# Einleitung

Die Arbeit basiert auf der Veröffentlichung *„Polygon Area Decomposition for Multiple-Robot Workspace Division“* von Susan Hert und Vladimir Lumelsky [<Literaturverweis>]. In dieser Veröffentlichung wird das Problem der Polygonzerlegung, das sogenannte „Problem der verankerten Flächenaufteilung“ (eng. *„anchored area partition problem“*) beschrieben und gelöst. Die Lösung erfolgt zunächst für konvexe Polygone und wird anschließend auf nicht konvexe, nicht einfache Polygone erweitert.

Die Polygonzerlegung ist eines der zentralen Probleme in der algorithmischen Geometrie und hat viele Anwendungsfälle, wie z.B. in der Kartographie, Bildverarbeitung oder in der Computergrafik. In vielen Fällen wird die Polygonzerlegung benötigt, um aus einem beliebigen Polygon eine Menge aus Teilpolygonen mit bestimmten Eigenschaften zu berechnen. Als Beispiel einer vielfach verwendeten Polygonzerlegung kann die Triangulation genannt werden, bei welcher ein gegebenes Polygon in eine Menge von Dreiecken zerlegt wird. Für die so berechnete Menge von Dreiecken stehen dann effiziente Algorithmen zur Lösung von Problemen zur Verfügung. Anschließend können die Lösungen der Teilpolygone zu einer Lösung für das Ausgangspolygon zusammengefasst werden.

Bei dem hier vorgestellten Problem der verankerten Flächenaufteilung ist die Anforderung an die resultierenden Teilpolygone nicht durch eine bestimmte Geometrie (z.B. ein Dreieck), sondern durch die Lage und Fläche der Teilpolygone gegeben. Bezüglich der Lage besteht die Anforderung darin, dass ein gegebener Punkt (*Standort* genannt) auf dem resultierenden Polygon liegen muss. Jeder Standort weist als Eigenschaft eine Flächenanforderung auf, welche durch die Größe des Teilpolygons erfüllt werden soll. Die Flächenanforderung kann je Standort den gleichen Wert aufweisen, kann aber auch unter den Standorten variieren. Das beschriebene Problem ist unter anderem durch die Flächenerkundung von Robotern motiviert:

Auf einem Polygons werden n Roboter Ri, i = 1, …, n, positioniert, welche die Aufgabe erhalten, zusammen die gesamte Fläche des Polygons zu erkunden. Hierzu muss jede Position innerhalb des Polygons von einem der n Roboter abgefahren werden. Um die Arbeit unter den Robotern aufzuteilen, ist es sinnvoll, jedem Roboter einen Polygonteil zuzuweisen, der von ihm bearbeitet werden muss. Die Teilpolygone sollen sich nicht überlappen, um ein mehrfaches Überfahren zu vermeiden. Bei der Flächenaufteilung muss berücksichtigt werden, dass der Startpunkt eines jeden Roboters auf dem Rand des zugewiesenen Teilpolygons liegt. Eine unterschiedliche Leistung der Roboter kann über die Flächenanforderung je Standort berücksichtigt werden.

Zur formalen Beschreibung des Problems sind als Eingangsdaten ein Polygon P sowie eine (nicht leere) Liste von Standorte S(P) gegeben. Für jeden der n Standorte Si ist der benötigte Flächenanteil ci mit 0 < ci < 1 gegeben, sodass gilt. Das Polygon P soll in n, nicht-überlappende Polygone zerlegt werden, sodass jeder Standort Si auf dem Rand eines Polygons Pi mit Fläche ci \* Area(P) liegt. Aus der Fläche des Polygons P kann für jeden der n Standorte die benötigte Fläche mit ci \* Area(P) bestimmt werden.

# Aufteilung eines einfachen, konvexen Polygons

## Grundidee

Bei der nachfolgend beschriebenen Lösung des Problems wird das konvexe Eingangs-Polygon CP mithilfe von Liniensegmenten schrittweise zerlegt. Jedes Liniensegment L ist hierbei vom Startpunkt Ls zum Endpunkt Le orientiert. Die bei jeder Teilung entstehen (zwei) Polygone erhalten entsprechend ihrer Lage zum Liniensegment L die Bezeichnungen für das *rechts* und für das *links* des Liniensegments liegenden Polygons.. An dieser Stelle soll genügen, dass die Liniensegmente so positioniert werden, dass die Fläche von der benötigten Fläche der auf dem Rand von liegenden Standorte entspricht ( analog). Die Zerlegung wird für jedes Polygon oder so oft wiederholt, bis je Polygon nur noch ein Standort auf dessen Rand liegt[[1]](#footnote-1).

|  |  |
| --- | --- |
|  |  |
|  |  |

Abbildung 1: Zerlegung eines konvexen Polygons CP in vier konvexe Polygone CP1 ... CP4

## Aufteilung eines einfachen, konvexen Polygons

Aus CP entstehende, konvexe Polygone, werden mit CPi notiert. Mit den genannten Überlegungen lässt sich ein rekursiver *divide-and-conquer* - Algorithmus zur Flächenaufteilung eines konvexen Polygons - basierend auf n Standorten - nun folgendermaßen skizzieren:

1 // Input: Convex polygon CP

2 Function ConvexDivide(CP)

3 if Length(S(CP)) == 1 then return CP end

4 // Here, the postion of L has to be calculated, see chapter xxx

5 PrL, PlL = cut(CP, L) // CP is cut into two pieces PrL and PlL

6 ConvexDivide(PrL) // recursive PrL

7 ConvexDivide(PlL) // recursive PlL

8 end

9 ConvexDivide(CP)

Bei jedem Aufruf von ConvexDivide(CP) wird zunächst geprüft, ob das übergebene Polygon nur noch einen Standortenthält . Falls ja, ist der Zielzustand für das übergebene Polygon erreicht und es ist keine weitere Flächenaufteilung erforderlich. Falls das Polygon mehrere Standorte enthält, erfolgt eine weitere Aufteilung des Polygons in zwei Teil-Polygone und , welche dann erneut mit ConvexDivide() aufgerufen werden.

