Ein optimaler paralleler Algorithmus zur Berechnung eines minimalen konvexen separierenden Polygons

Diplomarbeit Universität Dortmund

1993

Autor: Andreas Birkendorf
Betreuer: Prof. Dr. Hans-Ulrich Simon
Lehrstuhl Informatik II

Zusammenfassung

In dieser Diplomarbeit wird ein paralleler Algorithmus vorgestellt, der ein minimales konvexes Polygon findet, das zwei endlich große Punktmengen der Ebene separiert.

Auf dem Modell einer Concurrent-Read Exclusive-Write Parallel Random-Access Machine mit $O(N)$ Prozessoren erzielt der Algorithmus eine Laufzeit von $O(\log N)$ bei einem Platzbedarf von $O(N)$, wobei N die Gesamtgröße der zu separierenden Punktmengen bezeichnet.

Als Grundlage dient ein optimaler sequentieller $O(N \log N)$-Algorithmus, der das gegebene Problem in die allgemeinere kombinatorische Aufgabe überführt, eine minimale Transversale in einem Circular-Arc-System zu finden.

Wie diese Diplomarbeit zeigt, läßt sich jeder Schritt des sequentiellen Algorithmus derart parallelisieren, daß der entstehende parallele Algorithmus optimal ist.

Auf diese Weise ergibt sich zusätzlich ein optimaler paralleler Algorithmus zur Berechnung einer minimalen Transversale in einem Circular-Arc-System. Dieser Algorithmus ist bereits auf dem schwächeren Modell der Exclusive-Read Exclusive-Write Parallel Random-Access Machine lauffähig.

Außerdem wird eine neue Idee zur Ermittlung von Sichtbarkeitstangenten vorgestellt, die auf der Auswertung einer quadratischen Entscheidungsfunktion beruht.

Eine Untersuchung, unter welchen Voraussetzungen für die parallele Implementierung des Separierungs-Algorithmus bereits eine Exclusive-Read Exclusive-Write Parallel Random-Access Machine ausreicht, beschließt die Arbeit.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Gegeben ist die folgende Aufgabenstellung:

> Seien POS und NEG zwei endlich große Punktmengen in der Ebene. Finde ein konvexes Polygon mit minimaler Kantenanzahl, das POS von NEG dadurch separiert, daß es alle Punkte aus POS aber keinen Punkt aus NEG enthält.

Das Problem der Suche nach einem solchen *minimalen konvexen separierenden Polygon* hat neben seiner theoretischen Bedeutung für die algorithmische Geometrie praktische Anwendungsmöglichkeiten im Bereich des „Maschinellen Lernens" und der „Klassifikationstheorie":

Betrachtet man POS als legale und NEG als illegale Punkte, so kann man ein minimales konvexes separierendes Polygon als „kleinste konsistente Hypothese" bzgl. POS und NEG auffassen (vgl. [FS]).

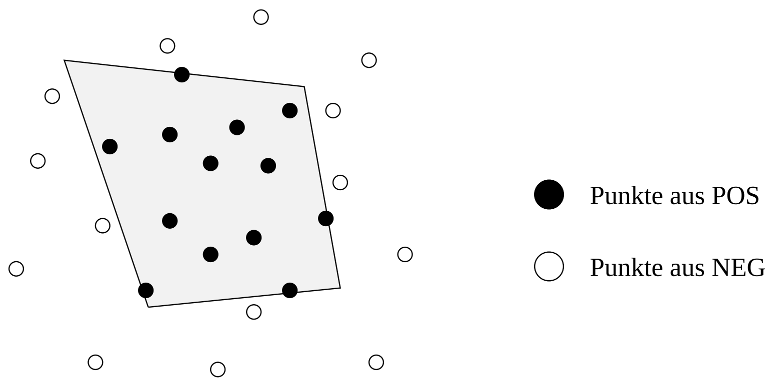


Bild 1.1: *Ein minimales konvexes separierendes Polygon*

Außerdem läßt sich ein konvexes Polygon mit k Kanten als Schnitt von k Halbebenen definieren. Somit reichen k lineare Tests aus, um einen neuen Punkt als Element von POS oder NEG zu klassifizieren (vgl. [EP]).

Fischer und Simon beschreiben in [FS] einen sequentiellen Algorithmus, der ein minimales konvexes separierendes Polygon für Punktmengen der Gesamtgröße N in $O(N \log N)$ Zeit ermittelt. Dabei bauen die Autoren auf einem Algorithmus von Edelsbrunner und Preparata ([EP]) auf und zeigen, daß sich das gegebene geometrische Problem als Spezialfall eines kombinatorischen Hypergraph-Problems auffassen läßt, nämlich als Suche nach „minimalen Transversalen in Circular-Arc-Systemen".

Ziel dieser Diplomarbeit ist es nun, den Algorithmus von Fischer und Simon „möglichst effizient" zu parallelisieren, also die einzelnen Algorithmusschritte derart umzuschreiben, daß sie eine möglichst geringe Laufzeit mit einer möglichst kleinen Prozessoranzahl erzielen.

Als Rechnermodell wird dazu das idealisierte Modell der sogenannten *Parallel Random-Access Machine* (PRAM) gewählt:

Eine PRAM besteht aus mehreren unabhängigen Prozessoren, die synchron arbeiten und jeweils über einen lokalen Speicher, *Local-Memory* genannt, verfügen. Zur Kommunikation untereinander teilen sich die einzelnen Prozessoren einen gemeinsamen Speicher, das *Global-Memory*. Pro Zeiteinheit kann jeder Prozessor eine Zelle des Local- oder Global-Memory lesen oder beschreiben.

Je nachdem, wie die im Global-Memory auftretenden Schreib- und Lesekonflikte behandelt werden, unterscheidet man die folgenden drei PRAM-Typen:

Die *Exclusive-Read Exclusive-Write* (EREW) PRAM verbietet es generell, daß zwei verschiedene Prozessoren zur gleichen Zeit auf ein und dieselbe Zelle im Global-Memory zugreifen, sei es zum Lesen oder zum Schreiben.
Dagegen können bei einer *Concurrent-Read Exclusive-Write* (CREW) PRAM zwei verschiedene Prozessoren gleichzeitig den Wert einer Zelle lesen. Weiterhin untersagt bleibt jedoch das simultane Beschreiben.
Das stärkste PRAM-Modell repräsentiert schließlich die *Concurrent-Read Concurrent-Write* (CRCW) PRAM. Bei ihr dürfen die Zellen des Global-Memory beliebig simultan gelesen und beschrieben werden.
Die CRCW PRAM spielt jedoch für die weiteren Betrachtungen keine Rolle. Deshalb informiere man sich über die einzelnen Schreibstrategien COMMON, ARBITRARY und PRIORITY zum Beispiel in [KR].

Diese Diplomarbeit zeigt, wie sich alle Schritte des sequentiellen Algorithmus von Fischer und Simon auf einer EREW PRAM bzw. CREW PRAM implementieren lassen, ohne eine logarithmische Gesamtlaufzeit zu überschreiten.

Das Ergebnis lautet:

Theorem 1.1
Sind zwei Punktmengen POS *und* NEG *der Ebene mit Gesamtgröße N gegeben, dann kann eine* CREW PRAM *mit $O(N)$ Prozessoren ein minimales konvexes Polygon, das* POS *von* NEG *separiert, in $O(\log N)$ Zeit bei einem Platzbedarf von $O(N)$ ermitteln.*

Sei $T_{seq}(n)$ die optimale Laufzeit eines sequentiellen Algorithmus mit Eingabelänge n. Benutzt man anstelle eines Prozessors $p(n)$ Prozessoren, so ist die untere Grenze an paralleler Laufzeit dieses Algorithmus offenbar durch $O(T_{seq}(n)/p(n))$ gegeben.

Deshalb nennt man einen parallelen Algorithmus mit Laufzeit $T(n)$ genau dann *optimal*, wenn $p(n) * T(n) = O(T_{seq}(n))$, denn es gilt $T(n) = O(T_{seq}(n)/p(n))$. (Das Produkt $p(n) * T(n)$ bezeichnet man auch als „Arbeit".)

In [EP] wird gezeigt, daß die optimale sequentielle Laufzeit eines Algorithmus zur Berechnung eines minimalen konvexen separierenden Polygons $O(N \log N)$ beträgt. Im Sinne der obigen Definition ist der in dieser Arbeit entwickelte parallele Algorithmus also optimal.

Die Diplomarbeit ist wie folgt gegliedert:

Im zweiten Kapitel wird der sequentielle Algorithmus von Fischer und Simon zur Berechnung eines minimalen konvexen separierenden Polygons ausführlich vorgestellt.

Das dritte Kapitel dient als Vorbereitung zur parallelen Implementierung auf einer PRAM. Hier werden insbesondere die parallelen Standardalgorithmen *Parallel-Prefix-Computation* und *List-Ranking* eingeführt und deren Verwendung an Beispielen demonstriert. Auf diese Beispiele wird in den folgenden Kapiteln dann immer wieder Bezug genommen werden.

Das vierte Kapitel enthält schließlich die eigentliche parallele Implementierung des Algorithmus.

Zwei größere Prozeduren, nämlich ein Verfahren zur Berechnung von Sichtbarkeitstangenten sowie ein optimaler paralleler Algorithmus zur Ermittlung einer minimalen Transversale in einem Circular-Arc-System, lassen sich auch in anderen Zusammenhängen verwenden und werden deshalb in zwei Extrakapiteln 5 und 6 vorgestellt.

Im siebten Kapitel befinden sich noch einmal ein tabellarischer Überblick über die gewonnenen Ergebnisse, sowie einige ergänzende und weiterführende Bemerkungen.

Insbesondere wird auf den Fall eingegangen, daß kein konvexes Polygon zur Separierung der Punktmengen POS und NEG existiert, und wie der Algorithmus eine solche Situation erkennen kann.
Außerdem zeigt eine Analyse, unter welchen Voraussetzungen der Algorithmus bereits auf einer EREW PRAM lauffähig ist.

Kapitel 2

Ein optimaler Algorithmus zur Berechnung eines minimalen konvexen separierenden Polygons

Ist eine endliche Punktmenge S der Ebene gegeben, dann nennt man das flächenmäßig kleinste konvexe Polygon, das alle Punkte aus S enthält, die *konvexe Hülle von S*, und bezeichnet es mit HULL(S).
Dabei fallen die Ecken von HULL(S) mit Punkten aus S zusammen, so daß der Rand von HULL(S), bezeichnet mit BOUNDARY(S), durch eine Auflistung dieser Eckpunkte im Uhrzeigersinn hinreichend genau charakterisiert ist.

Angenommen, zwei endliche Punktmengen POS und NEG der Ebene sind gegeben, und es existiert ein konvexes Polygon, das alle Punkte aus POS aber keinen Punkt aus NEG enthält. Dann ist die letzte Bedingung offenbar äquivalent mit der Aussage

$$\mathrm{HULL(POS)} \cap \mathrm{NEG} = \emptyset.$$

Sei also $\mathrm{HULL(POS)} \cap \mathrm{NEG} = \emptyset$ für die folgenden Betrachtungen vorausgesetzt:

Jede Gerade g zerlegt die Ebene topologisch gesehen in zwei offene Halbebenen. g separiert einen Punkt $q \in \mathrm{NEG}$ genau dann von POS , wenn q in einer dieser beiden offenen Halbebenen und ganz POS im anderen, abgeschlossenen Teil der Ebene liegt.
Man definiert ein *separierendes Geradensystem* bzgl. POS und NEG als eine Auswahl von Geraden, so daß zu jedem Punkt $q \in \mathrm{NEG}$ eine Gerade existiert, die q von POS separiert.

Daß konvexe separierende Polygone und separierende Geradensysteme in einer Eins-zu-Eins-Beziehung stehen, ist offensichtlich, denn separierende Geradensysteme sind nichts anderes als konvexe separierende Polygone, deren Kanten zu Geraden „verlängert" wurden.

Eine Gerade t, die BOUNDARY(POS) berührt aber nicht durch das Innere von HULL(POS) verläuft, heißt *Tangente* an HULL(POS).
Durch Parallelverschiebung läßt sich zu jeder Geraden g eine Tangente t finden, die dieselben Punkte $q \in$ NEG von POS separiert wie g. (Eventuell separiert t sogar einige Punkte mehr als g.)
Somit existiert zu jedem separierenden Geradensystem ein System derselben Größe, das nur aus Tangenten besteht, und es folgt:

Folgerung 2.1
Die Suche nach einem minimalen konvexen separierenden Polygon ist äquivalent mit der Suche nach einem separierenden Geradensystem minimaler Größe, das nur aus Tangenten an HULL(POS) *besteht. Diese Tangenten repräsentieren die (verlängerten) Kanten eines minimalen konvexen separierenden Polygons.*

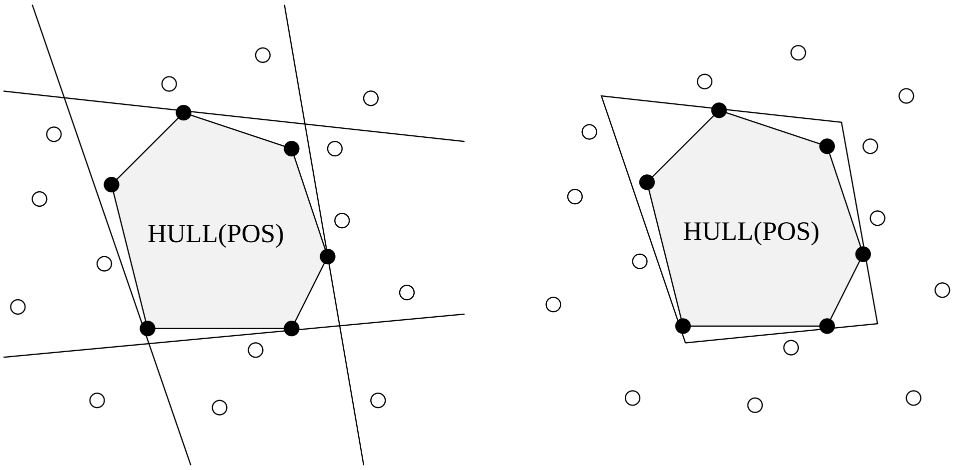


Bild 2.1: *Ein minimales separierendes Tangentensystem und das korrespondierende minimale konvexe separierende Polygon.*

Fischer und Simon stellen in [FS] den folgenden Algorithmus vor, der ein separierendes Tangentensystem minimaler Größe findet. Ihr Verfahren basiert auf dem Konzept der „Sichtbarkeit":

Durch jeden Punkt $q \in$ NEG lassen sich zwei Tangenten an HULL(POS) legen. Diese zerteilen BOUNDARY(POS) in zwei zusammenhängende Teilstücke. Das eine Teilstück $V(q)$ repräsentiert genau den von q aus gesehen „sichtbaren“ Anteil von BOUNDARY(POS), während das andere Teilstück $V'(q)$ den von q aus gesehen „unsichtbaren“ Teil darstellt.

Deshalb nennt man diese beiden Tangenten die rechte bzw. linke *Sichtbarkeitstangente* von q, und das offene, von $V(q)$ und den beiden Sichtbarkeitstangenten eingeschlossene Gebiet, heißt *Sichtbarkeitsbereich* von q.

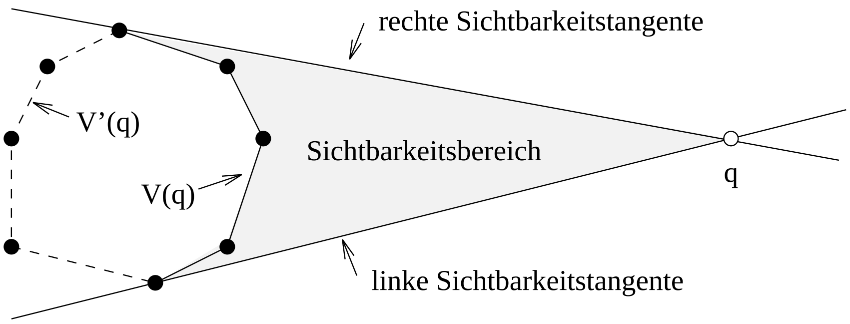


Bild 2.2

Man erkennt sofort:

Lemma 2.2
Eine Tangente separiert genau dann einen Punkt $q \in$ NEG von POS, wenn sie durch den Sichtbarkeitsbereich von q verläuft.

Seien $p_0, p_1, \ldots, p_{n-1}$ ($n \leq |$ POS $|$) die im Uhrzeigersinn sortierten Eckpunkte aus BOUNDARY(POS) und g_i, $i \in \{0, 1, \ldots, n-1\}$, die Geraden durch die Punkte $p_{(i-1) \bmod n}$ und p_i.
Da zwei verschiedene Punkte aus $q \in$ NEG durchaus dieselbe Tangente als Sichtbarkeitstangente besitzen können, gilt für s, die Anzahl unterschiedlicher Sichtbarkeitstangenten: $s \leq 2 \mid$ NEG $|$.

Um Lemma 2.2 für weitere kombinatorische Betrachtungen anwenden zu können, wird im folgenden ein spezielles (separierendes) Tangentensystem $\mathcal{T}$ erstellt. $\mathcal{T}$ besteht aus $2s$ Tangenten $T_0, T_1, \ldots, T_{2s-1}$, die bzgl. ihrer Lage um HULL(POS) „im Uhrzeigersinn sortiert“ sind.

Als Tangenten $T_0, T_2, \ldots, T_{2(s-1)}$ wählt man die s Sichtbarkeitstangenten, deren Sortierung im Uhrzeigersinn sich aus dem folgenden „Sweep-Line-Algorithmus“ ergibt:

Setze $k := 0$.
Durchlaufe nacheinander alle Eckpunkte $p_0, p_1, \ldots, p_{n-1}$ und „drehe“ für jeden Punkt p_i eine Sweep-Line um p_i im Uhrzeigersinn von g_i nach $g_{(i+1) \mathrm{mod} n}$.
Kommt die Sweep-Line dabei mit einer Sichtbarkeitstangente T zur Deckung, so setze $T_{2k} := T$ und erhöhe anschließend k um 1.

Auf diese Weise werden die Sichtbarkeitstangenten zu den Tangenten von $\mathcal{T}$ mit geradem Index.

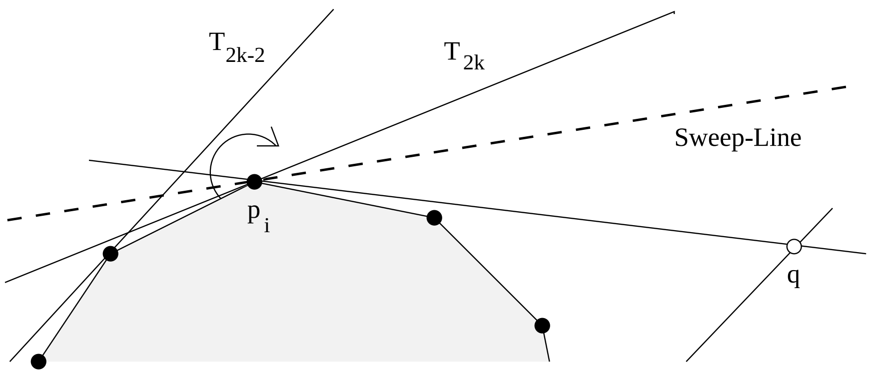


Bild 2.3: *Gleich wird der Sweep-Line-Algorithmus auf die rechte Sichtbarkeitstangente von q treffen und diese zur Tangenten T_{2k+2} machen ...*

Als Tangenten mit ungeradem Index wählt man sogenannte *Zwischentangenten*: Dabei ist für je zwei Tangenten T_{2k} und $T_{(2k+2) \mathrm{mod} 2s}$ ihre *Zwischentangente* $T_{(2k+1) \mathrm{mod} 2s}$ definiert als eine beliebige aber fest gewählte Tangente, die mit der Sweep-Line zur Deckung kommt, während man die Sweep-Line von T_{2k} nach $T_{(2k+2) \mathrm{mod} 2s}$ im Uhzeigersinn dreht (vgl. Bild 2.4).

Das so festgelegte Tangentensystem $\mathcal{T}$ besitzt die folgenden wichtigen Eigenschaften:

1. Betrachtet man einen Punkt $q \in$ NEG , dessen rechte Sichtbarkeitstangente zur Tangente T_{2r} und dessen linke Sichtbarkeitstangente zur Tangente T_{2l} wurde, so folgt auf Grund ihrer Sortierung, daß genau die Tangenten $\mathcal{T}(q) := \{T_{(2r+1) \mathrm{mod} 2s}, T_{(2r+2) \mathrm{mod} 2s}, \ldots, T_{(2l-1) \mathrm{mod} 2s}\}$ durch den Sichtbarkeitsbereich von q verlaufen.

2. Zu jeder beliebig gewählten Tangente T an HULL(POS) gibt es eine Tangente $T_i \in \mathcal{T}$, die genau dieselben Punkte $q \in$ NEG von POS separiert wie T. Also existiert zu jedem separierenden Tangentensystem ein separierendes Tangentensystem derselben Größe, das nur Tangenten aus $\mathcal{T}$ enthält.

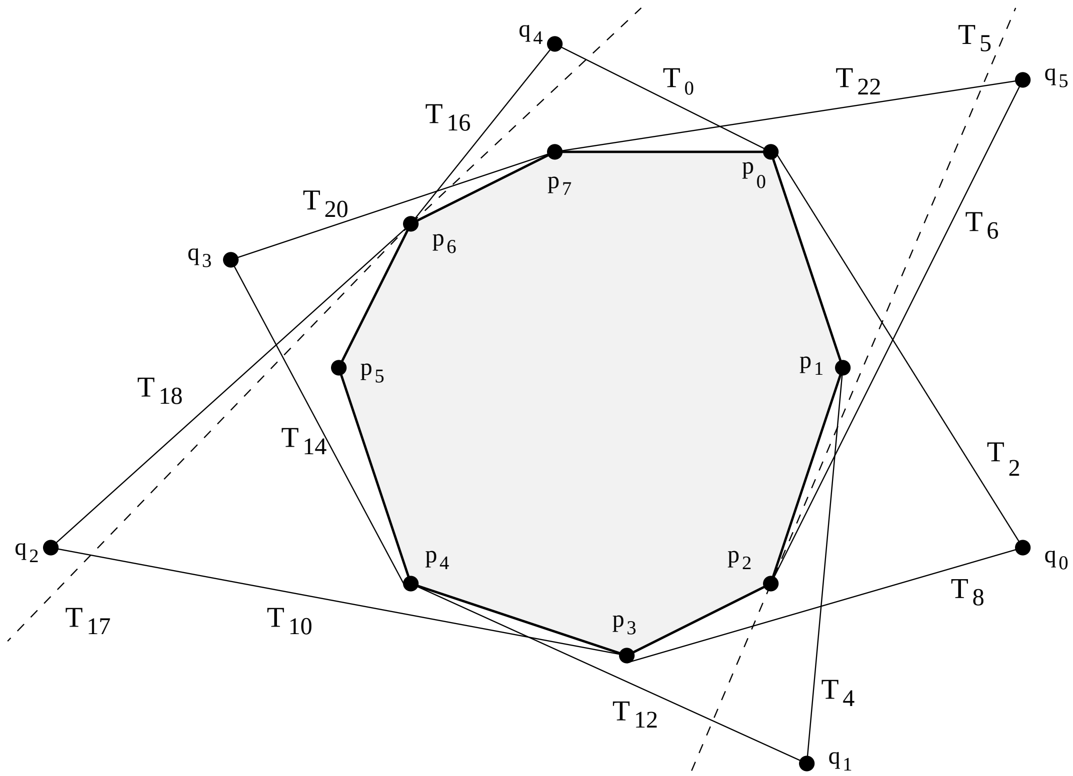


Bild 2.4: *Ein Tangentensystem $\mathcal{T}$ mit im Uhrzeigersinn sortierten Tangenten. Aus Gründen der Übersichtlichkeit wurden nur die Zwischentangenten T_5 und T_{17} eingezeichnet.*

Aus diesen Beobachtungen und Lemma 2.2 ergibt sich:

Folgerung 2.3
Ein Tangentensystem $\mathcal{T}^* \subset \mathcal{T}$ mit $\mathcal{T}^* \cap \mathcal{T}(q) \neq \emptyset$ für alle $q \in$ NEG ist ein separierendes Tangentensystem bzgl. POS *und* NEG.
Ein Tangentensystem $\mathcal{T}^* \subset \mathcal{T}$ minimaler Größe mit $\mathcal{T}^* \cap \mathcal{T}(q) \neq \emptyset$ für alle $q \in$ NEG *ist ein minimales separierendes Tangentensystem bzgl.* POS *und* NEG.

