Dreieck die zu (i, j) benachbarte Außenkante enthält. Wir fahren fort, bis das entstandene Dreieck wieder die Startkante enthält. Dann haben wir den Zyklus der Außenkanten konstruiert und kennen die „Form“ der Triangulation. Unter Umständen sind dabei abgeschlossene innere Gebiete entstanden. Diese werden weiter trianguliert, falls möglich, bis alle Knoten und Kanten verbaut sind. Existiert eine ebene Triangulation zu der gegebenen Distanzmatrix, so berechnet das vorgestellte Verfahren diese in $O(n^3)$.

1.3 Verwandte Probleme

Verwandte Problemstellungen, die sich mit der geometrischen Realisierung von metrischen Distanzmatrizen beschäftigen, finden sich z.B. bei H.-J. Bandelt in [7]. U.a. existiert bereits ein algebraisches Kriterium, wann ein metrischer Raum von einem Baum stammt:

$d(i,j) \leq max(d(i,k)+d(j,l), d(i,l)+d(j,k)) \ \forall \ i,j,k,l \in M.$

Ist dieses Kriterium erfüllt, so spricht man auch von einer *Baummetrik*. Es läßt sich nach [7] algorithmisch in $O(n^2 \log n)$ überprüfen (für $|M| \geq 3$), ob eine Distanzmarix $D := d(i,j)$ von einer Baummetrik herrührt. Dabei zeigt man zuerst, daß D genau dann einer Baummetrik entspricht, wenn das einfacher zu überprüfende Kriterium
$K := d(i,k) + d(j,r) = d(i,r) + d(j,k) \ \forall k \neq i,j,r \in M$ erfüllt ist. Dabei kann man r beliebig wählen, aber es muß $d(i,r) + d(j,r) - d(i,j) \ \forall i,j \neq r$ maximal sein. Außerdem bilden d und $M - \{i\}$ genau dann eine Baummetrik, wenn d und M eine Baummetrik sind.
Der Test erfolgt rekursiv. Zunächst wählt man ein beliebiges Element $r \in M$ und sortiert die restlichen Tupel (i,j) mit $i,j \neq r$ in $O(n^2 \log n)$ absteigend nach ihren Werten in $d(i,r) + d(j,r) - d(i,j)$ in eine Liste ein. Dann überprüft man die Eigenschaft K für das erste Element in der Liste, also dasjenige (i,j), für welches $d(i,r)+d(j,r)-d(i,j)$ maximal ist. Dies verursacht Kosten von $O(n)$, da man für k $n-3$ Möglichkeiten hat. Sollte K erfüllt sein, so streicht man i aus M und fährt mit dem nächstgrößeren Eintrag (j,k) mit $j,k \neq i$ in der Liste fort. Man bricht ab, sobald K nicht mehr erfüllt ist oder alle Elemente erfolgreich getestet wurden. Falls die Eigenschaft immer erfüllt war, bilden d und M eine Baummetrik.

In den Arbeiten von H.J. Bandelt und A. Dress ([8] und [9]) findet sich eine weitere Anwendung für Distanzmatrizen aus der Bioinformatik. In sogenannten *phylogenetischen Bäumen*, welche evolutionäre Verwandtschaftsbeziehungen einzelner Organismen darstellen, symbolisieren die Distanzen der zugehörigen Matrix den Grad der Verwandtschaft. Diese Bäume lassen sich mithilfe der Zerlegung endlicher metrischer Räume (split decomposition) analysieren.

Ein weiteres verwandtes Problem ist z.B. das *Euclidean distance matrix completion problem*, kurz EDM (siehe Arbeiten von Monique Laurent [5], Kapitel 3). Dabei geht es darum, festzustellen, ob sich eine gegebene, symmetrische $n \times n$-Matrix A, die Nullen als Diagonaleinträge und einige offene (bzw. unbekannte) Einträge enthält, zu einer euklidischen Distanzmatrix eines metrischen Raumes vervollständigen läßt. Dies ist genau dann der Fall, wenn n k-dimensionale Vektoren v_i existieren, deren euklidische Abstände den Einträgen in A entsprechen. Die Schwierigkeit des Problems hängt unter anderem von der Dimension k der Vektoren ab. Für $k \geq 2$ ist dieses Problem NP-hart (siehe Arbeiten von J.B. Saxe [6]).

Kapitel 2

Metrische Räume

2.1 Distanzmatrizen

Definition 2.1. *Ein endlicher* **metrischer Raum** *ist ein Tupel (M,d), wobei M eine endliche Menge und* $d : M \times M \to \mathbb{R}$ *eine Abstandsfunktion mit folgenden Eigenschaften ist:*

$\forall\ x, y, z \in M$ *gilt:*

1. $d(x, y) \geq 0$, *wobei* $d(x, y) = 0 \Leftrightarrow x = y$ *(Positivität)*
2. $d(x, y) = d(y, x)$ *(Symmetrie)*
3. $d(x, y) \leq d(x, z) + d(z, y)$ *(Dreiecksungleichung)*

Ein endlicher metrischer Raum wird häufig durch die Matrix D der Distanzen (Distanzmatrix) beschrieben.
Bsp.:

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 1 & 0 & 2 & 3 \\ 2 & 2 & 0 & 1 \\ 3 & 3 & 1 & 0 \end{pmatrix}$$

Die Elemente von M seien von 1 bis n durchnummeriert. Der Eintrag an der Position (i, j) in D ist der Abstand $d(i, j)$ der beiden Elemente i und j. Aus 1. folgt, daß die Diagonaleneinträge von $D = 0$ und daß alle anderen Einträge > 0 sind. Aus 2. folgt, daß D symmetrisch ist.

2.2 Eigenschaften von Graphen

Ein *Graph* G besteht aus einer endlichen Knotenmenge V und einer Kantenmenge E mit Elementen aus $V \times V$. Bildet man die Knoten auf paarweise verschiedenen Punkten im $\mathbb{R}^2$ ab und verbindet diese entsprechend, ohne dabei unterwegs andere Knoten zu besuchen, so sprechen wir von einem *geometrischen Graph*. Im weiteren Verlauf ist immer, wenn von Graphen die Rede ist, der geometrische Graph gemeint. Die Zusammenhangskomponenten von $\mathbb{R}^2$ ohne G heißen Flächen von G. Für eine ausführliche Definition sei auf Kapitel 1.2.2 im Buch von Rolf Klein [1] verwiesen.
Sei v die Anzahl der Knoten, e die Anzahl der Kanten und f die Anzahl der Flächen eines Graphen G. Außerdem wollen wir davon ausgehen, daß G *schlicht*, *zusammenhängend*, *nichtleer* und *planar* (d.h. kreuzungsfrei in der Ebene) realisierbar ist. Dann gilt die **Eulersche Formel** für zusammenhängende Graphen:

$$v - e + f = 2.$$

Beweis. (vgl. [1], Kapitel 1.2.2)
Besteht G nur aus einem Knoten, so ist $v = 1$, $f = 1$, $e = 0$ und die Behauptung erfüllt. Bei Hinzunahme eines weiteren Knotens und einer Kante (der Graph muß ja zusammenhängend sein) erhöhen sich v und e jeweils um 1, die Formel gilt weiterhin. Mit jeder weiteren Kante, die der neue Knoten impliziert, erhöhen sich e und f jeweils um 1, da eine neue Fläche entsteht. Dies ist analog zu dem Fall, in welchem eine neue Kante zwischen zwei schon vorhandenen Knoten hinzukommt. Falls ein neuer Knoten auf einer bereits vorhandenen Kante liegt, so erhöhen sich v und e jeweils um 1, die Formel gilt weiterhin. □

Als Folgerungen ergeben sich außerdem, wenn wir zusätzlich annehmen, daß der Grad[1] aller Knoten ≥ 3 ist:

1) $v \leq \frac{2}{3}e$
2) $v < 2f$
3) $e < 3f$
4) $f \leq \frac{2}{3}e$
5) $e < 3v$
6) $f < 2v$

[1] der Grad eines Knotens i bezeichnet die Anzahl der an i anliegenden Kanten

Aus 1) und 5) ergibt sich die für uns später wichtige Eigenschaft:

7) $\frac{3}{2}v \leq e < 3v$

Für den Beweis benötigen wir den zu G *dualen* Graphen G^*. Diesen erhalten wir folgendermaßen:
1.Wähle einen Punkt ${p_F}^*$ innerhalb jeder Fläche F von G. Diese Punkte sind die Knoten von G^*.
2.Verbinde ${p_F}^*$ mit ${p_{F'}}^*$ in G^* durch eine Kante e^* genau dann, wenn F und F' in G an eine gemeinsame Kante e angrenzend sind und so, daß e^* nur e kreuzt.
Der *duale* Graph G^* ist dann laut Definition kreuzungsfrei.

Beweis. (vgl. [1], Kapitel 1.2.2):

1) Da von jedem Knoten mindestens drei Kanten ausgehen, wir aber jede Kante doppelt zählen, wenn wir über die Anzahl der Knoten aufsummieren, ergibt sich: $2e \geq 3v$.

2) Setzt man 1) in die Eulerformel ein, so ergibt sich:

$\frac{2}{3}e - e + f \geq 2$
$\Leftrightarrow -\frac{1}{3}e + f \geq 2$
$\Leftrightarrow f \geq 2 + \frac{1}{3}e$
$\Rightarrow 2f \geq 4 + \frac{2}{3}e > v$

Das letzte „$>$“ ergibt sich aus 1).

3) Setzt man 2) in die Eulerformel ein, so erhält man:

$3f = 2f + f \geq v + f = 2 + e > e$

4) m_i bezeichne die Anzahl der Kanten auf dem Rand der i-ten Fläche, ist also laut Vorraussetzung ≥ 3, denn gäbe es eine Fläche, deren Rand nur aus zwei Kanten bestünde, so müßten diese Kanten zwischen denselben Knoten sein. Dann wäre der Graph aber nicht schlicht. Dies gilt auch für die äußere, den Graph „umschließende“ Fläche. Es gilt:

$3f \leq \sum_{i=1}^{f} m_i \leq 2e$

Das zweite „$\leq$“ folgt daraus, daß jede Kante maximal in zwei Flächen vorkommt, also doppelt gezählt wird.

5) Wir betrachten den zu G dualen Graphen G^*. Da in G alle Flächen mindestens 3 Randkanten haben, hat jeder Knoten in G^* einen Grad ≥ 3. Laut 3) gilt: $e^* < 3f^*$
Da aber $e = e^*$ und $v = f^*$ gilt für G $e < 3v$.

6) $e < 3v$ (laut 5))
$\Leftrightarrow \frac{2}{3}e < 2v$
$\Rightarrow f \leq \frac{2}{3}e < 2v$

Das erste „$\leq$“ folgt aus 4). □

2.3 Definition Triangulation

Normalerweise versteht man unter einer *Triangulation eines Polygons* P eine maximale Menge sich nicht kreuzender Diagonalen von P zusammen mit den Randkanten (vgl.[1], Kapitel 4.2). Andererseits ist eine *Triangulation einer Punktmenge* eine maximale Menge von Kanten, die sich nicht kreuzen. Diese Definition beinhaltet u.a. die konvexe Hülle der Punktmenge. In unserer Problemstellung haben wir jedoch nur die Kanten (bzw. die Abstände der Knoten zueinander) in Form der Distanzmatrix gegeben. Wir verwenden deshalb eine dritte, an unser Problem angepasste Definition. Eine Triangulation in unserem Sinne ist eine maximale Menge von Kanten, die sich nicht kreuzen und die innerhalb einer vorgegebenen Hülle verlaufen. Der Unterschied zur *Triangulation eines Polygons* ist, daß es Knoten im Inneren der Hülle gibt, die in der Triangulation vorkommen müssen. Der Unterschied zur *Triangulation einer Punktmenge* ist, daß die äußere Hülle nicht notwendig konvex ist. Formal also:

Definition 2.2. *Eine* **Triangulation** *in unserem Sinne ist ein schlichter, kreuzungsfreier, geometrischer Graph $T = (V, E)$ in der Ebene, bei dem jede Kante ein Liniensegment ist, und der folgende zusätzliche Bedingungen erfüllt:*
E enthält genau eine Teilmenge R (die der **Randkanten***), die eine einfache geschlossene polygonale Kette bildet. Sei P das durch R berandete, abgeschlossene Polygon (die* **äußere Hülle***). Dann muß E eine maximale (also nicht erweiterbare) Menge von kreuzungsfreien, geradlinigen Kanten über V sein, die alle ganz in P liegen.*

Laut dieser Definition sind alle entstandenen inneren Flächen Dreiecke, denn gäbe es Flächen mit mehr als drei Ecken, so könnte man noch zusätzliche Kanten hinzufügen, die diese unterteilen würden. Die Kantenmenge wäre also noch nicht maximal. Die Fälle in Abb. 2.1 - 2.3 schließen wir aus, da wir später die die Triangulation bis auf Rotation, Translation und Spiegelung eindeutig bestimmen möchten. Dreiecke, welche keine weiteren Dreiecke beinhalten, bezeichnen wir als **leere** Dreiecke. Konstruieren wir zu einer bereits verbauten Kante (i, j) das andere angrenzende leere Dreieck $\triangle(i, j, k)$, so bezeichnen wir k als neu konstruierte **Spitze**.

Satz 2.1. *Wenn zu einem endlichen metrischen Raum (M, d) eine Triangulation T gemäß Definition 2.2 existiert, deren Kürzeste-Wege-Distanz der Metrik d entspricht, dann ist T bis auf Rotation, Translation und Spiegelung eindeutig bestimmt.*

Beweis. Den Beweis verschieben wir auf später (Kap. 4.6), da er sich leicht anhand des Algorithmus nachvollziehen läßt. □

In den Fällen der Abbildungen 2.1 und 2.2 wäre die Form der äußeren Hülle nicht eindeutig festgelegt. Daher wollen wir diese Fälle nicht als Triangulation betrachten. Die Situation in Abb. 2.3 möchten wir aus dem gleichen Grund ausschließen. Sie kann jedoch gar nicht eintreten, wenn wir für die Distanzmatrix fordern, daß jeder Knoten i zu jedem anderen einen endlichen Distanzwert $\neq 0$ aufweist. In diesem Fall muß mindestens eine Kantenfolge existieren, so daß jeder Knoten von i aus erreicht werden kann.

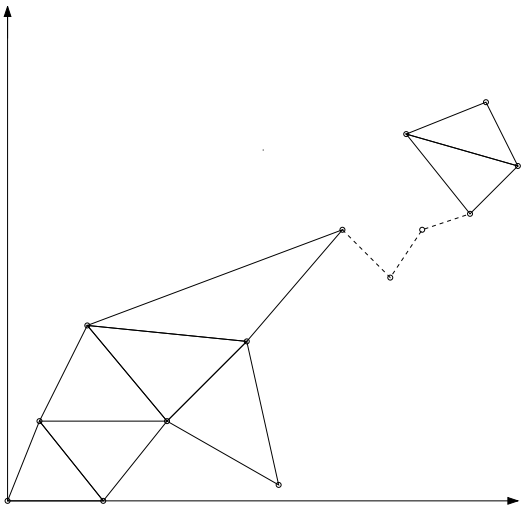

Abbildung 2.1: Keine Triangulation, denn wir haben nicht ein geschlossenes Polygon, in dem alle anderen Kanten liegen.

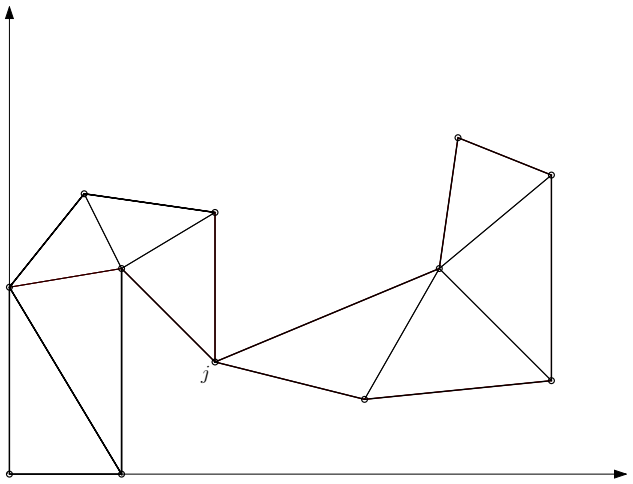

Abbildung 2.2: Keine Triangulation, denn wir haben zwei geschlossene Polygone.

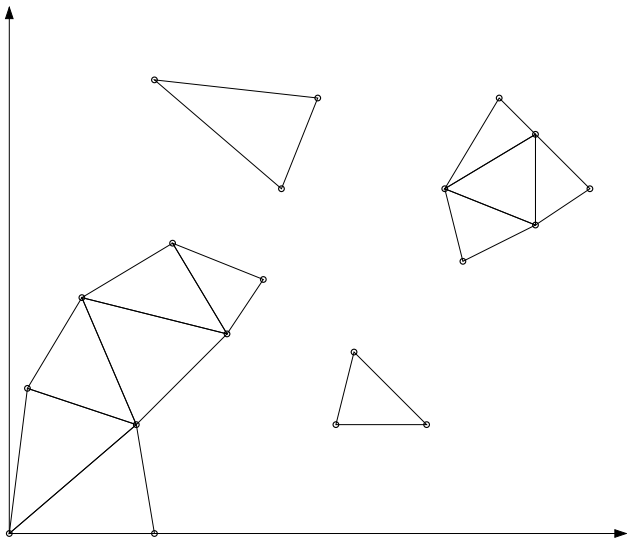

Abbildung 2.3: Keine Triangulation. Mehrere Zusammenhangskomponenten können nicht auftreten, wenn alle Einträge von $D > 0$ und $< \infty$ sind (außer den Diagonaleinträgen).

2.4 Zusammenhang zwischen Metrik und Graph

Man kann jeden endlichen metrischen Raum als Graphen derart darstellen, daß zwei Punkte x, y genau dann durch eine Kante verbunden werden, wenn gilt: $d(x,y) < d(x,z) + d(z,y)\ \forall\ z$.
Gewichten wir die entstehenden Kanten mit der zugehörigen Distanz, so entspricht die Metrik d der Kürzesten-Wege-Distanz im Graphen. Dies folgt leicht per Induktion, wenn wir uns vorstellen, daß der vollständige Graph iterativ um alle Kanten (m,n) ausgedünnt wird, die einen Zwischenpunkt z haben. Anschaulich bedeutet die Eigenschaft, daß wir zwei Knoten x und y genau dann durch eine Kante verbinden, wenn jeder Weg von x nach y über einen dritten Knoten z länger ist, als der direkte Weg von x nach y. Wir sprechen dann von **direkten Verbindungen**. In der Distanzmatrix sieht man nicht, ob die Distanzen über Zwischenpunkte ermittelt wurden oder äquivalent zu direkten Verbindungen im Graphen sind, weil wir für Metriken nur die Dreiecksungleichung (Eigenschaft 3 in 2.1) fordern. Das heißt, wir schließen nicht aus, daß ein Zwischenpunkt existiert, über den der Abstand zwischen x und y ermittelt wurde. Deshalb müssen wir in einem ersten Schritt diese Eigenschaft für jeden der n^2-vielen Kantenkandidaten überprüfen. Das heißt, wir ermitteln zunächst die direkten Verbindungen, indem wir jede mögliche Kante auf $n-2$ Zwischenpunkte testen. Damit ergibt sich als Schranke $O(n^3)$ für die Bestimmung der direkten Verbindungen. Diese Schranke ist ausschlaggebend für die Laufzeit des in Kapitel 4 vorgestellten Triangulationsalgorithmus.

2.5 Die Eulerformel für Triangulationen

Da in einer ebenen Triangulation unter Umständen Randknoten existieren, die nur vom Grad 2 sind, wollen wir nun untersuchen, wie sich 1.-7. aus 2.2 in diesem Fall verhalten. Zu diesem Zweck denken wir uns einen zusätzlichen Knoten x außerhalb unserer Triangulation und verbinden x direkt mit jedem Randknoten. Dadurch ist jeder Knoten vom Grad ≥ 3, (auch x, denn die Triangulation muß ja mindestens drei Randknoten aufweisen). e, v und f seien die Anzahl der Kanten, Knoten und Flächen in der ursprünglichen Triangulation. Die Anzahl der „neuen“ Kanten zwischen x und den Randknoten bezeichnen wir mit ${k_x}^2$. Ab der zweiten „neuen“ Kante erhöht sich die Anzahl der Flächen mit jeder Kante um 1. Also gibt es insgesamt $k_x - 1$ zusätzliche Flächen. Daraus folgt:

[2] k_x entspricht also der Anzahl der Randknoten

1. $v + 1 \leq \frac{2}{3}(e + k_x)$
$\Leftrightarrow v \leq \frac{2}{3}(e + k_x) - 1 \leq \frac{2}{3}(e + k_x)$

2. $v + 1 < 2(f + k_x - 1)$
$\Leftrightarrow v < 2(f + k_x) - 2 - 1 \leq 2(f + k_x)$

3. $e + k_x < 3(f + k_x - 1)$
$\Leftrightarrow e < 3f + 2k_x - 3 < 3f + 2k_x$

4. $f + k_x - 1 \leq \frac{2}{3}(e + k_x)$
$\Leftrightarrow f \leq \frac{2}{3}e - \frac{1}{3}k_x + 1 \leq \frac{2}{3}e + 1$

5. $e + k_x < 3(v + 1)$
$\Leftrightarrow e < 3v - k_x + 3 < 3v + 3$

6. $f + k_x - 1 < 2(v + 1)$
$\Leftrightarrow f < 2v - k_x + 2 + 1 < 2v + 3$

7. $\frac{3}{2}v - k_x \leq e < 3v + 3$

In ebenen Triangulationen gilt außerdem (siehe [10]), wenn $r := k_x$ die Anzahl der Kanten der äußeren Hülle ist:

8. $e = 3v - r - 3$
9. $f - 1 = 2v - r - 2$.

Beweis. **8.** Es gilt $3(f - 1) = 2e - r$, denn $f - 1$ ist die Anzahl der Dreiecke und jede Kante außer den r-vielen Außenkanten kommt in zwei Dreiecken vor. Eingesetzt in die Eulerformel ergibt sich:

$3v - 3e + (2e - r + 3) = 6$
$\Leftrightarrow 3v - e - r = 3$
$\Leftrightarrow -e = 3 - 3v + r$
$\Leftrightarrow e = 3v - r - 3$ □

Beweis. **9.**

Setzt man dieses Ergebnis für e in die Eulerformel ein, und löst diese nach f auf, so erhält man:

$v - (3v - r - 3) + f = 2$
$\Leftrightarrow -2v + r + 3 + f = 2$

$\Leftrightarrow f = 2v - r - 1$
$\Leftrightarrow f - 1 = 2v - r - 2$ □

Damit wissen wir also, sobald wir die Außenkanten berechnet haben, wie viele Kanten bzw. Dreiecke die gesamte Triangulation enthält. Solange wir die Anzahl der Außenkanten nicht kennen, muß e in folgendem Intervall liegen:

10. $2v - 3 \leq e \leq 3v - 6$.