Hierbei sei angemerkt, dass dieser Algorithmus stets terminiert, da die Anzahl an Standorten konstant ist und die Anzahl der Teil-Polygone je Schnitt um 1 erhöht wird. Nach n-1 Schnitten entspricht die Anzahl der Teil-Polygone der Anzahl der Standorte, siehe auch Kapitel xxx.

## Positionierung der Schnittlinie

Aus vorangegangenem Kapitel bleibt noch offen, wie genau die Aufteilung eines konvexen Polygons CP in die Polygone und erfolgt, sodass anschließend Area() == AreaRequired(S1 ... Si) und Area() == AreaRequired(Si+1 ... Sn) gilt. Konkret ist zu klären, wie Anfangs- und Endpunkt der Schnittlinien positioniert werden (siehe Zeile 4 oben).

Initialisierung von Ls und Le beim Aufruf von ConvexDivide():

* Der Startpunkt Ls der Linie L wird mit den Koordinaten des ersten Punkts der Liste W initialisiert, wobei dieser nach Definition ein Polygonpunkt (und kein Standort) ist. Es gilt daher w1 V.
* Der Endpunkt Le wird mit den Koordinaten des ersten Standorts in W initialisiert und mit wk notiert, wobei k der Index in W ist, bei welchem der erste Standort liegt. Da die Standorte nach ihrem Vorkommen auf dem Weg von v1 nach vn geordnet sind, ist bei einem konvexen Polygon sichergestellt, dass die Standorte S2 … Sn alle links der Linie L liegen.

Bei einer Zerlegung mit einer so initialisierten Linie würde S() == S1 und S() == S2 … Sn gelten, wobei S1 in einer Ecke von liegen würde. Je nach Fläche von und *AreaRequired*(S1) werden folgende Fälle unterschieden:

**Fall 1: Area() > AreaRequired(S1)**

Nach der Initialisierung der Linie L wird festgestellt, dass die Fläche von größer ist als die benötigte Fläche von S1. In diesem Fall erfolgt eine Verkleinerung von Area() unter Beibehaltung von S() == S1. Dies geschieht, indem Le als Drehpunkt fungiert und Ls inkrementell im Gegenuhrzeigersinn entlang des Polygons verschoben wird, bis Area() == AreaRequired(S1) gilt. Zur Verdeutlichung dieser Vorgehensweise sollen folgende Punkte nochmals herausgestellt werden:

* Durch die Initialisierung kann auf dem Weg von w1 zu wk kein weiterer Standort liegen, d.h. die *AreaRequired*(S()) ist konstant.
* Die Area() wird mit Verschiebung von Ls stetig kleiner. Bei Ls == S1 gilt Area() == 0.
* Le ist fest, d.h. S1 ist stets Teil von .

Wenn die Bedingung Area() == AreaRequired(S1) eintritt, erfolgt eine Polygonzerlegung. Für erfolgt keine weitere Zerlegung beim Aufruf von ConvexDivide() (siehe Zeile 3 oben), da nur S1 auf dessen Rand liegt. Falls auf dem Rand von mehr als 1 Standort verbleibt, erfolgt eine erneute Zerlegung beim Aufruf von ConvexDivide() (siehe Zeilen 4 und 5 oben).

**Fall 2: Area() < AreaRequired(S1)**

Nach der Initialisierung der Linie L wird festgestellt, dass die Fläche von kleiner ist als die benötigte Fläche von S1. In diesem Fall erfolgt eine Vergrößerung von Area() mit dem Ziel, die Anforderung von S1 zu erfüllen. Hierbei fungiert Ls als Drehpunkt und Le wird auf den nächsten in W vorkommenden Polygonpunkt oder Standort (wk+1) gesetzt. Die Anforderung wird erneut geprüft.[[2]](#footnote-2)

Hierbei kann es nun vorkommen, dass Le auf die Koordinaten eines Punktes wj in W gesetzt wird, welcher ein Standort S1 ist. Dieser Standort wird dann beim nächsten „Vorrücken“ (also bei wj+1) zur benötigten Fläche von hinzugenommen. Bei Fall 2 kann AreaRequired() demnach ansteigen, sodass ein „Weiterrücken“ von Le zwar zu einer größeren Area(), nicht aber unbedingt zu einem günstigeren Erfüllungsgrad aus Area() / AreaRequired(S()) führt.

Le wird so oft verschoben, bis eine der folgenden Bedingungen eintritt:

* Area() > AreaRequired(S())
* Le == Sn

Je nachdem, wie weit Le „vorrückt“ und wie die Fläche von PrL zur Flächenanforderung von S(PrL) ist, werden nun weiter zwei Fälle unterschieden:

Fall 2.1: Le Sn und Area() > AreaRequired(S()). In diesem Fall wird der Endpunkt Le inkrementell im Uhrzeigersinn entlang des Polygons bewegt, bis Area() == AreaRequired(S(PrL)) gilt.

Hinweis: Angenommen die Ausgangsposition von Le ist bei wj, dann muss es zwischen wj und wj-1 eine Position geben, bei der Area() == AreaRequired(S()) gilt, da beim „Vorrücken“ (s.o.) die Area() bei wj-1 zu klein und bei wj zu groß war. Dieser Zwischenpunkt kann auch durch Interpolation gefunden werden.