Damit läßt sich die Suche nach einem minimalen separierenden Tangentensystem als Suche nach einer *minimalen Transversale* in einem *Circular-Arc-System* auffassen:

Definition 2.4
Ein *Circular-Arc-System* S besteht aus einer Menge im Uhrzeigersinn orientierter abgeschlossener Intervalle I, sogenannter *Bögen*, auf einem Kreis C.
Eine Menge M von Punkten auf C heißt *Transversale* in S, wenn $M \cap I \neq \emptyset$ für alle $I \in S$.
Eine Transversale in S heißt *minimal*, wenn es keine Transversale in S mit weniger Punkten gibt.

Identifiziert man jede Tangente $T_i \in \mathcal{T}$ mit einem Punkt i auf einem Kreis C, so kann man zu jedem Punkt $q \in$ NEG mit rechter und linker Sichtbarkeitstangente T_{2r} bzw. T_{2l} das Tangentensystem $\mathcal{T}(q)$ als Bogen I_q auf C darstellen. I_q verläuft von $(2r+1) \bmod 2s$ über $(2r+2) \bmod 2s$ bis nach $(2l-1) \bmod 2s$.

Das so entstehende Circular-Arc-System S heißt *das zu* POS *und* NEG *assoziierte Circular-Arc-System.*

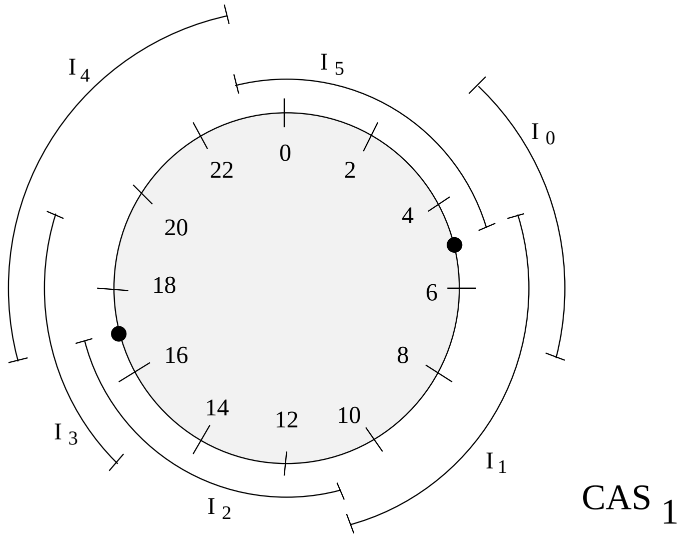


Bild 2.5: *Das zu den Punkten aus Bild 2.4 assoziierte Circular-Arc-System* CAS_1 *mit minimaler Transversale* $\{5,\ 17\}$.

Man kann zeigen, daß minimale Transversalen o.B.d.A nur aus Bogenendpunkten bestehen (vgl. Lemma 6.7). In S sind alle Bogenendpunkte ungerade. Auf Grund dieser Tatsache ergibt sich in Verbindung mit Folgerung 2.3 und Definition 2.4:

Satz 2.5
Für alle Teilmengen $M \subset \{1, 3, \ldots, 2s-1\}$ gilt:
$\{T_j \mid j \in M\}$ ist genau dann ein separierendes Tangentensystem bzgl. POS *und* NEG*, wenn M eine Transversale in dem zu* POS *und* NEG *assoziierten Circular-Arc-System darstellt. Insbesondere ergibt ein M minimaler Größe eine minimale Transversale und somit ein minimales konvexes separierendes Polygon.*

(CAS_1 hat eine minimale Transversale $\{5, 17\}$, und tatsächlich bilden die Tangenten T_5 und T_{17} in Bild 2.4 ein minimales konvexes separierendes Polygon, das allerdings ungebunden ist.)

Der Algorithmus von Fischer und Simon zur Berechnung eines minimalen konvexen separierenden Polygons lautet damit wie folgt:

ALGORITHMUS „MINIMALES KONVEXES SEPARIERENDES POLYGON“

Eingabe: Punktmengen POS und NEG in der Ebene.

Ausgabe: Tangenten, die ein minimales konvexes separierendes Polygon bilden.

Schritt 1: Berechne BOUNDARY(POS).

Schritt 2: Ermittle für jeden Punkt $q \in$ NEG seine rechte und linke Sichtbarkeitstangente.

Schritt 3: Sortiere die Sichtbarkeitstangenten im Uhrzeigersinn.

Schritt 4: Ermittle das zu POS und NEG assoziierte Circular-Arc-System S.

Schritt 5: Finde eine minimale Transversale M in S.

Schritt 6: Berechne die zu jedem Punkt in M korrespondierende Zwischentangente, und gebe sie aus.

Satz 2.6
Dieser Algorithmus berechnet ein minimales konvexes separierendes Polygon bzgl. POS *und* NEG *in einer sequentiellen Laufzeit von $O(N \log N)$.*

Beweis:
Die Korrektheit des Algorithmus folgt aus den Vorüberlegungen.

Mit $n' = |\,\mathrm{POS}|$, $n = |\,\mathrm{BOUNDARY(POS)}|$ und $m = |\,\mathrm{NEG}|$ ergeben sich für die einzelnen Schritte des Algorithmus folgende Laufzeiten:

Schritt 1 ist ein Standardproblem der algorithmischen Geometrie und läßt sich in $O(n' \log n')$ Zeit zum Beispiel mittels *Graham-Scan* (vgl. [PS]) lösen.

In Kapitel 5 wird ein Algorithmus vorgestellt, der in $O(\log n)$ die Sichtbarkeitstangenten eines Punktes $q \in \mathrm{NEG}$ ermittelt. Um Sichtbarkeitstangenten für alle Punkte $q \in \mathrm{NEG}$ zu berechnen, ergibt sich also eine Laufzeit von $O(m \log n)$.

Wie man Schritt 3 als „normales" Sortierproblem auffassen kann, zeigt Kapitel 4. Das Sortieren der $2m$ Sichtbarkeitstangenten benötigt demnach $O(m \log m)$ sequentielle Laufzeit.

Die Schritte 4, 5 und 6 lassen sich in $O(m)$ implementieren:
Für die Schritte 4 und 6 folgt dieses „straight-forward", und für Schritt 5 geben Fischer und Simon in [FS] einen Minimalen-Transversalen-Algorithmus an, der eine Laufzeit von $O(m)$ (auf sortierten Daten) erzielt.

Wegen $N \geq k$ für alle $k \in \{n, n', m\}$ ergibt sich die behauptete Gesamtlaufzeit.

$\square$

Kapitel 3

Vorbereitungen für die parallele Implementierung

3.1 Die Orientierungsdeterminante

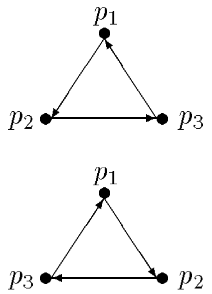
Eine wichtige Rolle in dieser Arbeit wird das Testen von Punkttripeln auf ihre *Orientierung* spielen.

Ein Punkttripel (p_1, p_2, p_3) besteht dabei aus drei paarweise verschiedenen Punkten p_1, p_2, p_3 der euklidischen Ebene, wobei die Reihenfolge der Punkte relevant ist, das heißt z. B. $(p_1, p_2, p_3) \neq (p_1, p_3, p_2)$.

Liegen p_1, p_2 und p_3 auf einer Geraden, nennt man das Punkttripel *kollinear*.

Jedem nichtkollinearen Punkttripel lassen sich nun auf kanonische Weise zwei Orientierungen zuordnen:

Definition 3.1
(p_1, p_2, p_3) heißt *links orientiert*, wenn seine drei Punkte in der Reihenfolge p_1, p_2, p_3 entgegen dem Uhrzeigersinn angeordnet sind.
(p_1, p_2, p_3) heißt *rechts orientiert*, wenn seine drei Punkte in der Reihenfolge p_1, p_2, p_3 im Uhrzeigersinn angeordnet sind.



Im ersten Fall sagt man auch: (p_1, p_2, p_3) bildet ein *Linkssystem*; im zweiten Fall spricht man von einem *Rechtssystem*.

Mit Hilfe des Orientierungsbegriffs lassen sich viele elementargeometrische Fragen in der Ebene beantworten, wie z.B. die Lage eines Punktes p_1 bzgl. einer Geraden g, die durch zwei Punkte p_2 und p_3 repräsentiert wird und ähnliches.

Das nun vorgestellte Verfahren zur Orientierungsermittlung benutzt eine quadratische Entscheidungsfunktion in Form einer Determinante (vgl. [PS]). Da keinerlei trigonometrische Funktionen verwendet werden, ergeben sich bei einer Implementierung auf Rechnern günstige Laufzeiten:

Definition 3.2
Seien drei Punkte p_1, p_2, p_3 der euklidischen Ebene durch ihre x-Koordinaten $x(p_1)$, $x(p_2)$, $x(p_3)$ sowie ihre y-Koordinaten $y(p_1)$, $y(p_2)$, $y(p_3)$ gegeben. Dann heißt die Determinante

$$\Delta(p_1, p_2, p_3) := \begin{vmatrix} x(p_1) & y(p_1) & 1 \\ x(p_2) & y(p_2) & 1 \\ x(p_3) & y(p_3) & 1 \end{vmatrix}$$

die *Orientierungsdeterminante* von p_1, p_2 und p_3.

Der folgende Satz erklärt den Namen der oben beschriebenen Determinante:

Satz 3.3
Für die Orientierungsdeterminante Δ dreier verschiedener Punkte p_1, p_2, p_3 gilt:

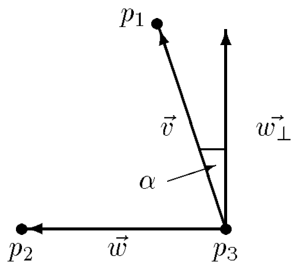
(i) $\Delta(p_1, p_2, p_3) > 0 \iff (p_1, p_2, p_3)$ *bildet ein Linkssystem*

(ii) $\Delta(p_1, p_2, p_3) < 0 \iff (p_1, p_2, p_3)$ *bildet ein Rechtssystem*

Beweis:
Der Einfachheit halber setze man $x_i := x(p_i)$, $y_i := y(p_i)$ $(i = 1, 2, 3)$.

Seien $\vec{p_1}$, $\vec{p_2}$ und $\vec{p_3}$ die (verschiedenen) Ortsvektoren der Punkte p_1, p_2 und p_3, das heißt $\vec{p_i} = \binom{x_i}{y_i}$ $(i = 1, 2, 3)$.

Setze
$\vec{v} := \vec{p_1} - \vec{p_3} = \binom{x_1 - x_3}{y_1 - y_3}$,
$\vec{w} := \vec{p_2} - \vec{p_3} = \binom{x_2 - x_3}{y_2 - y_3}$,
$\vec{w_\perp} := \binom{y_2 - y_3}{-(x_2 - x_3)}$.



Dann gilt für das Standardskalarprodukt zwischen $\vec{v}$ und $\vec{w_\perp}$

$$\vec{v} * \vec{w_\perp} \quad = \quad (\vec{p_1} - \vec{p_3}) * \vec{w_\perp} \quad = \quad \begin{pmatrix} x_1 - x_3 \\ y_1 - y_3 \end{pmatrix} * \begin{pmatrix} y_2 - y_3 \\ -(x_2 - x_3) \end{pmatrix}$$

$$= x_1y_2 - x_3y_2 - x_1y_3 + x_3y_3 + x_3y_1 - x_3y_3 - x_2y_1 + x_2y_3$$

$$= x_1y_2 + x_2y_3 + x_3y_1 - x_3y_2 - x_2y_1 - x_1y_3$$

$$= \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} \quad = \quad \Delta(p_1, p_2, p_3).$$

Für (i) bleibt also noch zu zeigen, daß

$$\vec{v} * \vec{w_\perp} > 0 \iff (p_1, p_2, p_3) \textit{ Linkssystem}.$$

Dies folgt jedoch aus der Beziehung

$$\vec{v} * \vec{w_\perp} = \|\vec{v}\| \cdot \|\vec{w_\perp}\| \cdot \cos\alpha,$$

wobei $\|.\|$ den Betrag der Vektoren und $\alpha \in [0, \pi]$ den Innenwinkel zwischen $\vec{v}$ und $\vec{w_\perp}$ angibt:

$$\vec{v} * \vec{w_\perp} > 0 \iff \cos\alpha > 0 \quad (\|\vec{v}\| \text{ und } \|\vec{w_\perp}\| \text{ sind } \neq 0 \text{ n. Vor.})$$

$$\iff \alpha \in [0, \frac{1}{2}\pi[\iff (p_1, p_2, p_3) \textit{ Linkssystem}.$$

Für (ii) läßt sich analog zeigen, daß

$$\vec{v} * \vec{w_\perp} < 0 \iff (p_1, p_2, p_3) \textit{ Rechtssystem}. \qquad \square$$

Aus obigem Satz folgt sofort, daß im Fall von $\Delta(p_1, p_2, p_3) = 0$ die Punkte p_1, p_2, p_3 weder ein Links- noch ein Rechtssystem bilden, also kollinear sind.

Damit sind die Vorbereitungen für den „geometrischen Teil“ der Diplomarbeit abgeschlossen. Jetzt werden die speziell für die parallele Programmierung benötigten Grundlagen geschaffen:

3.2 Parallel-Prefix-Computation

Das am vielseitigsten verwendbare Hilfsmittel beim Entwickeln paralleler Algorithmen ist mit Sicherheit der Algorithmus zur Prefix-Summen-Berechnung, *Parallel-Prefix-Computation* oder einfach nur *Parallel-Prefix* genannt.

Gegeben ein Array X mit Elementen $x_0, x_1, \ldots, x_{n-1}$ aus einem Definitionsbereich D und eine assoziative binäre Verknüpfung $\oplus$, besteht die Aufgabe darin, die *Prefix-Summen* $S_i := x_0 \oplus x_1 \oplus \ldots \oplus x_i$ für alle $i \in \{0, 1, \ldots, n-1\}$ zu berechnen.

Die $\oplus$-Operation kann z.B. die normale Addition oder Multiplikation sein, aber auch Minimums- oder Maximumsbildung sind denkbar.

Ein sequentieller Algorithmus zur Lösung dieser Aufgabe ist auf Grund der Tatsache $S_{i+1} = S_i \oplus x_{i+1}$ trivial und benötigt $O(n)$ Zeit. Für eine parallele Implementierung läßt sich aus dieser „sequentiellen" Beziehung auf Anhieb jedoch kaum Kapital schlagen.

Der folgende parallele Algorithmus zur Berechnung von Prefix-Summen steht z.B. ähnlich in [KR] und [KRS]:

ALGORITHMUS „PARALLEL-PREFIX-COMPUTATION"

Eingabe: Eine assoziative binäre Verknüpfung „$\oplus$" und ein Array $X = (x_0, x_1, \ldots, x_{n-1})$ mit Elementen $x_i \in D$.

Ausgabe: Ein Array $S = (S_0, S_1, \ldots, S_{n-1})$ aller n Prefix-Summen.

Schritt 1: Setze parallel $S_i := x_i$ für alle $i \in \{0, 1, \ldots, n-1\}$.

Schritt 2: Führe Schritt 3 für alle $j \in \{0, 1, \ldots, \lceil \log_2 n \rceil - 1\}$ aus:

Schritt 3: Setze parallel für alle $i \in \{2^j, 2^j+1, \ldots, n-1\}$: $S_i := S_{i-2^j} \oplus S_i$.

Eine PRAM mit $O(n)$ Prozessoren kann jeden Schritt des Algorithmus in konstanter Zeit ausführen, indem der k-te Prozessor für das Arrayelement S_k zuständig ist. Da der Schritt 3 genau $\lceil \log_2 n \rceil$-mal ausgeführt wird, ergibt sich eine Gesamtlaufzeit von $O(\log n)$.

Zerteilt man Schritt 3 in eine Phase zum Lesen aller Elemente S_{i-2^j}, eine Phase zum Lesen aller Elemente S_i und eine Phase zum Schreiben der neuen Werte von S_i, so treten bei der parallelen Ausführung des Algorithmus keinerlei Lese- oder Schreibkonflikte auf den Daten von S auf. Zur Durchführung genügt also eine EREW PRAM .

Die Korrektheit des Algorithmus folgt aus der Beobachtung, daß für alle Durchläufe $j \in \{0, 1, \ldots, \lceil \log_2 n \rceil - 1\}$ nach der Zuweisungsoperation in Schritt 3 gilt:

$$S_i = x_k \oplus x_{k+1} \oplus \ldots \oplus x_i \qquad (k := max\{0, i - 2^{j+1} + 1\})$$

(Mittels vollständiger Induktion über j läßt sich diese Beziehung leicht nachweisen.) Für $j = \lceil \log_2 n \rceil - 1$ enthalten demnach alle S_i die gewünschten Prefix-Summen.

Beispiel: Parallel-Prefix-Computation bzgl. der normalen Addition, angewandt auf das Array $X = (3, 1, -4, 1, 5, -9, 2, 6, -5, 4)$.

j	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9
	3	1	-4	1	5	-9	2	6	-5	4
0		4	-3	-3	6	-4	-7	8	1	-1
1			0	1	3	-7	-1	4	-6	7
2					6	-3	-1	5	-3	0
3									0	4

Der Parallel-Prefix-Algorithmus ist in seiner jetzigen Form noch nicht optimal, denn die verrichtete Arbeit beträgt bei $O(n)$ Prozessoren und einer Laufzeit von $O(\log n)$ genau $O(n \log n)$.
Der triviale sequentielle Algorithmus läuft jedoch in $O(n)$.

Mit Hilfe von „Brent's Scheduling-Prinzip" (vgl. [KR]) gelingt es aber, einen logarithmischen Faktor von Prozessoren einzusparen:

Schritt 1: Zuerst zerlegt man das Eingabearray X in m gleichgroße zusammenhängende Blöcke $X_0, X_1, \ldots, X_{m-1}$ der Größe $g := n/m$, d.h.

$$X_i := (x_{ig}, x_{ig+1}, \ldots, x_{(i+1)g-1}).$$

(Der Einfachheit halber sei n durch m teilbar. Auf die günstigste Wahl von m wird später eingegangen.)

Schritt 2: Parallel wird nun innerhalb jedes Blocks X_i auf sequentielle Weise

$$y_i := x_{ig} \oplus x_{ig+1} \oplus \ldots \oplus x_{(i+1)g-1}$$

berechnet. Es entsteht ein Array

$$Y = (y_0, y_1, \ldots, y_{m-1}).$$

Schritt 3: Anschließend ermittelt man mittels des eigentlichen Parallel-Prefix-Algorithmus alle Teilsummen

$$s_j := y_0 \oplus y_1 \oplus \ldots \oplus y_j$$

des verkürzten Arrays Y.

Schritt 4: Für jedes x_j aus X_i läßt sich seine tatsächliche Prefix-Summe S_j mit Hilfe des sequentiellen Algorithmus bestimmen als

$$S_j := s_{i-1} \oplus x_{ig} \oplus x_{ig+1} \oplus \ldots \oplus x_j, \quad (s_{-1} := 0).$$

Sämtliche Prefix-Summen ergeben sich, indem parallel innerhalb jedes Blocks X_i der sequentielle Prefix-Algorithmus angewandt wird.

Hat man m Prozessoren einer EREW PRAM zur Verfügung, so beträgt die Gesamtlaufzeit des modifizierten Parallel-Prefix-Algorithmus $O(n/m + \log m)$.
Wählt man m geschickt als $n/\log n$, so ergibt sich eine Laufzeit von $O(n/(n/\log n) + \log(n/\log n)) = O(\log n)$ bei nur noch $O(n/\log n)$ Prozessoren. Dies ist optimal.

Es ergibt sich:

Satz 3.4
*Auf einem Array der Größe n ist das Problem „Parallel-Prefix-Computation" mit einer $O(n/\log n)$-Prozessor-*EREW PRAM *in $O(\log n)$ Zeit und $O(n)$ Platz lösbar.*

Die Vielseitigkeit von Parallel-Prefix-Computation wird nun an drei Beispielen demonstriert, auf die im weiteren Verlauf der Diplomarbeit des öfteren Bezug genommen wird:

3.2.1 Anwendungsbeispiel 1: Die Broadcast-Phase

Bei dem Entwurf paralleler Algorithmen kommt es sehr häufig vor, daß eine Information, z.B. der Wert einer Variable, n Prozessoren $Proz_0$, $Proz_1$, …, $Proz_{n-1}$ zum gleichen Zeitpunkt zur Verfügung stehen muß.

Für eine CREW PRAM stellt dies kein Problem dar: Angenommen, die Information steht in einer Speicherzelle des Global-Memory, dann kann jeder Prozessor die Information dort einfach herauslesen.

Anders bei einer EREW PRAM: Hier sind gleichzeitige Leseaktionen auf ein und dieselbe Speicherstelle des Global-Memory verboten, und es stellt sich die Frage, ob es eine effizientere Möglichkeit der Informationsverbreitung gibt, als alle Prozessoren nacheinander auf das Global-Memory zugreifen zu lassen.

Eine Lösung besteht darin, eine *Broadcast-Phase* mittels Parallel-Prefix-Computation zu realisieren:

In einem ersten Schritt liest $Proz_0$ die Information (o.B.d.A eine Zahl aus $z \in \Re$) und schreibt sie nach x_0, dem ersten Element eines Arrays $X = (x_0, x_1, \ldots, x_{n-1})$ im Global-Memory. Gleichzeitig initialisieren die Prozessoren $Proz_i$ $(i \neq 0)$ die übrigen Elemente x_i mit dem Wert 0.
Werden anschließend alle Summen $S_i = x_0 + x_1 + \ldots + x_i$ in einem Parallel-Prefix-Durchlauf ermittelt, dann folgt $S_0 = S_1 = \ldots = S_{n-1} = z$. Der Wert von z hat sich also in logarithmischer Laufzeit n-mal kopiert und steht für den i-ten Prozessor in S_i zur Verfügung.

Es folgt:

Satz 3.5
Mittels Parallel-Prefix-Computation kann eine Broadcast-Phase durchgeführt werden, so daß eine Information allen n Prozessoren einer EREW PRAM *in* $O(\log n)$ *Zeit bekannt ist.*

3.2.2 Anwendungsbeispiel 2: Löschen von Arrayelementen

Aufgabenstellung: Aus einem Array $X = (x_0, x_1, \ldots, x_{n-1})$ werden einige Elemente x_i gelöscht. Gesucht ist das entstehende Array $X' = (x'_0, x'_1, \ldots, x'_{m-1})$.

Durch die Verwendung von Parallel-Prefix-Computation ist das Problem leicht zu lösen:

Zuerst ordnet man jedem Element x_i ein Gewicht $w(i)$ zu, wobei

$$w(i) := \begin{cases} 0, & \text{falls } x_i \text{ aus } X \text{ gelöscht werden soll,} \\ 1, & \text{sonst.} \end{cases}$$

Ein anschließender Parallel-Prefix-Durchlauf ergibt für jedes Element x_i ein Gesamtgewicht $W(i) := w(0) + w(1) + \ldots + w(i)$.
$W(i)$ gibt genau die Anzahl der vor x_i einsortierten Elemente (einschließlich x_i) an, die <u>nicht</u> gelöscht werden sollen.
Also setzt man $x'_{W(i)-1} := x_i$ für alle nicht zu löschenden Elemente x_i, und es ergibt sich das gewünschte Array X'.
(Man beachte, daß m, die Größe von X', implizit mitberechnet wurde, denn es gilt $m = W(n-1)$.)