Beweis. r muß im Intervall $[3; v]$ liegen, denn es kann maximal v Außenknoten geben und es muß mindesten drei geben, da sonst keine Triangulation vorliegen würde.
Setzen wir die beiden Intervallgrenzen für r in $e = f(r) = 3v - r - 3$ ein, so erhalten wir Maximum und Minimum, da $f(r)$ linear in r und damit monoton fallend ist. □

Eigenschaft **10.** können wir überprüfen, sobald wir die direkten Verbindungen bestimmt haben und damit die Anzahl der Kanten kennen. Unter Umständen können wir dann eine Triangulation bereits ausschließen.

Kapitel 3

Die äußere Hülle

3.1 Direkte Verbindungen

Die folgende Funktion $direkteVerbindungen()$ bestimmt zu einer gegebenen Distanzmatrix D eines metrischen Raumes (M, d) die Matrix der direkten Verbindungen V. $V(i, j)$ ist genau dann $null$, wenn es keine direkte Verbindung zwischen i und j gibt (für $i \neq j$), bzw. wenn $i = j$ ist. Ansonsten werden die Abstände aus D übernommen.
Der zugehörige abstrakte Graph $G = (M, E)$ sei so definiert, daß $(i, j) \in E$ genau dann wenn $V(i, j) \neq 0$. Zu einem Knoten $v \in M$ sei $N(v)$ die Menge der Nachbarn von v in G.
Für jede Distanz (i, j) unterhalb der Diagonalen (D ist symmetrisch) testet die Funktion, ob für irgendeinen Knoten z die Zwischenpunkteigenschaft $D(i, j) = D(i, z) + D(z, j)$ erfüllt ist. In diesem Fall existiert keine direkte Verbindung, $V(i, j)$ bzw. $V(j, i)$ wird nicht gesetzt. Andernfalls werden die Distanzen aus D übernommen. [1]
Wir ermitteln dabei die Anzahl e der direkten Verbindungen und überprüfen anschließend 10. aus 2.5, also ob $2n - 3 \leq e \leq 3n - 6$. Liegt e nicht in diesem geforderten Intervall, so können wir bereits an dieser Stelle eine ebene Triangulation ausschließen.

[1] der Übersichtlichkeit wegen möchten wir ab jetzt davon ausgehen, daß sobald wir eine Eintragung $M(i, j)$ in einer Matrix vornehmen, gleichzeitig der Eintrag $M(j, i)$ aktualisiert wird, da alle vorkommenden Matrizen symmetrisch sind

funktion direkteVerbindungen()

(**Intialisierung**)
initialisiere die Matrix V und Kantenzahl e mit *null*;

(**Berechnung der direkten Verbindungen**)
for each (Spalte j von D)
for each (Zeile i der aktuellen Spalte unterhalb der Diagonalen)
$V(i,j) := D(i,j)$;
for each (möglichen Zwischenpunkt $z \neq i,j$)
if $(D(i,j) = D(i,z) + D(z,j))$ $V(i,j) := null$;
end for
zähle dabei die in V verbleibenden Kanten e;
end for
end for
if $(2n-3 \leq e \leq 3n-6 \neq true)$ **STOP**;

V enthält nun alle durch D implizierten direkten Verbindungen. Diese bilden die Kanten der Triangulation.

Beweis. Wir haben alle möglichen Knotenkombinationen (i,j) auf $n-2$-viele Zwischenpunkte z getestet. Da $D(i,j) = D(j,i)$ laut 2. in Definition 2.1 gilt, genügt es, die untere Hälfte von D zu betrachten. Existiert kein Zwischenpunkt für (i,j), so müssen i und j laut 2.4 direkt verbunden werden, denn dann gilt laut 3. in 2.1 $D(i,j) < D(i,z) + D(z,j)$ $\forall z$. □

3.2 Berechnung der Außenkanten

3.2.1 Erkennungsmerkmale

In diesem Kapitel wird das Problem gelöst, zu einer gegebenen Distanzmatrix D bzw. der entsprechenden Verbindungsmatrix V die Menge der äußeren Kanten einer Triangulation (falls diese existiert) zu bestimmen.

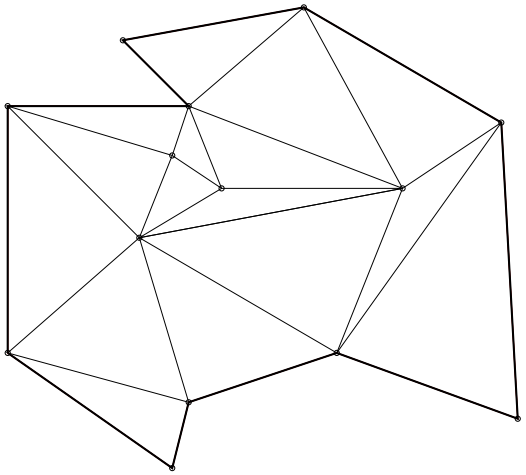

Abbildung 3.1: Eine ebene Triangulation, die äußeren Kanten sind fettgedruckt.

Vorüberlegung : Eine Kante (i, j) liegt genau dann auf der äußeren Hülle einer Triangulation, wenn sie in nur einem leeren Dreieck vorkommt. Dabei können anschaulich folgende Fälle auftreten:

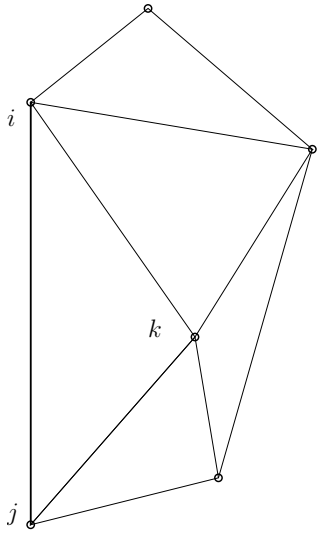

Abbildung 3.2:

Fall 1: (Abb. 3.2)

Es gibt nur einen Knoten k, der direkte Verbindungen sowohl zu i als auch zu j besitzt. In diesem Fall muß (i, j) Außenkante sein, falls eine Triangulation existiert. Anhand der Matrix der direkten Verbindungen V können wir zu jeder Kante (i, j) in $O(n)$ überprüfen, ob es nur einen Knoten k gibt, der sowohl mit i als auch j direkt verbunden ist, indem wir alle n Knoten einmal durchgehen.

Fall 2: (Abb. 3.3)

Gibt es zwei oder mehr Knoten $x_1, x_2....., x_n$, die direkt sowohl mit i als auch j verbunden sind, so müßten die implizierten Dreiecke alle ineinander liegen und die entstandene Figur eine Triangulation bilden, sollte (i, j) Außenkante einer Triangulation sein. Abb. 3.3 a) zeigt ein Beispiel für diesen Fall.

Die Situation in Fall 2 ist jedoch allein anhand von V kompliziert zu erkennen, denn wir haben keine eindeutige Information über die Lage der Dreiecke relativ zueinander. Deshalb wird in 3.2.2 ein anderer Ansatz zur Berechnung der Außenkanten vorgestellt, welcher dieses Problem umgeht.

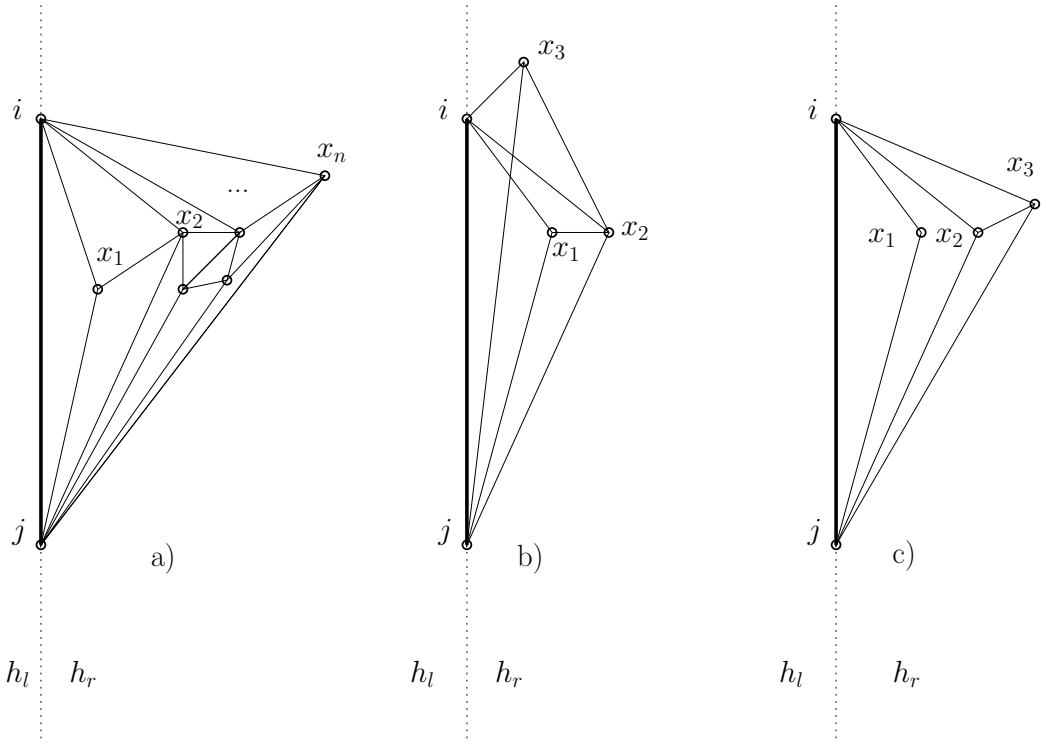

Abbildung 3.3: Nur in a) ist (i, j) Außenkante einer gültigen Triangulation.

3.2.2 Die Idee des Verfahrens zur Bestimmung der Außenkanten

Wir verwenden ein aus fünf Teilschritten bestehendes Verfahren, um die Außenkanten der Triangulation zu bestimmen. In diesem Abschnitt wird zunächst die Vorgehensweise der in 3.2.3 vorgestellten Funktion *aussenkanten*() verdeutlicht. Wir starten mit einem Knoten i. Unser Ziel ist es, zu bestimmen, ob i auf der äußeren Hülle der Triangulation liegt, falls diese existiert. Ist dies der Fall, bestimmt das Verfahren zusätzlich die beiden zu i benachbarten Knoten auf der äußeren Hülle und ruft sich dann mit einem dieser beiden erneut auf, bis ein geschlossener Außenkantenzyklus entdeckt wurde.

1. Nachbarn finden. Zuerst bestimmen wir in V alle mit i direkt verbundenen Punkte, also alle Nachbarn von i. Dies geht sogar für alle Aufrufe der Funktion *aussenkanten()* insgesamt in $O(n)$ (siehe 5.1).

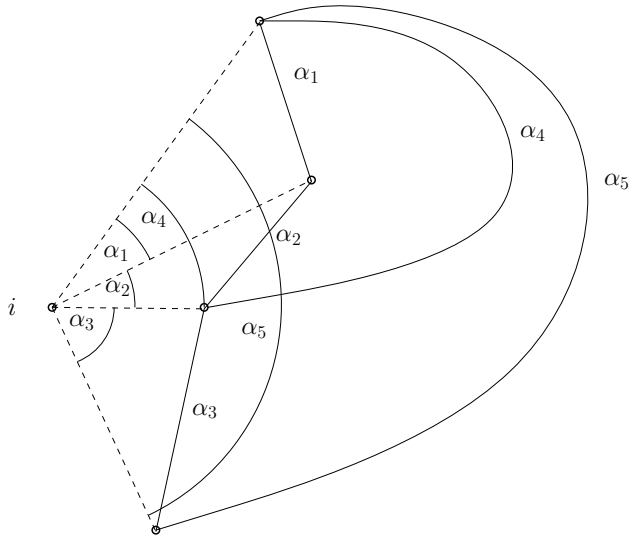

Abbildung 3.4: Der Winkelgraph: Die gestrichelten Kanten gehören nicht zum Winkelgraph. Die restlichen Kanten werden mit den Winkeln α_i gewichtet.

2. Dreiecke bestimmen. Wir können nun in $O(n)$ alle Dreiecke $\triangle(i,j,k)$ bestimmen, in denen i enthalten ist. Dazu durchlaufen wir pro Testknoten i alle $O(n)$ Kanten einmal.

3. Teilgraph und Innenwinkel bestimmen. Wir können in jedem der in 2. berechneten Dreiecke die Innenwinkel an i bestimmen. Daraus ergibt sich der *Winkelgraph* in Abb. 3.4:
Sei $N(i)$ die Menge der direkten Nachbarn von Knoten i im ausgedünnten Graphen G, der durch V impliziert wird (siehe 3.1). Dann ist der *Winkelgraph* $W(i)$ der Teilgraph von G auf $N(i)$, bei dem jede Kante (j,k) für die $V(j,k) \neq 0$ mit dem Innenwinkel $\alpha := \sphericalangle_i(j,k)$ des Dreiecks $\triangle(i,j,k)$ an i gewichtet ist. Die kürzeste-Wege-Distanz $d_{\sphericalangle_i}(.,.)$ im Winkelgraphen ist offensichtlich eine Metrik.
Da wir noch nicht wissen, wie die Dreiecke in einer möglichen Triangulation ineinander verschachtelt sind, müssen wir nun herausfinden, wie sie um i herum angeordnet sind.

4. Kanten eliminieren. Stammt die gegebene Distanzmatrix tatsächlich von einer Triangulierung in der Ebene, dann gehört offenbar eine Kante (j,k) aus dem Winkelgraphen $W(i)$ genau dann zu einem leeren Dreieck $\triangle(i,j,k)$ in der Triangulierung, wenn es zu (j,k) in $W(i)$ keinen Zwischenpunkt gibt, wenn also kein $l \in N(i)$ existiert mit $\sphericalangle_i(j,k) = \sphericalangle_i(j,l) + \sphericalangle_i(l,k)$. Wir müssen also in $W(i)$ nur analog zu Abschnitt 3.1 die Kanten eliminieren, für die

Zwischenpunkte existieren. In Abb. 3.5 wird z.B. wegen $\alpha_4 = \alpha_1 + \alpha_2 + \alpha_3$ die Kante (j,k) eliminiert. Benutzen wir zur Kantenelimination das naive Verfahren aus Abschnitt 3.1, dann benötigen wir pro Knoten i eine Laufzeit von $O(k_i^3)$ mit $k_i := |N(i)|$. Dies würde zu einer Gesamtlaufzeit von $O(n^3)$ führen, genauer gehen wir darauf in Kapitel 5 ein. Es würde uns hier nicht stören, da schon unser erster Schritt eine Laufzeit in $O(n^3)$ hat, ist aber verbesserbar. Eine schnellere Alternative ist die Benutzung des Dijkstra-Algorithmus (siehe Buch von A. Brandstädt, [3], Kapitel 5.3). Wir rufen ihn für jede Kante (j,k) aus $W(i)$ so auf, daß er die Länge l des kürzesten Weges von j nach k in dem Winkelgraphen bestimmt, in dem die Kante (j,k) entfernt wurde. Dann hat (j,k) genau dann einen Zwischenpunkt in $W(i)$, wenn $l = \sphericalangle_i(j,k)$ gilt. Der Dijkstra Algorithmus wird also k_i-mal aufgerufen, was zu einer Gesamtlaufzeit von $O(n^2 \log n)$ führt, wie wir in Kapitel 5 zeigen.

5. Außenknotentest. Stammt D von einer Triangulierung in der Ebene, so sind in dem Ergebnisgraphen der Kantenelimination nur noch Kanten enthalten, die mit i zusammen ein leeres Dreieck bilden. Sie bilden entweder eine einfache geschlossene Kette (siehe Abb. 3.6), dann ist i innerer Knoten, oder eine einfache, nicht geschlossene Kette (Abb. 3.7), dann ist i Randknoten und die beiden Endknoten ak_1 und ak_2 aus Abb. 3.7 der Kette bilden mit i je eine Außenkante. Wir können dies offensichtlich in $O(k_i)$ testen, indem wir mit einem Knoten x starten, uns von x ausgehend von Knoten zu Knoten weiterhangeln und dann überprüfen, ob wir nachdem wir alle Knoten durchlaufen haben, wieder bei x ankommen oder ob der Zyklus an einer Stelle unterbrochen ist. Der Algorithmus speichert dann eine der beiden gefundenen Außenkanten (i, ak_1) und ruft sich erneut mit ak_2 auf. Da wir bereits wissen, daß (i, ak_2) Außenkante der Triangulierung ist, liefert der erneute Funktionsaufruf genau einen neuen Randknoten z, so daß (ak_2, z) die zu (i, ak_2) benachbarte Außenkante auf der äußeren Hülle ist. Wir wiederholen diesen Schritt solange mit dem jeweils zuletzt neu berechneten Randknoten, bis wir wieder die Startkante erreichen. In diesem Fall haben wir einen geschlossenen Außenkantenzyklus ermittelt. Andernfalls brechen wir spätestens nach $3n-e-3$ Schritten ab, denn mehr Knoten können laut 8. in 2.5 nicht auf der äußeren Hülle der Triangulation liegen. Haben wir zu diesem Zeitpunkt keinen Zyklus entdeckt, so existiert keine Triangulation.

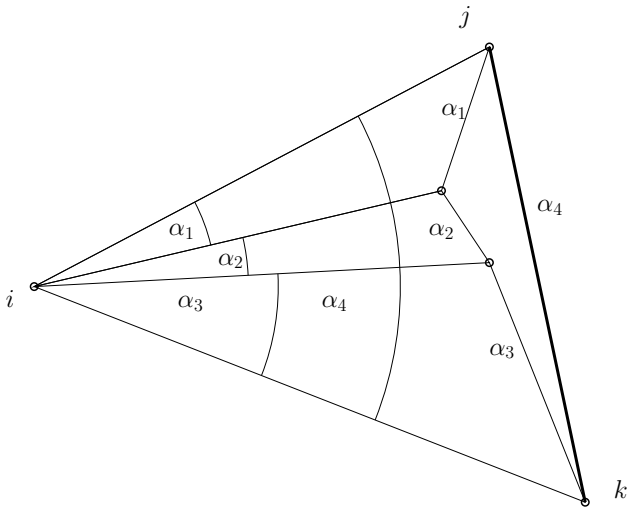

Abbildung 3.5: (j, k) wird eliminiert.

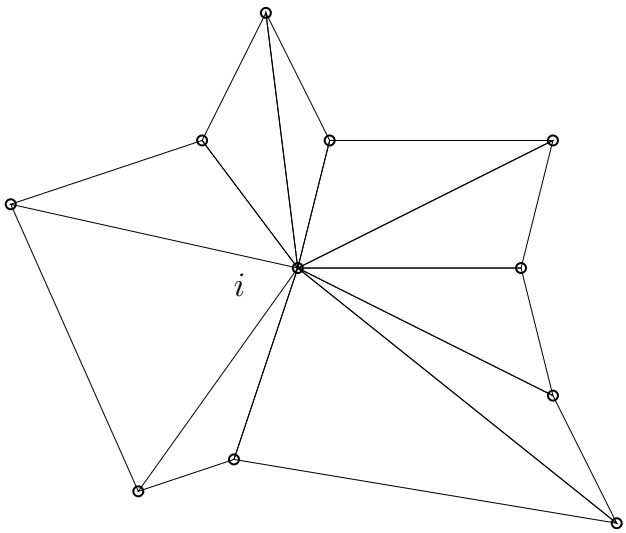

Abbildung 3.6: i ist innerer Knoten.

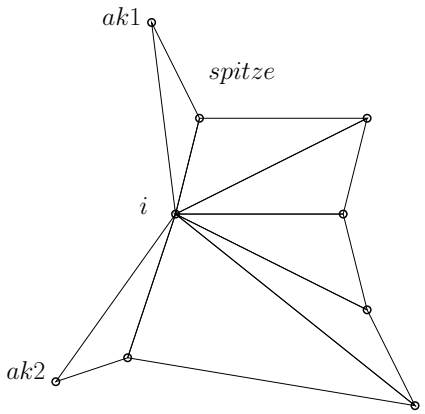

Abbildung 3.7: i liegt auf der äußeren Hülle.