Fall 2.2: Le == Sn und Area() < AreaRequired(S()). In diesem Fall wird der Anfangspunkt Ls inkrementell im Uhrzeigersinn entlang des Polygons bewegt, bis Area() == AreaRequired(S()) gilt.

Diese Vorgehensweise entspricht im Wesentlichen Fall 1, wobei Ls nun nicht in Richtung *zum ersten* Standort, sondern *zum letzten* Standort bewegt wird.

|  |  |  |  |
| --- | --- | --- | --- |
| Fall | Initialisierung  Ls = w1, Le = s1 | Fall 1 und 2.1:  Area() > AreaRequired(S())  Fall 2.2: wk == sn (2.2) | Ergebnis  Area() == AreaRequired(S()) |
| 1 |  |  |  |
| 2.1 |  |  |  |
| 2.2 |  |  |  |

Abbildung 2: Fall 1, 2.1 und 2.2 inklusive der jeweiligen Zwischenschritte. Die unterschiedlichen Fälle werden durch angepasste Flächenanforderungen von S01/S02 hervorgerufen (Fall 1: 0.20/0.80, Fall 2.1: 0.70/0.30, Fall 2.2: 0.95/0.05)

Mit den Fällen 1 und 2 lässt sich der Algorithmus von ConvexDivide() nun wie folgt erweitern:

1 // Input: Convex polygon CP

2 Function ConvexDivide(CP)

3 if Length(S(CP)) == 1 then return CP end

4 Ls = W(1), Le = W(k) // k = index of first Site in W

5 PrL, PlL = cut(CP, L) // partitioning, returns PrL and PlL

6 while Area(PrL) < AreaRequired(S(PrL)) and Le != Sn do

7 if W(k-1) != S1 and W(k-1) in S then

8 S(PrL) = S(PrL) + W(k-1) // add previous Site to S(PrL)

9 end

10 k += 1

11 Le = W(k) // move Le to next point in W

12 PrL, PlL = cut(CP, L)

13 if Area(PrL) > AreaRequired(S(PrL)) and Le == S1 then

14 move Le CCW until Area(PrL) == AreaRequired(S(PrL))

15 else if Area(PrL) < AreaRequired(S(PrL))

16 if Le != Sn then

17 move/interpolate Le CW until Area(PrL) == AreaRequired(S(PrL))

18 else if Le == Sn then

19 move Ls CW until Area(PrL) == AreaRequired(S(PrL))

20 end

21 end

22 PrL, PlL = cut(CP, L) // CP is cut into two pieces PrL and PlL

23 ConvexDivide(PrL) // recursive PrL

24 ConvexDivide(PlL // recursive PlL

25 end

26 ConvexDivide(CP)

## Beispiel oder Anhang 1 (je nach Umfang der Arbeit)

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

Von oben links nach unten rechts zeilenweise:

1. Ausgangspolygon mit 8 Standorte und einem jeweils benötigten Flächenanteil von 12.5 %
2. Fall 2.1: Le wird bis V04 bewegt, sodass Area() > AreaRequired(S()) gilt. S() enthält dann die Standorte S01 … S03. Le wird anschließend von V04 in Richtung V03 inkrementell bewegt, bis Area() == AreaRequired(S()) gilt.
3. Fall 2.2: Le wird bis S03 verschoben (letzter Standort in S()). Ls wird anschließend CW inkrementell verschoben, bis Area() == AreaRequired(S()) gilt.
4. Fall 2.2 mit dem Hinweis: Area() kann bei Le == Sn zunächst eine Fläche 0 aufweisen. Dies stellt jedoch für den Algorithmus kein Hindernis dar.
5. Fall 1: Nach der Initialisierung mit Le = S1 gilt bereits Area() > AreaRequired(S(PrL)). Ls wird CCW verschoben, bis Area() == AreaRequired(S()) gilt.
6. Fall 1 mit dem Hinweis, dass bei der inkrementellen Verschiebung von Ls (bzw. im Fall 2.2 gilt dies analog für Le) diese auch „um die Ecke“ möglich sein muss (Ls wurde bei V01 initialisiert).
7. Fall 1: Verschiebung von Ls CCW bis Area() == AreaRequired(S()) gilt.
8. Fall 1: Fall 1: Verschiebung von Ls CCW bis Area() == AreaRequired(S()) gilt.
9. Die resultierende Flächenaufteilung aus den Schritten 2-8 mit Teilflächen zu je 12.5 %

# Verallgemeinerung: Aufteilung eines nicht einfachen, nicht konvexen Polygons (SJ)

Es wurde gezeigt, dass ein einfaches, konvexes Polygon in polynomieller Zeit rekursiv in n 1-Standort-Polygone aufgeteilt werden kann. Dieses Kapitel dient dazu einen verallgemeinerten Algorithmus zu skizzieren, damit auch für nicht einfache, nicht konvexe Polygone (Abbildung XX) das Problem der verankerten Flächenaufteilung gelöst werden kann.

Bevor dieser Algorithmus vorgestellt wird, soll die Beschreibung der dahinterliegenden Grundidee einen Überblick über die Vorgehensweise verschaffen. Danach werden die vorbereitenden Schritte vorgestellt und die Aufteilung des Polygons erläutert. Ein Beispiel dient anschließend zur Veranschaulichung des vorgestellten Algorithmus und zum Schluss des Kapitels wird der Sonderfall geschildert, dass Standorte im Inneren des Polygons liegen. Abschließend wird die Komplexität der Vorgehensweise analysiert.