Bearbeitet ein PRAM-Prozessor $O(\log n)$ Arrayelemente, so treten keine Schreib- oder Lesekonflikte auf, und es folgt:

Lemma 3.6
Löscht man aus einem Array der Größe n eine beliebige Anzahl von Elementen, so kann eine EREW PRAM *mit $O(n/\log n)$ Prozessoren das neu entstehende Array in einer Laufzeit von $O(\log n)$ mittels Parallel-Prefix-Computation berechnen.*

Beispiel: Aus dem Array $X = (3, 1, -4, 1, 5, -9, 2, 6, -5, 4)$ sollen alle negativen Einträge gelöscht werden.

i	0	1	2	3	4	5	6	7	8	9
X_i	3	1	-4	1	5	-9	2	6	-5	4
$w(i)$	1	1	0	1	1	0	1	1	0	1
$W(i)$	1	2	2	3	4	4	5	6	6	7
$W(i)-1$	0	1		2	3		4	5		6
X'_i	3	1	1	5	2	6	4			

3.2.3 Anwendungsbeispiel 3: Das Umsortierproblem

Aufgabenstellung: In einem Array $X = (x_0, x_1, \ldots, x_{n-1})$ werden einige Elemente umsortiert, indem sie zunächst aus X herausgenommen und anschließend wieder an X angefügt werden. Dabei darf sich die Reihenfolge der umsortierten Elemente untereinander nicht ändern: War x_i vor x_j in X einsortiert, und wurden beide Elemente umsortiert, so steht x_i vor x_j auch in X'. Gesucht ist wieder das entstehende Array $X' = (x'_0, x'_1, \ldots, x'_{n-1})$.

Die Lösung dieses Problems besteht im Prinzip aus einer zweifachen Anwendung von 3.2.2:

Zunächst simuliert man ein Löschen aller umzusortierenden Elemente x_i aus X, d.h. man ordnet allen umzusortierenden Elementen das Gewicht 0 und allen anderen Elementen das Gewicht 1 zu und berechnet mittels Parallel-Prefix $W(i)$ wie oben.
Anschließend vertauscht man die Rollen, d.h. man simuliert ein Löschen aller Elemente, die <u>nicht</u> umsortiert werden sollen, und berechnet das analoge Gesamtgewicht $W'(i)$.
(Wegen $W'(i) = i + 1 - W(i)$ kann auf einen Parallel-Prefix-Durchlauf an dieser Stelle verzichtet werden.)
Nach einer Broadcast-Phase, in der der Wert von $m := W(n-1)$ allen Prozessoren bekannt gegeben wurde, ergibt sich das neue Array X', indem man alle Elemente x_i aus X betrachtet und $x'_{m+W(i)'-1} := x_i$ bzw. $x'_{W(i)-1} := x_i$ setzt, je nachdem, ob x_i umsortiert werden soll oder nicht.

Beispiel: In dem Array $X = (3, 1, -4, 1, 5, -9, 2, 6, -5, 4)$ sollen alle negativen Einträge umsortiert werden.

i	0	1	2	3	4	5	6	7	8	9
X_i	3	1	-4	1	5	-9	2	6	-5	4
$w(i)$	1	1	0	1	1	0	1	1	0	1
$W(i)$	1	2	2	3	4	4	5	6	6	7
$W'(i)$	0	0	1	1	1	2	2	2	3	3
$m + W'(i) - 1$			7			8			9	
$W(i) - 1$	0	1		2	3		4	5		6
X'_i	3	1	1	5	2	6	4	-4	-9	-5

Analog zu Lemma 3.6 ergibt sich:

Lemma 3.7
Für ein Array der Größe n läßt sich das „Umsortierproblem“ auf einer EREW PRAM *mit $O(n/\log n)$ Prozessoren in einer Laufzeit von $O(\log n)$ mittels Parallel-Prefix-Computation lösen.*

3.3 Parallel-Suffix-Computation

Durch eine leichte Modifikation ist es sofort möglich, anstelle aller Prefix-Summen $S_i = x_0 \oplus x_1 \oplus \ldots \oplus x_i$ die *Suffix-Summen* $S'_i := x_i \oplus x_{i+1} \oplus \ldots \oplus x_{n-1}$ zu berechnen.

Dazu „spiegelt“ man den entscheidenden 3-ten Schritt des Prefix-Algorithmus, indem man jede k-te Komponente des S-Arrays mit der $(n-k)$-ten Komponente des S'-Arrays identifiziert:

ALGORITHMUS „PARALLEL-SUFFIX-COMPUTATION“

Eingabe: Eine assoziative binäre Verknüpfung „$\oplus$“ und ein Array $X = (x_0, x_1, \ldots, x_{n-1})$ mit Elementen $x_i \in D$.

Ausgabe: Ein Array $S' = (S'_0, S'_1, \ldots, S'_{n-1})$ aller n Suffix-Summen.

Schritt 1: Setze parallel $S'_i := x_i$ für alle $i \in \{0, 1, \ldots, n-1\}$.

Schritt 2: Führe Schritt 3 für alle $j \in \{0, 1, \ldots, \lceil \log_2 n \rceil - 1\}$ aus:

Schritt 3: Setze parallel für alle $i \in \{2^j, 2^j + 1, \ldots, n-1\}$:
$S'_{n-i} := S'_{n-i+2^j} \oplus S'_{n-i}$.

Analog zu Satz 3.4 ergibt sich:

Satz 3.8
*Auf einem Array der Größe n ist das Problem „Parallel-Suffix-Computation“ mit einer $O(n/\log n)$-Prozessor-*EREW PRAM *in $O(\log n)$ Zeit und $O(n)$ Platz lösbar.*

3.4 List-Ranking

Eng verwandt mit dem Problem der Suffix-Summen-Berechnung ist das *List-Ranking*-Problem:

Dabei ist anstelle eines Arrays der Größe n eine verkettete Liste L bestehend aus n Elementen gegeben.

Jedes Listenelement L_i, $i \in \{0, 1, \ldots, n-1\}$, besitzt ein Gewicht $w(i)$, und die Summe (bzgl. $\oplus$) aus $w(i)$ und der Gewichte aller Nachfolgeelemente von L_i in L ergibt die i-te Suffix-Summe.

Definition 3.9
Der *Rang* (engl. *Rank*) eines Listenelementes ist definiert als die Anzahl seiner Nachfolgeelemente in der Liste.

Belegt man das letzte Listenelement mit einem Gewicht 0 und alle anderen Listenelemente mit dem Gewicht 1 und berechnet die eben beschriebenen Suffix-Summen bzgl. der normalen Addition, so ergibt sich als Suffix-Summe für jedes Listenelement sein Rang.
Diese spezielle Anwendung ist der Grund, warum sich für die Suffix-Summen-Berechnung auf Listen die Bezeichnung *List-Ranking* durchgesetzt hat.

Der folgende List-Ranking-Algorithmus steht ähnlich in [KR] und [KRS]:

ALGORITHMUS „LIST-RANKING“

Eingabe: Eine assoziative binäre Verknüpfung „$\oplus$“ und eine Liste L bestehend aus n Elementen $L_0, L_1, \ldots, L_{n-1}$. Jedes Element L_i wird repräsentiert durch sein Gewicht $w(i) \in D$ und einen Zeiger $next(i)$ auf sein Nachfolgeelement in L, d.h. $L_{next(i)}$ ist das nächste Element der Liste. Für das letzte Listenelement gilt symbolisch $next(i) = \emptyset$.

Ausgabe: Alle n Suffix-Summen im Array $w()$.

Schritt 1: Führe Schritt 2 $\lceil \log_2 n \rceil$-mal aus:

Schritt 2: Prüfe parallel für alle $i \in \{0, 1, \ldots, n-1\}$:
Gilt $next(i) \neq \emptyset$, so setze $w(i) := w(i) \oplus w(next(i))$,
$next(i) := next(next(i))$.

Sei $r(i)$ der Rang von Listenelement L_i und $r(\emptyset) := -1$. Dann folgt die Korrektheit des Algorithmus aus der Beobachtung, daß $w(i)$ jeweils zu Beginn von Schritt 2 die Summe der Gewichte aller Elemente mit Rang $r(i), r(i)-1, \ldots, r(next(i))+1$ enthält. Nach $\lceil \log_2 n \rceil$ Durchläufen gilt $next(i) = \emptyset$. Damit enthält $w(i)$ zum Schluß die Summe der Gewichte aller Elemente mit Rang $r(i), r(i)-1, \ldots, 0$, und dies ist genau die i-te Suffix-Summe.

Beispiel: Rangermittlung auf der Liste
$L = L_3 \to L_5 \to L_9 \to L_2 \to L_1 \to L_7 \to L_0 \to L_8 \to L_6 \to L_4$,
also $(w_0, w_1, \ldots, w_8, w_9) = (1, 1, \ldots, 1, 0)$
und $(next(0), next(1), \ldots, next(9)) = (8, 7, 1, 5, \emptyset, 9, 4, 0, 6, 2)$.

i	0	1	2	3	4	5	6	7	8	9
$w(i)$	1	1	1	1	0	1	1	1	1	1
$next(i)$	8	7	1	5	$\emptyset$	9	4	0	6	2
$w(i)$	2	2	2	2		2	1	2	2	2
$next(i)$	6	0	7	9		2	$\emptyset$	8	4	1
$w(i)$	3	4	4	4		4		4	2	4
$next(i)$	$\emptyset$	6	8	1		7		4	$\emptyset$	0
$w(i)$		5	6	8		8		4		7
$next(i)$		$\emptyset$	$\emptyset$	6		4		$\emptyset$		$\emptyset$
$w(i)$				9		8				
$next(i)$				$\emptyset$		$\emptyset$				

Bei der Verwendung des List-Ranking-Algorithmus ist natürlich zu beachten, daß durch die Änderung des $next()$-Arrays die Struktur der Liste zerstört wird. Eventuell muß also vorher eine Kopie dieses Zeigerarrays angelegt werden.

Ordnet man jedem Listenelement einen Prozessor zu und vermeidet eventuelle Lesekonflikte durch entsprechende globale Lesephasen, so genügt eine EREW PRAM, um eine logarithmische Laufzeit des Algorithmus zu erzielen.

Allerdings werden $O(N)$ Prozessoren benötigt, so daß diese Variante des List-Ranking-Algorithmus (noch) nicht optimal ist.

Die Idee der Blockbildung, die ja bei der Parallel-Prefix-Berechnung half, eine logarithmische Prozessoranzahl einzusparen, ist auf Listen leider nicht so einfach anwendbar, denn von keinem Listenelement ist seine genaue Position innerhalb der Liste bekannt.

In [CV] beschreiben Cole und Viskin, wie sich List-Ranking auf einer EREW PRAM mit $O(n/\log n)$ Prozessoren in $O(\log n)$ Zeit durchführen läßt:
Dazu führen sie in einem ersten Schritt eine Listenkontraktion durch und wenden den oben beschriebenen Standard-List-Ranking-Algorithmus auf die verkürzte Liste an. Zum Schluß rekonstruieren sie die Ursprungsliste wieder und gewinnen dabei alle Suffix-Summen.

Eine exaktere Beschreibung des Algorithmus von Cole und Vishkin wäre an dieser Stelle zu aufwendig, so daß nur festgehalten wird:

Satz 3.10
Auf einer Liste der Größe n ist List-Ranking mit einer $O(n/\log n)$-Prozessor- EREW PRAM *in $O(\log n)$ Zeit und $O(n)$ Platz möglich.*

Eine wichtige Anwendung von List-Ranking wird in Kapitel 6.1 im Rahmen der *Euler-Tour-Technik* vorgestellt, so daß nun mit der parallelen Implementierung des Hauptalgorithmus begonnen werden kann.

Kapitel 4

Die parallele Implementierung

Nach den Vorbereitungen im letzten Kapitel wird nun der Algorithmus von Fischer und Simon Schritt für Schritt parallelisiert.

Dazu befinden sich die unsortierten Punktmengen POS und NEG als Arrays $(p'_0, p'_1, \ldots, p'_{n'-1})$ und $(q_0, q_1, \ldots, q_{m-1})$ im Global-Memory einer PRAM. Jedes Arrayelement p'_i und q_j besteht dabei aus seinem Index und der x- und y-Koordinate des von ihm repräsentierten Punktes. p'_i und q_j kann man z.B. als dreikomponentige Vektoren der Form $p'_i = (i, x(p'_i), y(p'_i))$ und $q_j = (j, x(q_j), y(q_j))$ implementieren.

4.1 Berechnung von BOUNDARY(POS)

Wie bereits früher erwähnt, läßt sich BOUNDARY(POS) durch eine Liste der im Uhrzeigersinn sortierten Eckpunkte von HULL(POS) charakterisieren.

Ist eine endliche Punktmenge S der Ebene gegeben, dann gehört die Aufgabe, die Eckpunkte von HULL(S) zu berechnen, zu einem Standardproblem der algorithmischen Geometrie. Dementsprechend viele Veröffentlichungen gibt es zu diesem Thema. Parallele Verfahren, die auf einer CREW PRAM arbeiten, stehen beispielsweise in [CG], [Go], [AG] oder [Ja].

Miller und Stout stellen in [MS] einen Algorithmus zur Berechnung der Eckpunkte von HULL(S) vor, der zunächst für parallele Computermodelle wie *Hypercube* und *Mesh-Computer* konzipiert ist. In einem zusätzlichen Kapitel beschreiben die Autoren dann, wie sich ihr $O(|S|)$-Prozessor-Hypercube-Algorithmus mit Laufzeit $O(\log^2 |S|)$ auf eine EREW PRAM portieren läßt, wodurch die Laufzeit des Algorithmus auf $O(\log |S|)$ sinkt.

Die Eingabe dieses Algorithmus besteht aus den (unsortierten) Punkten von S, die durch ihre x- und y-Koordinate gegeben sind. (O.B.d.A sind die x-Koordinaten aller Punkte aus POS durch eine entsprechende Wahl des Koordinatensystems paarweise verschieden.)

Als Ausgabe erzeugt der Algorithmus ein boolesches Array $extreme()$, mit $extreme(i) = true$ genau dann, wenn der i-te Punkt in S ein Eckpunkt von HULL(S) ist. Außerdem steht ein weiteres Array $pos()$ zur Verfügung, bei dem $pos(i)$ im Fall von $extreme(i) = true$ die Position dieses Eckpunktes bzgl. einer Sortierung der Eckpunkte im Uhrzeigersinn angibt.

Auf eine genauere Beschreibung des Algorithmus von Miller und Scout wird an dieser Stelle verzichtet. Festzuhalten bleibt, daß er auf einer EREW PRAM mit $O(|S|)$ Prozessoren eine parallele Laufzeit von $O(\log |S|)$ erzielt. Der Platzbedarf ist $O(|S|)$.

Wendet man diesen Algorithmus auf die Punktmenge POS an, so läßt sich BOUNDARY(POS) anschließend aus den Werten des $pos()$-Arrays ablesen:
Dazu setzt der i-te PRAM-Prozessor genau dann $p_j := p'_i$, wenn $extreme(i) = true$ und $pos(i) = j$ gilt. Auf diese Weise entsteht ein Array $(p_0, p_1, \ldots, p_{n-1})$, das die Eckpunkte von BOUNDARY(POS) im Uhrzeigersinn sortiert enthält. Der Wert von n kann mittels Parallel-Prefix-Computation berechnet werden, denn es gilt $n = 1 + \max\{pos(0), pos(1), \ldots, pos(n'-1)\}$.

Als Resultat erhält man:

Satz 4.1
Für eine Punktmenge POS *der Größe* n' *läßt sich* BOUNDARY(POS) *als Array* $p_0, p_1, \ldots, p_{n-1}$ *seiner im Uhrzeigersinn sortierten Eckpunkte auf einer* EREW PRAM *mit* $O(n')$ *Prozessoren in* $O(\log n')$ *Zeit und* $O(n')$ *Platz ermitteln.*

4.2 Ermittlung der Sichtbarkeitstangenten

In Kapitel 5 wird ein sequentieller Algorithmus vorgestellt, der die Sichtbarkeitstangenten eines Punktes q an ein konvexes Polygon mit n Ecken in $O(\log n)$ Zeit und $O(n)$ Platz berechnet.

Als Eingabe verlangt der Algorithmus die x- und y-Koordinaten von q und der im Uhrzeigersinn sortierten Eckpunkte $p_0, p_1, \ldots, p_{n-1}$ des konvexen Polygons. Die Ausgabe besteht aus zwei Punkten p_r und p_l, mit der Bedeutung, daß die rechte Sichtbarkeitstangente von q durch die Punkte q und p_r und die linke Sichtbarkeitstangente von q durch die Punkte q und p_l verläuft.

Verwendet man eine PRAM mit $O(m)$ Prozessoren, so läßt sich dieser Algorithmus sofort parallel auf die Punkte $q_0, q_1, \ldots, q_{m-1}$ aus NEG anwenden, indem ein Prozessor eine konstante Anzahl von Sichtbarkeitstangenten berechnet. Das konvexe Polygon ist durch die im letzten Kapitel ermittelten Eckpunkte von BOUNDARY(POS) gegeben. Als Ausgabe sind zwei Arrays $(p_{r_0}, p_{r_1}, \ldots, p_{r_{m-1}})$ und $(p_{l_0}, p_{l_1}, \ldots, p_{l_{m-1}})$ im Global-Memory der PRAM denkbar.

Aus diesen beiden Arrays wird in einem letzten Schritt ein „Sichtbarkeitstangentenarray" $(S_0, S_1, \ldots, S_{2m-1})$ gebildet:
Dazu belegt der j-te Prozessor der PRAM die Arrayelemente S_{2j} und S_{2j+1} mit den Vektoren $(q_j, p_{r_j}, right)$ und $(q_j, p_{l_j}, left)$.

Auf diese Weise ist von jeder Sichtbarkeitstangente S_k ihre Lage in der Ebene und ihr Bezugspunkt $q \in$ NEG ablesbar. Außerdem gibt die letzte Komponente Auskunft darüber, ob S_k eine rechte oder linke Sichtbarkeitstangente darstellt.

Bei der parallelen Anwendung des Algorithmus zur Berechnung der Sichtbarkeitstangenten kann es zwischen den PRAM-Prozessoren auf den Daten von BOUNDARY(POS) zu Lesekonflikten kommen. Schreibkonflikte treten dagegen an keiner Stelle auf.

Es folgt:

Satz 4.2
Sind die n Eckpunkte von BOUNDARY(POS) *im Uhrzeigersinn sortiert gegeben, dann kann eine* CREW PRAM *mit $O(m)$ Prozessoren die rechten und linken Sichtbarkeitstangenten für alle m Punkte $q \in$* NEG *in $O(\log n)$ Zeit und $O(n+m)$ Platz berechnen.*

4.3 Sortieren der Sichtbarkeitstangenten

Die soeben ermittelten $2m$ Sichtbarkeitstangenten $S_0, S_1, \ldots, S_{2m-1}$ sollen nun im Uhrzeigersinn sortiert werden. Dazu wurde im zweiten Kapitel ein Sweep-Line-Algorithmus vorgestellt.

Ziel ist es nun, die Sichtbarkeitstangenten derart umzusortieren, daß das neu entstehende Sichtbarkeitstangentenarray $(S'_0, S'_1, \ldots, S'_{2m-1})$ die Sichtbarkeitstangenten in der Reihenfolge enthält, wie der Sweep-Line-Algorithmus sie entdeckt.

Dabei ist allerdings ist beachten, daß es Sichtbarkeitstangenten S_i, S_j $(i \neq j)$ geben kann, die dieselbe Tangente T_{2k} des Tangentensystems $\mathcal{T}$ repräsentieren. Tangenten S_i und S_j mit dieser Eigenschaft heißen *kollinear*.

Kollineare Sichtbarkeitstangenten werden „gleichzeitig“ von dem Sweep-Line-Algorithmus entdeckt, und man hat die Wahl, in welcher Reihenfolge man S_i und S_j in das neue Tangentenarray einfügt.

Für die weiteren Betrachtungen erweist es sich aber als vorteilhaft, die Reihenfolge zweier beliebiger Tangenten S_k und $S_{k'}$ im neuen Sichtbarkeitstangentenarray mittels einer totalen Ordnungsrelation „$\ll$“ eindeutig wie folgt festzulegen:

$$S_k \ll S_{k'} :\Longleftrightarrow \begin{cases} & \text{Der Sweep-Line-Algorithmus findet } S_k \text{ vor } S_{k'}, \\ \textit{oder} & S_k \text{ und } S_{k'} \text{ sind kollinear, und } S_k \text{ ist linke und } S_{k'} \\ & \text{rechte Sichtbarkeitstangente,} \\ \textit{oder} & S_k \text{ und } S_{k'} \text{ sind kollinear, und } S_k \text{ und } S_{k'} \text{ sind bei-} \\ & \text{des rechte bzw. linke Sichtbarkeitstangenten und} \\ & k < k'. \end{cases}$$

Auf diese Weise gilt für zwei beliebige Sichtbarkeitstangenten S_k und $S_{k'}$ entweder $S_k \ll S_{k'}$ oder $S_{k'} \ll S_k$, und ein Sichtbarkeitstangentenarray $(S'_0, S'_1, \ldots, S'_{2m-1})$ mit $S'_0 \ll S'_1 \ll \ldots \ll S'_{2m-1}$ enthält die Tangenten in der gewünschten Reihenfolge.

Ziel ist es nun, eine kompakte mathematische Charakterisierung von „$S_k \ll S_{k'}$“ zu finden:

Von jeder Sichtbarkeitstangente $S_k = (q_j, p_i, f)$, $f \in \{right, left\}$, definiert man deshalb einen Hilfspunkt Q_k vektoriell als

$$Q_k := \begin{cases} q_j, & \text{falls } f = right, \\ 2p_i - q_j, & \text{falls } f = left. \end{cases}$$

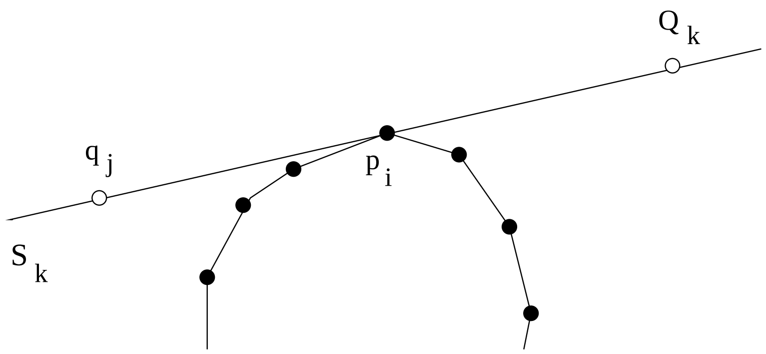


Bild 4.1: *Falls S_k eine linke Sichtbarkeitstangente repräsentiert, erhält man Q_k aus q_j durch Punktspiegelung an p_i.*

Daß S_k durch Q_k verläuft, ist offensichtlich. Insbesondere läßt sich S_k grundsätzlich als rechte Sichtbarkeitstangente von Q_k auffassen, wodurch sich bei den folgenden Betrachtungen einige Fallunterscheidungen vermeiden lassen:

Sei $S_k = (q_j, p_i, f)$ und $S_{k'} = (q_{j'}, p_{i'}, f')$ ($f, f' \in \{right, left\}$).

Setzt man $D(S_k, S_{k'}) := \Delta(Q_k, Q_{k'}, p_i)$, wobei Δ die in Kapitel 3 eingeführte Orientierungsdeterminante bezeichnet, dann gilt insbesondere $D(S_k, S_{k'}) = 0$ genau dann, wenn S_k und $S_{k'}$ kollinear sind.

Formal läßt sich „$S_k \ll S_{k'}$" mit Hilfe von $D(S_k, S_{k'})$ jetzt wie folgt ausdrücken:

$$S_k \ll S_{k'} \iff \begin{cases} \quad (i < i') & (1) \\ \vee \quad (i = i' \ \wedge \ D(S_k, S_{k'}) < 0) & (2) \\ \vee \quad (i = i' \ \wedge \ D(S_k, S_{k'}) = 0 \ \wedge \ f \neq f' \ \wedge \ f = left) & (3) \\ \vee \quad (i = i' \ \wedge \ D(S_k, S_{k'}) = 0 \ \wedge \ f = f' \ \wedge \ k < k') & (4) \end{cases}$$

(1) und (2) geben genau die Situation wieder, in denen der Sweep-Line-Algorithmus S_k vor $S_{k'}$ findet (vgl. Bild 4.2), während (3) und (4) den Fall charakterisieren, daß S_k und $S_{k'}$ kollinear sind und per Definitionem $S_k \ll S_{k'}$ gelten soll.