3.2.3 Die Funktion *aussenkanten()*

Die Funktion *aussenkanten()* überprüft, ob der Inputknoten i auf der äußeren Hülle der Triangulation liegt. Ist dies der Fall, so speichert sie die äußere Kante $(ak1, i)$ samt Spitze und rechtem Nachbarknoten $ak2$ an i in der Matrix AH ab (die Bezeichnungnen seien wie in Abb. 3.7 gewählt). Sie ruft sich dann mit dem nächsten gefundenen Außenknoten $ak2$ auf. Sobald ein geschlossener Zyklus aus Außenkanten entdeckt wurde, sind wir fertig. Der Außenkantenzyklus steht dann als doppelt verkettete Liste in AH. Nur in diesem Fall kann eine Triangulation mit Kürzester-Wege-Distanz d existieren, die Funktion liefert dann den Wert *eins* zurück. Liegt i nicht auf der äußeren Hülle bzw. wird kein Zyklus gefunden, der i enthält, so gibt die Funktion den Wert *null* zurück.
Zu diesem Zweck führen wir einen eigenen Datentyp für Kanten der Triangulation ein, den Datentyp *kante*. Er bietet die Möglichkeit, die beiden Knoten einer Kante k (*k.setzeKnotenl(), k.setzeKnotenr()*), die beiden benachbarten Knoten (*k.setzeNachbarl(), k.setzeNachbarr()*)[2], falls k zur äußeren Hülle gehört, sowie die Spitzen der beiden leeren Dreiecke, in denen k in der Triangulation vorkommt (*k.setzeSpitze1(), k.setzeSpitze2()*) zu speichern.
Nach der Initialisierung werden alle direkt mit i verbundenen Knoten in der Queue q gespeichert. Dann wird ermittelt, welche Knoten (j, k) in q zusammen mit i ein Dreieck in der Triangulation bilden. Die Winkel $\sphericalangle_i(j, k)$ werden berechnet und in der Winkelmatrix WM an der Stelle (j, k) gespeichert. Dann wird ein Kanteneliminierungs-Algorithmus (z.B. der von Dijkstra) zur Berechnung der Hüllkette um i wie in 3.2.2 beschrieben aufgerufen. Er eliminiert jede Kante (x, y) aus $W(i)$ (setzt $WM(x, y) := 0$), für die es Zwischen-

[2]wobei l und r den linken bzw. rechten Knoten bezeichnen, siehe Anhang A.2

punkte gibt. Als Ergebnis stehen dann die kleinsten Winkel $\sphericalangle_i(j,k)$ an der Stelle $WM(j,k)$, falls das Dreieck $\triangle(i,j,k)$ leer ist. Ansonsten führt der kürzeste Weg von j nach k über einen weiteren Knoten, die Kante (j,k) im Winkelgraph wird dann gelöscht, indem $WM(j,k)$ *null* gesetzt wird.
Nun muß überprüft werden, ob die entstandenen Dreiecke i vollständig umschließen, also ob die Hüllkette geschlossen ist. Dies ist genau dann nicht der Fall, wenn es in WM zwei Spalten x und y gibt, die nur jeweils einen Eintrag enthalten, d.h. x und y kommen nur in jeweils einem leeren Dreieck mit i zusammen vor, sind demnach die Außenknoten $ak1$ und $ak2$. In diesem Fall speichern wir die Kante $(ak1,i)$ in AH und rufen die Funktion $aussenkanten(ak2)$ auf, solange bis wir wieder den Startknoten erreichen oder alle Knoten verbaut haben.
Sobald wir wieder die Startkante erreichen, haben wir erfolgreich einen Außenkantenzyklus berechnet. Wir können an dieser Stelle eine Triangulation bereits ausschließen, falls 8. in Kapitel 2.5 nicht erfüllt ist. Aus Definition 2.2 folgt, daß es in einer Triangulation genau einen Außenkantenzyklus geben muß. Gäbe es mehrere, wäre die Triangulation nicht zusammenhängend oder wir hätten ein „Loch“ entdeckt. Wir werden deshalb, nachdem wir unseren gefundenen Zyklus trianguliert haben, noch überprüfen, ob alle Kanten verbaut wurden.

funktion aussenkanten(**knoten** i)

(**Initialisierung**)
initialisiere Winkelmatrix WM und Kantenmatrix AH mit *null*;
boolean *ausgabe* mit *null*;
queue q mit *null*;
kante *startkante* mit *null*;

(**Nachbarn finden**)
for each ($c \in N(i)$) q.enqueue(c);

(**Dreiecke bestimmen und Teilgraph bzw. Innenwinkel berechnen**)
for each (Kante (j,k) mit $j,k \neq i$)
if $(j,k \in N(i))$ $WM(j,k) := \sphericalangle_i(j,k)$;
end for

(**Kanten eliminieren**)
$Huellkette := Kantenelimination(WM)$;

(**Aussenknotentest**)
if (Huellkette offene Kette)
ermittele äußere Nachbarknoten $ak1$ und $ak2$ von i;
speichere Kante $(ak1, i)$ sowie zugehörige Spitze und Nachbar $ak2$ in $AH(ak1, i)$;
if$(ak2 = Startknoten)$ *ausgabe* := *eins*[3];
speichere Kante $(i, ak2)$ sowie zugehörige Spitze und Nachbarn in $AH(i, ak2)$;
else if ($ak2$ ist anderer vorher berechneter Außenknoten) **STOP**[4];
else $aussenkanten(ak2)$;
if (ermittelte Anzahl von Außenkanten 8. in 2.5 nicht erfüllt) **STOP**;

(**Ausgabe**)
ausgabe;

3.3 Realisierbarkeit in der Ebene

Aufgrund der Dimensionierung der Kanten könnte der Fall eintreten, daß wir einen gefundenen Kantenzyklus gar nicht in der Ebene darstellen kön-

[3] wir haben einen zulässigen Außenkantenzyklus gefunden
[4] wir haben einen unzulässigen Außenkantenzyklus gefunden

nen. Beispielsweise könnte eine Kante länger sein als alle anderen zusammen. Notwendiges Kriterium für die Existenz einer Triangulation ist, daß die äußeren Kanten einen Zyklus bilden, der im $\mathbb{R}^2$ darstellbar ist. Der in diesem Abschnitt bewiesene Satz 3.1 hilft uns, bereits eine ebene Triangulation auszuschließen, wenn wir nur den Zyklus der Außenkanten kennen. Dazu müßten wir zu jeder Außenkante l_i die Bedingung $l_i \leq \sum_{j=1,j\neq i}^{n} l_j$ überprüfen. Da wir die Summe in $O(n)$ einmal berechnen und dann für alle Kanten l_i verwenden können (wenn wir die Länge l_i jeweils abziehen), ergibt sich als Laufzeit für diesen Test $O(n)$. In unserem Algorithmus ist er nicht implementiert, da sich im Laufe unserer Triangulationsberechnung zeigt, ob der Außenkantenzyklus geometrisch realisierbar ist. Sollte dies nicht der Fall sein, erreichen wir bei der Berechnung der Triangulation die Startkante nicht mehr oder wir stoßen bei der Überprüfung der Kantenlängen am Ende des Algorithmus auf einen Widerspruch.

Satz 3.1. *Seien $l_1, l_2, ...l_n$, $l_i \in R^+$ die vorgegebenen Kantenlängen, so daß für alle l_i die Ungleichung $l_i \leq \sum_{j=1,j\neq i}^{n} l_j$ gilt. Außerdem nehmen wir an, daß $l_1 = \max\{l_1, ..., l_n\}$ gilt. Dann gibt es Indizes $j, k \in 1, ..., n$, so daß die Definitionen $a := l_1 + ... + l_{j-1}$, $b := l_j + ... + l_{k-1}$ und $c := l_k + ... + l_n$ dazu führen, daß a, b und c die Dreiecksungleichungen $a \leq b + c$, $b \leq a + c$, $c \leq a + b$ erfüllen.*

Beweis. **Fall 1:** (Abb. 3.8) Wir betrachten zunächst den Fall, in welchem die ersten beiden Kanten zusammenaddiert bereits mindestens so groß wie die Summe der restlichen Kanten ist, also falls $l_1 + l_2 \geq \sum_{j=3}^{n} l_j$ gilt.
Dann wählen wir $j = 2$, $k = 3$ und damit $a = l_1$, $b = l_2$, $c = \sum_{x=3}^{n} l_x$ und zeigen, daß die Behauptung gilt:

$a \leq b + c$ gilt laut Vorraussetzung des Satzes, denn a besteht nur aus einer Kante.

$b \leq a \leq a + c$ gilt wegen $a := l_1 = \max(l_1, ..., l_n)$.

$c \leq a + b$ gilt laut Vorraussetzung von Fall 1.

Fall 2: (Abb. 3.9)

Wir betrachten nun die Situation, in welcher $l_1 + l_2 < \sum_{x=3}^{n} l_x$ gilt.
Dann wählen wir $j = 3$ und für k denjenigen Index, ab dem erstmalig $\sum_{i=1}^{k-1} l_i \geq \sum_{i=k}^{n} l_i$ gilt. Dies impliziert direkt $a + b \geq c$, bzw. $a + b - c \geq 0$, aber auch folgende obere Schranke für $a + b - c$:

Behauptung: Für das oben definierte k gilt $a + b - c \leq 2l_1$.

Wäre die Differenz größer als $2l_1$, so bestünde das Differenzstück aus mindestens zwei Kanten der Länge l_1, bzw. aus mindestens drei kürzeren Kanten. In diesem Fall hätte man schon eine Kante früher „abknicken“ können, da die Bedingung für k bereits erfüllt gewesen wäre. Formal läßt sich dies folgendermaßen beweisen:

Sei $a + b - c > 2l_1$, d.h. $\sum_{i=1}^{k-1} l_i - \sum_{i=k}^{n} l_i > 2l_1$.

Dann gilt $\sum_{i=1}^{k-2} l_i - \sum_{i=k-1}^{n} l_i = \sum_{i=1}^{k-1} l_i - \sum_{i=k}^{n} l_i - 2l_{k-1} > 0$,

denn $\sum_{i=1}^{k-1} l_i - \sum_{i=k}^{n} l_i > 2l_1$ und $2l_{k-1} \leq 2l_1$.

Dies ist ein Widerspruch zur Wahl von k. Damit ist die Behauptung bewiesen.

Wir zeigen nun, daß auch in Fall 2 die drei Dreiecksungleichungen gelten:

$a \leq b + c$ gilt laut Vorraussetzung von Fall 2.

$c \leq a + b$ gilt laut Konstruktion, denn wir haben k so gewählt, daß diese Eigenschaft erfüllt ist.

Es bleibt also $b \leq a+c$ zu zeigen. Aufgrund der Behauptung gilt $a+b-c \leq 2l_1$. Nach einer Umformung ergibt sich daher $b \leq c+(2l_1 - a)$, wobei $2l_1 - a \leq a$, wegen $a = l_1 + l_2 > l_1$. Eingesetzt ergibt sich $b \leq a + c$. □

Korollar: Für gegebene Werte $l_1, ..., l_n \in \mathbb{R}^+$ gibt es genau dann eine einfache geschlossene polygonale Kette in der Ebene, deren Kantenlängen in der gleichen Reihenfolge den Werten l_i entsprechen, wenn für alle $i \in 1, ..., n$ die Dreiecksungleichung $l_i \leq \sum_{j=1, j\neq i}^{n} l_j$ gilt.

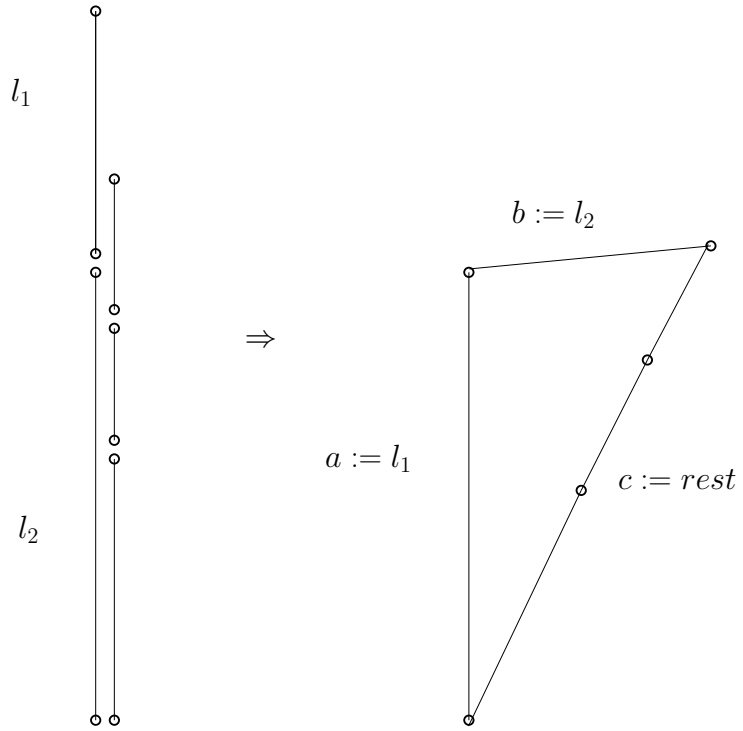

Abbildung 3.8: Fall 1

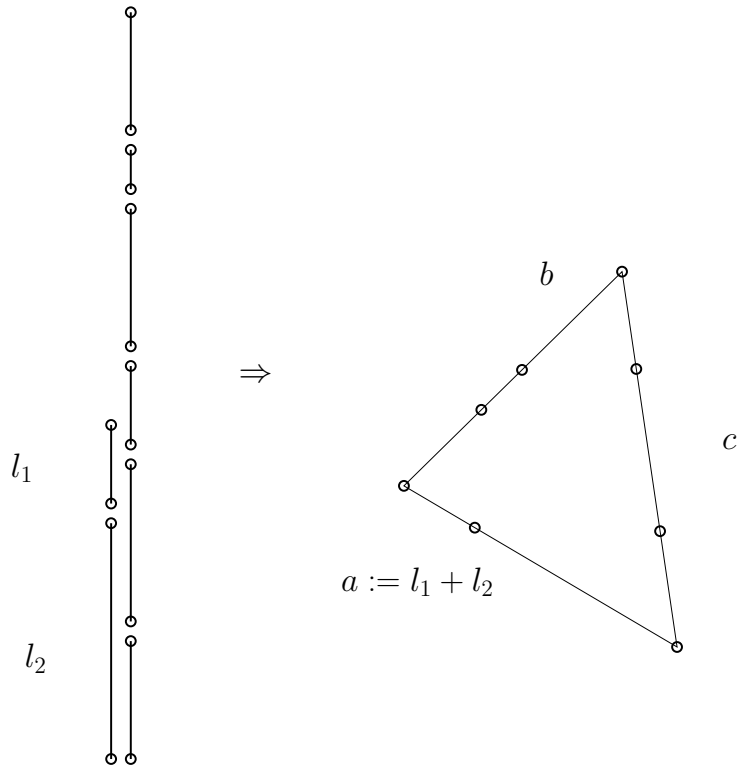

Abbildung 3.9: Fall 2

Beweis. Für Dreiecke ist dies leicht einzusehen. Gibt es nun für $n \geq 4$ zu gegebenen Werten l_i eine einfache polygonale Kette im $\mathbb{R}^2$, dann bilden alle Kanten außer der i-ten Kante einen Pfad, der beide Endpunkte von Kante i verbindet. Also muß der Pfad mindestens die Länge l_i haben,
es gilt $l_i \leq \sum_{j=1,j\neq i}^{n} l_j$.
Seien andersherum alle Dreiecksungleichungen erfüllt, dann können wir nach Satz 3.1 die l_i so zu drei zusammenhängenden Gruppen zusammenfassen, daß ihre Gesamtlängen a, b und c die drei Dreiecksungleichungen erfüllen.Wir können also eine zugehörige polygonale Kette als Dreieck realisieren. □

3.4 Welches Dreieck liegt innen?

Im folgenden Triangulationsalgorithmus tritt das Problem auf, zu entscheiden, welches Dreieck das innerste ist, wenn mehrere Dreiecke existieren, die die Kante (i, j) enthalten und beide auf der gleichen Seite von (i, j) liegen müssen. Wie können wir nun testen, ob zwei Dreiecke A und B, welche diesen Anforderungen entsprechen, ineinander liegen? Dies läßt sich leicht anhand der Innenwinkel überprüfen. A liegt genau dann in B, falls gilt: $\alpha_A < \alpha_B$ und $\beta_A < \beta_B$ (Abb. 3.10).

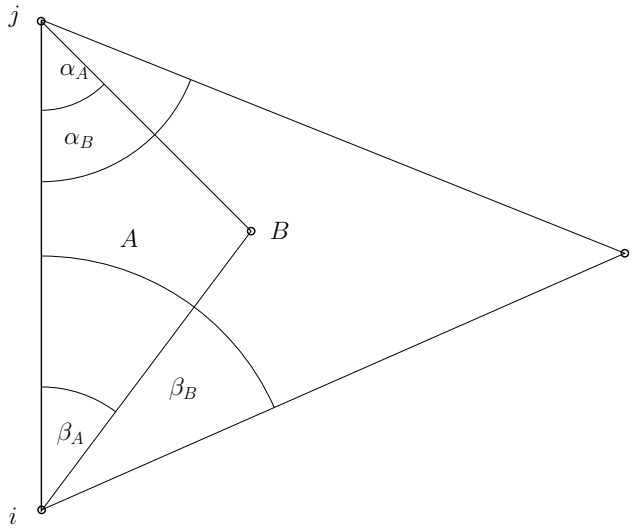

Abbildung 3.10: A liegt in B.

3.5 Schnitttest von Kanten

Nachdem der Triangulationsalgorithmus (Kap. 4) erfolgreich eine ebene Triangulation berechnet hat, müssen wir alle vorkommenden Kanten auf mögliche Schnittpunkte testen.
Wie können wir nun rechnerisch ermitteln, ob sich zwei Kanten schneiden? Stellen wir uns die Testkandidaten im Koordinatensystem vor (Abb. 3.11). Sollte mindestens eine der beiden Kanten vertikal verlaufen, so können wir beide Kanten um 30° drehen, indem wir die Knoten i, j, k und l jeweils mit der Rotationsmatrix $R_{30^\circ} := \begin{pmatrix} \cos 30^\circ & \sin 30^\circ \\ -\sin 30^\circ & \cos 30^\circ \end{pmatrix}$ multiplizieren (Fall 1 in der Funktion *schnitttest()*). Sollte dann immer noch eine Kante vertikal verlaufen, dann drehen wir beide Kanten erneut um 30°, und spätestens jetzt verläuft keine Kante mehr senkrecht. Wir können nun mithilfe der beiden Knoten zu jeder **Kante** eine **Geradengleichung** aufstellen, welche die Verlängerung der Kante beschreibt. Zwischen zwei windschiefen Geraden (Abb. 3.12) läßt sich leicht der Schnittpunkt berechnen, indem man diese gleichsetzt (Fall 3). Beispielsweise liefert $r(x) = s(x)$ den x-Wert des Schnittpunktes s (in Abb. 3.11). Da der x-Wert von s sowohl im x-Werteintervall von $[i, j]$ als auch im Intervall $[k, l]$ liegt, existiert tatsächlich ein Schnittpunkt der Kanten. Der Schnittpunkt zwischen $r(x)$ und $t(x)$ liegt beispielsweise zwar in $[m, n]$, jedoch nicht in $[i, j]$. Daher schneiden sich diese beiden Kanten nicht. Wir sprechen genau dann von einem Schnittpunkt, wenn dieser auf beiden Kanten und **echt** im Inneren mindestens einer Kante liegt, denn der Fall, daß sich zwei Kanten in einem Knoten berühren ist ja zulässig. Im Falle der Identität der Geraden (Abb. 3.13) ist zu überprüfen, ob sich die beiden Intervalle, die die Kanten auf der x-Achse durchlaufen, überschneiden (Fall 4). Ist dies der Fall, so liegt eine unzulässige Überlappung von zwei Kanten vor. Parallelität liegt nur vor, wenn die Steigungen übereinstimmen und die implizierten Geraden nicht identisch sind. In diesem Fall gibt es keinen Schnittpunkt.

funktion schnitttest(**kante** (i,j), (k,l))

(**Initialisierung**)
initialisiere boolean *schnitt* mit *null*;

(**1.Mindestens eine Kante ist senkrecht**)
if ((i,j) oder (k,l) senkrecht)
$i' = R_{30^\circ} i$, $j' = R_{30^\circ} j$, $k' = R_{30^\circ} k$, $l' = R_{30^\circ} l$;
schnitttest$((i',j'),(k',l'))$;

(**2.Keine Kante ist senkrecht**)
else berechne Geradengleichungen $g_{(i,j)}$ und $g_{(j,k)}$;

(**3.Beide Geraden sind windschief**)
if ($g_{(i,j)}$ und $g_{(j,k)}$ sind windschief)
berechne Schnittpunkt s_x;
if (s_x liegt echt im inneren mindestens einer Kante)
schnitt := 1;

(**4.Die Geraden sind identisch**)
else if ($g_{(i,j)}$ und $g_{(j,k)}$ identisch und Kanten überlappen[5])
schnitt := 1;

(**Ausgabe**)
schnitt;

[5] es reicht nicht, wenn die beiden Kanten sich berühren

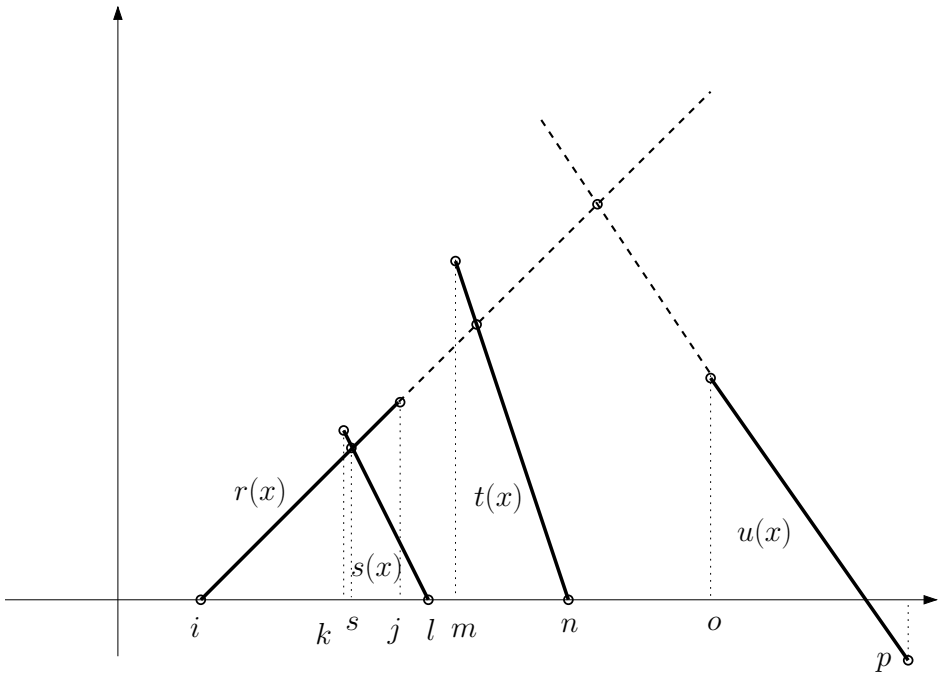

Abbildung 3.11: Die vier Kanten implizieren die Geraden $r(x)$, $s(x)$, $t(x)$, $u(x)$.

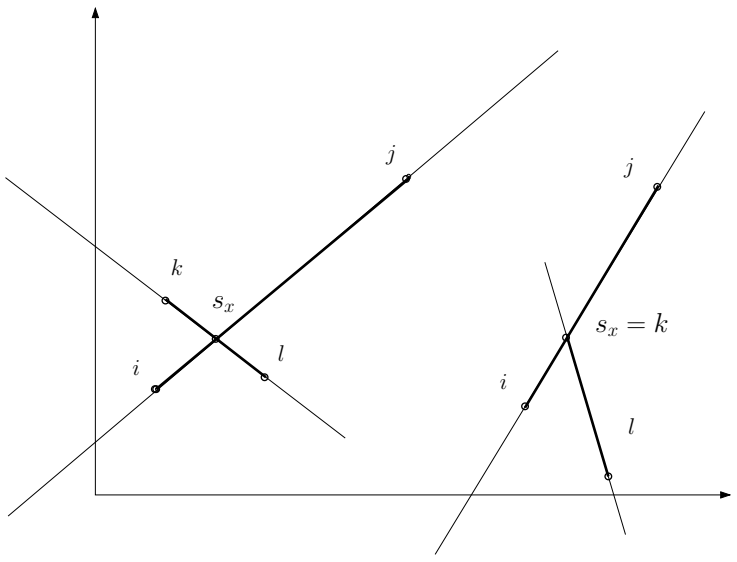

Abbildung 3.12: Der Schnittpunkt s_x liegt im inneren mindestens einer Kante.