## Grundidee

In Kapitel XX wurde bereits erläutert, wie ein einfaches, konvexes Polygon aufgeteilt werden kann. Dieses Vorgehen kann auch bei der Aufteilung komplexerer Polygone verwendet werden, muss jedoch in einigen Punkten erweitert werden.

Als Voraussetzung wird angenommen, dass ein nicht einfaches, nicht konvexes Polygon P bereits in konvexe Teilpolygone CP1,…,CPp zerlegt wurde. Im ersten Schritt werden die Teilpolygone mithilfe einer Tiefensuche neu geordnet, um eine feste Bearbeitungsfolge für das weitere Vorgehen zu erhalten. Anschließend werden die Teilpolygone rekursiv aufgeteilt, wie es bereits in Kapitel XX gezeigt wurde. Allerdings können nun Sonderfälle auftreten, die bei der Zerlegung eines einfachen, konvexen Polygons nicht vorkommen können. Einerseits kann CPi weniger Fläche ausfüllt, als durch *AreaRequired(S(CPi))* gefordert ist. In diesem Fall ist CPi flächenunvollständig und muss Flächen von anderen Teilpolygonen übernehmen. Andererseits kann es sein, dass einzelne Teilpolygone keinen Standort enthalten oder weniger Fläche ausfüllen, als durch *AreaRequired(S(CPi))* gefordert ist. In diesem Fall ist CPi standortunvollständig und andere Teilpolygone müssen Fläche von CPi übernehmen.

Die Neuordnung wird innerhalb der Prozedur *OrderPieces* umgesetzt und die Aufteilung inklusive der Sonderfallbehandlung wird durch die beiden Methoden *NonconvexDivide* und *DetachAndAssign* umgesetzt, die sich gegenseitig rekursiv aufrufen, bis ein n-Standort Polygon in n 1-Standort Polygone aufgeteilt wurde.

## Aufteilung in konvexe Teilpolygone

Als Voraussetzung für die gleichmäßige Aufteilung eines nicht einfachen, nicht konvexen Polygons wird angenommen, dass das Polygon bereits in konvexe Teilpolygone aufgeteilt wurde. In verschiedenen Werken [6, 15, 17, 19,21,31] werden Möglichkeiten einer solchen Aufteilung vorgestellt. Ein Vorgehen wäre zum Beispiel, eine Triangulation eines Polygons zu erzeugen. In diesem Fall würde jedoch eine hohe Anzahl von Teilpolygonen entstehen. Um Teilpolygone zusammenzufassen, können nacheinander Kanten der Triangulation entfernt werden, solange das dadurch entstehende Teilpolygon weiterhin konvex ist.

Hieraus wird ersichtlich, dass es verschiedene Möglichkeiten gibt, ein Polygon in konvexe Teilpolygone aufzuteilen. Zum Schluss dieser Arbeit wird besprochen, welche Auswirkungen diese vorbereitenden Schritte auf den Verlauf des vorgestellten Algorithmus haben können.

1 // Input: Nj - Node of the connectivity Graph

2 Function OderPieces(Nj)

3 if Nj has not beeing marked then

4 if Nj is a leaf node then

5 Mark(Nj)

6 output(CPj)

7 for each Nk in Neighbors(Nj)

8 OrderPieces(Nk)

9 else

10 Mark(Nj)

11 for each Nk in Neighbors(Nj)

12 OrderPieces(Nk)

13 output(CPj)

14 end OrderPieces(Nj)

## Ordnung der Teilpolygone

Es kann nun davon ausgegangen werden, dass das Polygon P bereits in konvexe Teilpolygone CP1,…,CPp zerlegt wurde. Die Indizes der Teilpolygone können willkürlich geordnet sein und geben keine Aussage über die tatsächliche Anordnung im Polygon P. Aus diesem Grund werden die Teilpolygone zuerst neu geordnet, was für den anschließend dargestellten Algorithmus erforderlich ist. Dazu wird ein Verbindungsgraph G gebildet und anhand dessen mittels einer Tiefensuche eine Ordnung erzeugt. Jedes Teilpolygon CPi wird durch einen Knoten Ni in G abgebildet. Für jeden Nachbarn CPk (i ungleich k) des Teilpolygons CPi wird eine Kante zum jeweils korrespondierenden Knoten Nk eingefügt.

Wir definieren einen Knoten Ni in G als Blatt, wenn Ni entweder nur einen Nachbar hat oder alle Nachbarn von Ni als besucht markiert wurden.

Die Prozedur *OrderPieces* beschreibt nun die Neuordnung der Teilpolygone. *OrderPieces* wird mit einem beliebigen Knoten Ni von G initial aufgerufen. Zuerst wird geprüft, ob Ni bereits markiert wurde. Ist dies der Fall, kann der Aufruf zurückkehren. Falls Ni noch nicht markiert wurde, wird geprüft, ob Ni nach obiger Definition ein Blatt ist. Falls Ni kein Blatt ist, dann wird der Knoten markiert und für alle Nachbarn Nk von Ni rekursiv *OrderPieces* aufgerufen. Nach dem Rücksprung der Aufrufe aller Nachbarn von CPi wird CPi ausgegeben. Falls Ni ein Blatt ist, dann wird Ni markiert und CPi ausgegeben. Anschließend wird für alle Nachbarn Nk von Ni rekursiv *OrderPieces* aufgerufen.

Die neue Ordnung der CPi ist nun die Reihenfolge, in der die Teilpolygone ausgegeben wurden. Diese neu entstandene Ordnung wird in den nächsten Abschnitten genutzt und die Teilpolygone anhand von PlL, NextNeighbor(CP) und PredPoly(CP), wie sie in Kapitel 6 beschrieben wurden, zu klassifizieren.