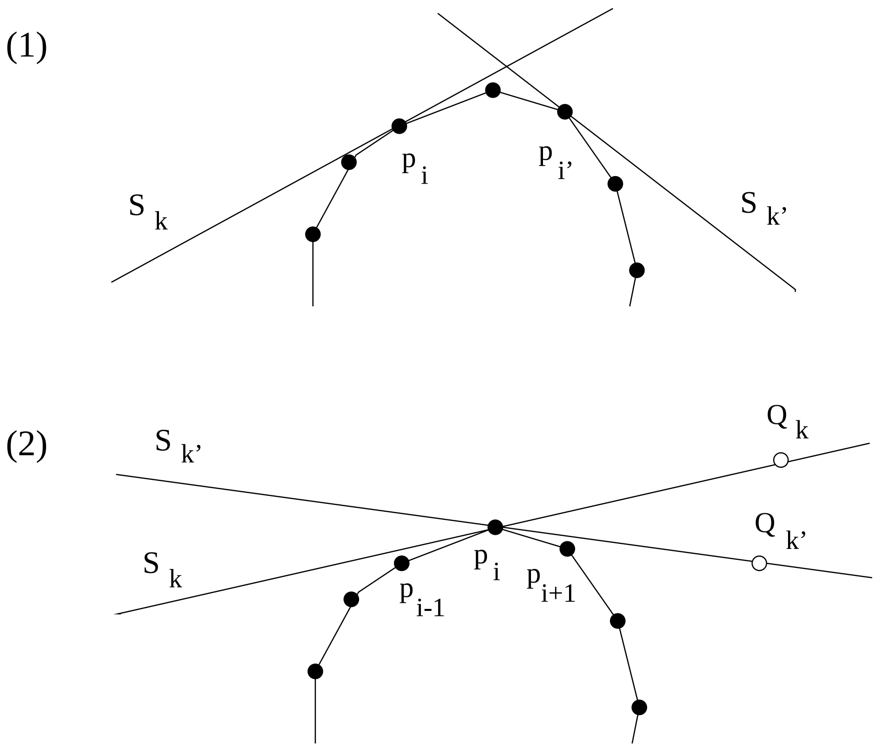


Bild 4.2: *Die Situationen (1) und (2).*

Mittels Pipelined-Parallel-Mergesort (vgl. [Co]) ist es jetzt möglich, die $2m$ Arrayelemente $S_0, S_1, \ldots, S_{2m-1}$ auf einer O(m)-Prozessor-EREW PRAM in $O(\log m)$ Zeit bzgl. „ $\ll$ " zu sortieren. Der Platzbedarf beträgt $O(m)$.

Als Ergebnis erhält man im Global-Memory der PRAM ein Tangentenarray $(S'_0, S'_1, \ldots, S'_{2m-1})$, das die Sichtbarkeitstangenten in der gewünschten Reihenfolge enthält.

Natürlich entspricht die Sortierung der Sichtbarkeitstangenten einer Sortierung bezüglich bestimmter Winkel. Durch die Einführung von $D(\cdot, \cdot)$ werden jedoch die in der Praxis teuren Winkelfunktionen vermieden.

Abschließend ordnet man jeder Tangente S'_i noch eine Variable $num(i)$ zu. Hat $num(i)$ den Wert $2k$, so bedeutet dies, daß S'_i zur Tangente T_{2k} des Tangentensystems $\mathcal{T}$ wird.

Kollineare Sichtbarkeitstangenten müssen denselben num-Wert erhalten. Solche Tangenten bilden in $(S'_0, S'_1, \ldots, S'_{2m-1})$ einen zusammenhängenden Block $(S'_i, S'_{i+1}, \ldots, S'_j)$, weshalb sich folgendes Verfahren zur $num()$-Berechnung anbietet:

Zuerst assoziiert man zu jeder Tangente S'_k, $k \in \{0, 1, \ldots, 2m-1\}$, das Gewicht

$$w(k) := \begin{cases} 0, & \text{falls } k = 0 \text{ oder } D(S'_{k-1}, S'_k) = 0, \\ 2, & \text{sonst.} \end{cases}$$

Dann ergibt sich $num(k)$ für alle $k \in \{0, 1, \ldots, 2m-1\}$ als $num(k) = \sum_{i=0}^{k} w(i)$. Diese Berechnung ist eine Aufgabe für Parallel-Prefix-Computation.

Beispiel: (Die Sichtbarkeitstangenten $S'_i, S'_{i+1}, \ldots, S'_j$ sind kollinear.)

S'	S'_0	S'_1	$\ldots$	S'_{i-1}	S'_i	S'_{i+1}	$\ldots$	S'_j	S'_{j+1}	$\ldots$	S'_{2m-1}
$w(i)$	0	2	$\ldots$	2	2	0	$\ldots$	0	2	$\ldots$	2
$num(i)$	0	2	$\ldots$	$2k-2$	$2k$	$2k$	$\ldots$	$2k$	$2k+2$	$\ldots$	$2(s-1)$
$\cong$	T_0	T_2	$\ldots$	T_{2k-2}	T_{2k}	T_{2k}	$\ldots$	T_{2k}	T_{2k+2}	$\ldots$	$T_{2(s-1)}$

Die Implementierung des Sweep-Line-Algorithmus besteht also im wesentlichen aus einem Sortierdurchlauf nach geschickt gewählten Kriterien. Die Identifizierung der Sichtbarkeitstangenten mit den Tangenten $T_0, T_2, \ldots, T_{2(s-1)}$ von $\mathcal{T}$ erfolgt daraufhin mittels eines einfachen Parallel-Prefix-Durchlaufs.

Man beachte, daß wegen $num(2m-1) = 2(s-1)$ der Wert von s für die weiteren Betrachtungen bekannt ist.

Bei der parallelen Implementierung auf einer PRAM kommt es zu keinerlei Schreibkonflikten. Lesekonflikte, beispielsweise auf den Daten des S'-Arrays bei der Auswertung der Determinante $D(S'_{k-1}, S'_k)$, lassen sich leicht vermeiden, indem in einem ersten Schritt alle S'_{k-1}-Daten und in einem zweiten Schritt alle S'_k-Daten gelesen werden.

Bearbeitet ein Prozessor $O(\log m)$ Arrayelemente, ergibt sich:

Satz 4.3
Die $2m$ Sichtbarkeitstangenten können in $O(\log m)$ Zeit und $O(m)$ Platz von einer EREW PRAM *mit $O(m)$ Prozessoren im Uhrzeigersinn sortiert werden.*

4.4 Ermittlung des Circular-Arc-Systems

Nachdem jede Sichtbarkeitstangente S_i' zu einer Tangente T_{2k} von $\mathcal{T}$ gemacht worden ist, bereitet die Ermittlung des zu POS und NEG assoziierten Circular-Arc-Systems keine größeren Schwierigkeiten mehr:

Das zu bildende Circular-Arc-System S besteht aus m Bögen $I_0, I_1, \ldots, I_{m-1}$, die durch ihre Anfangspunkte a_j und Endpunkte b_j, $j \in \{0, 1, \ldots, m-1\}$, repräsentiert werden.

Hat ein Punkt $q_j \in$ NEG als rechte Sichtbarkeitstangente die Tangente T_{2r} und als linke Sichtbarkeitstangente die Tangente T_{2l}, so sind a_j und b_j definiert als

$$a_j := (2r+1) \bmod 2s, \quad b_j := (2l-1) \bmod 2s.$$

Im Hinblick auf den nächsten Schritt der parallelen Implementierung erweist es sich als vorteilhaft, das Circular-Arc-System S in einem „Randpunktarray“ $R' = (R_0', R_1', \ldots, R_{2m-1}')$ zu repräsentieren. Ein Arrayelement R_k' enthält dabei den zur Sichtbarkeitstangente S_k' assoziierten Anfangspunkt a_i bzw. Endpunkt b_j in Form eines dreikomponentigen Vektors $(a_i, i, start)$ bzw. (b_j, j, end).

Mit Hilfe des $num()$-Arrays läßt sich R' leicht ermitteln:

> Inspiziere alle Sichtbarkeitstangenten $S_0', S_1', \ldots, S_{2m-1}'$:
> Ist S_k' eine rechte Sichtbarkeitstangente, also vom Typ $(q_j, p_i, right)$, so setze $R_k' := ((num(j)+1) \bmod 2s, j, start)$.
> Andernfalls ist S_k' eine linke Sichtbarkeitstangente und somit vom Typ $(q_j, p_i, left)$. Setze $R_k' := ((num(j)-1) \bmod 2s, j, end)$.

Berechnet ein Prozessor einer PRAM aus logarithmisch vielen Sichtbarkeitstangenten S_k' die entsprechenden Elemente R_k' und schreibt sie in das Global-Memory, so kommt es weder zu Schreib- noch zu Lesekonflikten, und es ergibt sich:

Satz 4.4
Das zu POS *und* NEG *assoziierte Circular-Arc-System S läßt sich mit den $O(m/\log m)$ Prozessoren einer* EREW PRAM *in $O(\log m)$ Zeit und $O(m)$ Platz ermitteln.*

4.5 Berechnung einer minimalen Transversale

Ein paralleler Algorithmus zur Berechnung einer minimalen Transversale auf einem Circular Arc-System wird in Kapitel 6 ausführlich vorgestellt. Bei einem Circular-Arc-System mit m paarweise verschiedenen Bögen benötigt dieser Algorithmus auf einer $O(m/\log m)$-Prozessor-EREW PRAM eine Laufzeit von $O(\log m)$ und einen Platzbedarf von $O(m)$.

(Man beachte, daß in dem zu POS und NEG assoziierten Circular-Arc-System tatsächlich alle Bögen paarweise verschieden sind, denn o.B.d.A enthält NEG keine doppelten Punkte.)

Als Eingabe verlangt der Transversalen-Algorithmus ein Randpunktarray $R = (R_0, R_1, \ldots, R_{2m-1})$, das die $2m$ Anfangs- und Endpunkte in aufsteigend sortierter Form enthält. Dabei sollen Anfangspunkte a_i grundsätzlich vor Endpunkten b_j einsortiert sein, falls $a_i = b_j$ gilt.

Eine Komponente R_k von R muß neben dem Wert des repräsentierten Randpunktes auch noch den Index des zugehörigen Bogens, sowie ein Flag, ob es sich bei dem Randpunkt um einen Anfangs- oder Endpunkt handelt, beinhalten.

Wie sich zeigt, erfüllt das im letzten Schritt ermittelte Array R' schon fast alle diese Vorgaben:

Daß ein Element R'_k aus R' die benötigten Informationen bzgl. Randpunktwert, Bogenindex und Anfangs- bzw. Endpunkt bereitstellt, ist offensichtlich; nicht so klar hingegen ist, daß die Randpunkte in R' bereits (fast) in der Weise sortiert sind, wie es die Eingabe des Transversalen-Algorithmus verlangt.

Dazu betrachte man die folgende Fallunterscheidung, in der für zwei beliebige aufeinanderfolgende Sichtbarkeitstangenten S'_j und S'_{j+1}, $j \in \{0, 1, \ldots, 2m-2\}$, die resultierenden Werte von R'_j und R'_{j+1} verglichen werden:

1. S'_j und S'_{j+1} sind <u>nicht</u> kollinear und repräsentieren die Tangenten T_{2k} und T_{2k+2}, d.h. $num(j) = 2k$ und $num(j+1) = 2k+2$.

(a)	$S'_j = (\cdot, \cdot, right)$ $R'_j = ((2k+1) \bmod 2s, \cdot, start)$	$S'_{j+1} = (\cdot, \cdot, right)$ $R'_{j+1} = ((2k+3) \bmod 2s, \cdot, start)$
(b)	$S'_j = (\cdot, \cdot, right)$ $R'_j = ((2k+1) \bmod 2s, \cdot, start)$	$S'_{j+1} = (\cdot, \cdot, left)$ $R'_{j+1} = ((2k+1) \bmod 2s, \cdot, end)$
(c)	$S'_j = (\cdot, \cdot, left)$ $R'_j = ((2k-1) \bmod 2s, \cdot, end)$	$S'_{j+1} = (\cdot, \cdot, right)$ $R'_{j+1} = ((2k+3) \bmod 2s, \cdot, start)$
(d)	$S'_j = (\cdot, \cdot, left)$ $R'_j = ((2k-1) \bmod 2s, \cdot, end)$	$S'_{j+1} = (\cdot, \cdot, left)$ $R'_{j+1} = ((2k+1) \bmod 2s, \cdot, end)$

2. S'_j und S'_{j+1} sind kollinear und repräsentieren beide die Tangente T_{2k}, d.h. $num(j) = num(j+1) = 2k$. Man beachte, daß in diesem Fall auf Grund der Definition von „$\ll$" niemals S'_j eine rechte und S'_{j+1} eine linke Sichtbarkeitstangente sein können.

(a)	$S'_j = (\cdot, \cdot, right)$ $R'_j = ((2k+1) \bmod 2s, \cdot, start)$	$S'_{j+1} = (\cdot, \cdot, right)$ $R'_{j+1} = ((2k+1) \bmod 2s, \cdot, start)$
(b)	$S'_j = (\cdot, \cdot, left)$ $R'_j = ((2k-1) \bmod 2s, \cdot, end)$	$S'_{j+1} = (\cdot, \cdot, right)$ $R'_{j+1} = ((2k+1) \bmod 2s, \cdot, start)$
(c)	$S'_j = (\cdot, \cdot, left)$ $R'_j = ((2k-1) \bmod 2s, \cdot, end)$	$S'_{j+1} = (\cdot, \cdot, left)$ $R'_{j+1} = ((2k-1) \bmod 2s, \cdot, end)$

Man sieht, daß falls ein Anfangs- und Endpunkt zusammenfallen, der Anfangspunkt vor dem Endpunkt in R' eingeordnet ist (Fall 1(b)).

Außerdem gibt es nur eine Konstellation, in der R'_j und R'_{j+1} nicht aufsteigend sortiert sind, nämlich die, daß eine linke Sichtbarkeitstangente S'_j in den Fällen 1(c), 1(d) oder 2(b) die Tangente T_0 repräsentiert. Auf Grund der *modulo*-Rechnung stellt R'_j dann den Randpunkt $2s-1$ und R'_{j+1} den Randpunkt 1 oder 3 dar.

Ein korrekt sortiertes Randpunktarray R entsteht jedoch durch eine „zyklische Linksverschiebung" von R': Dazu löscht man zunächst alle Arrayelemente R'_j vom Typ $R'_j = (2s-1, \cdot, end)$ aus R' und hängt sie anschließend wieder an R' an.

(Man beachte, daß nun auch alle Anfangspunkte mit Wert $(2s-1)$ vor Endpunkten mit Wert $(2s-1)$ einsortiert sind.)

Die Ermittlung von R ist eine Anwendung des „Umsortierproblems" in Kapitel 3.2.3 und läßt sich somit auf einer EREW PRAM mit $O(m/\log m)$ Prozessoren in $O(\log m)$ Zeit durchführen.

Damit sind die Voraussetzungen für den Einsatz des Transversalen-Algorithmus in Kapitel 6 erfüllt, und es ergibt sich:

Satz 4.5
Eine EREW PRAM *mit* $O(m/\log m)$ *Prozessoren findet in einer Laufzeit von* $O(\log m)$ *eine minimale Transversale in dem zu* POS *und* NEG *assoziierten Circular-Arc-System* S. *Der Platzbedarf beträgt* $O(m)$.

4.6 Ermittlung und Ausgabe der Zwischentangenten

Die Ausgabe des parallelen Transversalen-Algorithmus besteht aus einem Array $M = (M_0, M_1, \ldots, M_{t-1})$, wobei M_i den Wert des i-ten Transversalenelementes und t die Größe der Transversale angibt.

Dies bedeutet, daß jede Tangente T_{M_i} des Tangentensystems $\mathcal{T}$ eine Kante des minimalen konvexen separierenden Lösungspolygons darstellt.

Nach Lemma 6.7 gibt es immer eine minimale Transversale, die nur aus Bogenendpunkten besteht, und eine solche Transversale ermittelt auch der Algorithmus in Kapitel 6. Auf Grund der Definition des zu POS und NEG assoziierten Circular-Arc-Systems besitzt jeder Bogenendpunkt b_j einen ungeraden Wert $(2k+1)$ und repräsentiert damit eine Zwischentangente T_{2k+1}.

Es gibt genau s Zwischentangenten $T_1, T_3, \ldots, T_{2s-1}$, deren exakte Lage in der Ebene bis jetzt noch nicht festgelegt wurde.

Um einen einfacheren Zugriff auf die für das Lösungspolygon tatsächlich benötigten Zwischentangenten zu haben, wird in einem ersten Schritt ein Array *isEdge*() der Größe s erzeugt. Für ein Element *isEdge*(k) soll gelten:

$$\mathit{isEdge}(k) \;=\; \begin{cases} i\,(\geq 0), & \text{falls } T_{2k+1} \text{ die } i\text{-te Kante des Lösungspolygons bildet,} \\ -1, & \text{falls } T_{2k+1} \text{ keine Kante des Lösungspolygons bildet.} \end{cases}$$

Deshalb initialisiert man zunächst alle s Einträge von *Edge*() mit dem Wert -1 und setzt anschließend $\mathit{isEdge}((M_i - 1)/2) := i$ für alle $i \in \{0, 1, \ldots, t-1\}$.

Ziel ist nun die Ermittlung eines Arrays *Edge*() der Größe t. Ein Arrayelement *Edge*(i) besteht dabei aus einem Vektor (p, q), wobei $p = (x(p), y(p))$ und $q = (x(q), y(q))$ zwei Punkte der Ebene definieren, durch die die Zwischentangente T_{M_i} verläuft.

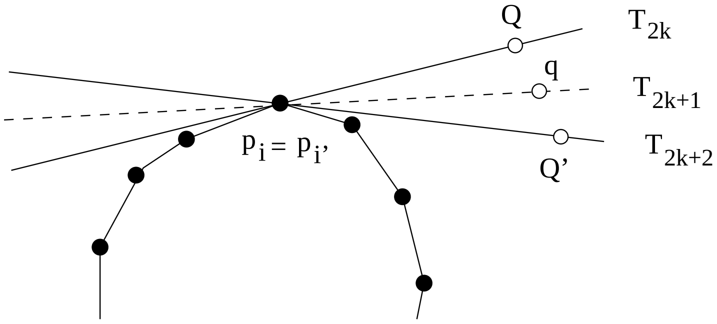
Im sortierten Sichtbarkeitstangentenarray S' existieren zu jeder Zwischentangente T_{2k+1} zwei aufeinanderfolgende Elemente $S'_{k'}$ und $S'_{(k'+1)\mathrm{mod}2m}$, die die Tangenten T_{2k} und $T_{(2k+2)\mathrm{mod}2s}$ repräsentieren, d.h. es gilt $num(k') = 2k$ und
$num((k'+1) \bmod 2m) = (2k+2) \bmod 2s$ (vgl. das Beispiel auf Seite 32).

Sei $S'_{k'} = (q_j, p_i, f)$ und $S'_{(k'+1)\mathrm{mod}2m} = (q_{j'}, p_{i'}, f')$ $(f, f' \in \{right, left\})$.

Aus den Punkten q_j, p_i, $q_{j'}$ und $p_{i'}$ lassen sich p und q, die entsprechenden Punkte einer Zwischentangente T_{2k+1}, wie folgt ermitteln:

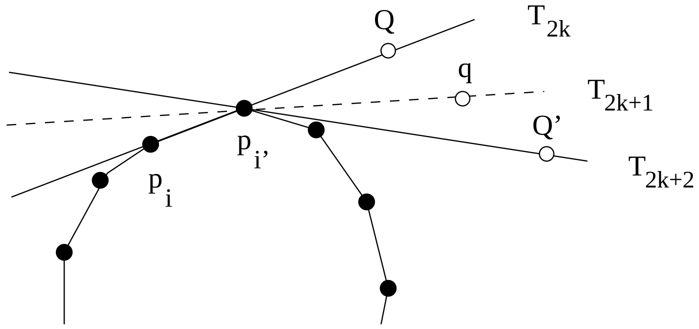
(Man beachte, daß von allen Punkten q_j, p_i, $q_{j'}$ und $p_{i'}$ neben ihrer x- und y-Koordinate auch ihr Index in den Datenstrukturen von $S'_{k'}$ bzw. $S'_{(k'+1)\mathrm{mod}2m}$ enthalten ist.)

Fall 1: $i = i'$.



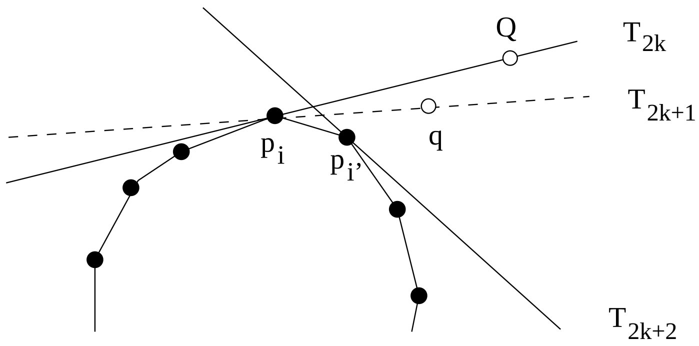
Als T_{2k+1} bietet sich eine Tangente durch die Punkte $p := p_i$ und $q := (Q + Q')/2$ an.
(Dabei bezeichnen Q und Q' die in Kapitel 4.3 beschriebenen Hilfspunkte auf den Sichtbarkeitstangenten $S'_{k'} = T_{2k}$ und $S'_{(k'+1) \bmod 2m} = T_{(2k+2) \bmod 2s'}$.)

Fall 2(a): $(i + 1 = i')$ und $(\Delta(Q, p_i, p_{i'}) = 0)$, d.h. Q, p_i und $p_{i'}$ sind kollinear.



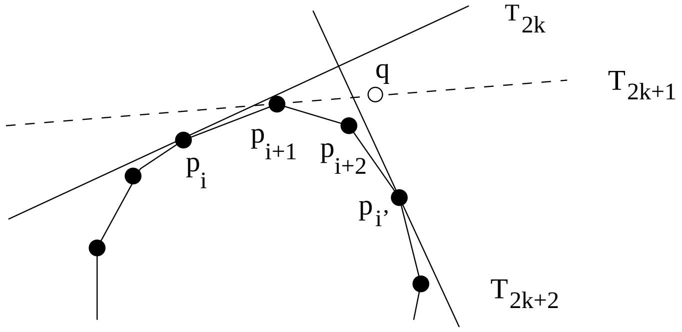
Ähnlich wie in Fall 1 kann man T_{2k+1} als Tangente durch die Punkte $(p, q) := (p_{i'}, (Q + Q')/2)$ wählen.

Fall 2(b): $(i + 1 = i')$ und $(\Delta(Q, p_i, p_{i'}) \neq 0)$.



Wähle T_{2k+1} als Tangente durch die Punkte $(p, q) := (p_i, (Q + p_{i'})/2)$.

Fall 3: $(i \neq i')$ und $(i + 1 \neq i')$.



Ansonsten definiere T_{2k+1} als Tangente durch die Punkte
$(p,\, q) := (p_{i+1},\, p_{i+1} + (p_{i+2} - p_i)/2)$.

(Auch hier wurde bewußt auf die Ermittlung irgendwelcher Winkel verzichtet. Die Berechnungsvorschriften für die Punkte p und q sind vektoriell und die Indexangaben bei den Punkten aus BOUNDARY(POS) „*modulo n*“ zu verstehen.)

Das komplette Kantenarray $Edge()$ ergibt sich, indem man alle Arrayelemente $S'_{k'}$ aus S' inspiziert:
Gilt $num(k') \neq num((k'+1) \bmod 2m)$, dann muß geprüft werden, ob $T_{num(k')+1}$ eine auszugebende Zwischentangente ist, also $i := isEdge(num(k')/2) \geq 0$ gilt.
In diesem Fall berechnet man die Punkte p und q von $Edge(i)$ aus $S_{k'}$ und $S_{(k'+1) \mathrm{mod} 2m}$ wie angegeben.

Bearbeitet ein Prozessor $O(\log m)$ Arrayelemente, so reicht für alle Schritte in diesem Abschnitt eine $O(m/\log m)$-Prozessor-PRAM aus, um eine Laufzeit von $O(\log m)$ nicht zu überschreiten.

Daß es zu keinen Schreibkonflikten kommt, ist klar.
Lesekonflikten auf den $num()$- und S'-Arrays kann mit entsprechenden globalen Lesephasen wie in Kapitel 4.3 begegnet werden.

Zur Ermittlung der Punkte p und q werden in Fall 3 die Punkte p_{i+1} und p_{i+2} benötigt. Die Koordinaten dieser Punkte stehen im allgemeinen nicht in den Datenstrukturen von $S'_{k'}$ oder $S'_{(k'+1) \mathrm{mod} 2m}$. Sie müssen also stattdessen aus dem BOUNDARY(POS)-Array $(p_0,\, p_1,\, \ldots,\, p_{n-1})$ bezogen werden, und es stellt sich die Frage, ob es bei der parallelen Ermittlung des $Edge()$-Arrays zu Lesekonflikten auf den Daten dieses Arrays kommen kann.
Dies ist jedoch auf Grund der Sortierung der Sichtbarkeitstangenten ausgeschlossen.