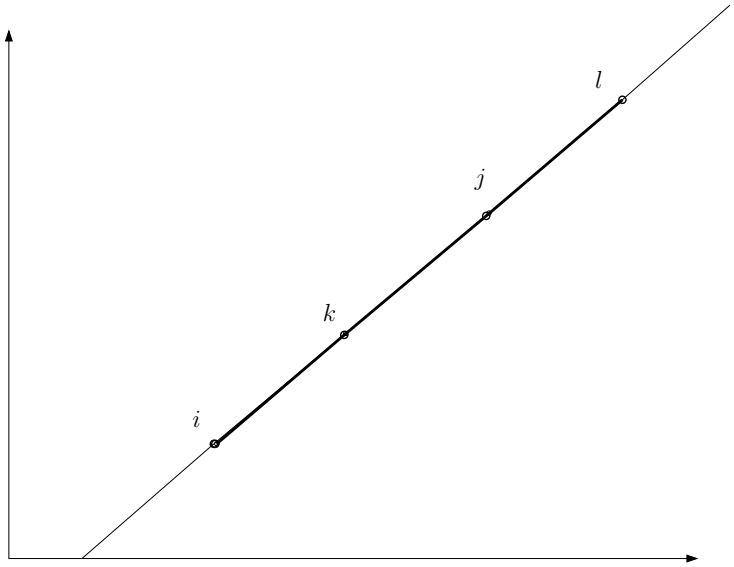

Abbildung 3.13: Die Geraden sind identisch und die Kantenintervalle überlappen sich.

3.6 Dreiecksberechnung

Da der in Kapitel 4 folgende Algorithmus inkrementell eine Triangulation berechnet, benötigen wir ein Verfahren, das uns zu einem gegebenen Dreieck die Nachbardreiecke liefert. Dazu setzen wir voraus, daß das gegebene Dreieck in der Triangulation nicht weiter unterteilt wird[6], also leer ist. Wir wollen nun das Problem betrachten, zu einer inneren[7] Kante einer Triangulation das andere angrenzende leere Dreieck zu berechnen. Die Außenkanten kommen nur in einem leeren Dreieck in der Triangulation vor. Stoßen wir jedoch auf eine innere Kante, so muß diese in zwei leeren Dreiecken enthalten sein, falls eine Triangulation existiert.
Zu einer Kante (i, j) lassen sich mithilfe der Matrix der direkten Verbindungen V alle Knoten x bestimmen, die mit (i, j) ein Dreieck bilden. Man durchlaufe Spalte i sowie Spalte j jeweils einmal und bestimme diejenigen Knoten, welche sowohl mit i als auch mit j verbunden sind (und im bisher berechneten Graphen noch nicht bereits mit i und j verbunden wurden). Uns interessiert an dieser Stelle nur das innerste Dreieck. Da nun (i, j) bereits in einem leeren Dreieck des bisherigen Graphen vorkommt, können alle anderen Dreiecke nur in der gegenüberliegenden Halbebene H_{frei} in Abb. 3.14 liegen.

[6] später werden wir sehen, daß dies keine Einschränkung für unseren Triangulationsalgorithmus darstellt

[7] „innen“ im Sinne von „keine äußere Kante in der endgültigen Triangulation“

Dies hängt mit der Arbeitsweise des Algorithmus zusammen. Daher verschieben wir den Beweis dieser Aussage bis zu Kapitel 4.6. Außerdem müssen die übrigen, noch nicht verbauten Dreiecke ineinander liegen. Im Beispiel 3.14 liegt $\triangle(i,j,k)$, im Gegensatz zu den Dreiecken $\triangle(m,n,o)$ und $\triangle(m,n,p)$, vollständig in $\triangle(i,j,l)$. Aufgrund der eindeutigen Lage von $\triangle(m,n,o)$ und $\triangle(m,n,p)$ können wir an dieser Stelle eine Triangulation ausschließen.
Die Funktion *angrenzendesDreieck()* berechnet den Index s der Spitze des noch nicht verbauten leeren Dreiecks. Sie erhält als *input* die zu testende Kante (i,j) und verwendet die Matrix V'. V' enthält diejenigen direkten Verbindungen, die noch nicht im Graphen „verbaut“ wurden. Am Anfang gilt $V = V'$, im Triangulationsalgorithmus (Kapitel 4) werden dann nach und nach Verbindungen gelöscht bzw. mit *null* überschrieben, sobald sie verbaut wurden.
In der äußeren *for each*-Schleife werden alle Knoten $x \neq i,j$ durchlaufen. Zu jedem Knoten wird der Winkel α berechnet und dieser mit dem bisher minimalen Winkel α_{min} verglichen. Ist α kleiner, so werden α_{min} und β_{min} aktualisiert, falls β ebenfalls kleiner als β_{min} ist, und die neue Spitze in *ausgabe* gespeichert. Sobald die Schleife verlassen wird, steht in *ausgabe* der Index der Spitze des Dreiecks an (i,j) mit minimalem Winkel α. Falls die Berechnung fehlgeschlagen ist, hat *ok* den Wert *null*. Dies ist der Fall, wenn zwei Winkel gleich groß sind (also zwei Kanten aufeinanderliegen müßten) oder Dreiecke nicht ineinander liegen.

funktion angrenzendesDreieck(**kante** (i, j))

(**Initialisierung**)
initialisiere *ok* mit *eins*, *ausgabe*, α_{min} und β_{min} mit *null*;

(**Bestimme α_{min} und β_{min}**)
for each (Knoten $x \neq i, j$ und falls $ok = eins$)
if ($V(i, x)$ und $V(j, x) \neq 0$ und ($V'(i, x)$ oder $V'(x, j) \neq 0$))
berechne Winkel α und β;
if (($\alpha < \alpha_{min}$ und $\beta < \beta_{min}$) oder $\alpha_{min} = 0$)
setze $\alpha_{min} := \alpha$ und $\beta_{min} := \beta$, $x := ausgabe$;
if (($\alpha < \alpha_{min}$ und $\beta \geq \beta_{min}$) oder ($\alpha \geq \alpha_{min}$ und $\beta < \beta_{min}$))
setze $ok = null$;

(**Ausgabe**)
if ($ok = 0$) $ausgabe := null$;
ausgabe;

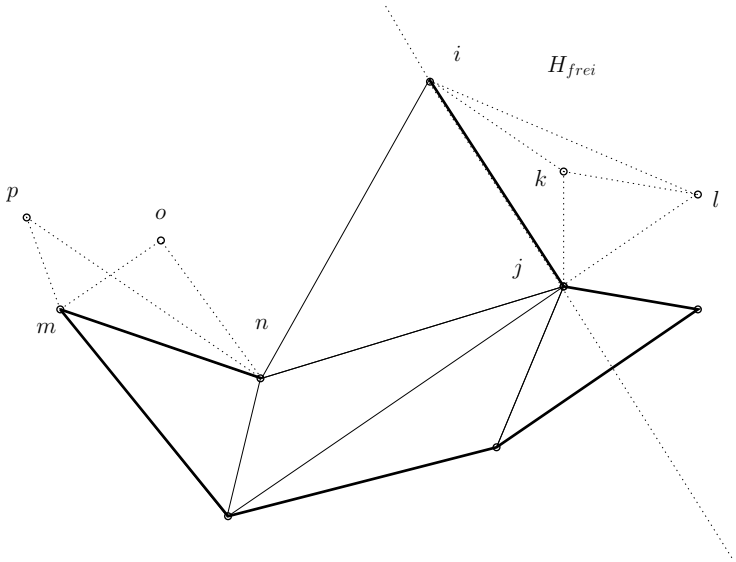

Abbildung 3.14: In welchen Dreiecken kommen (i, j) bzw. (m, n) vor, falls eine Triangulation existiert?

3.7 Koordinaten der Spitze

In Kapitel 3.6 wurde zu zwei gegebenen Punkten der dritte Punkt, die Spitze, des zugehörigen unverbauten Dreiecks bestimmt. Wir möchten nun in diesem Abschnitt zeigen, wie aus den Koordinaten der beiden gegebenen Punkte und der gegenüberliegenden Dreiecksspitze sowie den Längen des entstandenen Dreiecks die Koordinaten der zweiten Spitze berechnet werden. Die Bezeichnungen seien wie in Abb. 3.15 gewählt.

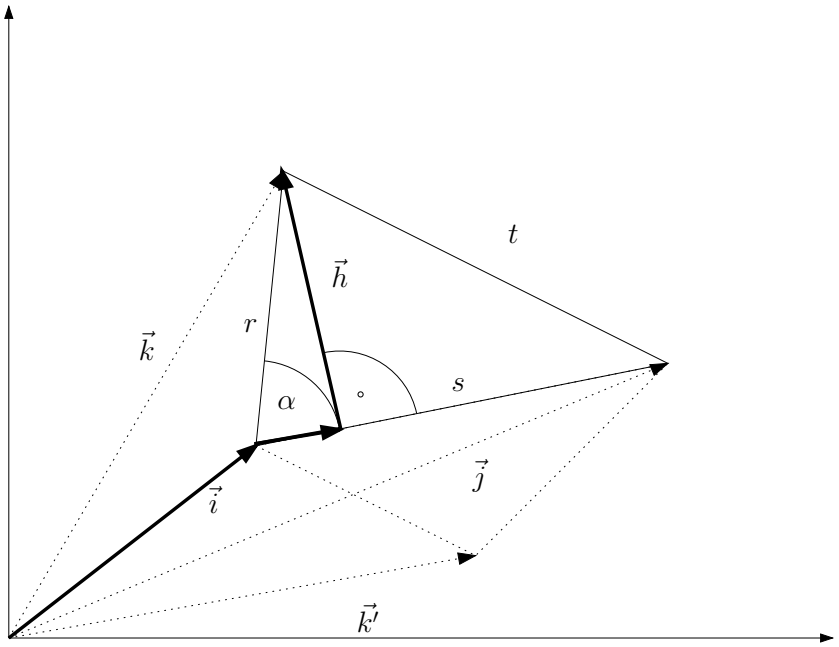

Abbildung 3.15: Berechnung von $\vec{k}$.

Gegeben: $\vec{i}$, $\vec{j}$, $\vec{k'}$ sowie die Längen r, s, t und α.

Gesucht: Ortsvektor $\vec{k}$.

Es gilt: $\vec{k} = \vec{i} + x_1(\frac{\vec{j}-\vec{i}}{\sqrt{<\vec{j}-\vec{i},\vec{j}-\vec{i}>}}) + x_2\frac{\vec{h}}{\sqrt{<\vec{h},\vec{h}>}}$.

$\cos\alpha = \frac{x_1}{r} \Rightarrow x_1 = r\cos\alpha$.

$x_2^2 + x_1^2 = r^2 \Rightarrow x_2 = \sqrt{r^2 - x_1^2}$.

$\vec{h}$ muß senkrecht zu $\vec{d} := \vec{j} - \vec{i}$ sein. Dies ist der Fall, wenn gilt: $< \vec{h}, \vec{d} >= 0$.

Da die Kante s bereits in einem Dreieck verbaut wurde, müssen wir den Vektor $\vec{h}$ zudem so wählen, daß er in die freie Halbebene zeigt, also in die, in der die Spitze $\vec{k}$ liegen soll. Die Spitze $\vec{k'}$ des bereits verbauten Dreiecks liegt dann in der anderen Halbebene. Wir benötigen die **Hessesche Normalenform** der durch s implizierten Geraden. Diese liefert den Abstand des Vektors $\vec{k'}$ zur Geraden. Da dieser nur negativ ist, wenn der Normalenvektor der Geraden in die Halbebene zeigt, in der $\vec{k}$ liegt, muß gelten:

$HN(\vec{k'}) = (\vec{k'} - \vec{i}) \cdot \frac{\vec{h}}{\sqrt{<\vec{h},\vec{h}>}} < 0.$

Wir können $\vec{h}$ folgendermaßen wählen: $\begin{pmatrix} x \\ y \end{pmatrix} := \vec{j} - \vec{i} \Rightarrow \vec{h} := \begin{pmatrix} -y \\ x \end{pmatrix}$ falls $HN(\vec{k'}) < 0$ (dies entspricht einer Drehung von $\begin{pmatrix} x \\ y \end{pmatrix}$ um 90° gegen den Uhrzeigersinn).
Sonst setzen wir $\vec{h} := \begin{pmatrix} y \\ -x \end{pmatrix}$ (dies entspricht einer Drehung von $\begin{pmatrix} x \\ y \end{pmatrix}$ um 90° im Uhrzeigersinn).

Beispiel: $\vec{i} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$, $\vec{j} = \begin{pmatrix} 5 \\ 2 \end{pmatrix}$, $\vec{j} - \vec{i} = \begin{pmatrix} 4 \\ 1 \end{pmatrix} =: \vec{d}$, $\vec{k'} = \begin{pmatrix} 3 \\ 0,5 \end{pmatrix}$

$r = \sqrt{2^2 + 1^2} = \sqrt{5}$, $s = \sqrt{17}$, $t = \sqrt{10}$

$\Rightarrow \alpha = \arccos\frac{r^2+s^2-t^2}{2rs} \approx 49,39870535°$

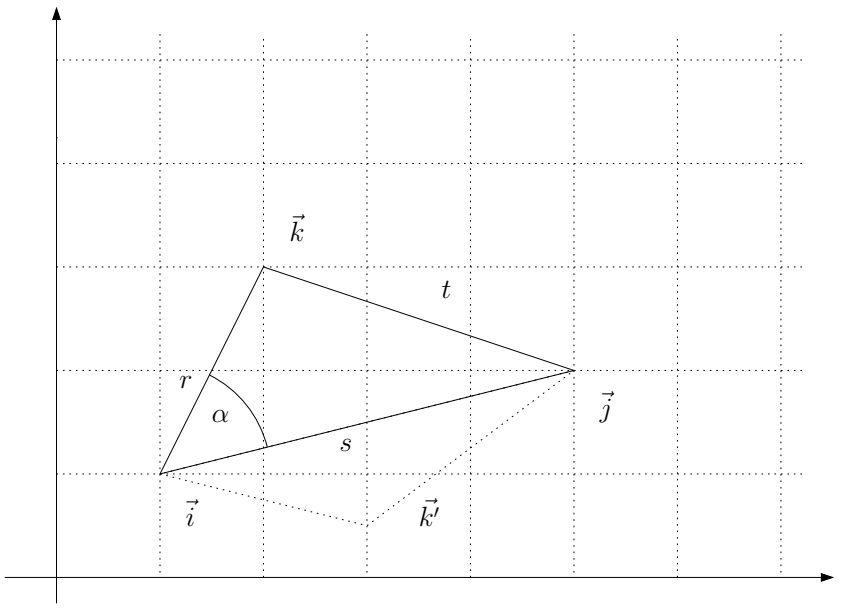

Abbildung 3.16: Berechnung von $\vec{k}$.

$\Rightarrow x_1 = r\cos\alpha \approx 0.650790735\sqrt{5} \approx 1,45521375,$

$x_2 = \sqrt{r^2 - x_1^2} \approx \sqrt{5 - 2,117642907} \approx 1,697749375$

$\vec{h} = \begin{pmatrix} -1 \\ 4 \end{pmatrix}$, denn $< \vec{h}, \vec{d} >= 0,$

und $HN(\vec{k'}) = (\vec{k'} - \vec{i}) \cdot \frac{\vec{h}}{\sqrt{<\vec{h},\vec{h}>}} = \begin{pmatrix} 2 \\ -0,5 \end{pmatrix} \cdot \begin{pmatrix} \frac{-1}{\sqrt{17}} \\ \frac{4}{\sqrt{17}} \end{pmatrix} = \frac{-4}{\sqrt{17}} < 0.$

$\vec{k} \approx \vec{i} + \frac{1,455212324}{\sqrt{17}}\vec{d} + \frac{1,697749375}{\sqrt{17}}\vec{h} \approx \begin{pmatrix} 1,999998616 \\ 2,999999654 \end{pmatrix} \approx \begin{pmatrix} 2 \\ 3 \end{pmatrix}.$

Die Funktion *berechneKoordinaten()* arbeitet genau nach diesem Schema. Sie berechnet die Koordinaten der Spitze k aus den Koordinaten der Knoten i und j. Dabei wollen wir davon ausgehen, daß die Koordinaten eines Punktes in der globalen $n \times 2$-Matrix *koordinaten* gespeichert werden.

funktion berechneKoordinaten(**kante** (i,j), **knoten** k)

(**Initialisierung**)

initialisiere r mit $V(i,k)$, s mit $V(i,j)$, t mit $V(j,k)$, k' mit $M(i,j).spitze1()$;
if $(k'=0)$ setze Koordinaten von k' auf $(0,-1)$;
initialisiere Vektoren v_i, v_j, v_k, $v_{k'}$, v_d, v_h, und weise
diesen Koordinaten wie folgt zu:

$v_i(1) := koordinaten(i)(1)$, $v_i(2) := koordinaten(i)(2)$;
$v_j(1) := koordinaten(j)(1)$, $v_j(2) := koordinaten(j)(2)$;
$v_{k'}(1) := koordinaten(k')(1)$, $v_{k'}(2) := koordinaten(k')(2)$;
berechne $v_d := v_j - v_i$;

(**Berechnung der Koordinaten**)

berechne $alpha := \arccos \frac{r^2+s^2-t^2}{2rs}$ (Cosinussatz);

berechne Abschnitt x_1 auf v_d: $x1 := r \cdot \cos alpha$;

berechne Abschnitt x_2 auf v_h: $x2 := \sqrt{r^2 - x1^2}$;

berechne $v_h := (-d_y, d_x)$;

if $((v_{k'} - v_i) \cdot \frac{v_h}{\sqrt{<v_h,v_h>}} \geq 0)$ $v_h := (d_y, -d_x)$;

$v_k = v_i + x1\frac{v_d}{\sqrt{<v_d,v_d>}} + x2\frac{v_h}{\sqrt{<v_h,v_h>}}$;

speichere Koordinaten von v_k in *koordinaten*;

Kapitel 4

Der Triangulationsalgorithmus

4.1 Arbeitsweise des Algorithmus

Um bestimmen zu können, ob eine Metrik, gegeben durch die zugehörige Distanzmatrix D, von einer Triangulation im R^2 herrührt, benötigt der Algorithmus zunächst die Verbindungsmatrix V sowie die Matrix der Kanten der äußeren Hülle AH (vgl. Kapitel 3). Es folgt zunächst ein Überblick über die Funktionsweise des Algorithmus, ohne daß bereits auf Spezialfälle eingegangen wird. Im ersten Teil wird die *äußere Schicht* der Triangulation berechnet.

Definition 4.1. *Ein Dreieck $\triangle(x, y, z)$ gehört genau dann zur* **äußeren Schicht** *der Triangulation, wenn mindestens ein Eckpunkt zur äußeren Hülle gehört (Abb. 4.1).*

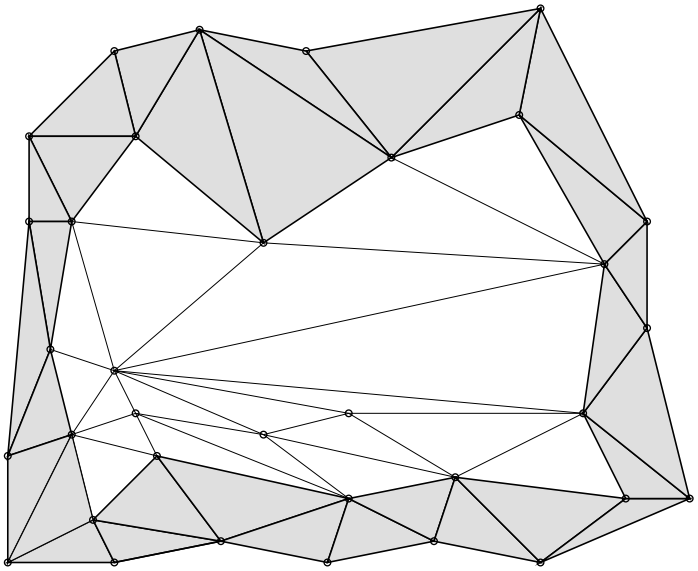

Abbildung 4.1: Die **äußere Schicht** ist grau gefärbt.

Der folgende Algorithmus berechnet die Dreiecke der äußeren Schicht und legt damit die Form der äußeren Hülle fest.

Algorithmus 4.1. *„Äußere Schicht" (Überblick)*

***Schritt 1**: Bilde das eindeutige leere Dreieck $\triangle(r,s,t)$ zu einer beliebigen Außenkante r. Lege das Koordinatensystem wie in Abb. 4.2 fest. Dies kann geschehen, indem wir z.B. dem Knoten i die Koordinaten $(0,0)$ zuweisen, dem Knoten j die Koordinaten $(d(i,j),0)$ und für den dritten Knoten k eine positive y-Koordinate erzwingen. Die Koordinaten der folgenden Knoten können dann aus den bereits bekannten Koordinaten bzw. den neu entstandenen Dreiecken berechnet werden.*

***Schritt 2**: Falls s in Abb. 4.3 nicht Außenkante ist, bestimme das noch nicht verbaute, an s angrenzende leere Dreieck. Falls u nicht Außenkante ist, setze neues Dreieck als aktuelles Dreieck $\triangle(r,s,t)$ und wiederhole Schritt 2 solange, bis wir die an r angrenzende Außenkante erreicht haben. Breche ab, falls es kein eindeutiges angrenzendes Dreieck gibt.*

***Schritt 3**: Wiederhole Algorithmus für alle äußeren Dreiecke solange, bis wieder das Startdreieck erreicht ist oder wir kein angrenzendes Dreieck mehr finden, z.B. wenn wir alle Knoten und Kanten verbaut haben (Abb. 4.4).*

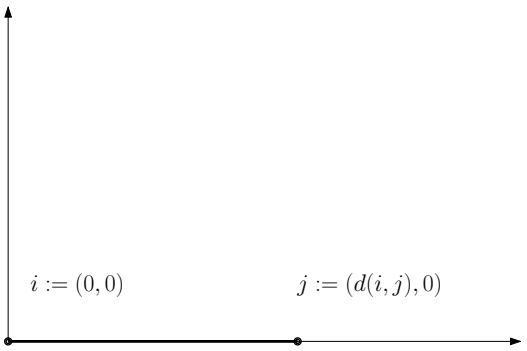

Abbildung 4.2: Wir legen ein Koordinatensystem fest.

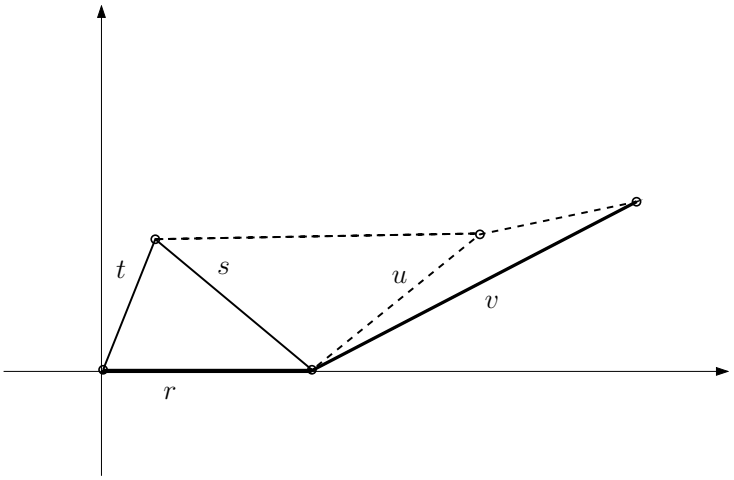

Abbildung 4.3: Die äußere Hülle nimmt eine feste Form an.