1 // Input: Convex polygon CP

2 Function NonConvexDivide(CP)

3 Ls = W(1), Le = W(k) // k = index of first Site CCW from w1 in W

4 PrL, PlL = cut(CP, L) // partitioning, returns PrL and PlL

5 while Area(PrL) < AreaRequired(S(PrL)) and Le != wm do

6 if W(k-1) != S1 and W(k-1) in S then

7 S(CPrL) = S(CPrL) + W(k-1) // add previous Site to S(CPrL)

8 end

9 k += 1

10 Le = W(k) // move Le to next point in W

11 PrL, PlL = cut(CP, L)

12 if Area(PrL) > AreaRequired(S(CPrL)) then

13 if Le == Si then

14 k1 = 1

15 while Area(PrL) > AreaRequired(S(PrL)) do

16 k1 = k1 + 1

17 Ls = w(k1)

18 L1 = (w(k1), Le)

19 T(t1,t2,t3) = (w(k1), w(k1-1), Le)

20 else

21 L1 = (Ls, w(k-1))

22 T(t1,t2,t3) = (w(k-1), w(k1), Ls)

23 end

24

25 if Area(PrL1 + T) > AreaRequired(S(CPrL)) then

26 interpolate point t on (t1,t2) until Area(PrL + T) == AreaRequired(S(CPrL))

27 T(t1,t2,t3) = (t1, t, t3)

28 DetachAndAssign(PrL1 + T)

29 DetachAndAssign(PlL1 – T – PredPoly(CP,(t1,t))

30 else if Area(PrL1 + PredPoly(CP,(t1,t2)) < AreaRequired(S(CPrL)) then

31 interpolate point t on (t1,t2) until Area(PrL + T) == AreaRequired(S(CPrL))

32 T(t1,t2,t3) = (t1, t, t3)

33 DetachAndAssign(PrL1 + T)

34 DetachAndAssign(PlL1 – T – PredPoly(CP,(t1,t))

35 else

36 PS = interiorPoint(t1,t2) //PS is new Pseudosite

37 T(t1,t2,t3) = (t1,PS,t3)

38 AreaRequired(PS) = AreaRequired(S(CPrL) – Area(PrL1 + T)

39 Order(W(PredPoly(CP,(t1,t2)) //such that w1 = PS if Le != Si and wm = PS if Le == Si

40 DetachAndAssign(PredPoly(CP,(t1,t2))

41 DetachAndAssign(PrL1 + T)

42 DetachAndAssign(PlL1 + T)

43 end

44 else

45 t = interiorPoint(wm, w1)

46 L1 = (t,Si)

47 DetachAndAssign(PrL1)

48 DetachAndAssign(PlL1)

49 end

50 end NonConvexDivide()

## Aufteilung eines nicht einfachen, nicht konvexen Polygons

Für die Aufteilung wird jedes Teilpolygon CP1,…,CPp betrachtet. Konkret wird das Polygon PredPoly(CPi) so aufgeteilt, dass ein Teilstück einem Standort in CPi (falls vorhanden) zugeordnet wird und der Rest dem Polygon PredPoly(CPk) mit k > i angehangen wird. Diese Aufteilung wird durch die sich gegenseitig rekursiv aufrufenden Prozeduren *NonconvexDivide* und *DetachAndAssign* erreicht. Erstere erzeugt ein Liniensegment, welches PredPoly(CPi) in zwei Teile aufteilt und letztere ordnet die Teile entweder einem Standort zu oder teilt sie erneut auf.

Zuerst wird die Prozedur *NonconvexDivide* beschrieben, die die Teilpolygone in zwei Teile aufteilt. Listing XX beschreibt diese Prozedur.

Als Eingabe dient ein konvexes Teilpolygon, beschrieben durch die Liste W(CPi) (= wk, k = 1, … m) mit allen Polygonpunkten inklusive Steiner-Punkten und die Liste S(CPi) mit den Standorten des Teilpolygons inklusive der jeweils benötigten Fläche. Anders als *ConvexDivide* aus Kapitel 7 ist die Reihenfolge der Polygonpunkte in W(CPi) für die Bearbeitung relevant. Die Kante, die durch die Polygonpunkte (wm, w1) erzeugt wird, sei nun die Kante zu NextNeighbor(CPi). Hat CPi keinen nächsten Nachbarn, muss wm gleich einem Standort sein.

Wie es auch schon bei *ConvexDivide* der Fall war, lässt die Prozedur erneut ein Liniensegment L gegen den Uhrzeigersinn durch das Polygon CPi wandern, wobei Ls als Drehpunkt dient. L wird durch (Ls, Le) = (w1, Si) initialisiert, wobei Si der erste Standort aus S(CPi) ist.

Nun können zwei Fälle eintreten, in denen die Schleife stoppt:

1. Die Fläche rechts der Linie ist größer oder gleich der benötigten Fläche der Standorte, die sich in diesem Gebiet befinden. Es gilt:  
   Area(PrL) >= AreaRequired(S(CPrL))
2. Das Ende des Polygons wird erreicht, also Le = wm.

Durch die Bearbeitung von vorherigen Teilpolygonen kann es sein, dass nicht zugewiesene Teile dieser Polygone in die Aufteilung von CPi miteinbezogen werden müssen. Außerdem kann nun der Fall eintreten, dass Die Fläche des Teilpolygons kleiner ist, als AreaRequired(S(CPi)). Aus diesem Grund müssen die oberen beiden Fälle noch feingranularer aufgeteilt werden.

**Fall 1:** Wie auch in der Prozedur *ConvexDivide* wird ein Ende der Linie L entlang des Polygons bewegt, um die Fläche Area(PrL) zu verkleinern. Hierbei unterscheiden wir zwei Fälle. Falls Le = Si für einen beliebigen Wert für i gilt, dann wird der Startpunkt Ls im Gegenuhrzeigersinn bewegt, ansonst wird der Endpunkt Le im Uhrzeigersinn bewegt.