Es folgt:

Satz 4.6
Eine EREW PRAM *mit* $O(m/\log m)$ *Prozessoren kann die Kanten eines minimalen konvexen separierenden Polygons bzgl. der Punktmengen* POS *und* NEG *in* $O(\log m)$ *Zeit ermitteln und als Kantenarray ausgeben. Der Platzbedarf beträgt* $O(m)$.

Kapitel 5

Ermittlung von Sichtbarkeitstangenten

In diesem Kapitel wird ein sequentieller Algorithmus vorgestellt, der die rechte und linke Sichtbarkeitstangente eines Punktes q an ein konvexes Polygon POL mit n Ecken in $O(\log n)$ Zeit ermittelt. Einziges Hilfsmittel wird die in Kapitel 3.1 eingeführte Orientierungsdeterminante sein, was bedeutet, daß keinerlei trigonometrische Funktionen verwendet werden müssen. Genau wie bei einem Verfahren in [PS] wird die logarithmische Laufzeit durch eine *Divide-And-Conquer*-Methode erzielt, jedoch benötigt der Algorithmus in [PS] mehr Fallunterscheidungen pro Divide-Schritt.

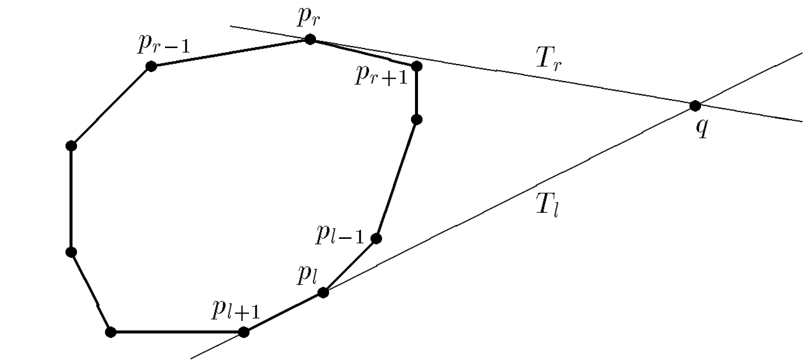
Folgende **Generalvoraussetzungen** sollen von nun an gelten:

- $p_0, p_1, \ldots, p_{n-1}$ ($n \geq 3$) bezeichnen die *im Uhrzeigersinn sortierten* Eckpunkte von POL und q einen beliebigen, aber fest gewählten Punkt außerhalb von POL. Von allen Punkten sei ihre x- und y-Koordinate bekannt.
- Aus Gründen der Übersichtlichkeit wird bei jedem Punkt p_i bei der Indexangabe auf den Ausdruck *„modulo n“* verzichtet, d.h. p_i repräsentiert den Punkt $p_{i \bmod n}$.

Stellt man sich das konvexe Polygon POL als nicht transparent vor, dann ist für q ein Teil seiner Eckpunkte $p_0, p_1, \ldots, p_{n-1}$ *sichtbar* und ein anderer Teil *nicht sichtbar*.

Die rechte Sichtbarkeitstangente T_r und die linke Sichtbarkeitstangente T_l von q sind genau so definiert, daß sie die Punkte $p_0, p_1, \ldots, p_{n-1}$ disjunkt in eine Menge $p_r, p_{r+1}, \ldots, p_l$ sichtbarer und eine Menge $p_{l+1}, p_{l+2}, \ldots, p_{r-1}$ nicht sichtbarer Punkte zerteilen, wobei T_r durch q und p_r und T_l durch q und p_l verläuft.

Die Punkte p_r und p_l heißen dementsprechend *Stützpunkte* der rechten bzw. linken Sichtbarkeitstangente von q.



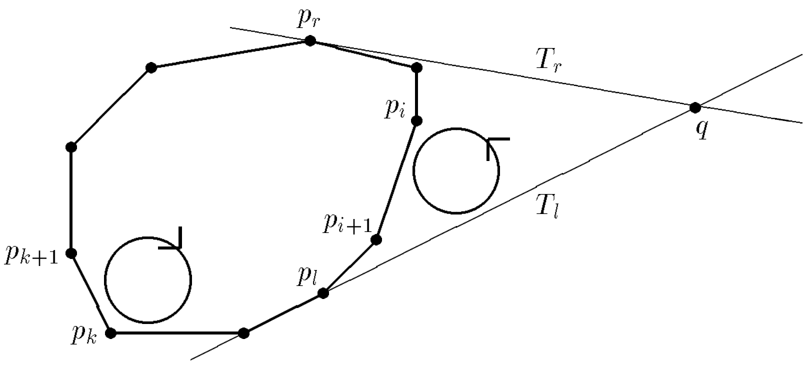
Ziel eines Algorithmus zur Berechnung der Sichtbarkeitstangenten muß es also sein, die Stützpunkte p_r und p_l zu ermitteln, da die Lagen von T_r und T_l in der Ebene durch die Punktpaare (q, p_r) bzw. (q, p_l) eindeutig charakterisiert sind.

Auf Grund der Sortierung und der Konvexität von POL folgt:

Lemma 5.1

(i) *Für alle* $i \in \{r, (r+1) \bmod n, \ldots, (l-1) \bmod n\}$ *gilt:*
(q, p_i, p_{i+1}) *ist Linkssystem*

(ii) *Für alle* $k \in \{l, (l+1) \bmod n, \ldots, (r-1) \bmod n\}$ *gilt:*
(q, p_k, p_{k+1}) *ist Rechtssystem oder kollinear*



Damit lassen sich die Stützpunkte p_r und p_l in eindeutiger Weise wie folgt beschreiben:

Lemma 5.2

$$\text{(i)}\quad p_r = p_i \iff \begin{cases} & (q,\, p_{i-1},\, p_i) \textit{ ist Rechtssystem oder kollinear} \\ \wedge & (q,\, p_i,\, p_{i+1}) \textit{ ist Linkssystem} \end{cases}$$

$$\text{(ii)}\quad p_l = p_i \iff \begin{cases} & (q,\, p_{i-1},\, p_i) \textit{ ist Linkssystem} \\ \wedge & (q,\, p_i,\, p_{i+1}) \textit{ ist Rechtssystem oder kollinear} \end{cases}$$

Mit 3.3, dem Satz über die Orientierungsdeterminante folgt:

Lemma 5.3

$$\text{(i)}\quad p_r = p_i \iff (\Delta(q,\, p_{i-1},\, p_i) \leq 0 \quad \wedge \quad \Delta(q,\, p_i,\, p_{i+1}) > 0)$$

$$\text{(ii)}\quad p_l = p_i \iff (\Delta(q,\, p_{i-1},\, p_i) > 0 \quad \wedge \quad \Delta(q,\, p_i,\, p_{i+1}) \leq 0)$$

Ein trivialer Algorithmus zur Berechnung der Sichtbarkeitstangenten testet also einfach für alle Punkte p_i, ob die Orientierungsdeterminanten der Punkttripel $(q,\, p_{i-1},\, p_i)$ und $(q,\, p_i,\, p_{i+1})$ die Voraussetzungen für $p_r = p_i$ oder $p_l = p_i$ erfüllen.

Im ersten Fall ist die Gerade durch die Punkte q und p_i die rechte, im zweiten Fall die linke Sichtbarkeitstangente von q.

Während dieser triviale Algorithmus noch $O(n)$ Zeit benötigt, bildet das folgende Lemma die Basis für einen Algorithmus mit logarithmischer Laufzeit:

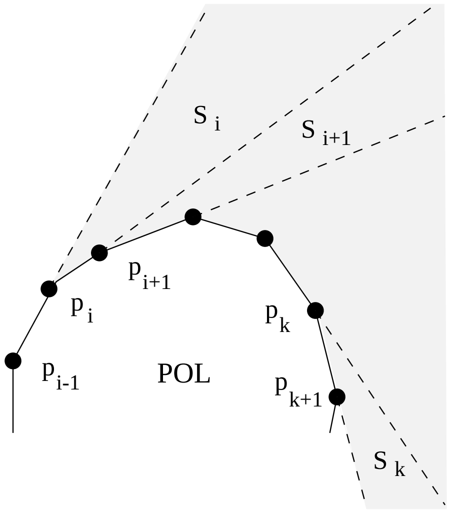
Lemma 5.4
Für alle $i,\, k \in \{0, 1, \ldots, n-1\}$ *gilt:*

$$\text{(i)}\quad p_r \in \{p_i,\, p_{i+1}, \ldots, p_k\}$$
$$\iff \begin{cases} & (\ \Delta(q,\, p_i,\, p_k) > 0 \quad \wedge \quad \Delta(q,\, p_{i-1},\, p_i) \leq 0\ ) \\ \vee & (\ \Delta(q,\, p_i,\, p_k) \leq 0 \quad \wedge \quad \Delta(q,\, p_k,\, p_{k+1}) > 0\ ) \end{cases}$$

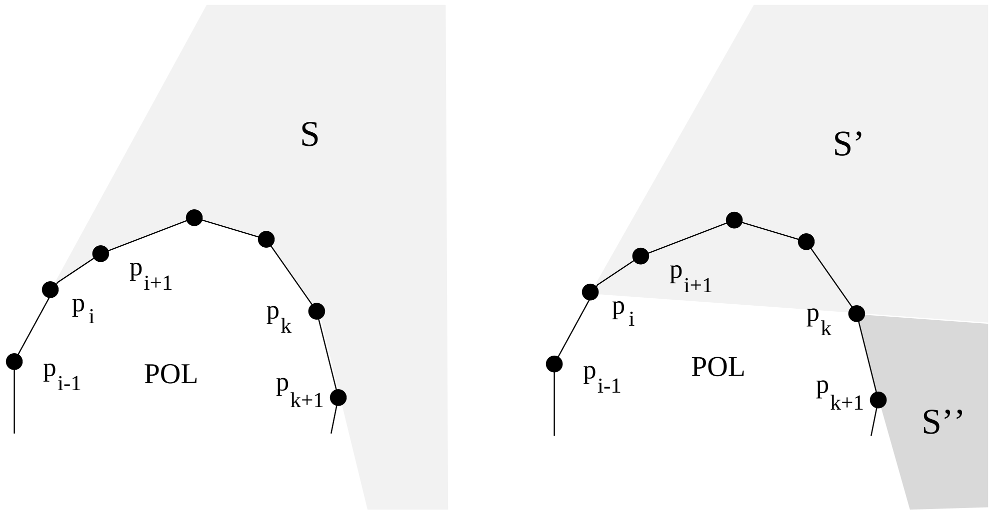
$$\text{(ii)}\quad p_l \in \{p_i,\, p_{i+1}, \ldots, p_k\}$$
$$\iff \begin{cases} & (\ \Delta(q,\, p_i,\, p_k) \leq 0 \quad \wedge \quad \Delta(q,\, p_{i-1},\, p_i) > 0\ ) \\ \vee & (\ \Delta(q,\, p_i,\, p_k) > 0 \quad \wedge \quad \Delta(q,\, p_k,\, p_{k+1}) \leq 0\ ) \end{cases}$$

Beweis:
Gezeigt wird nur Behauptung (i), Behauptung (ii) folgt analog:

Bezeichnet S_j den Sektor der Ebene in dem q liegt, damit $p_j = p_r$ gilt, d.h. $S_j = \{\, s \in \Re^2 \,|\, \Delta(s, p_{j-1}, p_j) \le 0 \;\wedge\; \Delta(s, p_j, p_{j+1}) > 0 \,\}$ (vgl. 5.3), dann gilt: $p_r \in \{p_i, p_{i+1}, \ldots, p_k\} \iff q \in \bigcup_{j=i}^{k} S_j =: S$.



Setzt man weiterhin $S' := \{\, s \in \Re^2 \,|\, \Delta(s, p_i, p_k) > 0 \;\wedge\; \Delta(s, p_{i-1}, p_i) \le 0\}$ und $S'' := \{\, s \in \Re^2 \,|\; \Delta(s, p_i, p_k) \le 0 \;\wedge\; \Delta(s, p_k, p_{k+1}) > 0\}$, so folgt auf Grund der Sortierung und Konvexität von POL $S \subset S' \cup S''$ und somit $q \in S \;\Rightarrow\; q \in S' \cup S''$.



Da $(S' \cup S'') \setminus S \subset$ POL und $q \notin$ POL gilt auch die Rückrichtung, d.h. $q \in S' \cup\; S'' \;\Rightarrow\; q \in S$.

Insgesamt ergibt sich $p_r \in \{p_i, p_{i+1}, \ldots, p_k\} \iff q \in S \iff q \in S' \cup S''$.
Dies ist die Behauptung (i). $\square$

Lemma 5.4 impliziert sofort den folgenden Divide-And-Conquer-Algorithmus zur Ermittlung von z.B. p_r, dem Stützpunkt der rechten Sichtbarkeitstangente von q:

Algorithmus „Rechte Sichtbarkeitstangente"

Eingabe: Die im Uhrzeigersinn sortierten Eckpunkte p_0, p_1, ..., p_{n-1} eines konvexen Polygons POL und ein Punkt q außerhalb von POL.

Ausgabe: Der Stützpunkt der rechten Sichtbarkeitstangente von q.

Schritt 1: Setze $i := 0$ und $j := n - 1$.

Schritt 2: Gilt $i = j$, dann gebe p_i aus und stoppe.

Schritt 3: Berechne k, den Index des mittleren Punktes aus $p_i, p_{i+1}, \ldots, p_j$, d.h. $k := i + \lfloor\, |\{p_i, p_{i+1}, \ldots, p_j\}| \,/\, 2 \,\rfloor$.

Schritt 4: Teste mit Hilfe von Lemma 5.4, ob p_r in $\{p_i, p_{i+1}, \ldots, p_k\}$ liegt, und setze in diesem Fall $j := k$. Ansonsten setze $i := k + 1$.

Schritt 5: Fahre fort mit Schritt 2.

Dieser Algorithmus berechnet in $O(\log n)$ Schritten den Stützpunkt p_r, wie die folgenden Überlegungen zeigen:

Bezeichnet $M^{(s)}$ die Punktmenge $\{p_i, p_{i+1}, \ldots, p_j\}$ beim s-ten Aufruf von Schritt 2, dann gilt $p_r \in M^{(1)} = \{p_0, p_1, \ldots, p_{n-1}\}$ und mit Lemma 5.4 folgt induktiv $p_r \in M^{(s)}$ für alle $s > 1$.

Auf Grund der Definition von k gilt außerdem $|M^{(s)}| \leq \lceil\, |M^{(s-1)}| \,/\, 2 \,\rceil$.

Wegen $|M^{(1)}| = n$ existiert also ein $s_0 \leq \lceil \log_2(n) \rceil + 1$ mit $|M^{(s_0)}| = 1$. In diesem Fall gilt $p_i = p_r$, und wegen $i = j$ gibt der Algorithmus den Punkt p_r aus.

Völlig analog läßt sich p_l, der Stützpunkt der linken Sichtbarkeitstangente von q ermitteln, und es folgt:

Satz 5.5
Die rechte und linke Sichtbarkeitstangente eines Punktes q an ein konvexes Polygon POL können in $O(\log n)$ Zeit und $O(n)$ Platz ohne Verwendung trigonometrischer Funktionen ermittelt werden, wenn die n Eckpunkte von POL in sortierter Form vorliegen.

Kapitel 6

Ein optimaler paralleler Algorithmus zur Berechnung einer minimalen Transversale in einem Circular-Arc-System

In diesem Kapitel wird ein optimaler paralleler Algorithmus entwickelt, der eine minimale Transversale eines Circular-Arc-Systems S, bestehend aus m Bögen I_j mit Anfangspunkt a_j und Endpunkt b_j, $j \in \{0, 1, \ldots, m-1\}$, in $O(\log m)$ Zeit auf einer EREW PRAM mit $O(m/\log m)$ Prozessoren ermittelt.

Als Grundlage dazu dienen hauptsächlich ein in [RR] vorgestellter paralleler Algorithmus zur Berechnung einer minimalen Cliquen-Überdeckung in einem Circular-Arc-Graphen, sowie einige theoretische Überlegungen, ähnlich denen in [FS].

Bei der eigentlichen parallelen Implementierung wird neben den bereits früher eingeführten Algorithmen *Parallel-Prefix-Computation* und *List-Ranking* die sogenannte *Euler-Tour-Technik* von Bedeutung sein, die im folgenden vorgestellt wird:

6.1 Die Euler-Tour-Technik

Die Idee vieler Algorithmen, gerade in der algorithmischen Geometrie, besteht darin, die gestellte Aufgabe in ein graphentheoretisches Problem zu überführen und statt der eigentlichen Aufgabe das äquivalente graphentheoretische Problem zu lösen. Unter anderem deswegen ist es wichtig, effiziente parallele Algorithmen für den Bereich der Graphentheorie zu besitzen.

Eine wichtige Graphenklasse ist die Klasse der Bäume, aber nicht sofort ist zu erkennen, auf welche Weise sich grundlegende Baumfunktionen wie z.B. Depth-First-Search-Numerierung, Ausgabe des Pfades von einem Knoten zur Baumwurzel usw. effizient parallel implementieren lassen.

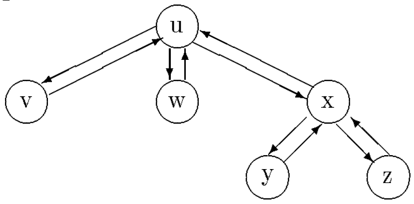
Eine Lösung bietet die in [TV] und [KR] beschriebene *Euler-Tour-Technik*, die die Berechnung vieler elementarer Baumfunktionen auf ein List-Ranking-Problem reduziert:

Definition 6.1
Eine *Euler-Tour* durch einen gerichteten Graphen G ist ein geschlossener Weg durch G, der alle Kanten aus G enthält. Ein Graph, in dem eine Euler-Tour existiert, heißt *Euler-Graph.*

Ist ein ungerichteter Baum T gegeben, so läßt sich dieser in einen gerichteten Eulergraphen G_T überführen, indem man jede ungerichtete Kante $\{x, y\}$ durch zwei gerichtete Kanten (x, y) und (y, x) ersetzt.

Hat T als Wurzelelement den Knoten u, dann ist leicht zu erkennen, daß eine Euler-Tour ET durch G_T, die in u startet, einen Depth-First-Search-Durchlauf durch T repräsentiert.

Beispiel: G_T :



$ET = (u, v), (v, u), (u, w), (w, u), (u, x), (x, y), (y, x), (x, z), (z, x), (x, u)$

In ET tritt jede ungerichtete Kante $\{x, y\}$ zweimal auf, einmal als (x, y), ein anderes Mal als (y, x). Steht (x, y) vor (y, x), so ist x der Vater von y in T, andernfalls ist y der Vater von x. Somit läßt sich die „Vater-Sohn"-Beziehung in T durch List-Ranking ermitteln (vgl. Kapitel 3.4):

$$x \text{ ist Vaterknoten von } y \text{ in } T \iff \mathit{Rang}(x, y) > \mathit{Rang}(y, x)$$

Eine Kante (x, y), wobei x der Vater von y ist, repräsentiert eine *Vorwärtskante* und (y, x) entsprechend eine *Rückwärtskante* in einem Depth-First-Search-Durchlauf.

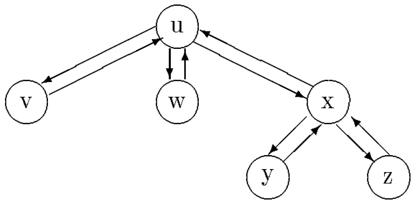
Doch wie erhält man bei gegebenem Baum T einen Eulerweg ET durch den korrespondierenden Eulergraphen G_T?

Dazu benötigt man für jeden Knoten aus G_T eine Adjazenzliste aller von ihm ausgehenden Kanten. Auf diese Weise befindet sich die ungerichtete Kante $\{x, y\}$ einerseits als (x, y) in der Adjazenzliste von x und andererseits als (y, x) in der Adjazenzliste von y. Voraussetzung ist, daß von jeder Kante (x, y) ein Zeiger $inv()$ zu ihrem inversen Pendant (y, x) existiert, so daß $inv(x, y) = (y, x)$.

Eine Euler-Tour ET durch G_T ergibt sich nun einfach, indem man zu jeder Kante (x, y) aus G_T ihre Nachfolgekante $nexttour(x, y)$ in ET definiert als $nexttour(x, y) := succ(inv(x, y))$.

Dabei bezeichnet $succ()$, angewandt auf eine Kante (y, x), die nachfolgende Kante in der Adjazenzliste von y. Ist (y, x) die letzte Kante in der Adjazenzliste, so ergibt $succ(y, x)$ entsprechend die erste Kante dieser Liste. $succ()$ realisiert somit eine zyklische Verknüpfung aller Elemente einer Adjazenzliste.

Beispiel: G_T :



Knoten	Adjazenzliste	$inv()$	$succ()$	$nexttour()$
u	(u, v)	(v, u)	(u, w)	(v, u)
	(u, w)	(w, u)	(u, x)	(w, u)
	(u, x)	(x, u)	(u, v)	(x, y)
v	(v, u)	(u, v)	(v, u)	(u, w)
w	(w, u)	(u, w)	(w, u)	(u, x)
x	(x, z)	(z, x)	(x, u)	(z, x)
	(x, u)	(u, x)	(x, y)	(u, v)
	(x, y)	(y, x)	(x, z)	(y, x)
y	(y, x)	(x, y)	(y, x)	(x, z)
z	(z, x)	(x, z)	(z, x)	(x, u)

$ET = (u, v), (v, u), (u, w), (w, u), (u, x), (x, y), (y, x), (x, z), (z, x), (x, u), \ldots$

Ein Baum T mit n Knoten hat $n - 1$ Kanten. Somit enthält die durch $nexttour()$ gegebene Euler-Tour-Liste L_T durch den Graphen G_T genau $2(n - 1)$ Kanten, ist also von der Größenordnung $O(n)$.

Um auf L_T List-Ranking-Verfahren durchführen zu können, stört die Tatsache, daß L_T zyklisch ist. Deshalb „öffnet“ man L_T an einer günstigen Stelle, so daß

eine beliebige Kante $(u, \cdot)$, wobei u die Wurzel von T sein muß, zur *Startkante* wird:
Sei (u, v) die erste Kante in der Adjazenzliste des Wurzelknotens u, dann setzt man beispielsweise $nexttour(inv(u, v))$ auf $\emptyset$, wodurch die Kante $succ(u, v)$ zur Startkante von L_T wird.

Mittels List-Ranking auf L_T können nun viele elementare Baumfunktionen implementiert werden. Im folgenden wird die Realisierung einer Funktion vorgestellt, die bei der späteren Transversalenberechnung eine entscheidende Rolle spielen wird.

6.1.1 Anwendungsbeispiel: Pfadermittlung

Ziel ist die Ermittlung des Pfades von einem fest gewählten Knoten z zur Baumwurzel u. Realisiert werden soll zum Beispiel eine Indikatorfunktion $Path_z$, mit

$$Path_z(x) \;=\; \begin{cases} 1, & \text{falls } x \text{ auf dem Pfad von } z \text{ zur Wurzel } u \text{ liegt,} \\ 0, & \text{sonst.} \end{cases}$$

Voraussetzung ist, daß von jeder Kante ihr *Rang* in L_T bekannt ist, ansonsten läßt sich dieser mit Hilfe eines List-Ranking-Durchlaufs ermitteln.
Somit steht insbesondere von jeder Kante fest, ob sie Vorwärts- oder Rückwärtskante ist (s.o.).

Bezeichnet $(z, father(z))$ die Rückwärtskante des Knotens z zu seinem Vaterknoten, so folgt, da L_T einen Depth-First-Search-Durchlauf durch T repräsentiert:

> Eine Rückwärtskante (y, x) liegt genau dann auf dem Pfad von z zur Baumwurzel u, wenn $Rang(y, x) \leq Rang(z, father(z)) < Rang(x, y)$.