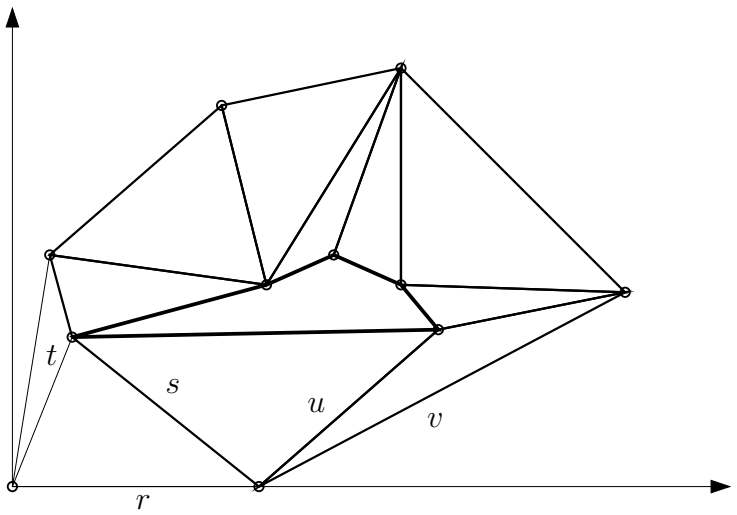

Abbildung 4.4: Die äußere Schicht wurde erfolgreich konstruiert. Für die weitere Triangulation wird der fettgedruckte Kantenzyklus als neue äußere Hülle verwendet.

Der Algorithmus für die inneren Schichten verwendet im Normalfall den von Algorithmus 4.1 gefundenen Zyklus der inneren Kanten als äußere Hülle (Abb 4.4). Außerdem wird er immer dann aufgerufen, wenn ein geschlossenes Polygon während der Triangulation entsteht, welches weiter trianguliert werden muß.

Algorithmus 4.2. *„Innere Schichten“ (Überblick)*

***Schritt 1**: Bilde leeres Dreieck $\triangle(r, s, t)$ zu einer beliebigen Startkante r (siehe Abb. 4.5) des Zyklus der inneren Kanten. Aktualisiere den Kantenzyklus.*

***Schritt 2**: Falls ein weiteres leeres Dreieck existiert, welches s enthält, wiederhole Algorithmus 4.2 mit s als Startkante.*

***Schritt 3**: Andernfalls überprüfe, ob der Zyklus nur noch aus drei Kanten besteht. In diesem Fall haben wir das ursprüngliche Polygon erfolgreich trianguliert. Besteht er aus mehr als drei Kanten, existiert keine Triangulation.*

Der erste Algorithmus wird einmal aufgerufen, der zweite solange, bis alle leeren Dreiecke im Inneren des Polygons verbaut wurden, also nicht weiter unterteilt werden kann.
Nach diesem groben Überblick über die Arbeitsweise folgt nun zunächst die Behandlung der Spezialfälle, die bisher nicht berücksichtigt wurden.

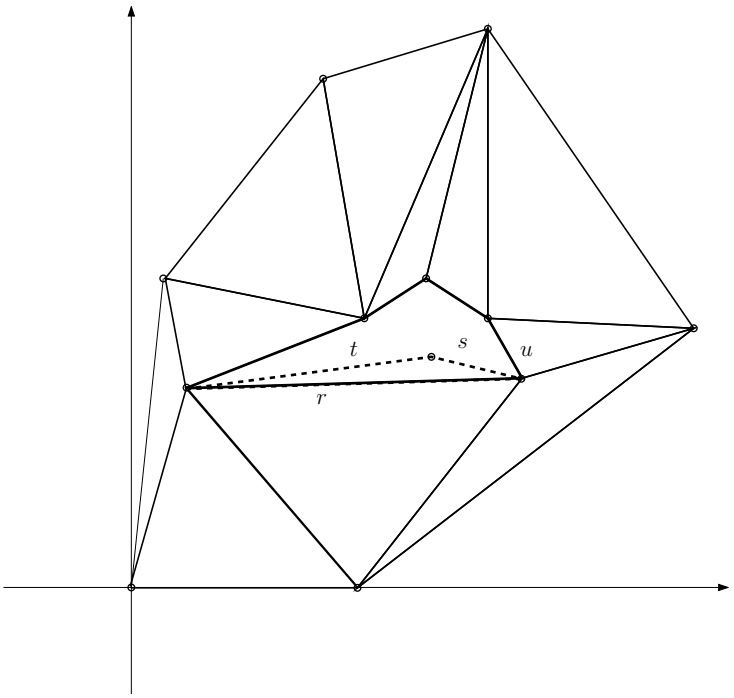

Abbildung 4.5: Die fettgedruckte „innere“ Hülle muß nun trianguliert werden. Wir starten mit dem gestrichelten Dreieck.

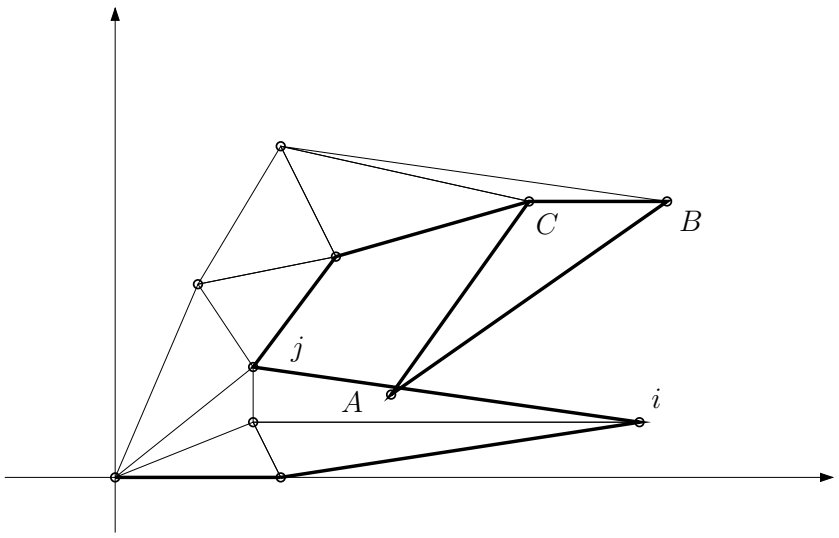

Abbildung 4.6: Das Dreieck ABC schneidet die Kante (i, j) der äußeren Hülle.

4.2 Spezialfälle

Dieser Abschnitt enthält die sowohl in Algorithmus 4.1 als auch in 4.2 während der Triangulationsberechnung auftretenden Spezialfälle.

4.2.1 Dreieck schneidet äußere Hülle

Während der Triangulationsberechnung kann es sein, daß das neu berechnete Dreieck Kanten schon verbauter Dreiecke schneidet (wie im Beispiel Abb. 4.6). Deshalb testen wir, falls der Triangulationsalgorithmus erfolgreich war, danach noch, ob sich die verbauten Kanten schneiden.

4.2.2 Dreieck mit zwei Außenkanten

Auf unserem Weg durch die äußere Schicht stoßen wir nicht zwangsläufig auf Dreiecke, die keine bzw. nur eine passende Außenkante enthalten. Unter Umständen tritt der Fall ein, daß ein Dreieck $\triangle(i, j, k)$ aus zwei Außenkanten besteht (Abb. 4.7). In diesem Fall fahren wir mit der nächsten inneren Kante, die auf der bisher berechneten äußeren Hülle ermittelt wird (zum Beispiel $(k, nachbarr)$ in Abb. 4.7), fort, denn die äußere Hülle muß ab dort weiterberechnet werden. Zu diesem Zweck wollen wir die äußere Hülle der bisher berechneten Triangulation als doppelt verkettete Liste speichern, die wir in jedem Schritt aktualisieren.

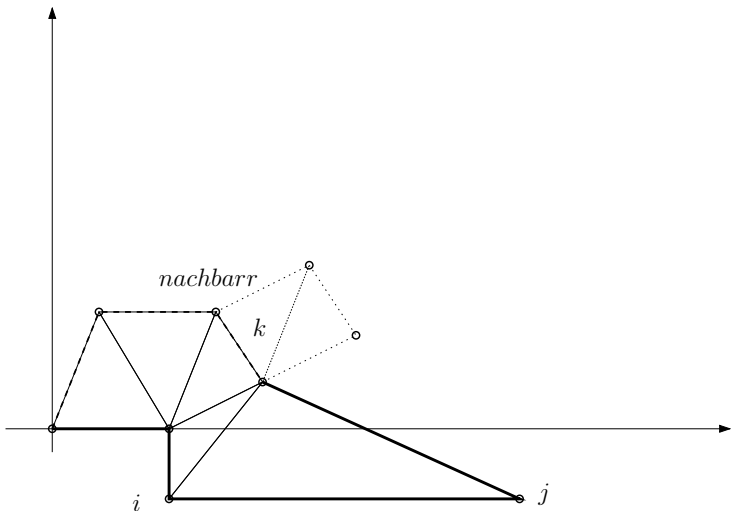

Abbildung 4.7: Die Außenkanten sind fettgedruckt.

4.2.3 Bereits verbaute Spitze

Da der Triangulationsalgorithmus nacheinander die Dreiecke einer Schicht berechnet, kann der Fall eintreten, daß der Knoten k an der Spitze des aktuellen Dreiecks bereits verbaut wurde (Abb. 4.8). Dann wird der Algorithmus *innereTriangulation()* für das neu entstandene, abgeschlossene innere Gebiet aufgerufen.

4.2.4 Dreieck enthält „spätere“ Außenkante

Ein weiterer Spezialfall tritt ein, falls wir in einem früheren Schritt schon ein Dreieck verbaut hatten, welches die nun im Zyklus folgende Außenkante enthält (Abb. 4.9). Deshalb müssen wir, immer wenn wir ein Dreieck mit einer Außenkante zu unserem Graphen hinzufügen, testen, ob die nachfolgende Kante bereits Außenkante ist. In diesem Fall müssen wir sie überspringen, und mit der nächsten inneren Kante auf der äußeren Hülle der bisher berechneten Figur fortfahren und an dieser die Triangulation fortsetzen. Dies läßt sich ebenfalls leicht mit einer verketteten Liste realisieren.

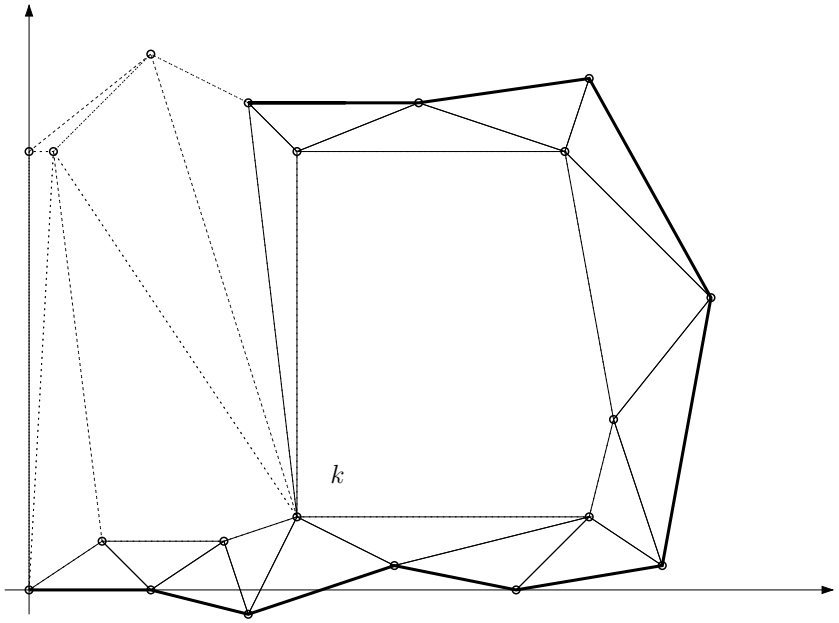

Abbildung 4.8: k ist bereits fest.

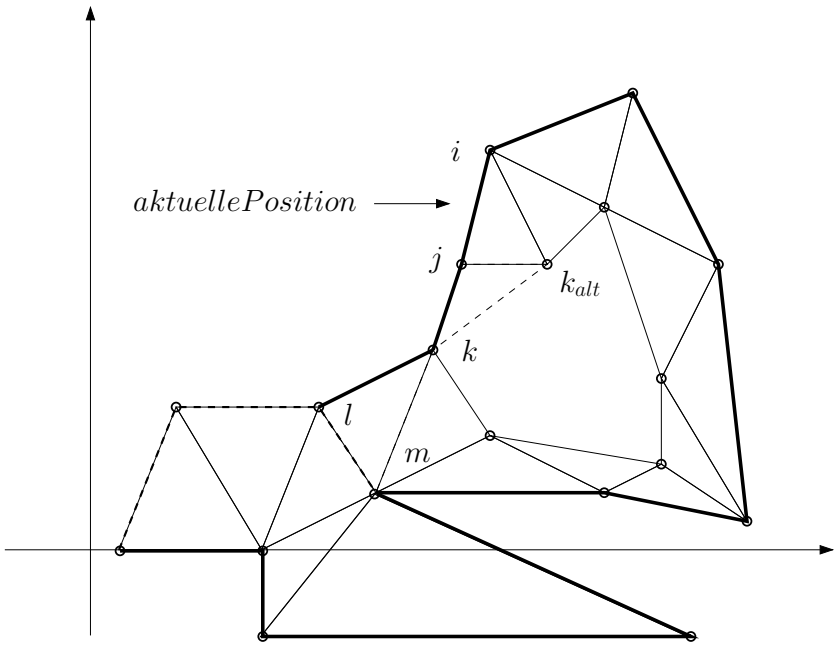

Abbildung 4.9: Die nächste Außenkante im Zyklus ist (j, k). Wir haben jedoch zuvor bereits die übernächste Außenkante (k, l) berechnet, als wir das Dreieck $\triangle(k, l, m)$ konstruiert haben.

4.3 Die Triangulation

4.3.1 Ersetzen von Kanten

Der Triangulationsalgorithmus berechnet inkrementell die äußere Hülle und trianguliert entstandene Polygone. Dabei speichert er die äußeren Kanten der bereits berechneten Triangulation als doppelt verkettete Liste in der Kantenmatrix M, die alle bisher verbauten Kanten der Triangulation enthält. Eine elementare Operation ist das Ersetzen einer Kante (j, k_{alt}) (in Abb. 4.10) durch zwei neue Kanten (j, k) und (k_{alt}, k) und die Aktualisierung der Liste. Dies geschieht in der Funktion *ersetzeKante()*. Dabei werden zuerst die beiden Nachbarn der Kante (j, k_{alt}) zwischengespeichert und als Nachbarn von $M(j, k_{alt})$ gelöscht, da diese Kante im Zyklus nicht mehr vorkommt. (j, k_{alt}) ist somit aus dem Zyklus gelöscht. Dann werden die Nachbarn der Kanten, die an (j, k_{alt}) angrenzend waren, sowie die der beiden neu eingefügten Kanten aktualisiert. Damit ist die neue Verkettung hergestellt. Zuletzt werden die den Kanten gegenüberliegenden Punkte im leeren Dreieck $\triangle(j, k_{alt}, k)$ als Spitze gespeichert, da diese für die Koordinatenberechnung (siehe 3.7) gebraucht werden. Dabei wird erst überprüft, ob die jeweilige Kante schon in einem leeren Dreieck vorkommt (*spitze1* schon einen Wert $\neq$ *null* gespeichert hat), oder ob die Kante zum ersten mal in einem Dreieck vorkommt.

funktion ersetzeKante(**kante** (k_{alt}, j), **knoten** k);

(**Initialisierung**)
$rechterNachbar := M(k_{alt}, j).nachbarr();$
$linkerNachbar := M(k_{alt}, j).nachbarl();$

(**Entferne (k_{alt}, j) aus Zyklus**)
$M(k_{alt}, j).setzeNachbarl(null),\ M(k_{alt}, j).setzeNachbarr(null);$

(**Aktualisiere Nachbarn**)
$M(k, j).setzeNachbarr(rechterNachbar), M(k, j).setzeNachbarl(k_{alt});$
$M(k_{alt}, k).setzeNachbarl(linkerNachbar), M(k_{alt}, k).setzeNachbarr(j);$
$M(linkerNachbar, k_{alt}).setzeNachbarr(k);$
$M(j, rechterNachbar).setzeNachbarl(k);$

(**Speichere Spitzen des neuen Dreiecks**)
if $(M(j, k).spitze1() = null)\ M(j, k).setzeSpitze1(k_{alt});$
else $M(j, k).setzeSpitze2(k_{alt});$

if $(M(k, k_{alt}).spitze1() = null)\ M(k, k_{alt}).setzeSpitze1(j);$
else $M(k, k_{alt}).setzeSpitze2(j);$

if $(M(k_{alt}, j).spitze1() = null)\ M(j, k).setzeSpitze1(k);$
else $M(k_{alt}, j).setzeSpitze2(k);$

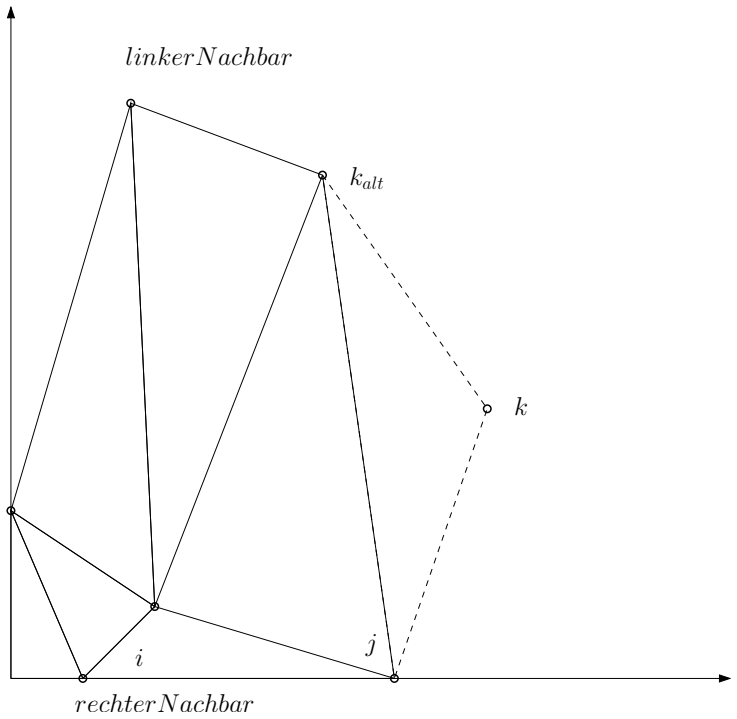

Abbildung 4.10: (j, k_{alt}) wird durch (j, k) und (k_{alt}, k) ersetzt. Die Nachbarn in der Liste werden entsprechend neu verkettet.

4.4 Der Triangulationsalgorithmus

Der Algorithmus *Triangulation* berechnet von der Startkante ausgehend die Kanten der Triangulation und speichert diese in M. Dabei ist (j, k_{alt}) immer die Kante, an der wir unser nächstes Dreieck bestimmen (Abb. 4.12). An verschiedenen Stellen stoßen wir auf das Problem, im Inneren entstandene Polygone weiter triangulieren zu müssen. Wir verwenden dann die Funktion *innereTriangulation()*, welche in Kapitel 4.5 erläutert wird. Zuerst wollen wir nun die Phasen des komplexen Algorithmus einzeln analysieren, um am Ende des Abschnittes den gesamten Algorithmus verstehen zu können.

4.4.1 Phase 1: Initialisierung und vorbereitende Maßnahmen

Nach der Initialisierung wird die Funktion *direkteVerbindungen()* aufgerufen. Diese berechnet wie in Abschnitt 3.1 beschrieben aus der Distanzmatrix D die Kanten der Triangulation und schreibt diese in V. Dabei überprüft sie auch, ob die Anzahl der Kanten im für Triangulationen gültigen Intervall (Eigenschaft 10. aus 2.5) liegt. Die Funktion *aussenkanten()* aus Abschnitt 3.2 wird solange mit unterschiedlichen Knoten aufgerufen, bis sie erfolgreich den Zyklus der Außenkanten berechnet und in AH gespeichert hat. Sie liefert dann den Wert *eins* zurück und speichert eine Außenkante (i, j) als Startkante der Triangulation ab. Sollte kein Außenkantenzyklus existieren, so wird dies spätestens dann erkannt, wenn alle n Knoten einmal durchlaufen wurden.
Der Startkante werden dann gemäß Abb. 4.2 Koordinaten zugewiesen und das zugehörige leere Dreieck zu (i, j), welches in der Funktion *aussenkanten*() ermittelt wurde, wird als Startdreieck $\triangle(i, j, k)$ festgelegt (Abb. 4.11). k werden die enstprechenden (positiven y-)Koordinaten (siehe 3.7) zugewiesen. Die Kanten des Startdreiecks werden dann aus V' entfernt. Nun haben wir also ein leeres Dreieck $\triangle(i, j, k)$, von welchem ausgehend wir die Triangulation weiter berechnen.

Algorithmus *Triangulation*

(**Initialisierung**)
initialisiere die Knotenmatrizen V, V' sowie die Kantenmatrizen AH und M jeweils mit *null*, Zählindex x mit *eins*;
berechne die Matrix V mithilfe der Funktion $direkteVerbindungen()$;
$V' := V$;

(**Bestimme Außenkantenzyklus**)
while ($x \leq n$ und $aussenkanten(x) = 0$) $x := x + 1$;
weise der Startkante (i, j) Koordinaten zu;
k sei die zugehörige Spitze zu (i, j), $\triangle(i, j, k)$ damit unser Startdreieck;
$berechneKoordinaten((i, j), k)$;
speichere $\triangle(i, j, k)$ als doppelt verkettete Liste in M und streiche die drei Kanten aus V';

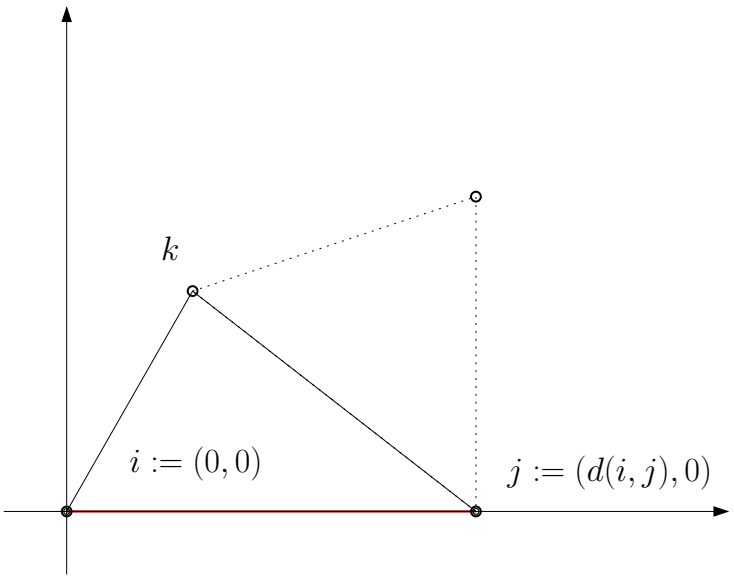

Abbildung 4.11: Der erste Schritt: Das Dreieck (i, j, k) wird als Startdreieck festgelegt. Als nächstes wird das gestrichelte Dreieck an der Kante (j, k) bestimmt.