Nun seien L1 und L2 zwei Liniensegmente, die einen gemeinsamen, festen Endpunkt haben. Dieser gemeinsame Endpunkt ist für beide Linien entweder Ls oder Le und damit das Gegenstück zum oben bestimmten Punkt, welcher entlang des Polygons bewegt wurde. Die Linien sind so positioniert, dass Area(PrL1) < AreaRequired(S(CPrL1)) und Area(PrL2) > AreaRequired(S(CPrL2)) gilt. Die Linie L2 ist demnach durch (w1, wk) und die Linie L1 durch (w1, wk-1) beschrieben.

Dadurch entsteht ein Dreieck T = (t1, t2, t3), dass die Differenz zwischen CPrL1 und CPrL2 bildet. Außerdem sei (t1, t2) das Liniensegment von CPi, das L1 und L2 verbindet. Der gemeinsame Endpunkt von L1 und L2 ist demnach t3.

Nun müssen CPrL1 mit Teilen des Dreiecks T und Teilen der Reste der Vorgängerpolygone verbunden werden, damit die Flächenanforderungen der Standorte in CPrL1 erfüllt ist. Dabei entstehen 3 Fälle:

* 1. Area(PrL1 + T) > AreaRequired(S(CPrL))  
     Die Flächenanforderung der Standorte kann durch die Fläche rechts von L1 und T vollständig gedeckt werden. Insbesondere wird kein Flächenanteil von PredPoly(CP, (t1, t2)) benötigt.
  2. Area(PrL1 + T) <= AreaRequired(S(CPrL)) und  
     Area(PrL1 + PredPoly(CP, (t1, t2)) < AreaRequired(S(CPrL)) (\*)  
     Die Flächenanforderung der Standorte kann genau oder nicht komplett durch die Fläche rechts von L1 und T komplett abgedeckt werden, jedoch nicht durch die Fläche rechts von L1 und die Reste der Vorgängerpolygone.
  3. Area(PrL1 + T) <= AreaRequired(S(CPrL)) und  
     Area(PrL1 + PredPoly(CP, (t1, t2)) >= AreaRequired(S(CPrL))  
     Die Flächenanforderung der Standorte kann genau oder nicht komplett durch die Fläche rechts von L1 und T komplett abgedeckt werden, jedoch durch die Fläche rechts von L1 und die Reste der Vorgängerpolygone.

Die je nach Fall entstehenden Polygone werden anschließend an die Prozedur *DetachAndAssign* übergeben und dort entweder Standorten zugewiesen oder durch einen rekursiven Aufruf von *NonConvexDivide* erneut aufgeteilt.

**Fall 1.1:** Die Flächenanforderung von S(CPrL) kann durch das Polygon PrL1 zusammen mit dem Dreieck T erfüllt werden. In diesem Fall reicht es aus, mittels linearer Interpolation einen Punkt t zwischen t1 und t2 zu finden, sodass für das Dreieck T‘ = (t1, t, t3) gilt:

Area(PrL1 + T‘ - PredPoly(CP, (t1, t)) = AreaRequired(S(CPrL))

Es müssen also keine Reste der Vorgängerpolygone genutzt werden, die links von L1 liegen. Durch diese Aufteilung entstehen die beiden Polygone (PrL1 + T‘ - PredPoly(CP, (t1, t))) und (PlL1 – T‘), die der Prozedur *DetachAndAssign* übergeben werden.

**Fall 1.2:** Damit die Flächenanforderung erfüllt werden kann, wird zunächst das Vorgängerpolygon PredPoly(CP, (t1, t2)) hinzugenommen und dieses um die Flächen PrL1 und einem Teile des Dreiecks T erweitert. Erneut wird durch lineare Interpolation der Punkt t gefunden und wie oben das Dreieck T‘ gebildet, sodass die Flächenanforderung erfüllt ist. Das Dreieck T‘ kann durch die strikte Ungleichung (\*) nicht kollabieren. Somit entstehen die beiden Polygone (PrL1 + T‘) und (PlL1 – T‘ – PredPoly(CP, (t1, t))), die der Prozedur *DetachAndAssign* übergeben werden.

**Fall 1.3:** Die Restfläche der Vorgängerpolygone ist größer als die Flächenanforderung der Standorte. In diesem Fall muss die Fläche der Vorgängerpolygone wiederrum aufgeteilt werden. Ein Teil wird dann S(CPrL1) zugeordnet, der andere Teil bleibt unbehandelt. Um dies zu erreichen, wird auf der Kante von CP und zwischen t1 und t2 willkürlich ein Pseudostandort PS hinzugefügt. Dadurch entsteht das Dreieck T‘ = (t1, PS, t3). Es gilt:

AreaRequired(PS) = AreaRequired(S(CPrL)) – Area(PrL1 + T‘)

PS bekommt also die fehlende Flächenanforderung zugewiesen. Dadurch kann PredPoly(CP, (t1, PS)) ebenfalls durch *NonconvexDivide* aufgeteilt und ein Teilpolygon PS zugewiesen werden. Das zugewiesene Teilpolygon kann dann dem Polygon (PrL1 + T‘) hinzugefügt werden. Dieses Polygon und das Polygon (PlL1 – T‘) werden dann an *DetachAndAssign* übergeben und weiter aufgeteilt.

Weiterhin betrachtet werden, muss der Fall, dass das Liniensegment L das Polygon einmal komplett durchlaufen hat.