Die gesuchte Indikatorfunktion ergibt sich somit, indem man alle Rückwärtskanten (y, x) aus L_T testet und

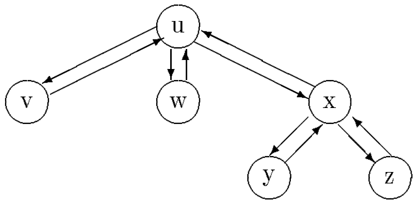
$$Path_z(y) \;:=\; \begin{cases} 1, & \text{falls } Rang(y, x) \leq Rang(z, father(z)) < Rang(inv(y, x)), \\ 0, & \text{sonst,} \end{cases}$$

setzt. Außerdem definiert man $Path(u) := 1$.

Bei der parallelen Implementierung dieser Funktion teilt man vorher den Wert von $Rang(z, father(z))$ allen Prozessoren in einer Broadcast-Phase mit. Somit kommt es zu keinen Lese- oder Schreibkonflikten, und es folgt:

Lemma 6.2
Ist ein Baum der Größe $O(n)$ gegeben, dann können $O(n/\log n)$ Prozessoren einer EREW PRAM *alle Knoten des Pfades von einem fest gewählten Knoten z zur Baumwurzel in $O(\log n)$ Zeit mit Hilfe der Euler-Tour-Technik ermitteln.*

Beispiel: Gesucht ist der Pfad von y nach u.



Kante	(u,w)	(w,u)	(u,x)	(x,y)	(y,x)	(x,z)	(z,x)	(x,u)	(u,v)	(v,u)
$Rang()$	9	8	7	6	5	4	3	2	1	0

$Path_y(u) := 1$
$Rang(w,u) \not\leq Rang(y,x) \Rightarrow Path_y(w) := 0$
$Rang(y,x) \leq Rang(y,x) < Rang(x,y) \Rightarrow Path_y(y) := 1$
$Rang(z,x) \leq Rang(y,x) \not< Rang(x,z) \Rightarrow Path_y(z) := 0$
$Rang(x,u) \leq Rang(y,x) < Rang(u,x) \Rightarrow Path_y(x) := 1$
$Rang(v,u) \leq Rang(y,x) \not< Rang(u,v) \Rightarrow Path_y(v) := 0$

Auf ähnliche Weise wie in diesem Demonstrationsbeispiel lassen sich viele weitere elementare Funktionen auf Bäumen der Größe $O(n)$ mit einer $O(n/\log n)$-Prozessor-EREW PRAM in $O(\log n)$ Zeit implementieren (vgl. [TV] und [KR]).

6.2 Der Transversalen-Algorithmus

Nachfolgend wird ein Algorithmus zur Berechnung einer minimalen Transversale in einem Circular-Arc-System S entwickelt. Der entstehende Algorithmus hat große Ähnlichkeiten mit dem Transversalen-Algorithmus von Fischer und Simon (vgl. [FS]), läßt sich jedoch einfacher parallelisieren.

Grundvoraussetzungen sind:

- Alle Bögen $I_0, I_1, \ldots, I_{m-1}$ des Circular-Arc-Systems sind paarweise verschieden.
- S ist in Form eines „Randpunktarrays" $R = (R_0, R_1, \ldots, R_{2m-1})$ gegeben, das die $2m$ Randpunkte a_j und b_j als aufsteigend sortierte Folge enthält. Dabei müssen Anfangspunkte a_i <u>vor</u> Endpunkten b_j einsortiert sein, falls $a_i = b_j$ gilt.

O.B.d.A wird zusätzlich davon ausgegangen, daß die Bögen $I_0, I_1, \ldots, I_{m-1}$ in sortierter Form gegeben sind, d.h. es gilt $a_0 \leq a_1 \leq \ldots \leq a_{m-1}$. Andernfalls läßt sich eine entsprechende Umsortierung der Bögen mit Hilfe des gegebenen Randpunktarrays R leicht erzielen (vgl. S. 56).

Es folgt eine Reihe wichtiger Definitionen und Bezeichnungen, wie sie auch in [FS] und [RR] zu finden sind:

Definition 6.3
Existiert in einem Circular-Arc-System ein Bogen I, der einen anderen Bogen I' vollständig überdeckt, so heißt I *überdeckender Bogen* und I' entsprechend *überdeckter Bogen*.

Es ergibt sich sofort:

Folgerung 6.4

$$I_i \text{ überdeckt } I_j \iff \begin{cases} \quad a_i \leq a_j \leq b_j \leq b_i \ (1) \ \vee \ b_i < a_i \leq a_j \leq b_j \ (2) \\ \vee \ b_j \leq b_i < a_i \leq a_j \ (3) \ \vee \ a_j \leq b_j \leq b_i < a_i \ (4) \end{cases}$$

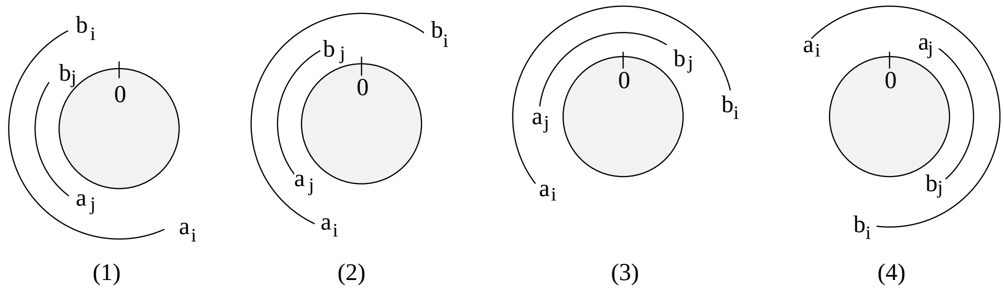


Bild 6.1: *Die vier Fälle aus Folgerung 6.4.*

Definition 6.5
Eine Teilmenge S_p eines Circular-Arc-Systems S heißt *proper*, wenn S_p keinen überdeckenden Bogen enthält.

Eliminiert man aus S sämtliche überdeckenden Bögen, so erhält man ein properes Circular-Arc-System S_p. (Ein Verfahren zur Elimination aller überdeckenden Bögen aus S wird später vorgestellt. Unter welchen Bedingungen die zu POS und NEG assoziierten Circular-Arc-Systeme überdeckende Bögen enthalten, steht in Kapitel 7.)

Da eine minimale Transversale von S_p auch eine minimale Transversale von S ist, sei von nun an S als *properes Circular-Arc-System* vorausgesetzt. (Man beachte, daß es in S_p keine zwei Bögen mit gleichem Anfangs- oder Endpunkt geben kann.)

Zu jedem Bogen I_j, $j \in \{0, 1, \ldots, m-1\}$, bezeichne $N(j)$ den Index des *Nachfolgebogens* von I_j.
Dabei ist der *Nachfolgebogen* von I_j in eindeutiger Weise definiert als der erste Bogen, dessen Anfangspunkt bei einem Durchlauf durch das Circular-Arc-System im Uhrzeigersinn nach dem Endpunkt des Bogens I_j entdeckt wird.
Induktiv definiert man für $k > 1$ den *k-ten Nachfolgebogen* von I_j als den Bogen mit Index $N^k(j) := N(N^{k-1}(j))$.

Ferner sei $\ell(j)$ das kleinste k, für das $I_{N^k(j)} \cap I_j \neq \emptyset$ gilt, d.h. $I_{N^{\ell(j)}(j)}$ ist der erste Nachfolgebogen von I_j, der I_j wieder überlappt.

Dann ist $t(j)$ definiert als

$$t(j) \ := \begin{cases} \ell(j), & \text{falls } I_{N^{\ell(j)}(j)} = I_j, \\ \ell(j)+1, & \text{sonst.} \end{cases}$$

Außerdem bezeichnet LEFT(0) die Indexmenge aller Bögen, die den Bogen I_0 links überlappen, d.h. $\text{LEFT}(0) := \{i \mid b_i \in I_0\}$. Insbesondere gilt also immer $\text{LEFT}(0) \neq \emptyset$, da $0 \in \text{LEFT}(0)$.

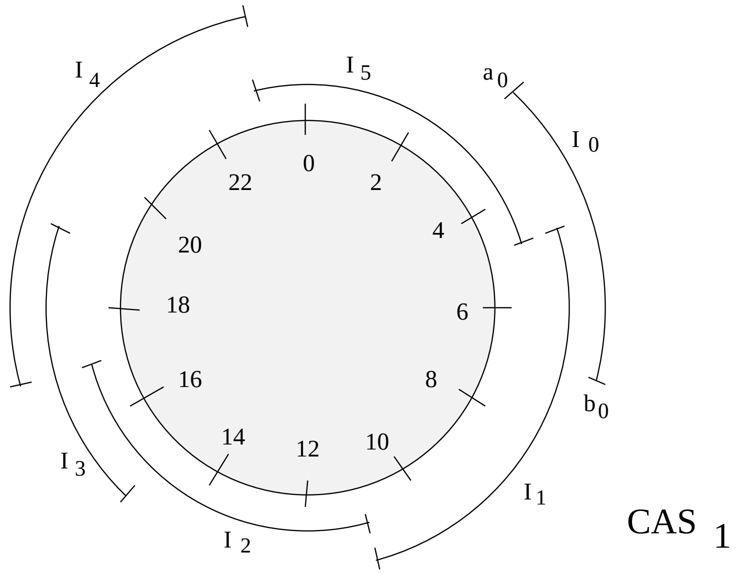


Bild 6.2: *Für* CAS_1*, das zu den Punkten aus Bild 2.4 assoziierte Circular-Arc-System, gilt:*
$N(0) = 2, \ \ N(1) = 3, \ \ N(2) = 5, \ \ N(3) = 5, \ \ N(4) = 0, \ \ N(5) = 2,$
$\ell(0) = \ell(1) = \ell(2) = \ell(3) = \ell(4) = \ell(5) = 2,$
$t(0) = t(1) = t(3) = t(4) = 3, \quad t(2) = t(5) = 2,$
$\text{LEFT}(0) = \{0, 5\}.$

Folgerung und Definition 6.6
Für alle $j \in \{0, 1, \ldots, m-1\}$ *bilden die Endpunkte* $b_j, b_{N(j)}, \ldots, b_{N^{t(j)-1}(j)}$ *eine Transversale der Größe* $t(j)$, *die sogenannte* Greedy-Transversale *mit Startpunkt* j, *bezeichnet mit* $GT(j)$.
Gilt $t(j) = \ell(j) + 1$, *so nennt man* $GT(j)$ überlappend, *denn der Bogen* $I_{N^{t(j)-1}(j)}$ *überlappt den Bogen* I_j *in diesem Fall; andernfalls nennt man* $GT(j)$ nicht-überlappend.

(In CAS_1 ist beispielsweise $GT(5) = \{b_5, b_2\} = \{17, 5\}$ eine nicht-überlappende und $GT(1) = \{b_1, b_3, b_5\} = \{11, 19, 5\}$ eine überlappende Transversale.)

Das folgende Lemma zeigt, wie sich eine minimale Transversale des properen Circular-Arc-Systems S mit Hilfe der eben eingeführten Begriffe berechnen läßt. (Die Punkte (i) bis (iv) des Lemmas stammen aus [FS], wo sie bis auf (iv) ohne Beweis aufgeführt sind.)

Lemma 6.7

(i) Zu jeder Transversale T *existiert eine Transversale* T' *derselben Größe, die nur aus Endpunkten* b_j, $j \in \{0, 1, \ldots, m-1\}$, *besteht.*

(ii) Es existiert immer eine Greedy-Transversale $GT(j)$, $j \in \{0, 1, \ldots, m-1\}$, *die minimal ist.*

(iii) Jede nicht-überlappende Greedy-Transversale ist minimal.

(iv) Sind alle Greedy-Transversalen überlappend, so ist jede Greedy-Transversale minimal.

(v) Es existiert immer eine Greedy-Transversale $GT(j)$, $j \in \mathrm{LEFT}(0)$, *die minimal ist.*

Beweis:
Die Transversale $T' = \{t'_0, t'_1, \ldots, t'_{n-1}\}$ von Behauptung (i) läßt sich aus $T = \{t_0, t_1, \ldots, t_{n-1}\}$ wie folgt konstruieren:
Zu jedem $t_i \in T$ existiert eine Menge M_i von Bögen $I \in S$ mit $t_i \cap I \neq \emptyset$. Betrachtet man das Schnittsegment $S_i := \bigcap_{I \in M_i} I$, so muß der Endpunkt von S_i notwendigerweise der Endpunkt b eines $I \in M_i$ sein. Setzt man $t'_i := b$ und wendet diese Prozedur auf alle $i = 0, 1, \ldots, n-1$ an, so erhält man die gewünschte Transversale T'.

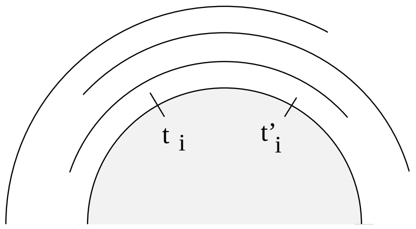


Bild 6.3: *Beweisskizze zu (i)*

Behauptung (ii) ergibt sich aus folgender Überlegung:
Sei $T = \{b_{t_0}, b_{t_1}, \ldots, b_{t_{n-1}}\}$ mit $b_{t_0} < b_{t_1} < \ldots < b_{t_{n-1}}$ eine minimale Transversale, die wegen (i) o.B.d.A nur aus den Endpunkten der Bögen $I_{t_0}, I_{t_1}, \ldots, I_{t_{n-1}}$ besteht und keine Greedy-Transversale ist.
Dann enthält T mindestens zwei Endpunkte b_{t_i} mit $b_{t_{(i+1) \bmod n}} \neq b_{N(t_i)}$.
Definiert man $i^* := \min\{i \in \{0, 1, \ldots, n-1\} \mid b_{t_{(i+1) \bmod n}} \neq b_{N(t_i)}\}$, so folgt, daß auch $T' := (T \setminus \{b_{t_{i^*+1}}\}) \cup \{b_{N(t_{i^*})}\}$ eine minimale Transversale ist.
Bildet T' bereits eine Greedy-Transversale, folgt die Behauptung, ansonsten setzt man $T := T'$ und bildet erneut T'. Dieser Bildungsprozeß ist endlich, denn es gilt $n > i^{*\prime} > i^*$, wobei $i^{*\prime}$ für T' definiert ist, wie i^* für T.

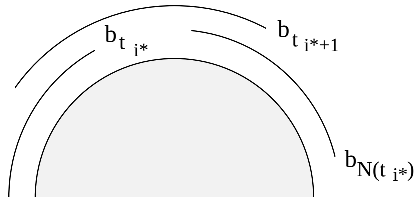


Bild 6.4: *Beweisskizze zu (ii)*

Für den Beweis von (iii) sei $GT(i)$ eine nicht-überlappende Greedy-Transversale und $B(i)$ die Menge der Bögen, deren Endpunkte in $GT(i)$ enthalten sind.
Da die Bögen aus $B(i)$ paarweise disjunkt sind, benötigt jede Transversale mindestens $|B(i)| = t(i) = |GT(i)|$ Punkte zur Repräsentation der Bögen aus $B(i)$. Also ist $GT(i)$ minimal.

Behauptung (iv) ergibt sich mit Hilfe des folgenden Widerspruchbeweises:
Angenommen alle Greedy-Transversalen sind überlappend, und es gibt Greedy-Transversalen unterschiedlicher Größe. Dann existieren zwei im Uhrzeigersinn aufeinanderfolgende Endpunkte b_i und b_j mit $t(i) < t(j)$.
Nach Konstruktion trifft $b_i \in GT(i)$ den Bogen I_j, aber keinen anderen Bogen aus $B(j)$. ($B(j)$ ist definiert wie bei (iii).) Da sich keine zwei Bögen aus $B(j) \setminus \{I_j\}$ paarweise überlappen und $GT(i)$ eine Transversale bildet, muß jeder Bogen aus $B(j) \setminus \{I_j\}$ von einem anderen Endpunkt aus $GT(i)$ getroffen werden.
Es folgt $t(i) \geq 1 + |B(j) \setminus \{I_j\}| = 1 + |B(j)| - 1 = t(j)$ im Widerspruch zur Voraussetzung.
Damit sind im Fall, daß alle Greedy-Transversalen überlappend sind, alle Greedy-Transversalen gleich groß, und es folgt die Behauptung.

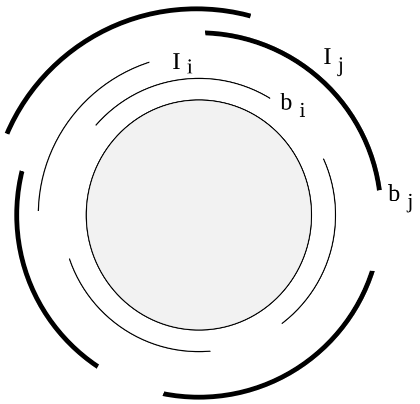


Bild 6.5: *Beweisskizze zu (iv). Bögen aus $B(j)$ sind fett gezeichnet.*

Für den Beweis von (v) sei $GT(i)$ eine minimale Greedy-Transversale.
Ist $GT(i)$ überlappend, dann ist nach (iv) insbesondere auch $GT(0)$ minimal, und 0 liegt in LEFT(0).
Andernfalls ist $GT(i)$ nicht-überlappend:
Für $i = 0$ folgt sofort die Behauptung. Sei also $i \neq 0$:
Weil $GT(i)$ eine Transversale ist, enthält $GT(i)$ einen Endpunkt b_j, der den Bogen I_0 trifft. Dann besteht $GT(j)$ aber aus denselben Endpunkte wie $GT(i)$, denn $GT(j)$ ist eine „zyklische Verschiebung“ von $GT(i)$. Somit ist auch $GT(j)$ minimal, und außerdem gilt $j \in \text{LEFT}(0)$. □

Ein Algorithmus zur Ermittlung einer minimalen Transversale besteht damit aus folgenden Schritten:

ALGORITHMUS „MINIMALE TRANSVERSALE“

Eingabe: Die Anfangs- und Endpunkte von Bögen eines Circular-Arc-Systems S.

Ausgabe: Eine minimale (Greedy-)Transversale M.

Schritt 1: Mache das gegebene Circular-Arc-System proper.

Schritt 2: Berechne $N(j)$ für alle $j \in \{0, 1, \ldots, m-1\}$.

Schritt 3: Ermittle für alle $j \in \mathrm{LEFT}(0)$, ob $GT(j)$ eine überlappende oder nicht-überlappende Greedy-Transversale bildet, und setze ein binäres Flag

$$overlap(j) := \begin{cases} 0, & \text{falls } GT(j) \text{ nicht-überlappend,} \\ 1, & \text{sonst.} \end{cases}$$

Schritt 4: Ermittle ein $j^* \in \mathrm{LEFT}(0)$ mit $overlap(j^*) \leq overlap(j)$ für alle $j \in \mathrm{LEFT}(0)$.

Schritt 5: Gebe $GT(j^*) = (b_{j^*}, b_{N(j^*)}, \ldots, b_{N^{t(j^*)-1}(j^*)})$ aus.

Die Korrektheit des Algorithmus folgt mit Lemma 6.7.

6.3 Die parallele Implementierung

Bei einem Circular-Arc-System der Größe m läßt sich jeder dieser fünf Algorithmusschritte auf einer $O(m/\log m)$-Prozessor-EREW PRAM in $O(\log m)$ Zeit durchführen, wie die folgenden Betrachtungen zeigen:

Dazu befindet sich das eingangs dieses Kapitels erwähnte Randpunktarray $R = (R_0, R_1, \ldots, R_{2m-1})$ im Global-Memory der PRAM. Eine Komponente R_k aus R enthält neben dem eigentlichen Wert des Randpunktes auch noch Informationen darüber, zu welchem Bogen R_k gehört, und ob R_k Anfangs- oder Endpunkt ist.
Technisch läßt sich R_k zum Beispiel als dreikomponentiger Vektor der Form $(a_i, i, start)$ oder (b_j, j, end) implementieren, je nachdem, ob R_k einen Anfangspunkt a_i oder einen Endpunkt b_j darstellt. Abkürzend wird jedoch oft nur „$R_k = a_i$“ oder „$R_k = b_j$“ geschrieben.

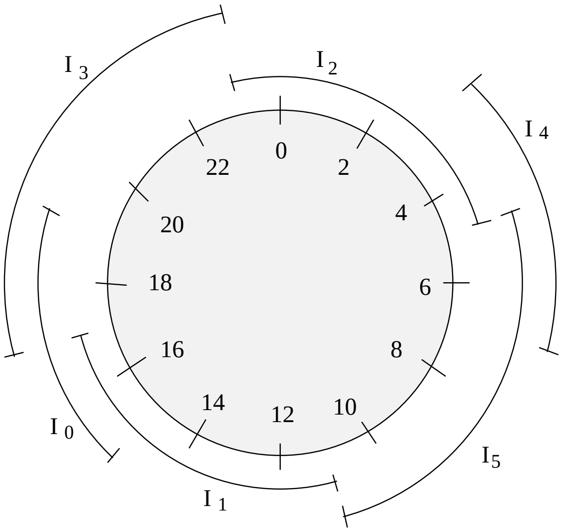
In einem Preprocessing-Schritt müssen zunächst die Bögen $I_0, I_1, \ldots, I_{m-1}$ so sortiert werden, daß für ihre Anfangspunkte $a_0 \leq a_1 \leq \ldots \leq a_{m-1}$ gilt. Dies läßt sich sehr effizient erreichen, indem man die bereits vorhandene Sortierung des Randpunktarrays ausnutzt:

Löscht man aus R alle Endpunkte, so ergibt sich ein neues Array $A = (A_0, A_1, \ldots, A_{m-1})$, das nur noch aus Anfangspunkten besteht. Jedes Element A_k ist vom Typ $(a_{j_k}, j_k, start)$ und auf Grund der Sortierung in R gilt $a_{j_0} \leq a_{j_1} \leq \ldots \leq a_{j_{m-1}}$.

Ziel muß es also sein, die Bögen in die Reihenfolge $I_{j_0}, I_{j_1}, \ldots, I_{j_{m-1}}$ zu bringen, indem der Bogen I_{j_k} zum Bogen I_k wird.
Da die Bögen durch ihre Anfangs- bzw. Endpunkte repräsentiert sind, gilt es, jeden Anfangspunkt „a_{j_k}“ und jeden Endpunkt „b_{j_k}“ in „a_k“ bzw. „b_k“ umzubenennen. Für das Randpunktarray R bedeutet dies nichts anderes, als alle Elemente $R_i = (*, k', *)$ durch $(*, k, *)$ zu ersetzen, falls $k' = j_k$ gilt.

Dazu ermittelt man ein Hilfsarray $\pi()$, indem man jedes Element $A_k = (a_{j_k}, j_k, start)$ aus A inspiziert und $\pi(j_k) := k$ setzt. Die korrekte Umindizierung ergibt sich dann durch einen Ersetzungsprozeß aller Elemente $R_i = (*, k', *)$ durch $(*, \pi(k'), *)$.

Zur Demonstration betrachte man das folgende Circular-Arc-System CAS_{1A}, das bis auf die Bogennumerierung identisch ist mit CAS_1:



CAS $_{1A}$

i	0	1	2	3	4	5	6	7	8	9	10	11
R_i	a_4 3	a_5 5	b_2 5	b_4 7	a_1 11	b_5 11	a_0 15	a_3 17	b_1 17	b_0 19	a_2 23	b_3 23

Löschen aller Endpunkte aus R:

i	0	1	2	3	4	5
A_i	a_4 3	a_5 5	a_1 11	a_0 15	a_3 17	a_2 23

Ermittlung des Hilfarrays $\pi()$:

$\pi(4) := 0,\ \pi(5) := 1,\ \pi(1) := 2,\ \pi(0) := 3,\ \pi(3) := 4,\ \pi(2) := 5$

Das neue Randpunktarray R enthält die Anfangspunkte in aufsteigender Form:

i	0	1	2	3	4	5	6	7	8	9	10	11
R_i	a_0 3	a_1 5	b_5 5	b_0 7	a_2 11	b_1 11	a_3 15	a_4 17	b_2 17	b_3 19	a_5 23	b_4 23

Abschließend ermittelt man noch ein Endpunktarray $B = (B_0, B_1, \ldots, B_{m-1})$, indem man alle Randpunkte R_k aus R betrachtet, und $B_j := R_k$ setzt, falls R_k den Endpunkt b_j enthält.
Auf diese Weise sind nun die Werte aller Randpunkte a_i und b_j in konstanter Zeit verfügbar, denn sie stehen in den Arrays A und B an den Positionen i bzw. j.