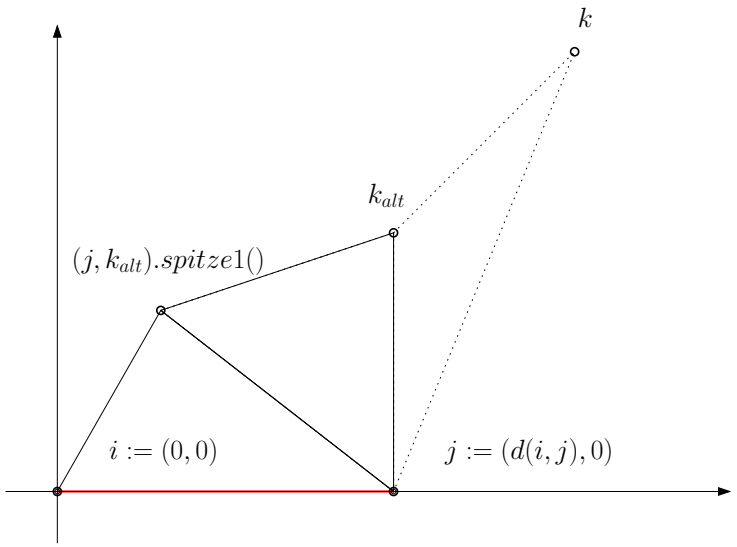

Abbildung 4.12: Die nächsten Schritte: Die äußere Hülle wächst weiter.

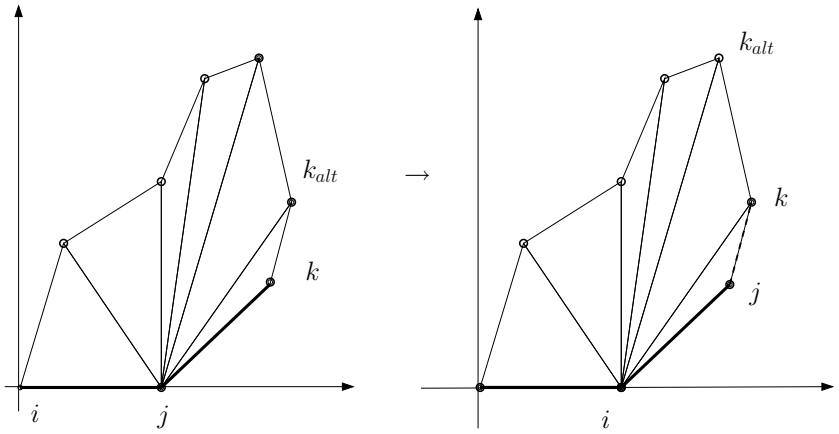

Abbildung 4.13: Die innere Schleife wird verlassen, sobald (j, k) die zu (i, j) benachbarte Außenkante ist. Dann werden die Bezeichnungen entsprechend neu gesetzt.

4.4.2 Phase 2: k ist noch nicht verbaut

Mit dem Startdreieck wird die äußere *repeat until*-Schleife betreten. Diese terminiert, sobald das aktuelle Dreieck wieder das Startdreieck ist (d.h. (i, j) ist wieder die Startkante). Falls die Kante (j, k) bereits die zu (i, j) benachbarte Außenkante ist[1], so wird die innere *repeat until*-Schleife nicht betreten (siehe Spezialfälle 4.2.2 bzw. 4.2.4). Die äußere *repeat-until*-Schleife durchläuft dann die aktuelle äußere Hülle, bis sie auf die nächste Kante stößt, die keine Außenkante ist, d.h. an der wir die Berechnung der äußeren Schicht fortsetzen müssen, oder bis (i, j) wieder die Startkante ist, denn dann haben wir einen Außenkantenzyklus konstruiert.
Anderenfalls werden in der inneren Schleife solange die angrenzenden leeren Dreiecke zu (j, k_{alt}) bestimmt (Abb. 4.11 - 4.13), bis die zu (i, j) folgende Kante der äußeren Hülle, die wir ja in Kapitel 3 bereits gefunden haben, im aktuellen Dreieck enthalten ist. Dabei ist k der neu berechnete Knoten und k_{alt} die jeweils im vorherigen Schritt neu berechnete Spitze. Es können drei Fälle eintreten:

1. k wurde noch nicht verbaut. Dann wird die Funktion $ersetzeKante()$ aufgerufen, die Kante (j, k_{alt}) durch die beiden neuen Kanten (j, k) und (k, k_{alt}) ersetzt und die Nachbarschaftsbeziehungen aktualisiert. Die beiden neuen Kanten werden dann in V' gelöscht.

```
(*bestimme angrenzendes Dreieck*)
    repeat until ((i, j) ist wieder die Startkante)
        repeat until ((j, k) ist die an j angrenzende Kante
        auf der äußeren Hülle)
            setze k_alt := k;
            k := angrenzendesDreick(j, k_alt);
            berechneKoordinaten((k_alt, j), k);

(*k noch nicht verbaut*)
            if (k ist noch nicht verbaut)
                ersetzeKante(j, k_alt);
                streiche die beiden Kanten aus V';
```

[1] wir haben ja den Zyklus der Außenkanten in der Funktion *aussenkanten*() bereits bestimmt und wissen daher, welche Außenkante als nächstes kommen muß

4.4.3 Phase 3: k ist schon verbaut

2. Der neue Knoten k ist bereits verbaut, liegt jedoch nicht auf der äußeren Hülle des bisher berechneten Graphen, also im Inneren. Dann bricht der Algorithmus sofort ab, denn es gibt Schnittpunkte der neuen Kanten (j,k), (k_{alt},k) mit schon verbauten Kanten.
3. Der neue Knoten k ist bereits verbaut, liegt jedoch auf der äußeren Hülle des bisher berechneten Graphen. Dabei ensteht in jedem Fall ein zusätzliches abgeschlossenes Polygon, welches unter Umständen weiter trianguliert werden muß. Abb. 4.14 zeigt ein Beispiel für den Fall, in welchem der neuen Knoten k schon verbaut, jedoch nicht bereits vorher unmittelbarer Nachbar von k_{alt} war. Die Kanten des neu entstandenen Polygons werden in M als doppelt verkettete Liste gespeichert, indem (k,k_{alt}) an der entsprechenden Stelle eingefügt wird und wir einmal um das Polygon „herumlaufen“ und die zu (k,k_{alt}) benachbarten Kanten aktualisieren[2]. Ebenso wird die äußere Hülle des gesamten Graphen aktualisiert, indem (j,k) für die Kanten des inneren Polygons hinzugefügt wird. Nachdem beide Zyklen aktualisiert worden sind, wird für das geschlossene Polygon die Funktion *innereTriangulation*() mit (k,k_{alt}) als Startkante aufgerufen (siehe Kap. 4.5). Diese trianguliert ein abgeschlossenes Gebiet vollständig, soweit dies möglich ist. Der Triangulationsalgorithmus fährt danach ganz normal fort mit der aktualisierten äußeren Hülle.
Ein Spezialfall liegt vor, falls die neue Spitze k bereits vorher unmittelbarer linker Nachbar von k_{alt} auf der äußeren Hülle war (Abb. 4.15). In diesem Fall entsteht nur eine neue Kante (j,k). Das Dreieck $\triangle(j,k_{alt},k)$ ist ja leer (dies wird von der Funktion *innereTriangulation*() erkannt), demnach muß nur die äußere Hülle des Graphen aktualisiert werden und der Triangulationsalgorithmus fährt fort.
Ein weiterer Spezialfall liegt vor, falls k bereits verbaut wurde, jedoch Randknoten der *endgültigen* Triangulation ist (Abb. 4.16). In diesem Fall entstünde zwar ein abgeschlossenes Polygon, dieses kann jedoch nicht weiter trianguliert werden, da es Außenkanten enthält und diese bereits in einem leeren Dreieck verbaut sind und daher in keinem weiteren leeren Dreieck mehr vorkommen können. Diesen Fall können wir jedoch leicht in $O(1)$ erkennen wenn wir uns für jeden Knoten bei der Berechnung der Außenkanten merken, ob er Randknoten der engültigen Triangulation ist. Ist dies der Fall, so liegt der diskutierte Spezialfall vor und wir können den Algorithmus abbrechen.

Sollte die Kante (j,k) bereits die gesuchte, nächste Außenkante sein, so wird die innere Schleife verlassen. Die Bezeichnungen werden umgesetzt

[2]speichern wir zu jedem Randknoten der aktuellen Triangulation dessen Nachbarn auf der äußeren Hülle, so können wir in $O(1)$ die neuen Nachbarknoten zu k und k_{alt} bestimmen

(Abb. 4.17 bzw. 4.18). Die neu entdeckte Außenkante ist nun (i, j), die nächste folgende Kante (j, k). Den anderen, zu k benachbarten Knoten nennen wir (k_{alt}), um unerwünschte Seiteneffekte zu vermeiden. Sollte nun (j, k) bereits eine Außenkante sein, die wir nebenbei in einem vorherigen Schritt bereits berechnet haben (siehe 4.2.4), so läuft die äußere Schleife so lange weiter, bis die nächste innere Kante (j, k) ermittelt wurde, an welcher wir die Berechnung der Triangulation fortsetzen müssen (oder natürlich bis wir wieder die Startkante erreichen, dann sind wir fertig). Erst wenn (j, k) innere Kante ist, wird die innere Schleife wieder durchlaufen.

(**k schon verbaut**)
else if (k ist schon verbaut und liegt nicht auf der äußeren Hülle des bisher berechneten Graphen) **STOP**;
else if (k ist schon verbaut und liegt auf der äußeren Hülle des bisher berechneten Graphen)
aktualisiere neu entstandene Zyklen in M;
streiche die neue(n) Kante(n) aus V';
$innereTriangulation(k, k_{alt})$;
end repeat
$(i, j) := (j, k)$;
$k := M(i, j).nachbarl()$;
$k_{alt} := M(j, k).nachbarl()$;
end repeat

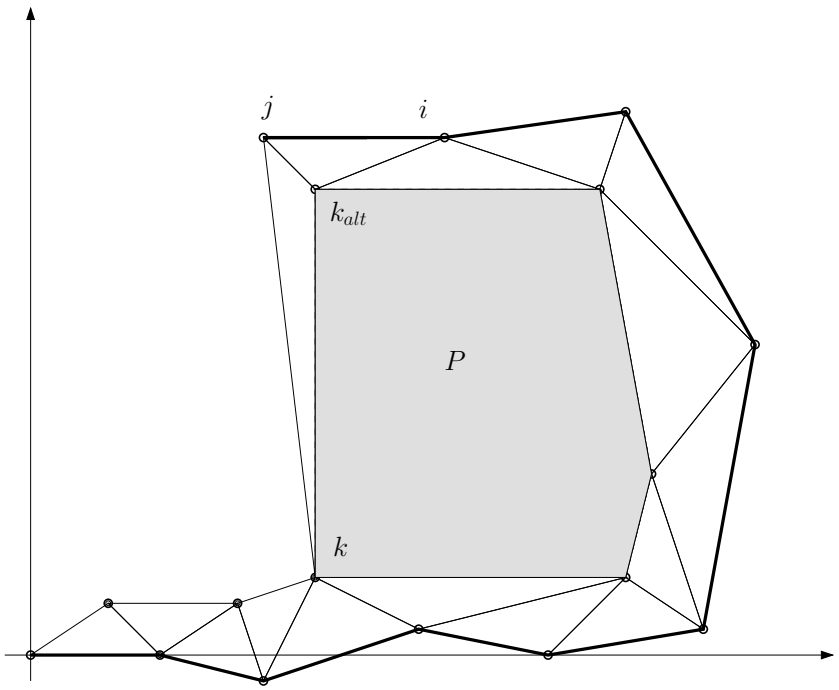

Abbildung 4.14: Das entstandene Polygon muß weiter trianguliert werden.

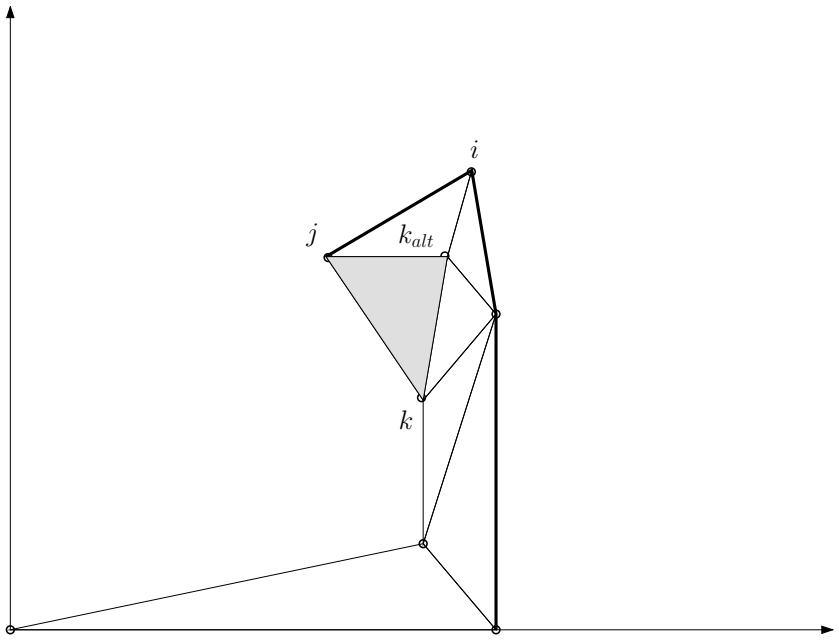

Abbildung 4.15: Die neue Spitze k ist direkter Nachbar von k_{alt}.

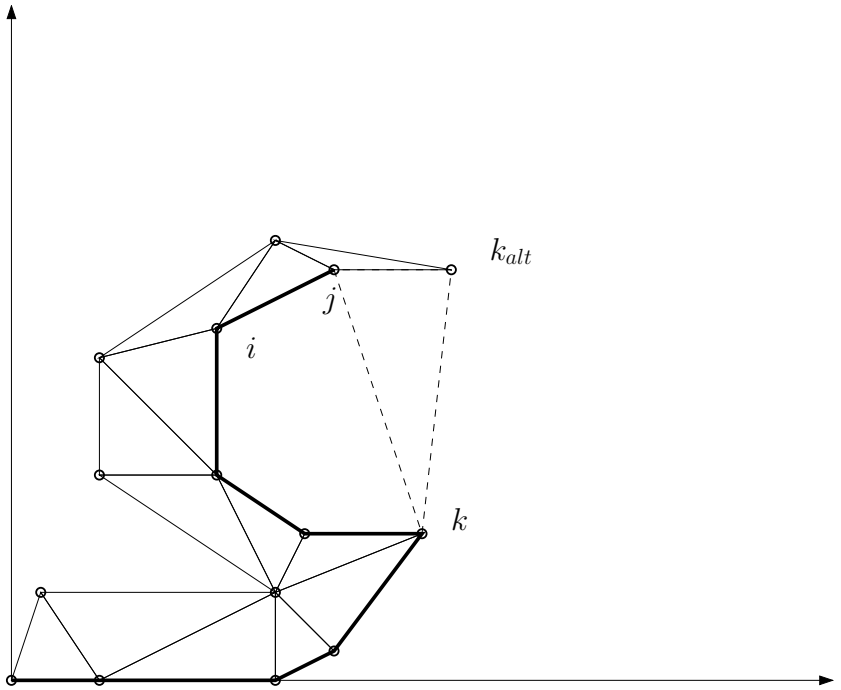

Abbildung 4.16: Durch das neu berechnete Dreieck (j, k_{alt}, k) entstünde ein „Loch“ in der Triangulation. Diesen Fall haben wir in 2.3 ausgeschlossen.

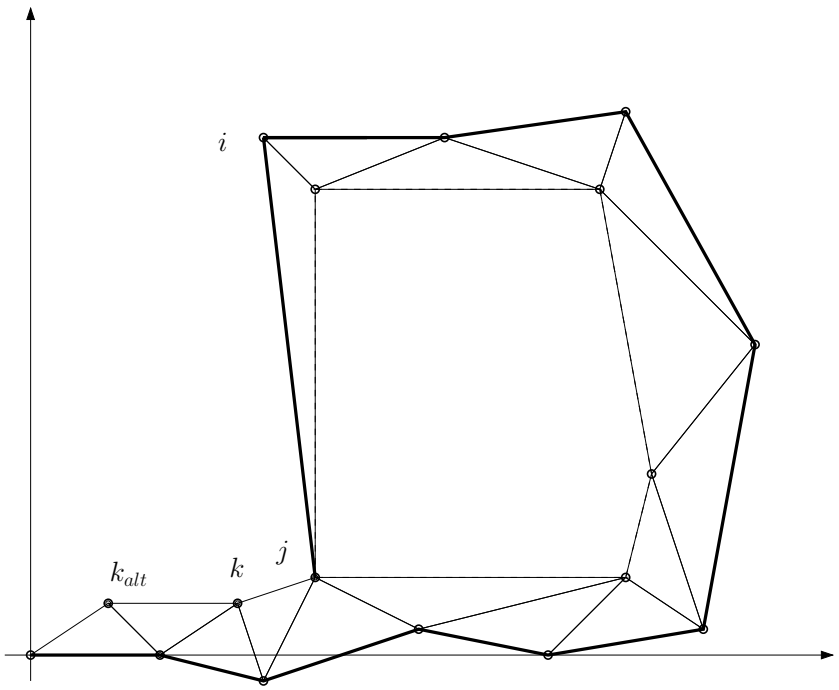

Abbildung 4.17: Wir wandern solange weiter, bis (j, k) keine Außenkante mehr ist oder wir wieder die Startkante erreichen.

4.4.4 Phase 4: Die hinreichenden Triangulationsbedingungen werden überprüft

Wurden die beiden Schleifen erfolgreich (also ohne vorzeitigen Abbruch) durchlaufen, so haben wir eine Triangulation konstruiert. Wir müssen nun zunächst überprüfen, ob alle Kanten verbaut wurden, denn wir beginnen unsere Triangulation ja mit der ersten gefundenen Außenkante, konstruieren den zugehörigen Außenkantenzyklus und triangulieren diesen. Sollte es jedoch mehrere Zusammenhangskomponenten geben (was nur vorkommen kann, wenn in D manche Einträge ∞ sind), so erkennen wir dies daran, daß nach der Triangulation noch nicht verbaute Kanten existieren. Deshalb überprüfen wir, ob V' leer ist. Sollte dies nicht der Fall sein, so könnten wir entweder versuchen, die restlichen Zusammenhangskomponenten zu konstruieren, indem wir den Triangulationsalgorithmus mit den in V' verbliebenen Kanten erneut aufrufen, oder wir brechen ab, weil wir diesen Fall nicht als gültige Triangulation anerkennen. In unserem Algorithmus brechen wir an dieser Stelle ab.
Als nächstes überprüfen wir, ob die Kantenlängen in unserer Triangulation auch den Distanzen in der Distanzmatrix D entsprechen. Da wir unter Um-

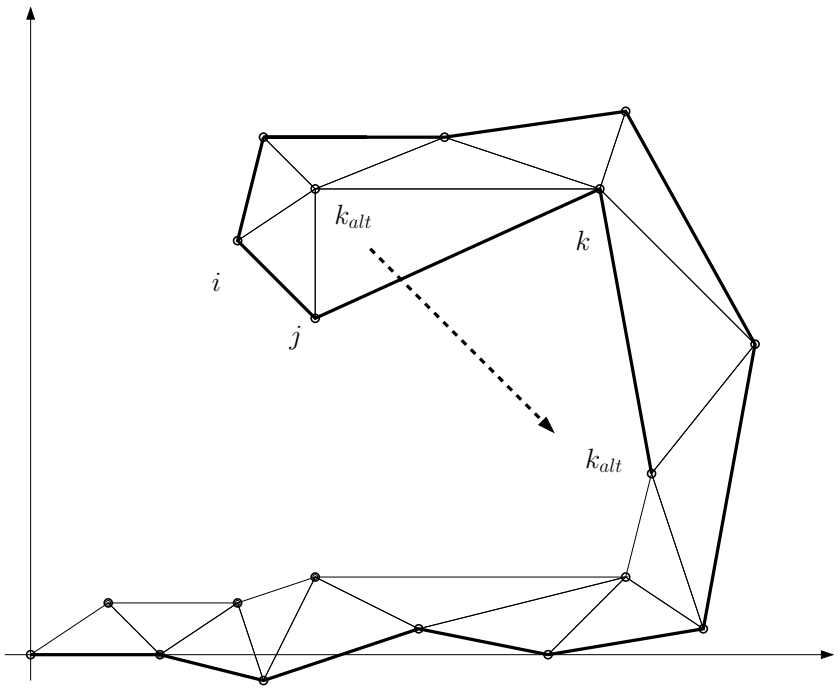

Abbildung 4.18: k_{alt} wird umgesetzt, für den Fall das (j, k) Außenkante ist und die nächste innere Kante, an welcher die Berechnung der äußeren Hülle fortgesetzt wird, gesucht werden muß.

ständen einen Knoten x mehrfach verbaut, ihm jedoch lediglich beim ersten Auftreten Koordinaten gemäß seines ersten gefundenen leeren Dreiecks zugewiesen haben, könnte es sein, daß manche Kantenlängen in anderen Dreiecken, die x enthalten, nicht mehr stimmen.
Sollte bisher alles korrekt sein, so müssen wir zuletzt testen, ob sich Kanten schneiden. Dazu verwenden wir die in 3.5 eingeführte Funktion *schnitttest*(). Wir rufen diese für alle möglichen Kantenpaare der Triangulation auf.