**Fall 2:** Dieser Fall tritt ein, wenn ein Teilpolygon und die Reste der Vorgängerpolygone weniger Fläche enthält, als die Standorte beanspruchen. In diesem Fall ist CPi Flächen-unvollständig und es muss mindestens ein Pseudostandort erzeugt werden, für mindestens einen Standort aus S(CPrL). Dazu wirkt ein Punkt t auf der Kante (wm, w1), also der Kante zu NextNeighbor(CPi), erzeugt. Nun sei L = (t, Si) der erste Standort aus S(CPi), wenn man CP gegen den Uhrzeigersinn von w1 aus durchläuft. Nun wird W(PrL) so geordnet, dass t = w1 ist. Dann werden PrL und PlL an *DetachAndAssign* übergeben und aufgeteilt. Hierbei entsteht auf der Seite (w1, w2) entweder ein Pseudostandort oder ein Teil von PrL wird dem Standort Si zugeordnet. Wenn ein Pseudostandort entsteht, dann wird diesem die Flächenanforderung von Si abzüglich der Fläche von PrL zugeordnet und das Polygon PrL von CPi entfernt. Mit den restlichen Standorten von CPi wird dasselbe Verfahren angewandt.

Die Pseudostandorte werden nun bei der Aufteilung von NextNeighbor(CPi) behandelt. Wenn den Pseudostandorten bei dieser Bearbeitung ein Polygon zugeteilt wird, dann wird dieses Polygon auf die korrespondierenden Standorte übertragen.

Durch *NonconvexDivide* wird ein q-Standort Polygon entweder in ein q1-Standort Polygon und ein q2-Standort Polygon aufgeteilt mit q1, q2 > 0 und q1 + q2 = q oder es wird ein 1-Standort Polygon abgetrennt und einem Standort zugeteilt und es bleibt ein q‘-Standort Polygon mit kleinerer Fläche und mit q = q‘ + 1 über.

1 // Input: Poly(CP) - Polygon rooted at convex piece CP

2 Function DetachAndAssign(Poly(CP))

3 if Length(S(CP)) == 0 then return

4 if PredPoly(CP) is AreaComplete then

5 if S(CP) == {Si} then //for some i

6 Assign PredPoly(CP) to Si

7 Detach PredPoly(CP) from Poly(CP)

8 else

9 Detach PredPoly(CP) from Poly(CP)

10 Order(W(CP)) //such that wm = Si for some i

11 NonConvexeDivide(CP)

12 end

13 else if PredPoly(CP) is areaIncomplete then

14 if S(CP) == {Si} then //for some i

15 Assign PredPoly(CP) to Si

16 Detach PredPoly(CP) from Poly(CP)

17 PS = interiorPoint(w(j),w(k)) //with (w(j),w(k)) is edge to NextNeighbor(CP)

18 else

19 Order(W(CP)) //such that edge (w(m),w(1)) is edge to NextNeiggbor(CP)

20 NonConvexeDivide(CP)

21 end

22 else

23 Order(W(CP)) //such that edge (w(m),w(1)) is edge to NextNeiggbor(CP)

24 NonConvexeDivide(CP)

25 end DetachAndAssign()

Die Prozedur *DetachAndAssign* teilt ein Teilpolygon einem Standort zu oder teilt ein Teilpolygon erneut mittels *NonconvexDivide* auf. *DetachAndAssign* ist durch Listing XX beschrieben.

*DetachAndAssign* wird nun das Polygon Poly(CP) übergeben, das, wie oben beschrieben, mittels *NonconvexDivide* ermittelt wurde. Nun können 3 Fälle auftreten.

1. PredPoly(CP) ist Flächen-vollständig
2. PredPoly(CP) ist Flächen-unvollständig
3. PredPoly(CP) ist Standort-unvollständig

Im ersten Fall kann es nun sein, dass PredPoly(CPi) lediglich einen Standort besitzt. Dann kann PredPoly(CPi) komplett diesem Standort zugeteilt (*Assign*) werden und vom Polygon Poly(CP) getrennt (*Detach*) werden.

Falls PredPoly(CP) mehrere Standorte enthält, dann wird PredPoly(CP) von Poly(CP) getrennt und PredPoly(CP) wird rekursiv mittels *NonconvexDivide* aufgeteilt.

Im zweiten Fall treten ebenfalls die beiden Unterfälle auf. Falls PredPoly(CP) lediglich einen Standort Si hat, dann kann PredPoly(CP) dem Standort zugeteilt werden und danach von Poly(CP) getrennt werden. Da PredPoly(CP) Flächen-unvollständig war, muss nun ein Pseudostandort auf der Kante zum NextNeighbor(CP) erzeugt werden, der die restliche Flächenanforderung von Si enthält. Wenn Flächen später dem Pseudostandort zugeteilt werden, dann werden diese stattdessen Si zugeteilt.

Falls PredPoly(CP) mehrere Standorte hat, dann wird PredPoly(CP) zuerst neu geordnet, sodass (wm, w1) die Kante zu NextNeighbor(CP) ist und dann rekursiv mittels *NonconvexDivide* aufgeteilt, da nicht klar ist, welcher Standort für die Flächen-unvollständigkeit verantwortlich ist.

Im dritten Fall hat PredPoly(CP) mehr Fläche, als die Standorte von CP benötigen. In diesem Fall wird PredPoly(CP) ebenfalls neu geordnet, sodass (wm, w1) die Kante zu NextNeighbor(CP) ist und dann rekursiv mittels *NonconvexDivide* aufgeteilt.