Wie man das Array A gewinnt, ist in Kapitel 3.2.2 beschrieben, und auch die anderen Schritte lassen sich auf einer EREW PRAM mit $O(m/\log m)$ Prozessoren in $O(\log m)$ Zeit durchführen, indem ein Prozessor $O(\log m)$ Arrayelemente bearbeitet.

Nach diesem Preprocessing-Schritt kann mit der eigentlichen parallelen Implementierung des Transversalen-Algorithmus begonnen werden:

Für Implementierung der Schritte 1 und 2 schlagen Rao und Rangan in [RR] bereits Techniken vor. Die dort angegebenen Verfahren müssen allerdings erweitert werden, um auch auf Circular-Arc-Systemen mit koinzidierenden Bogenrandpunkten anwendbar zu sein. (Man beachte, daß es bei den zu POS und NEG assoziierten Circular-Arc-Systemen durchaus Randpunkte mit gleichem Wert geben kann, insbesondere dann, wenn zwei verschiedene Punkte $q \in$ NEG kollineare rechte oder linke Sichtbarkeitstangenten besitzen.)
Die Idee, die Beziehung zwischen einem Bogen und seinem Nachfolgebogen als Kante eines Graphen zu interpretieren und aus dem so definierten Graphen die gewünschten Informationen mittels Euler-Tour-Technik zu gewinnen, stammt ebenfalls aus [RR].
Kapitel 6.4 zeigt detailliert, wie sich die für die Euler-Tour-Technik benötigten Datenstrukturen anlegen lassen. Dieser Schritt fehlt in [RR] ganz.

6.3.1 Schritt 1: Elimination aller überdeckenden Bögen

Zum einfacheren Aufspüren aller überdeckender Bögen des Circular-Arc-Systems bildet man aus dem Randpunktarray R ein neues Array R^*, indem zunächst alle Endpunkte b_j mit $b_j < a_j$ aus R gelöscht und anschließend wieder an das Arrayende von R angefügt werden.
Wie sich dieser Schritt mittels Parallel-Prefix auf einer $O(m/\log m)$-Prozessor-EREW PRAM in $O(\log m)$ Zeit durchführen läßt, zeigt Kapitel 3.2.3 („Umsortierproblem").

Nach dieser Aktion steht in R^* jeder Anfangspunkt a_j eines Bogens I_j vor dem zugehörigen Endpunkt b_j.

R^* besteht aus mehreren zusammenhängenden Blöcken $Block_0$, $Block_1$, $Block_2, \ldots$ von Randpunkten mit gleichem Randpunktwert.
Die Nummer des Blocks, zu dem ein Randpunkt a_i oder b_j gehört, läßt sich mittels Parallel-Prefix ermitteln:
Dazu assoziert man zu jedem Randpunkt R^*_k folgendes Gewicht w:

$$w(R^*_k) := \begin{cases} 0, & \text{falls } k = 0 \text{ oder } R_k \text{ hat denselben Wert wie } R^*_{k-1}, \\ 1, & \text{sonst.} \end{cases}$$

Anschließend ergibt sich $Bl(R^*_k)$, die Blocknummer von R^*_k, als $Bl(R^*_k) = \sum_{j=0}^{k} w(j)$.
Diese Berechnung ist Aufgabe für Parallel-Prefix-Computation.

Das folgende Beispiel zeigt die Berechnung der Blocknummern bei einem Circular-Arc-System mit extrem vielen überdeckenden Bögen:

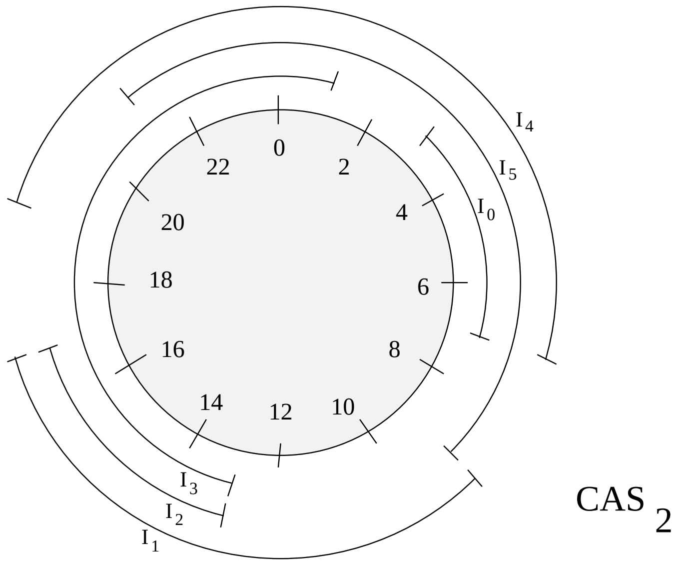


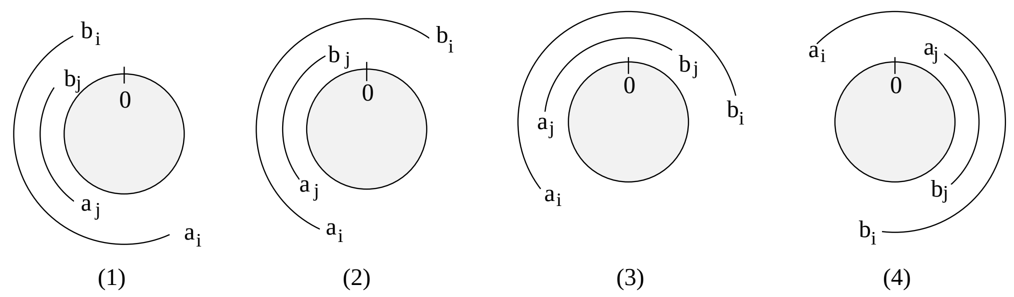
Bild 6.6: *In* CAS_2 *überdecken* I_4 *und* I_5 *den Bogen* I_0 *und* I_1 *und* I_3 *den Bogen* I_2.

i	0	1	2	3	4	5	6	7	8	9	10	11
R_i	b_3	a_0	b_4	b_0	a_1	b_5	a_2	a_3	b_1	b_2	a_4	a_5
	1	3	7	7	9	9	13	13	17	17	19	21
R_i^*	a_0	b_0	a_1	a_2	a_3	b_1	b_2	a_4	a_5	b_3	b_4	b_5
	3	7	9	13	13	17	17	19	21	1	7	9
$w(i)$	0	1	1	1	0	1	0	1	1	1	1	1
$Bl(i)$	0	1	2	3	3	4	4	5	6	7	8	9

Durch die Einführung von R^* und der Blocknummern vereinfacht sich die Charakterisierung überdeckender Bögen im Vergleich zu Folgerung 6.4 wie folgt:

Lemma 6.8

$$I_i \textit{ überdeckt } I_j \Longleftrightarrow \begin{cases} & Bl(a_i) \leq Bl(a_j) \leq Bl(b_j) \leq Bl(b_i) \\ \vee & a_j \leq b_j \leq b_i < a_i \end{cases}$$



($Bl(a_i) \leq Bl(a_j) \leq Bl(b_j) \leq Bl(b_i)$ gilt genau in den Fällen 1 bis 3 von Folgerung 6.4 , während $a_j \leq b_j \leq b_i < a_i$ den Fall 4 von Folgerung 6.4 darstellt.)

Das folgende Verfahren spürt alle Bögen I_i auf, die einen Bogen I_j mit $Bl(a_i) \leq Bl(a_j) \leq Bl(b_j) \leq Bl(b_i)$ überdecken:

Die Idee besteht darin, jedem Anfangspunkt a_i ein Gewicht $w(a_i) := Bl(b_i)$ und jedem Endpunkt b_i ein Gewicht $w'(b_i) := Bl(a_i)$ zuzuordnen.
Dann ist nämlich die Bedingung $Bl(a_i) \leq Bl(a_j) \leq Bl(b_j) \leq Bl(b_i)$ genau dann erfüllt, wenn in R^*
(Situation 1) ...hinter a_i ein Anfangspunkt a_j mit $w(a_j) \leq w(a_i)$, oder
(Situation 2) ...vor b_i ein Endpunkt b_j mit $w'(b_j) \geq w'(b_i)$
einsortiert ist. (Daß wirklich beide Situationen beachtet werden müssen, zeigt das Beispiel auf Seite 61.)

Das komplette Verfahren lautet wie folgt:

Betrachte alle Randpunkte R_k^* aus R^*:
Enthält R_k^* einen Anfangspunkt a_i, so setze $w(k) := Bl(b_i)$, $w'(k) := -\infty$.
Ansonsten enthält R_k^* einen Endpunkt b_i: Setze $w'(k) := Bl(a_i)$, $w(k) := +\infty$.
Berechne anschließend für alle Randpunkte R_k^* die Werte von
$W(k) := \min\{w(k+1), w(k+2), \ldots, w(2m-1)\}$ und
$W'(k) := \max\{w'(0), w'(1), \ldots, w'(k-1)\}$.
Gilt $W(k) \leq w(k)$ für einen Anfangspunkt a_i oder $W'(k) \geq w'(k)$ für einen Endpunkt b_i, so markiere den Bogen I_i als *„zu löschen“*.

Man kann schnell erkennen, daß $W(k) \leq w(k)$ genau in Situation 1 und $W'(k) \geq w'(k)$ genau in Situation 2 gilt. (Das Minimum bzw. Maximum einer leeren Menge wird als „$-\infty$“ bzw. „$+\infty$“ definiert.)

Beispiel: Anwendung auf CAS_2. Die Bögen I_1 und I_3 werden als überlappend erkannt.

k	0	1	2	3	4	5	6	7	8	9	10	11
R_k^*	a_0 3	b_0 7	a_1 9	a_2 13	a_3 13	b_1 17	b_2 17	a_4 19	a_5 21	b_3 1	b_4 7	b_5 9
$Bl(k)$	0	1	2	3	3	4	4	5	6	7	8	9
$w(k)$	1	$+\infty$	4	4	7	$+\infty$	$+\infty$	8	9	$+\infty$	$+\infty$	$+\infty$
$W(k)$	4	4	4	7	8	8	8	9	$+\infty$	$+\infty$	$+\infty$	$+\infty$
lösche			I_1									
$w'(k)$	$-\infty$	0	$-\infty$	$-\infty$	$-\infty$	2	3	$-\infty$	$-\infty$	3	5	6
$W'(k)$	$-\infty$	$-\infty$	0	0	0	0	2	3	3	3	3	5
lösche										I_3		

Nun müssen noch alle Bögen I_i entdeckt werden, die einen Bogen I_j mit $a_j \leq b_j \leq b_i < a_i$ überdecken.

Gesucht werden deshalb alle Endpunkte b_i mit $b_i < a_i$, zu denen ein Endpunkt b_j mit $a_j \leq b_j$ existiert, der in dem Ursprungsrandpunktarray R entweder
(Situation 3)...vor b_i, oder
(Situation 4)...im Fall von $b_i = b_j$ eventuell hinter b_i
einsortiert ist.

Durch eine Minimums- bzw. Maximumsbildung auf geschickt gewählten Gewichten lassen sich die gesuchten Endpunkte ähnlich wie zuvor ermitteln:

Inspiziere alle Randpunkte R_k aus R:
Enthält R_k einen Endpunkt b_j mit $a_j \leq b_j$, so setze $w(k) := 1$ und $w'(k) := b_j$.
Ansonsten setze $w(k) := 0$, $w'(k) := +\infty$.
Bilde anschließend $W(k) := \max\{w(0), w(1), \ldots, w(k-1)\}$ und
$W'(k) := \min\{w(k+1), w(k+2), \ldots, w(2m-1)\}$.
Gilt danach $W(k) = 1$ oder $W'(k) = b_i$ für einen Endpunkt $b_i < a_i$, so markiere den Bogen I_i als „*zu löschen*".

Hier ist der Fall $W(k) = 1$ äquivalent zu Situation 3 und $W'(k) = b_i$ gleichbedeutend mit Situation 4.

Beispiel: Anwendung auf CAS_2. Diesmal werden die Bögen I_4 und I_5 als überlappend erkannt.

k	0	1	2	3	4	5	6	7	8	9	10	11
R_k	b_3 1	a_0 3	b_4 7	b_0 7	a_1 9	b_5 9	a_2 13	a_3 13	b_1 17	b_2 17	a_4 19	a_5 21
$w(k)$	0	0	0	1	0	0	0	0	1	1	0	0
$W(k)$	$+\infty$	0	0	0	1	1	1	1	1	1	1	1
lösche						I_5						
$w'(k)$	$+\infty$	$+\infty$	$+\infty$	7	$+\infty$	$+\infty$	$+\infty$	$+\infty$	17	17	$+\infty$	$+\infty$
$W'(k)$	7	7	7	17	17	17	17	17	17	$+\infty$	$+\infty$	$+\infty$
lösche			I_4									

Bei der parallelen Implementierung dieser Verfahren auf einer PRAM läßt sich für die Minimums- bzw. Maximumsbildung Parallel-Prefix- bzw. Parallel-Suffix-Computation einsetzen. Ordnet man jedem Prozessor eine logarithmische Anzahl von Arrayelementen aus R und R^* zu, dann kommt es an keiner Stelle zu Schreibkonflikten. Lesekonflikte (z.B. bei der Ermittlung der Blocknummern) können durch die Einführung zweier aufeinanderfolgender Lesephasen leicht vermieden werden.

Gemäß Lemma 6.8 sind damit alle überdeckenden Bögen aus S als *„zu löschen“* markiert worden, und mittels Parallel-Prefix-Computation können abschließend die Anfangs- und Endpunkte dieser Bögen aus dem Randpunktarray R eliminiert werden (vgl. Kapitel 3.2.2).

Nach einer Umindizierung analog zum Preprocessing-Schritt enthalten $A = (A_0, A_1, \ldots, A_{m_p-1})$, $B = (B_0, B_1, \ldots, B_{m_p-1})$ und $R = (R_0, R_1, \ldots, R_{2m_p-1})$ die Anfangs- und Endpunkte eines properen Circular-Arc-Systems S_p, bestehend aus $m_p \leq m$ Bögen $I_0, I_1, \ldots, I_{m_p-1}$.

Es folgt:

Lemma 6.9
Alle überdeckenden Bögen des gegebenen Circular-Arc-Systems S können von einer EREW PRAM *mit $O(m/\log m)$ Prozessoren in $O(\log m)$ Zeit entdeckt und eliminiert werden.*

Für die nächsten Schritte sei S als properes Circular-Arc-System der Größe m vorausgesetzt:

6.3.2 <u>Schritt 2</u>: Ermittlung der Nachfolgeindizes N()

Auf Grund der Sortierung von R ist $N(j)$ der kleinste Index aller Anfangspunkte, die hinter b_j in R einsortiert sind. Diese Beobachtung nutzt man wie folgt aus:

Zuerst belegt man jedes R_k aus R mit einem Gewicht

$$w(k) := \begin{cases} j, & \text{falls } R_k = a_j, \\ +\infty, & \text{sonst.} \end{cases}$$

Anschließend wird mit Hilfe von Parallel-Suffix-Computation für jedes R_k das Minimum $W(k)$ aller Nachfolgegewichte ermittelt, also
$W(k) := \min(w(k+1), w(k+2), \ldots, w(2m-1))$.

Betrachtet man nun alle Elemente R_k aus R, die einen Endpunkt b_j enthalten, so ergibt sich $N(j)$ als

$$N(j) = \begin{cases} 0, & \text{falls } W(k) = +\infty, \\ W(k), & \text{sonst,} \end{cases}$$

denn $W(k)$ enthält den Index von $a_{N(j)}$. (Im Fall von $W(k) = +\infty$ ist kein Anfangspunkt mehr hinter b_j einsortiert, und $N(j)$ ergibt sich zyklisch als Index des ersten in R einsortierten Anfangspunktes a_0.)

Analog zu Lemma 6.9 ergibt sich:

Lemma 6.10
Mit $O(m/\log m)$ Prozessoren einer EREW PRAM *können die Nachfolgeindizes $N(j)$ für alle $j \in \{0, 1, \ldots, m-1\}$ in $O(\log m)$ Zeit berechnet werden.*

Als Vorbereitung für die Schritte 3 bis 5 soll zu jedem Bogen I_j neben dem Nachfolgeindex $N(j)$ auch die Indexmenge $N^{-1}(j)$ seiner *Vorgängerbögen* ermittelt werden, d.h. gesucht wird $N^{-1}(j) := \{i \mid N(i) = j\}$ für alle $j \in \{0, 1, \ldots, m-1\}$.

Betrachtet man drei Bögen I_{i-1}, I_i, I_{i+1} (alle vorkommenden Bogenindizes sind für die folgenden Überlegungen *„modulo m“* zu nehmen), und gilt $N(i-1) = N(i+1) = j$, dann folgt auf Grund der Tatsache, daß S ein properes Circular-Arc-System bildet, sogar $N(i-1) = N(i) = N(i+1) = j$. Verallgemeinert man diese Beobachtung induktiv, so ergibt sich:

Folgerung 6.11

(i) *Für alle $j \in \{0, 1, \ldots, m-1\}$ gilt entweder $N^{-1}(j) = \emptyset$, oder es existieren zwei Indizes u_j und v_j mit $N^{-1}(j) = \{u_j, u_j+1, \ldots, v_j-1, v_j\}$.*

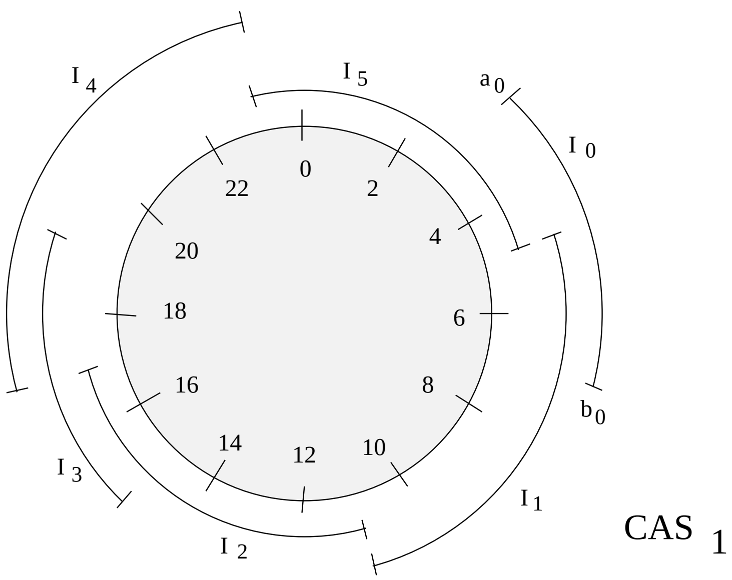
(ii) $N(j) \neq N(j+1) \implies (v_{N(j)} = j) \wedge (u_{N(j+1)} = j+1)$

Somit ist $N^{-1}(j)$ durch das Paar $(u_j,\, v_j)$ eindeutig charakterisiert.
u_j und v_j lassen sich für alle $j \in \{0, 1, \ldots, m-1\}$ wie folgt berechnen:

Schritt 1: Initialisiere $u_j := \emptyset$, $v_j := \emptyset$.

Schritt 2: Setze $k := N(j)$ und $k' := N((j+1) \bmod m)$.
Gilt $k \neq k'$, setze $v_k := j$ und $u_{k'} := (j+1) \bmod m$.

Beispiel: Berechnung der Nachfolge- und Vorgängerindizes auf CAS_1.



k	0	1	2	3	4	5	6	7	8	9	10	11
R_k	a_0 3	a_1 5	b_5 5	b_0 7	a_2 11	b_1 11	a_3 15	a_4 17	b_2 17	b_3 19	a_5 23	b_4 23
$w(k)$	0	1	$+\infty$	$+\infty$	2	$+\infty$	3	4	$+\infty$	$+\infty$	5	$+\infty$
$W(k)$	1	2	2	2	3	3	4	5	5	5	$+\infty$	$+\infty$
$N()$ =			$N(5)$ 2	$N(0)$ 2		$N(1)$ 3			$N(2)$ 5	$N(3)$ 5		$N(4)$ 0

$N(0)=2$	$N(1)=3$	$N(2)=5$	$N(3)=5$	$N(4)=0$	$N(5)=2$
$v_2=0$	$v_3=1$		$v_5=3$	$v_0=4$	
$u_3=1$	$u_5=2$		$u_0=4$	$u_2=5$	

$N^{-1}(0)=\{4\}$, $N^{-1}(1)=\emptyset$, $N^{-1}(2)=\{5,0\}$
$N^{-1}(3)=\{1\}$, $N^{-1}(4)=\emptyset$, $N^{-1}(5)=\{2,3\}$.

Führt man beide Schritte auf einer PRAM jeweils parallel für alle $j \in \{0, 1, \ldots, m-1\}$ aus, so treten keinerlei Schreibkonflikte bzgl. u_j und v_j auf. Eventuelle Lesekonflikte in Schritt 2 lassen sich durch die Einführung zweier getrennter Lesephasen für $N(j)$ und $N((j+1) \bmod m)$ verhindern. Es folgt:

Lemma 6.12
Die Indexmenge der Vorgängerbögen $N^{-1}(j)$ kann für alle $j \in \{0, 1, \ldots, m-1\}$ auf einer EREW PRAM *mit $O(m/\log m)$ Prozessoren in $O(\log m)$ Zeit ermittelt werden.*

6.3.3 <u>Schritt 3</u>: Ermittlung von overlap()

Ziel dieses Schrittes ist die Berechnung des Flags *overlap*(j), das für alle Greedy-Transversalen $GT(j)$ mit $j \in \text{LEFT}(0)$ definiert wurde als

$$overlap(j) := \begin{cases} 0, & \text{falls } GT(j) \text{ nicht-überlappende Greedy-Transversale ist,} \\ 1, & \text{sonst.} \end{cases}$$

Dabei hilft der folgende graphentheoretische Ansatz:

Identifiziert man die Bögen $I_0, I_1, \ldots, I_{m-1}$ von S mit der Knotenmenge $V = \{0, 1, \ldots, m-1\}$ eines Graphen G, so läßt sich die Relation „Bogen-Nachfolgebogen" als gerichtete Kantenmenge $E = \{(j, N(j))\} \subset V \times V$ darstellen. Der entstehende Graph $G = (V, E)$ ist ein eventuell zyklischer Graph mit *fan-out* 1, d.h. jeder Knoten von G hat genau eine ausgehende Kante.

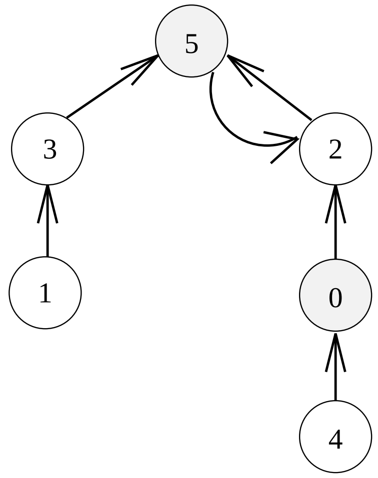


Bild 6.7: *Der Graph G zu* CAS_1*.(Knoten j, $j \in$* LEFT(0)*, wurden hervorgehoben.)*

Nun erweitert man die Knotenmenge V um Kopien $new(j)$ aller Knoten $j \in \mathrm{LEFT}(0)$ und ersetzt alle Kanten (i, j), $j \in \mathrm{LEFT}(0)$, durch Kanten $(i, new(j))$, d.h. $N(i) := new(j)$ für alle i mit $N(i) = j \in \mathrm{LEFT}(0)$, und $N(new(j)) := \emptyset$ für alle Knoten $new(j)$.

Ob $j \in \mathrm{LEFT}(0)$ gilt, läßt sich leicht testen, denn auf Grund der Sortierung der Bögen gilt:

$$j \in \mathrm{LEFT}(0) \iff (j = 0) \vee (a_j > b_j).$$

Der neu entstandene Graph $G' = (V', E')$ hat die folgenden Eigenschaften:

- G' ist nicht zyklisch und besteht aus genau $|\mathrm{LEFT}(0)|$ vielen Bäumen mit den kopierten Knoten $new(j)$, $j \in \mathrm{LEFT}(0)$, als Wurzelelement.
- Jede Kante $(j, N(j))$ eines Baumes zeigt in Richtung der Baumwurzel.
- Alle Knoten $j \in \mathrm{LEFT}(0)$ befinden sich in den Baumblättern.
- Die Knoten j und $new(j)$ repräsentieren beide den Bogen I_j.

Definition 6.13
Befindet sich ein Knoten $j \in V$ in einem Baum mit Wurzelelement $new(k)$, so definiert man $root(j)$ als k.