(**Wurden alle Kanten verbaut?**)
for each (Spalte j in V')
for each (Zeile i der aktuellen Spalte unterhalb der Diagonalen)
if ($V' \neq null$) **STOP**;
end for
end for

(**Überprüfe Distanzen**)
for each (Kante (i, j) der Triangulation)
berechne die Distanz D_T zwischen i und j mithilfe deren Koordinaten;
if ($D(i, j) \neq D_T$) **STOP** ;
end for

(**Schnittest**)
for each (Kante (i, j) der Triangulation)
for each (Kante $(k, l) \neq (i, j)$ der Triangulation)
if ($schnitttest((i, j), (k, l)) = 1$) **STOP**;
end for
end for

4.4.5 Der gesamte Algorithmus im Überblick

Der gesamte Triangulationsalgorithmus sieht demnach wie folgt aus:

Algorithmus *Triangulation*

(**Initialisierung**)
initialisiere die Knotenmatrizen V, V' sowie die Kantenmatrizen AH und M jeweils mit *null*, Zählindex x mit *eins*;
berechne die Matrix V mithilfe der Funktion $direkteVerbindungen()$;
$V' := V$;

(**Bestimme Außenkantenzyklus**)
while ($x \leq n$ und $aussenkanten(x) = 0$) $x := x + 1$;
weise der Startkante (i, j) Koordinaten zu;
k sei die zugehörige Spitze zu (i, j), $\triangle(i, j, k)$ damit unser Startdreieck;
$berechneKoordinaten((i, j), k)$;
speichere $\triangle(i, j, k)$ als doppelt verkettete Liste in M und streiche die drei Kanten aus V';
————————————————————————————Ende Phase 1

(**bestimme angrenzendes Dreieck**)
repeat until ((i, j) ist wieder die Startkante)
repeat until ((j, k) ist die an j angrenzende Kante auf der äußeren Hülle)
setze $k_{alt} := k$;
$k := angrenzendesDreick(j, k_{alt})$;
$berechneKoordinaten((k_{alt}, j), k)$;

(**k noch nicht verbaut**)
if (k ist noch nicht verbaut)
$ersetzeKante(j, k_{alt})$;
streiche die beiden Kanten aus V';
————————————————————————————Ende Phase 2

(**k schon verbaut**)
else if (k ist schon verbaut und liegt nicht auf der äußeren Hülle des bisher berechneten Graphen) **STOP**;
else if (k ist schon verbaut und liegt auf der äußeren Hülle des bisher berechneten Graphen)
aktualisiere neu entstandene Zyklen in M;

streiche die neue(n) Kante(n) aus V';
$innereTriangulation(k, k_{alt})$;
end repeat
$(i, j) := (j, k)$;
$k := M(i, j).nachbarl()$;
$k_{alt} := M(j, k).nachbarl()$;
end repeat

———————————————————————————Ende Phase 3

(**Wurden alle Kanten verbaut?**)
for each (Spalte j in V')
for each (Zeile i der aktuellen Spalte unterhalb der Diagonalen)
if $(V' \neq null)$ **STOP**;
end for
end for

(**Überprüfe Distanzen**)
for each (Kante (i, j) der Triangulation)
berechne die Distanz D_T zwischen i und j mithilfe deren Koordinaten;
if $(D(i, j) \neq D_T)$ **STOP** ;
end for

(**Schnittest**)
for each (Kante (i, j) der Triangulation)
for each (Kante $(k, l) \neq (i, j)$ der Triangulation)
if $(schnitttest((i, j), (k, l)) = 1)$ **STOP**;
end for
end for

———————————————————————————Ende Phase 4

4.5 Die innere Triangulation

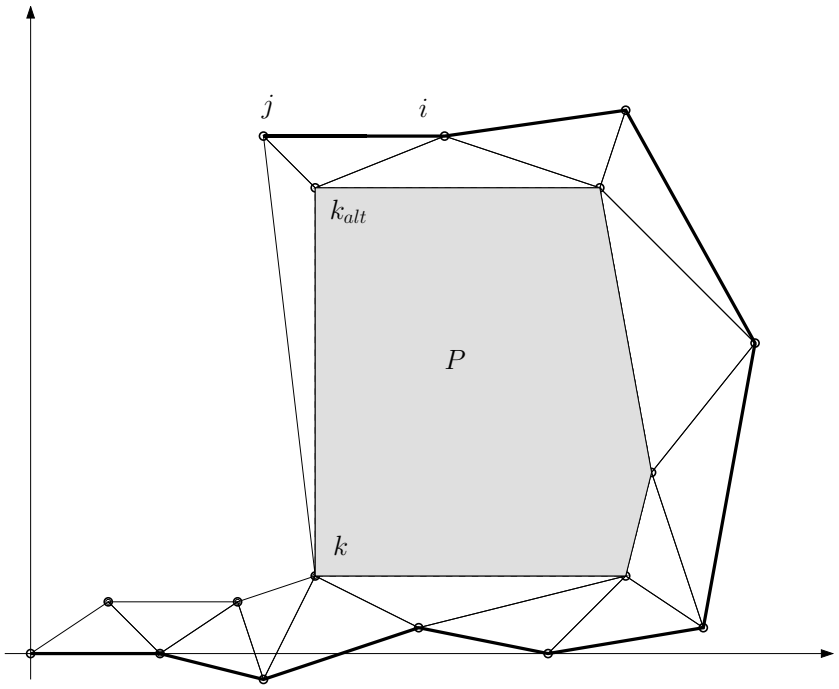

Abbildung 4.19: Das entstandene Polygon P muß weiter trianguliert werden.

Die Funktion *innereTriangulation()* wird aufgerufen, sobald ein geschlossener Zyklus innerer Kanten gefunden wird (Abb. 4.19), welcher weiter trianguliert werden muß. Dabei wird eine Kante des Zyklus als *Startkante* übergeben, z.B. (k, k_{alt}) in 4.19. Die restlichen Kanten sind als doppelt verkettete Liste in M gespeichert. Analog zu Kapitel 4.4 wollen wir zuerst die einzelnen Phasen des Algorithmus betrachten.

4.5.1 Phase 1: Berechnung des leeren Dreiecks zur Startkante

Nachdem die beiden Variablen z und k jeweils mit *null* inititalisiert wurden, wird das zur Startkante (i, j) gehördende noch nicht verbaute leere Dreieck mithilfe der Funktion *angrenzendesDreieck*() berechnet. Falls keins existiert, liefert diese den Wert *null* zurück und die innere Triangulation kann nicht weiter fortgesetzt werden. Besteht das zu triangulierende Polygon nun noch aus mehr als drei Kanten, so kann keine Triangulation existieren, der Algorithmus bricht ab. Andernfalls haben wir das Polygon erfolgreich trianguliert.

funktion innereTriangulation(**kante** (i, j))

(**Initialisierung**)
    initialisiere z und k mit *null*;

(**Berechne leeres Dreieck zur Startkante**)
    $k := angrenzendesDreieck(i, j)$;

(**Keine weitere Triangulation möglich?**)
    if $(k = 0)$
        bestimme Anzahl z der Kanten im Zyklus;
        if $(z > 3)$ **STOP**;

4.5.2 Phase 2: k ist noch nicht verbaut

Analog zu 4.4 können wir den Zyklus der Außenkanten mithilfe der Funktion $ersetzeKante()$ aktualisieren, wenn k noch nicht verbaut wurde. Wir berechnen dann auch die Koordinaten von k.

    else
        if (k ist noch nicht verbaut) $ersetzeKante((i, j), k)$;
            $berechneKoordinaten((i, j), k)$;

4.5.3 Phase 3: k ist bereits verbaut

Falls k bereits verbaut wurde, können entweder die in Abb. 4.20 und 4.21 skizzierten Fälle auftreten, in denen k bereits vor dem Einfügen des neuen Dreiecks Nachbar von (i, j) auf der Hülle des Polygons ist. In diesem Fall ist das grau gefärbte Dreieck bereits leer und braucht daher nicht weiter trianguliert werden. Der Kantenzyklus des verbleibenden Polygons wird aktualisiert, und die Funktion $innereTriangulation()$ wird, je nachdem ob k zuvor direkter Nachbar von i oder von j auf der Polygonhülle war, mit (k, j) oder (k, i) als Startkante aufgerufen.
Falls k weder zu i noch zu j zuvor direkt benachbart war (Abb. 4.22), so entstehen zwei neue Polygone, für die die Funktion $innereTriangulation()$ aufgerufen wird, nachdem die beiden Zyklen aktualisiert wurden.

if (k ist schon verbaut)
    if (k ist bereits Nachbar von (i, j))
    aktualisiere Zyklus;
    $innereTriangulation$(„(j, k) oder (i, k)“);
    if (k ist noch nicht Nachbar von (i, j))
    aktualisiere beide Zyklen;
    $innereTriangulation(j, k)$;
    $innereTriangulation(i, k)$;

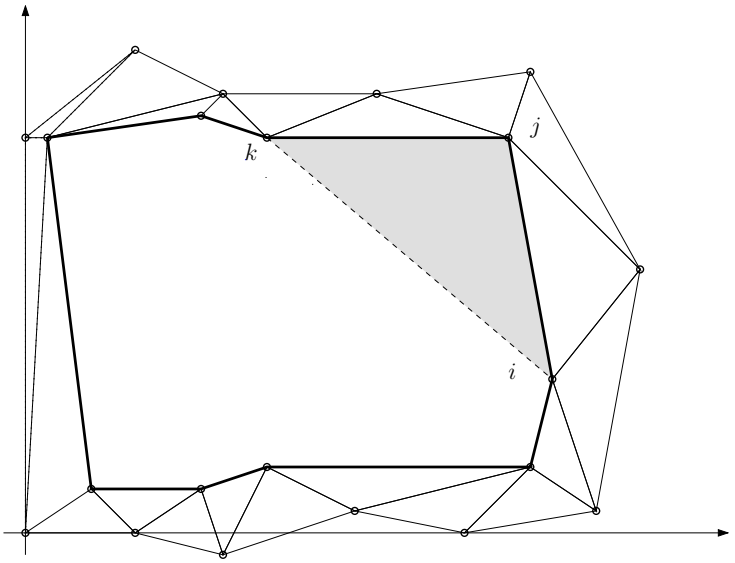

Abbildung 4.20: k ist rechter Nachbar von (i, j).

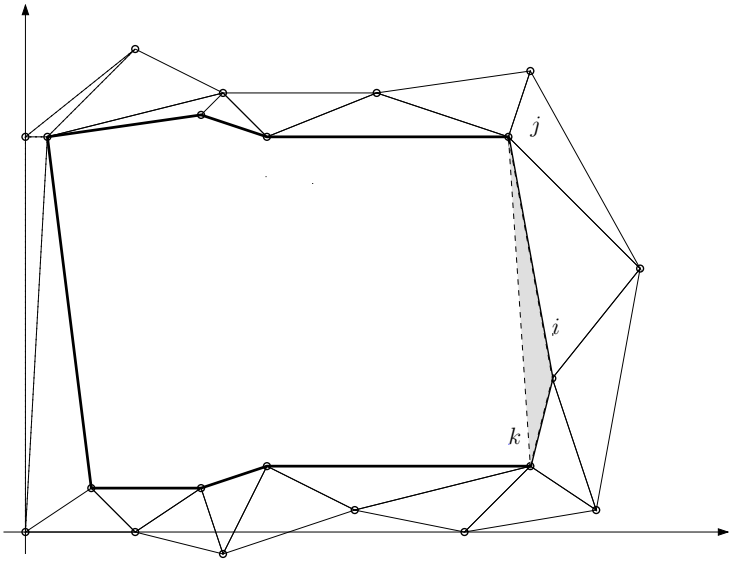

Abbildung 4.21: k ist linker Nachbar von (i, j).

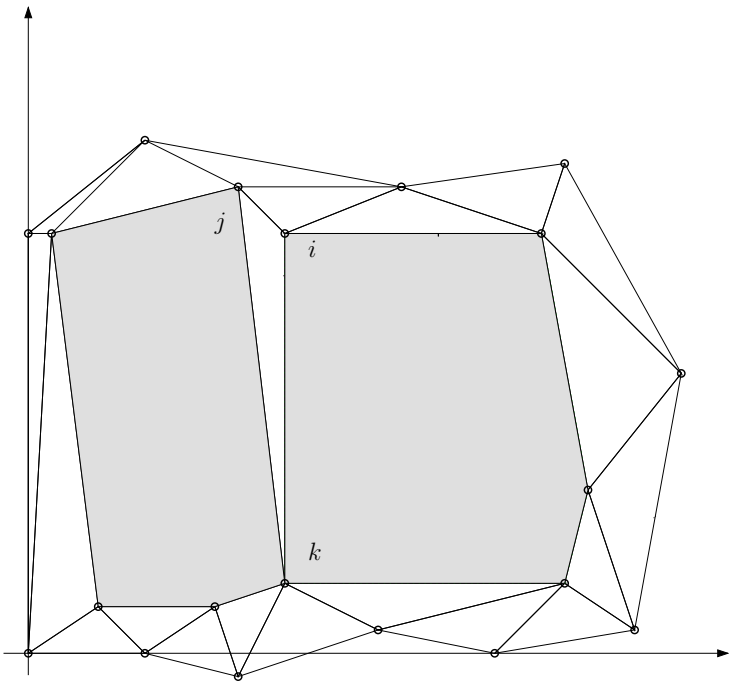

Abbildung 4.22: Die grauen Gebiete müssen unter Umständen weiter trianguliert werden.

4.5.4 Die Funktion *innereTriangulation*() im Überblick

funktion innereTriangulation(**kante** (i, j))

(**Initialisierung**)
    initialisiere z und k mit *null*;

(**Berechne leeres Dreieck zur Startkante**)
    $k :=$ *angrenzendesDreieck*(i, j);

(**Keine weitere Triangulation möglich?**)
    if $(k = 0)$
        bestimme Anzahl z der Kanten im Zyklus;
        if $(z > 3)$ **STOP**;
——————————————————————————— Ende Phase 1

(**k wurde noch nicht verbaut**)
    else
        if (k ist noch nicht verbaut) *ersetzeKante*$((i, j), k)$;
            berechneKoordinaten$((i, j), k)$;
——————————————————————————— Ende Phase 2

(**k wurde bereits verbaut**)
        if (k ist schon verbaut)
            if (k ist bereits Nachbar von (i, j))
            aktualisiere Zyklus;
            innereTriangulation(„(j, k)*oder*(i, k)“);
            if (k ist noch nicht Nachbar von (i, j))
            aktualisiere beide Zyklen;
            innereTriangulation(j, k);
            innereTriangulation(i, k);
——————————————————————————— Ende Phase 3

4.6 Korrektheit des Algorithmus und Eindeutigkeit der Triangulation

Wir möchten nun zeigen, daß zu einer gegebenen Distanzmatrix die Triangulation, sofern sie existiert, eindeutig bestimmt ist (also daß Satz 2.1 gilt) und daß unser Algorithmus diese dann auch korrekt berechnet. Dies läßt sich leicht anhand der Arbeitsweise des Algorithmus begründen.
Der Algorithmus beginnt zunächst mit einem leeren Dreieck, das eine Außenkante enthält. Sollte eine Triangulation existieren, so berechnet die Funktion *aussenkanten*() alle Kanten, die genau in einem leeren Dreieck vorkommen. Diese sind eindeutig festgelegt und müssen Außenkanten der Triangulation sein, da die Funktion und der darin verwendete Kürzeste-Wege-Algorithmus für jeden Knoten bestimmt, ob dieser vollständig eingeschlossen ist.
Wir wählen also ein leeres Dreieck $\triangle(i,j,k)$ in Abb. 4.23, das eine Außenkante enthält, als Startdreieck. Dieses ist bis auf Rotation, Translation und Spiegelung offensichtlich eindeutig durch die Angabe der drei Seitenlängen bestimmt (siehe A.1). Dann muß in diesem Dreieck mindestens eine innere Kante (j,k) existieren, falls eine Triangulation existiert und diese nicht nur aus einem Dreieck besteht. Diese innere Kante muß außerdem in einem weiteren leeren Dreieck $\triangle(j,k,l)$ vorkommen, welches in der Halbebene H_{frei} in Abb. 4.23 liegt. Andernfalls müßte es $\triangle(i,j,k)$ vollständig einschließen (denn $\triangle(i,j,k)$ ist ja leer), was jedoch ein Widerspruch zu der Feststellung, daß (i,j) Außenkante ist, wäre. Das leere Dreieck $\triangle(j,k,l)$ ist ebenfalls eindeutig bestimmt. Es muß dasjenige bisher unverbaute Dreieck sein, welches (j,k) enthält und das die kleinsten Innenwinkel aller restlichen (j,k) enthaltenden Dreiecke besitzt (siehe 3.6). Aus diesem Grund muß es auch leer sein. Laut A.1 ist also auch die Position von l relativ zu (j,k) eindeutig bestimmt, da alle drei Seitenlängen in $\triangle(j,k,l)$ durch D gegeben sind. Dieses Argument läßt sich auf jedes weitere leere Dreieck, welches wir „anbauen“, übertragen. Da immer alle drei Seitenlängen der leeren Dreiecke bekannt sind, eine Kante bereits verbaut ist, und die Halbebene, in der die neue Spitze liegen muß, feststeht, ist die Lage eines jeden neuen Dreiecks relativ zur bereits berechneten Triangulation eindeutig. Da wir natürlich zwangsläufig irgendwann auf die anderen Dreiecke der äußeren Hülle stoßen, ist es egal mit welchem Startdreieck wir beginnen, solange es eine Außenkante enthält, denn durch die eindeutige Konstruktion ist auch die relative Lage der Dreiecke der äußeren Hülle zueinander eindeutig. Wurden alle Knoten und Kanten verbaut, weist nun diese eindeutige Triangulation keine Kantenschnitte auf, und entsprechen auch wirklich alle verbauten Kanten den durch D vorgegebenen Distanzen, so haben wir erfolgreich die Triangulation konstruiert. Damit ist Satz 2.1 und

die Korrektheit des vorgestellten Triangulationsalgorithmus bewiesen.

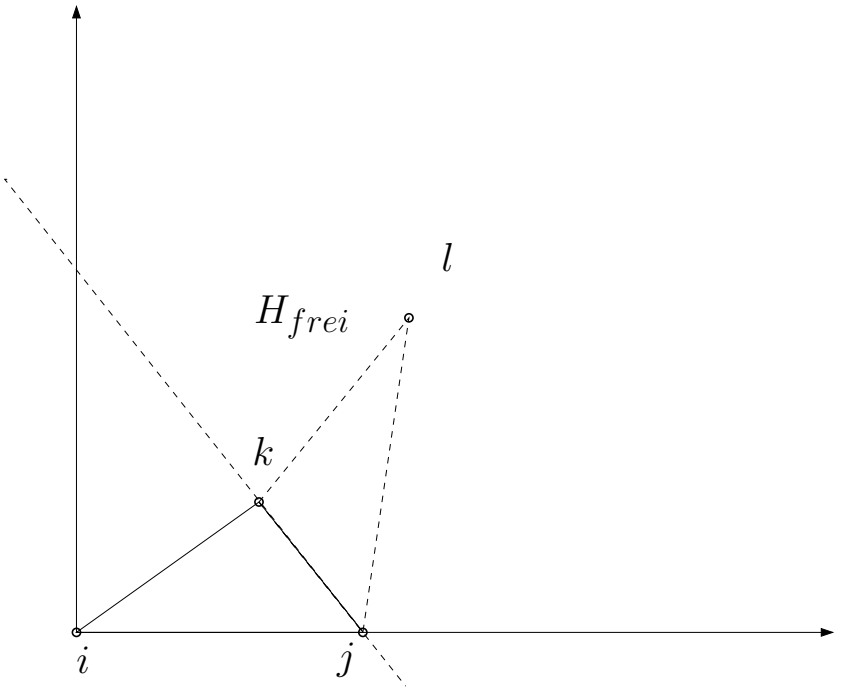

Abbildung 4.23: Das andere leere Dreieck, welches (j, k) enthält, muß in H_{frei} liegen.

Kapitel 5

Laufzeitanalyse

5.1 Kantenelimination und Berechnung der äußeren Hülle

Wir möchten nun eine obere Laufzeitschranke für unseren Triangulationsalgorithmus bestimmen. Dazu betrachten wir nacheinander alle Schritte des Verfahrens. Zuerst wird aus der Distanzmatrix D die Matrix V, welche nur die direkten Verbindungen enthält, berechnet. Um die direkten Verbindungen zu bestimmen, wird jede der $\binom{n}{2}$-vielen ungerichteten Kanten auf $n-2$ Zwischenpunkte getestet. Damit ergibt sich als Laufzeit $O(n^3)$. Laut 6) und 7) in 2.5 existieren $O(n)$ Kanten e und Dreiecke f, falls eine Triangulation existiert. Wir testen nun, ob dies für unseren durch Kantenelimination erhaltenen Graphen zutrifft. Speichern wir den Graphen nicht als Adjazenzmatrix, sondern in einer für dünne Graphen ökonomischeren Variante, wie etwa der Adjazenzliste, so geht dies offenbar in $O(n)$. Ab jetzt können wir sicher sein, daß der betrachtete Graph nur $O(n)$ Kanten hat, ansonsten haben wir schon abgebrochen.
Nun möchten wir herausfinden, welche Kanten Außenkanten in unserer Triangulation sind, falls diese existiert (siehe 3.2.2-3.2.3). Dazu müssen wir zu einem zu testenden Knoten i zunächst alle direkten Nachbarn finden. Verwenden wir eine Datenstruktur wie etwa die Adjazenzliste, die es uns ermöglicht, zu jedem Knoten die direkten Nachbarn in $O(1)$ zu bestimmen, so können wir für alle Knoten i die Menge der Nachbarn $N(i)$ gleichzeitig in $O(n)$ bestimmen, da es nur $O(n)$-viele Kanten gibt, und jede Kante (i,j) höchstens zweimal benutzt wird, nämlich bei der Berechnung von $N(i)$ und $N(j)$.
Wir benötigen dann alle Kanten, die mit dem Testknoten i ein (nicht notwendig leeres) Dreieck bilden, um die Dreiecke um i „herum“ zeichnen zu können. Diese können wir leicht finden, indem wir alle $O(n)$-vielen Kanten

einmal durchlaufen und testen, ob die jeweiligen Knoten der Kante in $N(i)$ liegen. Für alle Aufrufe ergibt sich also eine Laufzeit in $O(n^2)$ für die Bestimmung aller Dreiecke.
Ausschlaggebend für die obere Schranke ist jedoch der letzte Schritt (4. in 3.2.2), in welchem die Kanten „nichtleerer" Dreiecke eliminiert werden. Wir benutzen einen Kürzeste-Wege-Algorithmus, welcher uns zu zwei Knoten die kürzeste Distanz in dem erzeugten Winkelgraphen liefert, in dem wir zuvor die direkte Verbindung entfernt haben. Laut 10. in 2.5 gilt für ebene Triangulationen $e < 3n - 6$. Sollten wir diese Schranke überschreiten, können wir den Algorithmus abbrechen und eine Triangulation ausschließen. Der Algorithmus von Dijkstra[1] läßt sich damit und laut dem Buch von M.L. Fredman und D.E. Willard in [4] in $O(n \log n/(\log(\log n)))$ (oder grob: $O(n \log n)$) implementieren. Wenn k_i die Anzahl der Knoten ist, die direkt mit i verbunden sind, so müssen wir den Algorithmus von Dijkstra $O(k_i)$-mal aufrufen, denn der Winkelgraph ist als Teilgraph der Triangulierung auch planar und enthält daher laut Euler nur $O(n)$ viele Kanten. Sollte er mehr enthalten, können wir wieder abbrechen, weil dann die Distanzmatrix nicht von einer Triangulierung herrühren kann. Pro Knoten i ergäben sich also Kosten von $O(k_i^2 \log k_i)$. Um die Gesamtlaufzeit für alle n Knoten abzuschätzen, benötigen wir folgende Ungleichungen für Zahlen $a_1, ..., a_n \geq 0$:

* $\sum_{i=1}^{n} a_i^2 \leq (\sum_{i=1}^{n} a_i)^2$.

