# 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 ein neues Problem der Polygonzerlegung, das sog. „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 mit Löchern erweitert. Nachstehend wird diese Veröffentlichung vorgestellt.

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 einfacheren 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 einen bestimmten Geometrietyp (z.B. ein Dreieck), sondern durch die Lage und Fläche der Teilpolygone gegeben. Bzgl. der Lage besteht die Anforderung darin, dass ein gegebener Punkt („Standort“ genannt) auf dessen Rand 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 maßgebend durch die Flächenerkundung von Robotern motiviert:

*Auf dem Rand eines 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 „seinen“ Polygonteil zuzuweisen, der von ihm bearbeitet werden muss. Die Teilpolygone sollen sich nicht überlappen, um ein ineffizientes, 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), die auf dem Rand von P liegen, 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 \* Fläche(P) liegt. Aus der Fläche des Polygons P kann für jeden der n Standorte die benötigte Fläche mit ci \* Fläche(P) bestimmt werden.

# Notation

xxx

# 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. Die exakte Positionierung der Liniensegmente wird in Kapitel xxx erläutert. 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

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() wird zunächst geprüft, ob das übergebene Polygon nur noch einen Standort auf dem Rand besitzt. Falls ja, ist der Zielzustand für diesen Standort / dieses Polygon erreicht und es ist keine weitere Flächenaufteilung erforderlich. Das Polygon wird mit der return-Anweisung zurückgegeben. Falls nein, erfolgt eine weitere Aufteilung des Polygons in wiederrum 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 Fläche() == BenötigteFläche(S1 ... Si) und Fläche() == BenötigteFläche(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() == Si+1 … Sn gelten, wobei S1 in einer Ecke von liegen würde. Je nach Fläche von und *BenötigteFläche*(S1) werden folgende Fälle unterschieden:

**Fall 1: Fläche() > BenötigteFläche(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 Fläche() unter Beibehaltung von S() == S1. Dies geschieht, indem Le als Drehpunkt fungiert und Ls inkrementell im Gegenuhrzeigersinn entlang des Polygons verschoben wird, bis Fläche() == BenötigteFläche(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 *BenötigteFläche*(S()) ist konstant.
* Die Fläche() wird mit Verschiebung von Ls stetig kleiner. Bei Ls == S1 gilt Fläche() == 0.
* Le ist konstant, d.h. S1 ist stets Teil von .

Wenn die Bedingung Fläche() == BenötigteFläche(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: Fläche() < BenötigteFläche(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 Fläche() 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 BenötigeFläche() demnach ansteigen, sodass ein „Weiterrücken“ von Le zwar zu einer größeren Fläche(), nicht aber unbedingt zu einem günstigeren Erfüllungsgrad aus Fläche() / BenötigteFläche(S()) führt.

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

* Fläche() > BenötigteFläche(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 && Fläche() > BenötigteFläche(S()). In diesem Fall wird der Endpunkt Le inkrementell im Uhrzeigersinn entlang des Polygons bewegt, bis Fläche() == BenötigteFläche(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 Fläche() == BenötigteFläche(S()) gilt, da beim „Vorrücken“ (s.o.) die Fläche() bei wj-1 zu klein und bei wj zu groß war. Dieser Zwischenpunkt kann auch durch Interpolation gefunden werden.

Fall 2.2: Le == Sn && Fläche() < BenötigteFläche(S()). In diesem Fall wird der Anfangspunkt Ls inkrementell im Uhrzeigersinn entlang des Polygons bewegt, bis Fläche() == BenötigteFläche(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:  Fläche() > BenötigteFläche(S())  Fall 2.2: wk == sn (2.2) | Ergebnis  Fläche() == BenötigteFläche(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 Fläche() > BenötigteFläche(S()) gilt. S() enthält dann die Standorte S01 … S03. Le wird anschließend von V04 in Richtung V03 inkrementell bewegt, bis Fläche() == BenötigteFläche(S()) gilt.
3. Fall 2.2: Le wird bis S03 verschoben (letzter Standort in S()). Ls wird anschließend CW inkrementell verschoben, bis Fläche() == BenötigteFläche(S()) gilt.
4. Fall 2.2 mit dem Hinweis: Fläche() 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 Fläche() > BenötigteFläche(S(PrL)). Ls wird CCW verschoben, bis Fläche() == BenötigteFläche(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 Fläche() == BenötigteFläche(S()) gilt.
8. Fall 1: Fall 1: Verschiebung von Ls CCW bis Fläche() == BenötigteFläche(S()) gilt.
9. Die resultierende Flächenaufteilung aus den Schritten 2-8 mit Teilflächen zu je 12.5 %

# Komplexitätsanalyse

**Konvexes Polygon:**

Der Algorithmus *ConvexDivide()* benötigt lineare Zeit bezogen auf die Anzahl der Elemente der Liste W(P), um einen einzelnen Schnitt durchzuführen. Beim Aufbau der Polygone PrL bzw. PlL muss hierbei im worst-case jeder Polygonpunkt besucht werden, wobei das Hinzufügen und Entfernen von Polygonpunkten sowie die Ermittlung der Fläche in konstanter Zeit durchgeführt werden können. Das Finden der Punkte, bei denen Fläche(PrL) == BenötigteFläche(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)