## Behandlung innen liegender Standorte

Falls Standorte innerhalb von P liegen, wird P so in konvexe Teilpolygone aufgeteilt, dass die Standorte auf Kanten der konvexen Aufteilung liegen. Ist dies nicht direkt möglich, können für die Standorte auch weitere Kanten eingefügt werden und die Aufteilung in konvexe Teilpolygone wird etwas feingranularer. Für den korrekten Ablauf des Algorithmus spielt diese Art der Einteilung keine Rolle.

## Beispiel

Das Beispiel aus Abbildung XX ist aus dem Artikel von XX übernommen.

Abbildung 17 zeigt verschiedene Stadien der gleichmäßigen Aufteilung eines nicht konvexen Polygons mit 12 Ecken und sieben Standorten. Polygon *(a)* zeigt die initiale Aufteilung des Polygons in 5 konvexe Teilpolygone CP1, …, CP5. In den darauffolgenden Polygonen *(b) – (f)* werden die Teilpolygone, die bereits einem Standort zugeteilt sind dunkelblau markiert. Die Teilpolygone, die einem Standort zugeteilt sind, aber noch nicht ausreichend viel Fläche beinhalten, sind hellblau markiert.

In Polygon *(b)* wurde das Teilpolygon CP1 bearbeitet. Dabei wurde es in zwei Teilpolygone aufgeteilt. P3 wurde dem Standort S3 zugeordnet und P4 wurde dem Standort S4 zugeordnet. P4 erfüllt die Flächenanforderung von S4 nicht vollständig, weshalb ein Pseudostandort S‘4 an der Kante zu NextNeighbor(CP1) erzeugt wurde. Dieser Pseudostandort wird zu einem späteren Zeitpunkt bearbeitet.

Polygon *(c)* zeigt den Stand nach der Bearbeitung von Teilpolygon CP2. Hier tritt erneut der Fall auf, dass die Fläche, die dem Standort S2 zugeteilt wird, zu klein ist. Aus diesem Grund wird der Pseudostandort S‘2 an der Kante zu NextNeighbor(CP2) erzeugt.

Als nächstes wird in Polygon *(d)* der Stand nach der Bearbeitung von Teilpolygon CP3 gezeigt. Dort wurde das Teilpolygon P7 dem Standort S7 zugeteilt und das Teilpolygon P1 dem Standort S1. Der Rest von CP3 wurde mit P2 vereint und daher ebenfalls S2 zugewiesen. Da die Fläche von P2 weiterhin nicht groß genug ist um die Flächenanforderung von S2 zu genügen, wird ein neuer Pseudostandort S‘2 an der Kante zu NextNeighbor(CP3) erzeugt.

Polygon *(e)* zeigt Stand nach der Bearbeitung von CP4. Dort hat die Sweepline das Ende von CP4 erreicht, ohne an einer Stelle genug Fläche zu beinhalten für die enthaltenen Standorte. Aus diesem Grund wird P5 und der Pseudostandort S‘5 erzeugt und diese S5 zugeordnet. Anschließend wird derselbe Schritt für S6 wiederholt. Nach abziehen der Fläche von P5 und P6 von CP4 werden Teile von CP4 den Pseudostandorten S‘2 und S‘4 zugeordnet und mit den Teilpolygonen P2 und P4 vereint.

Im letzten Polygon (Polygon *(f)*) wurde CP5 und der verbliebene Teil von CP4 den Pseudostandorten S‘5 und S‘6 zugeordnet. Damit wurde das Polygon vollständig in gleichmäßige Teilpolygone P1, …, P7 aufgeteilt.

# Komplexitätsanalyse

**Konvexes Polygon:**

Der Algorithmus *ConvexDivide()* benötigt lineare Zeit bezogen auf die Anzahl der Elemente der Liste W(P), um einen einzelnen Schnitt durchzuführen. Der dabei erforderliche Aufbau der Polygone PrL bzw. PlL sowie die Ermittlung der Fläche ist in konstanter Zeit möglich. Das Finden der Punkte, bei denen Area(PrL) == AreaRequired(S(PrL)) gilt, kann über Interpolation ebenso in konstanter Zeit erfolgen. Der Algorithmus ConvexDivide() benötigt daher O(n+v) Zeit (n = Anzahl an Sites, v = Anzahl an Polygonpunkten).

Im worst-case trennt *ConvexDivide()* von einem konvexen Polygon mit q Standorten nur ein Dreieck (v = 3, n = 1) ab. Neben dem Dreieck verbleibt dann ein q-1 konvexes Polygon mit v+1 Polygonpunkten. Für dieses Polygon gilt wiederrum selbes. Um eine gesamte Flächenzerlegung eines konvexen Polygons zu berechnen, wird O((n-1)(n+v)) Zeit benötigt.

**Nicht konvexes Polygon:**

Der Algorithmus *OrderPieces()* besucht jeden Knoten im Nachbarschaftsgraphen maximal zwei mal, sodass die Herstellung der Ordnung in linearer Zeit bezogen auf die Anzahl der konvexenTeile hergestellt werden kann.

Der Algorithmus NonconvexDivide() benötigt O(pn2 + nv) Zeit, um alle konvexen Teile (p) unter den Standorten (n) aufzuteilen. Im worst case wird jedes der p konvexen Teile in n Polygone zerlegt, wobei jedes der Polygone einen Teil eines Standorts abbildet.

1. Da aus einer Zerlegung eines konvexen Polygons mit einem Liniensegment immer zwei *konvexe* Polygone und insbesondere kein nicht-konvexes Polygon resultiert, ist dieser Ansatz möglich. [↑](#footnote-ref-1)
2. In diesem Schritt wird Le von Polygonpunkt zu Polygonpunkt verschoben. Eine inkrementelle Verschiebung entlang des Polygons erfolgt dann unter Fall 2.1 bzw. 2.2. [↑](#footnote-ref-2)