Beweis. $(\sum_{i=1}^{n} a_i)^2 = (a_1 + a_2 + a_3 + ... + a_n)^2$

$= \sum_{i=1}^{n} \sum_{j=1}^{n} a_i a_j = \sum_{i=1}^{n} a_i^2 + Rest$, wobei $Rest \geq 0$ gilt.

$\geq \sum_{i=1}^{n} a_i^2$. □

Um die Laufzeit für die Bestimmung aller Außenkanten zu berechnen, müssen wir n Knoten betrachten, die jeweils Kosten $(k_v^2 \cdot \log k_v)$ verursachen. Also ergibt sich:

$O(\sum_{v=1}^{n}(k_v^2 \cdot \log k_v))$

$\leq O(\log n \cdot \sum_{v=1}^{n} k_v^2)$

$\leq O(\log n \cdot (\sum_{v=1}^{n} k_v)^2)$ laut *.

[1] der Dijkstra-Algorithmus berechnet die kürzesten Pfade zwischen einem Knoten s und allen anderen im Graphen vorkommenden Knoten

$\leq O(n^2 \cdot \log n)$, da jede Kante nur zweimal angesehen wird.

5.2 Die Berechnung eines an (i, j) angrenzenden leeren Dreiecks

Nachdem diese Vorbereitungsmaßnahmen erfolgreich abgeschlossen wurden, betrachten wir nun eine elementare Operation des Triangulationsalgorithmus: Die Berechnung des angrenzenden Dreiecks zu einer bereits verbauten inneren Kante (i, j). Dabei werden zuerst alle $n-2$ Knoten auf direkte Verbindungen mit i und j getestet ($O(n)$). Wir können leicht den „Kandidaten“ des innersten Spitzenpunktes bestimmen, indem wir die Winkel der entstandenen Dreiecke vergleichen. Wie in Abschnitt 4.6 beschrieben gilt, daß, falls D von einer ebenen Triangulation stammt und klar ist, daß ein leeres Dreieck nur noch auf einer Seite der betrachteten Kante (i, j) liegen kann (z.B. weil auf der anderen die äußere Hülle liegt), dann derjenige unverbaute Knoten k mit (i, j) das gesuchte leere Dreieck bildet, der den Innenwinkel des Dreiecks $\triangle(i, j, k)$ minimiert. Finden wir kein angrenzendes Dreieck, so können wir eine Triangulation ausschließen. Damit ergibt sich als obere Schranke für das Bestimmen des angrenzenden Dreiecks $O(n)$, da wir alle $n-2$ Knoten einmal durchgehen und uns dabei merken müssen, welcher Knoten zum bisher innersten Dreieck gehört.

5.3 Die Triangulation

Der Triangulationsalgorithmus bestimmt ausgehend von einem Startdreieck, welches die Startkante s enthält so lange angrenzende Dreiecke, bis eine zu s benachbarte Außenkante enthalten ist. Jedes Dreieck verursacht laut 5.2 Kosten $O(n)$, da aus n Knoten die Spitze des innersten Dreiecks bestimmt werden muß. Allerdings ergibt sich für alle Dreiecke zusammen eine bessere Abschätzung, denn im Mittel gilt für alle Kanten (i, j) einer Triangulation $N(i) \cup N(j) \leq 12 + \frac{12}{n} \leq 16$, also gibt es im Mittel nur konstant viele Nachbarknoten. Laut 7. in 2.5 gilt $e < 3n+3$. Damit ist $\sum_V d_v = 2e < 6+6n$, wenn d_v der Grad von v ist. Im Mittel ergibt sich dann für den Grad jedes Knotens $\frac{\sum_V d_v}{n} < 6 + \frac{6}{n}$ und damit $N(i) \cup N(j) \leq d_i + d_j \leq 12 + \frac{12}{n}$.
Wenn wir annehmen, daß die Anzahl n der Knoten ≥ 3 ist, so ergibt sich $12+\frac{12}{n} \leq 16$. Also können wir alle leeren Dreiecke der Triangulation insgesamt in $O(n)$ berechnen.

5.4 Kantenlängentest und Schnitttest

Am Ende des Triangulationsalgorithmus haben wir für jeden Knoten eine Position im $\mathbb{R}^2$ bestimmt. Wir können in konstanter Zeit für eine Kante (i, j) testen, ob der sich aus den Positionen ergebende euklidische Abstand $|ij|$ gleich der Vorgabe $d(i, j)$ ist. Für alle Kanten geht dies daher insgesamt in $O(n)$. Außerdem können wir in $O(n^2)$ alle Kantenpaare auf Schnitt testen. Insgesamt liegt die Laufzeit des vorgestellten Verfahrens also in $O(n^3)$, wobei der Kanteneliminierungsschritt entscheidend ist. Die Ermittlung der Außenkanten funktioniert in $O(n^2 \log n)$ und die restliche Berechnung der Triangulation benötigt aufgrund des Kantenschnitttests $O(n^2)$.

Kapitel 6

Zusammenfassung und Ausblick

Wir haben nun ein Triangulationsverfahren kennengelernt, welches zu einer gegebenen Distanzmatrix eines metrischen Raumes eine ebene Triangulation in $O(n^3)$ bestimmt. Ausschlaggebend für diese Schranke war dabei die Funktion, die die direkten Verbindungen, also die in der Triangulation darzustellenden Kanten, berechnet.
Der nächste Teilschritt war die Berechnung der äußeren Hülle. Die obere Schranke hierfür ($O(n^2 \log n)$) wurde durch Verwendung des Dijkstra-Algorithmus ermittelt. Die Kanteneliminierung ist gleich zweifach für die Laufzeit unseres Algorithmus entscheidend.
Nach Beendigung der Forschungsarbeit für diese Diplomarbeit wurde die Laufzeit der Kanteneliminierung von mehreren Autoren weiter ($O(n^2)$) verbessert (siehe [11]). Dadurch, dass man die Kanteneliminierung in $O(n^2)$ schaffen kann, ergibt sich eine Gesamtlaufzeit in $O(n^2)$.
Der eigentliche Triangulationsalgorithmus läßt sich dann, wenn wir die äußere Hülle und alle Kanten kennen, in $O(n^2)$ realisieren. Da laut Euler in Triangulationen $O(n)$-viele Kanten und Dreiecke existieren, ließe sich hierbei die Laufzeit nur verringern, indem wir den Aufwand für den Schnittpunkttest reduzieren würden.
Ein anderer Algorithmus ist möglich, indem wir die Idee von der Bestimmung der Außenkanten zur Konstruktion der gesamten Triangulation benutzen. Schließlich haben wir nach der Kanteneliminierung im Winkelgraphen $W(i)$ schon alle an i angrenzendenn leeren Dreiecke und sogar deren Reihenfolge bestimmt. So könnte man sich von Knoten zu Knoten hangeln und jeweils alle angrenzenden noch nicht verbauten leeren Dreiecke verbauen.
Da letzteres in $O(k_i)$ also insgesamt in $O(n)$ funktioniert, wenn die angrenzenden leeren Dreiecke bekannt sind, ist auch für diesen Algorithmus die Kanteneliminierung im Winkelgraphen bzw. in der Distanzmatrix entscheidend, die nach unserem Verfahren in $O(n^2 \log n)$ bzw. $O(n^3)$ läuft.

Anhang A

A.1 Elementares zur Dreiecksberechnung

Im Verlauf der Arbeit ist des öfteren von Dreiecken die Rede. Deshalb lohnt es sich, an dieser Stelle einige generelle Eigenschaften allgemeiner Dreiecke festzuhalten.

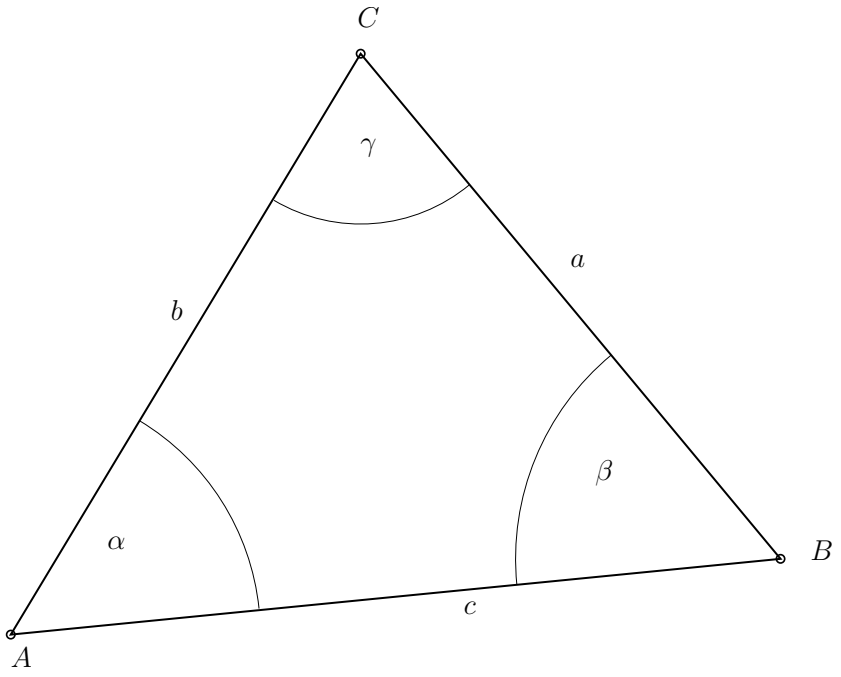

Abbildung A.1: Ein allgemeines Dreieck.

1. Drei Strecken a, b, c lassen sich genau dann zu einem echten Dreieck verbinden, wenn gilt: $a < b + c$, $b < a + c$ und $c < a + b$.

2. Die Winkelsumme in einem Dreieck beträgt immer 180°.

3. Ein Dreieck ist eindeutig durch die Angabe zweier Seitenlängen und des eingeschlossenen Winkels definiert. Mit Hilfe trigonometrischer Funktionen erhält man die fehlende Seitenlänge:

Cosinussatz:

$$a^2 = b^2 + c^2 - 2bc\cos\alpha \Rightarrow a = \sqrt{b^2 + c^2 - 2bc\cos\alpha}$$

$$b^2 = a^2 + c^2 - 2ac\cos\beta \Rightarrow b = \sqrt{a^2 + c^2 - 2ac\cos\beta}$$

$$c^2 = a^2 + b^2 - 2ab\cos\gamma \Rightarrow c = \sqrt{a^2 + b^2 - 2ab\cos\gamma}$$

Für $\gamma = 90°$ ergibt sich als Spezialfall der **Satz des Pythagoras**:

$c^2 = a^2 + b^2$.

Außerdem gilt dann:

$\cos\alpha = \sin\beta = \frac{b}{c}$;
$\sin\alpha = \cos\beta = \frac{a}{c}$;
$\tan\alpha = \frac{a}{b}$ bzw. $\tan\beta = \frac{b}{a}$.

4. Ein Dreieck ist eindeutig durch die Angabe aller drei Seitenlängen bestimmt. Mit Hilfe des umgeformten Cosinussatzes erhält man die fehlenden Winkelgrößen:

$$a^2 = b^2 + c^2 - 2bc\cos\alpha \Rightarrow \alpha = \arccos\frac{b^2+c^2-a^2}{2bc}$$

$$b^2 = a^2 + c^2 - 2ac\cos\beta \Rightarrow \beta = \arccos\frac{a^2+c^2-b^2}{2ac}$$

$$c^2 = a^2 + b^2 - 2ab\cos\gamma \Rightarrow \gamma = \arccos\frac{a^2+b^2-c^2}{2ab}$$

5. Die Verhältnisse von Winkel und gegenüberliegender Seite sind gleich:

Sinussatz: $\frac{a}{\sin\alpha} = \frac{b}{\sin\beta} = \frac{c}{\sin\gamma} = 2r$, wobei r der Umkreisradius ist.

Sind nur drei beliebige Größen bekannt (SWW, WSW oder SSW), so lassen sich immer die jeweils fehlenden Größen zur Anwendung des Cosinussatzes mittels Sinussatz bzw. Winkelsumme berechnen. Zur eindeutigen Beschreibung eines Dreiecks benötigt man also **drei Größen**.

A.2 Die Bezeichnungen „linker“ und „rechter“ Knoten

Befinden wir uns auf der äußeren Hülle der Triangulation, so wählen wir als Blickrichtung den Blick weg vom Graphen (Abb. A.2). Die Bezeichungen *rechts* und *links* gelten dann entsprechend der Abbildung. Da wir mit der Kante (i, j) im Nullpunkt des Koordinatensystems starten (dabei ist i rechter und j linker Knoten), entwickeln wir zu jeder Zeit die äußere Hülle nach links weiter.
Befinden wir uns in der Phase, in der ein Polygon trianguliert werden muß (also in der Funktion $innereTriangulation()$), so ist die Blickrichtung von der Kante (i, j) weg ins Innere des zu triangulierenden Polygons (Abb. A.3).

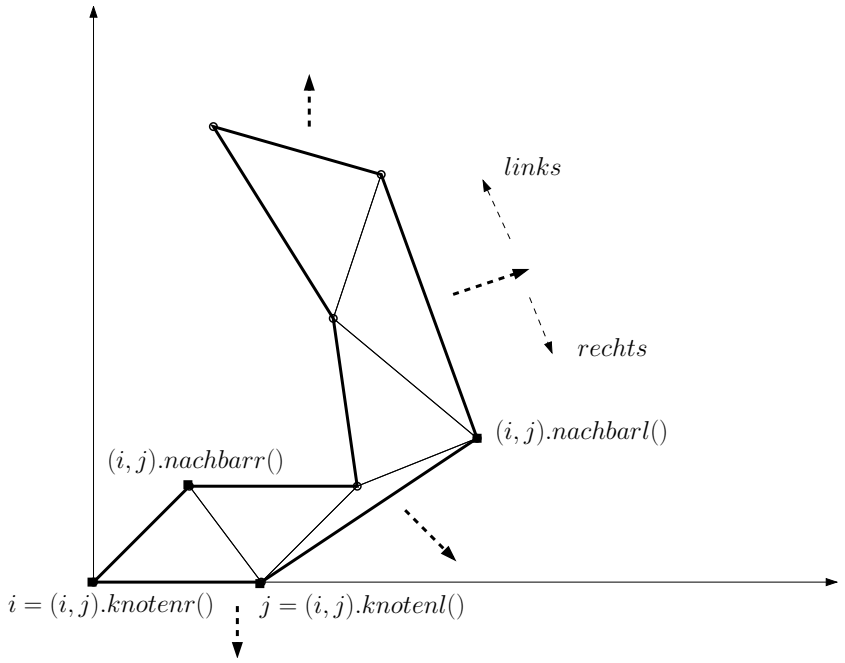

Abbildung A.2: Die Blickrichtung zeigt weg von der Hülle.

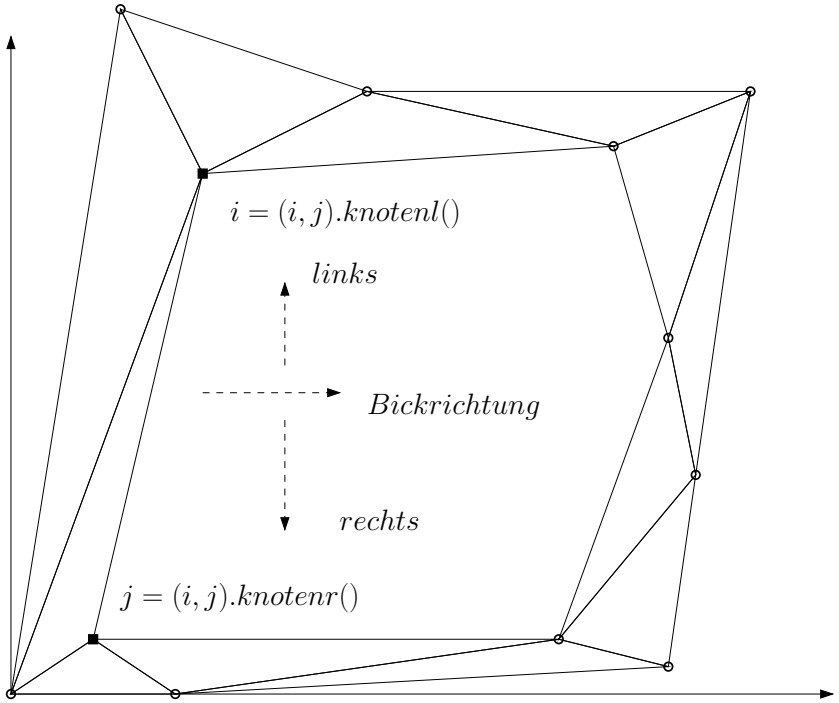

Abbildung A.3: Die Blickrichtung zeigt ins Polygoninnere.

Anhang B

B.1 Überblick über die verwendeten Elemente

Datentypen:

knoten i: Speichert Index eines Knotens.

queue q: Speichert Indizes von Knoten.
q.anfang(): Liefert erstes Element der Queue.
q.laenge(): Liefert Länge der Queue.
q.enqueue(): Fügt ein Element am Ende der Queue hinzu.
q.dequeue(): Entfernt erstes Element der Queue.

kante k: Speichert eine Kante.
k.knotenl(): Liefert den Index des linken Knotens von k.
k.knotenr(): Liefert den Index des rechten Knotens von k.
k.nachbarl(): Liefert den Index des linken Nachbarn von k.
k.nachbarr(): Liefert den Index des rechten Nachbarn von k.
k.spitze1(): Liefert den Index der ersten entdeckten Spitze zu k.
k.spitze2(): Liefert den Index der zweiten entdeckten Spitze zu k, falls k innere Kante ist (sonst *null*).
k.setzeKnotenl(): Speichert den Index des linken Knotens von k.
k.setzeKnotenr(): Speichert den Index des rechten Knotens von k.
k.setzeNachbarl(): Speichert den Index des linken Nachbarn von k.
k.setzeNachbarr(): Speichert den Index des rechten Nachbarn von k.
k.setzeSpitze1(): Speichert den Index der ersten entdeckten Spitze zu k.
k.setzeSpitze2(): Speichert den Index der zweiten entdeckten

Spitze zu k.

Matrizen:

[n][2] *koordinaten*$(i)(.)$: Speichert Koordinaten vom Knoten i.

[n][n] D: Distanzmatrix des metrischen Raumes.

[n][n] V: Distanzmatrix, die nur direkte Verbindungen enthält.

[n][n] WM: Speichert die Winkel an i im Dreieck $\triangle(i,j,k)$ an der Stelle $WM(j,k)$.

[n][n] V': Speichert noch nicht verbaute Kanten der Triangulation.

kante[n][n] AH: Enthält Kanten der äußeren Hülle als doppelt verkettete Liste.

kante[n][n] M: Speichert bereits verbaute Kanten der Triangulation.

Funktionen:

direkteVerbindungen(): Berechnet V.
output: -

*aussenkanten(**knoten** i)*: Speichert Zyklus der Außenkanten in AH, falls i auf der äußeren Hülle liegt.
output: boolean

eliminiereKanten([n][n] WM): Berechnet kürzeste Wege im Winkelgraphen und eliminiert überflüssige Kanten.
output: -

*schnitttest(**kante** (i,j), (k,l))*: Die Kanten (i,j) und (k,l) werden auf Schnitt gestestet.
output: boolean

*angrenzendesDreieck(**kante** (i,j))*: Liefert den Index der zu (i,j) gehörenden Spitze, bzw. *null*, falls diese nicht existiert.
output: knoten

*berechneKoordinaten(**kante** (i, j), **knoten** k)*: Berechnet die Koordinaten der Spitze k mit Hilfe der Koordinaten von i und j.
output: -

*ersetzeKante(**kante** (i, j), **knoten** k)*: Speichert die Kanten (i, k) und (j, k) in M und aktualisiert die Verkettungen der drei Knoten.
output: -

*innereTriangulation(**kante** (i, j))*: Trianguliert ein geschlossenes inneres Gebiet rekursiv.
output: -

Algorithmen:

Triangulation: Speichert die Triangulation in M,
wobei zu jeder Kante die eine bzw. die beiden Spitzen der Dreiecke gespeichert werden, in denen sie vorkommt.
output: -

Literaturverzeichnis

[1] Rolf Klein. Algorithmische Geometrie. Springer-Verlag, 2. überarbeitete Auflage, 2006.

[2] H.-P. Gumm und M. Sommer. Einführung in die Informatik. Oldenbourg-Verlag, 4. Auflage, 2000.

[3] Andreas Brandstädt. Graphen und Algorithmen. B. G. Teubner-Verlag Stuttgart, 1994.

[4] M.L. Fredman and D.E.Willard. Trans-dichotomous Algorithms for Minimum Spanning Trees and Shortest Paths. Journal of Computer and System Sciences 48, pages 533-551, 1994.

[5] Monique Laurent. Matrix completion problems. The Encyclopedia of Optimization, 3, pages 221-229, 2001.

[6] J.B. Saxe. Embeddability of weighted graphs in k-space is strongly NP-hard. Proceedings of the 17th Allerton Conference in Communications, Control and Computing, pages 480-489, 1979.

[7] H.-J. Bandelt. SIAM J.Disc. Math. 3(1), pages 1-6, 1990.

[8] H.-J. Bandelt and A. Dress. Split decomposition: a new and useful approach to phylogenetic analysis of distance data. Mol.Phylogen.Evolution 1, pages 242-252, 1992.

[9] H.-J. Bandelt and A. Dress. A canonical decomposition theory for metrics on a finite set. Adv. Math. 92, pages 47-105, 1992.

[10] Mark de Berg, Marc van Krefeld, Mark Overmars, Otfried Schwarzkopf. Computational Geometry: Algorithms and Applications, Theorem 9.1. Springer-Verlag, 2nd rev. edition, 2000.

[11] R. Klein, A. Grüne, C. Knauer, A. Lopez-Ortiz und R. Seidel. mündliche Kommunikation. Mai 2006.