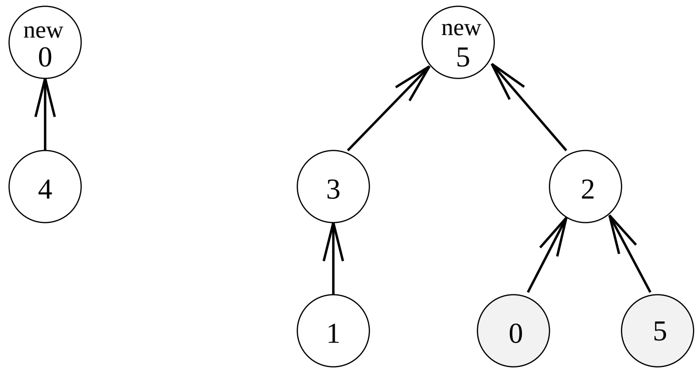


Bild 6.8: *Der Graph G' zu CAS_1 mit Wurzelelementen $new(0)$ und $new(5)$. (Knoten j, $j \in \mathrm{LEFT}(0)$, wurden hervorgehoben.)*

Betrachtet man die Knoten $v_0, v_1, v_2, \ldots, v_k$ auf dem Pfad von $j \in V$ zur Wurzel $new(root(j))$, so gilt $v_0 = j$, $v_1 = N(j)$, $v_2 = N^2(j), \ldots, v_k = new(root(j))$.

Insbesondere repräsentiert $new(root(j))$ den ersten Nachfolgebogen von I_j mit einem Index aus $\mathrm{LEFT}(0)$. Dieser Index ist $root(j)$, und es ergibt sich:

Folgerung 6.14
Eine Greedy-Transversale $GT(j)$ mit $j \in \mathrm{LEFT}(0)$ ist genau dann nicht-überlappend, wenn $root(j) = j$.

(In CAS_1 ist beispielsweise $GT(0)$ überlappend und $GT(5)$ nicht-überlappend. Man vergleiche diese Beobachtung mit Bild 6.8.)

An dieser Stelle wird auch verständlich, warum G' auf keinen Fall mehr zyklisch sein kann:
Ein zyklischer Pfad $v_0, v_1, \ldots, v_k, v_0, \ldots$ in G repräsentiert eine nicht-überlappende Greedy-Transversale im korrespondierenden Circular-Arc-System. Deshalb gibt es in dem Pfad einen Knoten (o.B.d.A. v_0) mit $v_0 \in \mathrm{LEFT}(0)$. Durch die Konvertierung von G in den Graphen G' wird die Kante (v_k, v_0) zur Kante $(v_k, new(v_0))$. Dies eliminiert den zyklischen Pfad.

Aufgabe ist es nun, $root(j)$ für alle $j \in \mathrm{LEFT}(0)$ zu berechnen, denn dann ergeben sich die gesuchten $overlap()$-Werte als

$$overlap(j) = \begin{cases} 0, & \text{falls } root(j) = j, \\ 1, & \text{sonst.} \end{cases}$$

Aus diesem Grund wurde in Kapitel 6.1 die *Euler-Tour-Technik* eingeführt. Zuvor müssen jedoch die für die Euler-Tour-Technik benötigten Datenstrukturen angelegt werden:

Zu jedem Knoten j aus V' ist somit eine zyklisch verkettete Adjazenzliste aller von j ausgehenden Kanten gesucht, wobei die Kantenmenge E' von G', die bis jetzt nur die *Rückwärtskanten* $\{(j, N(j)) \mid j \in V'\}$ enthielt, um die invers gerichteten *Vorwärtskanten* $\{(N(j), j) \mid j \in V'\}$ erweitert wird. Außerdem muß von jeder Kante (i, j) ein Zeiger zu ihrem inversen Pendant (j, i) existieren.

An dieser Stelle erweist es sich als hilfreich, daß bereits in Schritt 2 des Algorithmus zu jedem $j \in \{0, 1, \ldots, m-1\}$ neben dem Index des Nachfolgebogens $N(j)$ die Indizes der Vorgängerbögen $N^{-1}(j)$, repräsentiert durch das Paar (u_j, v_j), ermittelt worden sind:

Setzt man $N^{-1}(new(j)) := N^{-1}(j)$ und anschließend $N^{-1}(j) := \emptyset$ für alle $j \in \mathrm{LEFT}(0)$, dann ergeben sich für jeden Knoten $j \in V'$ seine ausgehenden Kanten als $(j, N(j)), (j, u_j), (j, (u_j + 1) \bmod m), \ldots, (j, v_j)$. (Im Fall von $N(j) = \emptyset$ oder $N^{-1}(j) = \emptyset$ sind es entsprechend weniger Kanten.)

Wie sich diese durch $N()$ und $N^{-1}()$ implizit gegebenen Adjazenzlisten explizit in das Global-Memory einer EREW PRAM mit $O(m/\log m)$ Prozessoren in $O(\log m)$ Zeit abbilden lassen, ist technisch nicht ganz einfach und wird in Kapitel 6.4 ausführlich erläutert.

Basis des in Kapitel 6.4 vorgestellten Konvertierungsalgorithmus ist ein *Informationsarray*, das zu jeder Kante (i, j) sämtliche den Knoten i betreffenden Daten wie $N(i)$, u_i, v_i usw. bereithält.

In Kapitel 6.4 werden zusätzlich die Implementierungen zweier Zeigerarrays $succ()$ und $inv()$ vorgenommen, wobei $succ()$ die zyklische Verkettung innerhalb der einzelnen Adjazenzlisten realisiert und $inv()$, angewandt auf eine Kante (i, j), die inverse Kante (j, i) ergibt.
O.B.d.A kann also angenommen werden, daß $nexttour(i, j)$, die nächste Kante von (i, j) in einer Eulertour durch den Baum von (i, j), für alle Kanten gegeben ist durch $nexttour(i, j) := succ(inv(i, j))$. Auf diese Weise entstehen genau $|\mathrm{LEFT}(0)|$ viele zyklisch verkettete Listen, bestehend aus insgesamt $|E'|$ vielen Kanten. Jede Liste repräsentiert eine Euler-Tour durch einen Baum T von G'.

Als letzte Vorbereitung zur Anwendung von List-Ranking muß in jedem Baum T mit Wurzel $new(j)$ eine beliebige Kante $(new(j), \cdot)$ zur Startkante der Euler-Tour durch T gemacht werden.
Analog zu Kapitel 6.1 leistet die Operation $nexttour(inv(new(j), u_{new(j)})) := \emptyset$ für alle Wurzelelemente $new(j)$ das Gewünschte. (Man beachte, daß $u_{new(j)}$ als der erste Knoten in der Adjazenzliste von $new(j)$ in konstanter Zeit adressierbar ist (vgl. Kapitel 6.4)).

Für CAS_1 ergeben sich auf diese Weise die beiden Listen
$L_1 = (new(0), 4) \rightarrow (4, new(0))$ und
$L_2 = (new(5), 3) \rightarrow (3, 1) \rightarrow (1, 3) \rightarrow (3, new(5)) \rightarrow (new(5), 2) \rightarrow (2, 0) \rightarrow (0, 2) \rightarrow (2, 5) \rightarrow (5, 2) \rightarrow (2, new(5))$.

Ermittlung von *root*() :

Enthält eine Euler-Tour-Liste als letztes Element die Kante $(i, new(j))$, dann repräsentiert diese Liste den Baum T mit Wurzelelement $new(j)$ und für alle in der Liste vorkommenden Knoten k gilt $root(k) = j$. Zur Ermittlung von $root()$ muß also der Wert j allen Knoten dieser Liste bekannt gegeben werden:

Dazu ordnet man jeder Kante $(i, new(j))$ mit $nexttour(i, new(j)) = \emptyset$ das Gewicht j und allen anderen Kanten das Gewicht 0 zu.
Dann ergibt sich für alle Vorwärtskanten $(\cdot, k)$ der Wert von $root(k)$ als maximales Gewicht aller nachfolgenden Kanten in der Euler-Tour-Liste von $(\cdot, k)$. Dieses maximale Gewicht ist j und läßt sich mittels List-Ranking berechnen.

Somit reduziert sich die eigentliche Berechnung von $root()$ auf einen einzigen List-Ranking-Durchlauf in $|\mathrm{LEFT}(0)|$ vielen Euler-Tour-Listen der Gesamtgröße $O(m)$. Diese Euler-Tour-Listen lassen sich als eine, wenn auch zerstückelte Liste der Größe $O(m)$ auffassen.

Zur Durchführung in $O(\log m)$ Zeit genügt also eine EREW PRAM mit $O(m/\log m)$ Prozessoren. Noch einmal dieselben Resourcen werden für die abschließende Berechnung der $overlap()$-Werte mittels der obigen Formel benötigt.

Zur Vereinfachung berechnet man zusätzlich einen formalen $overlap()$-Wert „$+\infty$“ für Knoten j, $j \notin$ LEFT(0).

Für die Vorwärtskanten $(new(0), 4)$, $(new(5), 3)$, $(3, 1)$, $(new(5), 2)$, $(2, 0)$ und $(2, 5)$ in den Listen L_1 und L_2 ergeben sich auf diese Weise die Werte $overlap(4) = +\infty$, $overlap(3) = +\infty$, $overlap(1) = +\infty$, $overlap(2) = +\infty$, $overlap(0) = 1$ und $overlap(5) = 0$.

Es folgt:

Lemma 6.15
Eine EREW PRAM *mit $O(m/\log m)$ Prozessoren kann in $O(\log m)$ Zeit für alle Greedy-Transversalen $GT(j)$ mit $j \in$* LEFT(0) *entscheiden, ob GT(j) überlappend oder nicht-überlappend ist und das Flag $overlap(j)$ entsprechend setzen.*

6.3.4 Schritt 4: Ermittlung einer minimalen Transversale

Nach Lemma 6.7 und der Definition von $overlap()$ ist $GT(j^*)$, $j^* \in$ LEFT(0), mit $overlap(j^*) \leq overlap(j)$ für alle $j \in$ LEFT(0) eine minimale Transversale.

Ein solches j^* läßt sich recht einfach mit Hilfe der Parallel-Prefix-Technik ermitteln:

Dazu wird zunächst jedem $j \in \{0, 1, \ldots, m-1\}$ ein zweikomponentiger Gewichtsvektor der Form $w(j) := (overlap(j), j)$ zugeordnet.

In einem anschließenden Parallel-Prefix-Durchlauf berechnet man dann den Wert von $W(k) := \min\{w(0), w(1), \ldots, w(k)\}$ für alle $k \in \{0, 1, \ldots, m-1\}$, wobei

$$w(i) < w(j) \quad :\Longleftrightarrow \quad overlap(i) < overlap(j).$$

Auf diese Weise ergibt $W(m-1)$ ein Paar $(overlap(j^*), j^*)$, wobei $overlap(j^*) \leq overlap(j)$ für alle $j \in$ LEFT(0).

Somit ist j^*, die 2. Komponente von $W(m-1)$, eine gesuchte Lösung, und es folgt:

Lemma 6.16
Die Nummer $j^* \in$ LEFT(0) *einer minimalen Greedy-Transversale $GT(j^*)$ läßt sich auf einer $O(m/\log m)$ Prozessor-*EREW PRAM *in $O(\log m)$ Zeit ermitteln.*

Für CAS_1 erhält man in diesem Schritt das Lösungspaar $(0, 5)$. Dies bedeutet, daß $GT(5)$ eine (nicht-überlappende) minimale Greedy-Transversale in CAS_1 darstellt.

6.3.5 <u>Schritt 5</u>: Ausgabe einer minimalen Transversale

In einem letzten Schritt müssen nun noch alle Elemente der minimalen Transversale $GT(j^*)$ ausgegeben werden:

Ziel ist deshalb die Erstellung eines Arrays $M = (M_0, M_1, \ldots, M_{t(j^*)-1})$ im Global-Memory der PRAM, bei dem ein Arrayelement M_i den Wert des i-ten Transversalenelementes enthält.
Dazu wird in einem ersten Schritt ein Hilfsarray $H = (H_0, H_1, \ldots, H_{m-1})$ angelegt, mit

$$H_i = \begin{cases} b_i, & \text{falls } b_i \in GT(j^*), \\ -1, & \text{sonst.} \end{cases}$$

Löscht man anschließend alle Elemente $H_i = -1$ aus H heraus (vgl. Kapitel 3.2.2), ergibt sich das gewünschte Array M. (Die Größe $t(j^*) = |GT(j^*)|$ des Ausgabearrays wird bei diesem Löschvorgang implizit mitberechnet und kann somit ein Teil der Ausgabe sein.)

$GT(j^*)$ besteht aus den Endpunkten $b_{j^*}, b_{N(j^*)}, \ldots, b_{N^{t(j^*)-1}(j^*)}$ der Bögen $I_{j^*}, I_{N(j^*)}, \ldots, I_{N^{t(j^*)-1}(j^*)}$.

Die Werte $j^*, N(j^*), \ldots, N^{t(j^*)-1}(j^*)$ sind im wesentlichen die Knoten des Pfades P vom Blatt j^* zur Baumwurzel $new(root(j^*))$ im Graphen G'.

Genauer gilt: $\{j^*, N(j^*), \ldots, N^{t(j^*)-1}(j^*)\} = V_P \setminus \{new(root(j^*))\} \cup \{root(j^*)\}$, wobei V_P die Knotenmenge des Pfades P bezeichnet.

Also läßt sich das in Kapitel 6.1.1 vorgestelle Verfahren zur Pfadberechnung anwenden, das für alle Knoten i aus dem Baum von j^* (das sind die Knoten mit $root(i) = root(j^*)$) eine Funktion $Path_{j^*}$ mit

$$Path_{j^*}(i) \;=\; \begin{cases} 1, & \text{falls } i \text{ auf dem Pfad von } j^* \text{ zur Wurzel } new(root(j^*)) \text{ liegt,} \\ 0, & \text{sonst,} \end{cases}$$

implementiert.

Mittels der *Path*-Funktion läßt sich H_i definieren als

$$H_i \;:=\; \begin{cases} b_i, & \text{falls } root(i) = root(j^*) \text{ und } Path_{j^*}(i) = 1, \\ -1, & \text{sonst.} \end{cases}$$

Gibt man den Wert von $root(j^*)$ allen Prozessoren in einer vorherigen Broadcast-Phase bekannt, so folgt für die parallele Implementierung auf einer PRAM:

Lemma 6.17
Die minimale Greedy-Transversale $GT(j^*)$ kann von einer EREW PRAM *mit $O(m/\log m)$ Prozessoren in $O(\log m)$ Zeit ausgegeben werden.*

Im letzten Schritt wurde für CAS_1 die Greedy-Transversale $GT(5) = \{b_2, b_5\} = \{17, 5\}$ als minimal entdeckt.
In diesem Fall ermittelt der Transversalen-Algorithmus zunächst das Hilfsarray $H = (-1, -1, 17, -1, -1, 5)$ und dann ein endgültiges Ausgabearray $M = (17, 5)$.

Das Gesamtresultat dieses Kapitels lautet schließlich:

Satz 6.18
Sind die m Bögen eines Circular-Arc-Systems S durch ihre Anfangs- und Endpunkte gegeben, und liegen alle Bogenrandpunkte in sortierter Form vor, dann kann eine EREW PRAM *mit $O(m/\log m)$ Prozessoren eine minimale Transversale von S in $O(\log m)$ Zeit und $O(m)$ Platz ermitteln.*

6.4 Ermittlung der Euler-Tour-Datenstrukturen

In diesem Kapitel wird gezeigt, wie sich die durch $N()$ und $N^{-1}()$ nur implizit gegebenen Adjazenzlisten aller Knoten $j \in V'$ konkret im Global-Memory einer EREW PRAM erzeugen lassen.

Anschließend werden alle Kanten innerhalb einer Adjazenzliste mit Hilfe eines Arrays $succ()$ zyklisch miteinander verkettet. Außerdem wird ein Array $inv()$ erzeugt, das zu jeder Kante (i, j) die invers gerichtete Kante (j, i) ergibt.

Am Schluß dieses Kapitels existiert im Global-Memory der EREW PRAM ein Kantenarray $(E_0, E_1, \ldots, E_{|E'|-1})$, wobei jede Komponente E_k eine Kante (i, j) enthält.

$E_{nexttour(k)}$, die Kante nach E_k in einer Euler-Tour durch den Baum von E_k, ergibt sich dann als $E_{succ(inv(k))}$.

Zur einfacheren Handhabung der Knoten aus V' wird jeder Knoten $j \in V$ mit der Nummer j und jeder kopierte Knoten $new(j)$ mit der Nummer $j + m$ identifiziert.

Die Werte von $N()$ und $N^{-1}()$ sind demnach als Arrays der Größe $2m$ im Global-Memory der PRAM gegeben, mit der Voraussetzung, daß $N(j) = N^{-1}(j) = \emptyset$, falls ein Knoten mit Nummer $j \quad (j \geq m)$ gar nicht existiert.

Es folgt der Konvertierungsalgorithmus:

Schritt 1:
Zuerst wird die Größe $grad(j)$ der Adjazenzliste des Knotens mit Nummer j für alle $j \in \{0, 1, \ldots, 2m-1\}$ ermittelt.
$grad(j)$ ergibt sich aus $N(j)$ und $N^{-1}(j)$ einfach als $grad(j) := |N(j)| + |N^{-1}(j)|$, wobei $|N(j)|$ und $|N^{-1}(j)|$ gegeben sind durch

$$|N^{-1}(j)| = \begin{cases} 0, & \text{falls } N^{-1}(j) = \emptyset, \\ |\{u_j, (u_j+1) \bmod m, \ldots, v_j\}|, & \text{sonst,} \end{cases}$$

und

$$|N(j)| = \begin{cases} 0, & \text{falls } N(j) = \emptyset, \\ 1, & \text{sonst.} \end{cases}$$

Schritt 2:
Nun berechnet man $base(j) := \sum_{i=0}^{j-1} grad(i)$ für alle $j \in \{0, 1, \ldots, 2m-1\}$.
Der Index $base(j)$ gibt im Fall von $grad(j) > 0$ an, wo die zum Knoten j adjazenten Kanten im Kantenarray $(E_0, E_1, \ldots, E_{|E'|-1})$ zu finden sein werden, nämlich als zusammenhängendes Arraystück an den Positionen $E_{base(j)}$, $E_{base(j)+1}, \ldots, E_{base(j)+grad(j)-1}$.

Schritt 3:
Weiß man von einem Arrayelement E_k, daß es eine vom Knoten j ausgehende Kante enthalten soll, d.h. $k \in \{base(j), base(j)+1, \ldots, base(j)+grad(j)-1\}$, so wird der Inhalt von E_k mit Hilfe der folgenden Formel explizit ausgerechnet und in E_k eingetragen:

$$E_k := \begin{cases} (j, N(j)), & \text{falls } N(j) \neq \emptyset \\ & \text{und } k = base(j), \\ (j, (u_j + k - base(j) - |N(j)|) \bmod m), & \text{sonst.} \end{cases}$$

Auf diese Weise füllen sich alle Arraystücke $E_{base(j)}, \ldots, E_{base(j)+grad(j)-1}$ mit den gewünschten Kanten:

$\ldots$	$E_{base(j)}$	$E_{base(j)+1}$	$E_{base(j)+2}$	$\ldots$	$E_{base(j)+grad(j)-1}$	$E_{base(j+1)}$
$\ldots$	$(j, N(j))$	(j, u_j)	$(j, (u_j+1) \bmod m)$	$\ldots$	(j, v_j)	$\ldots$

(Die Ausnahmen $N(j) = \emptyset$ oder $N^{-1}(j) = \emptyset$ werden ebenfalls korrekt berücksichtigt.)

Damit es zu keinen Leseproblemen kommt, müssen also die Werte von j, $N(j)$, u_j und $base(j)$ für jedes Arrayelement E_k bekannt sein. Aus diesem Grund wird in einem Preprocessing-Schritt ein *Informationsarray* $(info_0, info_1, \ldots, info_{2m-1})$ eingeführt, wobei ein Element $info_k$ die Werte enthält, die zur Berechnung von E_k benötigt werden:

Schritt 3*:
Zuerst belegt man für alle $j \in \{0, 1, \ldots, 2m-1\}$ mit $grad(j) > 0$ die Elemente $info_{base(j)}$ mit den entsprechenden Werten von j, $N(j)$, u_j und $base(j)$. Alle anderen Arrayelemente werden mit dem Vektor $(-1, -1, -1, -1)$ initialisiert:

$\ldots$	$info_{base(j)}$	$info_{base(j)+1}$	$\ldots$	$info_{base(j)+grad(j)-1}$	$info_{base(j+1)}$
$\ldots$	$j,\ N(j),\ u_j,\ base(j)$	$-1, -1, -1, -1$	$\ldots$	$-1, -1, -1, -1$	$j+1, \ldots$

Jedes Arrayelement $info_k$ vom Typ $(-1, -1, -1, -1)$ muß nun noch die Werte des letzten vor ihm eingeordneten Arrayelementes $info_{k'} \neq (-1, -1, -1, -1)$ annehmen. $info_{k'}$ ist jedoch genau das Arrayelement mit maximaler *base*-Komponente aller vor $info_k$ eingeordneten Elemente.

Somit lassen sich alle $info_k$ berechnen als $info_k := \max\{info_0, info_1, \ldots, info_{k-1}\}$, wobei

$info_i > info_j \quad :\Longleftrightarrow \quad$ Die *base*-Komponente von $info_i$ ist größer als die *base*-Komponente von $info_j$.

Diese Maximumsbildung ist Aufgabe für Parallel-Prefix-Computation, und nach einem entsprechenden Durchlauf enthält das Informationsarray die gewünschten Daten:

$\ldots$	$info_{base(j)}$	$info_{base(j)+1}$	$\ldots$	$info_{base(j)+grad(j)-1}$	$info_{base(j+1)}$
$\ldots$	$j,\ N(j),\ u_j,\ base(j)$	$j,\ N(j),\ u_j,\ base(j)$	$\ldots$	$j,\ N(j),\ u_j,\ base(j)$	$j+1, \ldots$

Anschließend lassen sich die Kanten des Kantenarrays mit Hilfe der oben angegebenen Formel ohne Lesekonflikte berechnen, indem jedes E_k seine Informationen exklusiv aus $info_k$ bezieht.

Schritt 4:
Die Funktion $succ()$ ergibt sich nun für alle Kanten E_k einfach als

$$succ(k) := \begin{cases} base(j), & \text{falls } k = base(j) + grad(j) - 1, \\ k+1, & \text{sonst,} \end{cases}$$

wobei der Wert von $grad(j)$ nach einer entsprechenden Erweiterung des Informationsarrays in $info_k$ steht und somit für alle k bekannt ist.

Enthält E_k eine Kante aus der Adjazenzliste des Knotens j, dann gibt $E_{succ(k)}$ den zyklischen Nachfolger innerhalb der Adjazenzliste an.
Wie man leicht sieht, realisiert $succ()$ für alle Adjazenzlisten eine zyklische Verkettung der Form $\ldots \to (j,\ N(j)) \to (j,\ u_j) \to \ldots \to (j,\ v_j) \to (j,\ N(j)) \to \ldots$

Schritt 5:
Zuletzt muß noch die $inv()$-Funktion implementiert werden, so daß zu allen Kanten E_k die inverse Kante gegeben ist durch $E_{inv(k)}$. Dabei macht man sich zunutze, daß alle Rückwärtskanten vom Typ $(j,\, N(j))$ (sofern sie existieren) an den Stellen $E_{base(j)}$ im Kantenarray stehen:

Für alle Kanten $E_k = (j, i)$ prüft man, ob $i \neq N(j)$ gilt. In diesem Fall repräsentiert E_k eine Vorwärtskante, und es gilt $j = N(i)$. Die inverse Rückwartskante $(i, j) = (i,\, N(i))$ existiert somit und befindet sich an der Stelle $E_{base(i)}$.

Also setzt man $inv(k) := base(i)$ und $inv(base(i)) := k$ und erzeugt auf diese Weise alle Verknüpfungen $(j,\, i) \rightarrow (i,\, j)$ und $(i,\, j) \rightarrow (j,\, i)$.

Betrachtet man alle sechs Schritte des Konvertierungsalgorithmus, so stellt man fest, daß bei einer parallelen Implementierung auf einer PRAM insbesondere durch die Verwendung des Informationsarrays an keiner Stelle Lese- oder Schreibkonflikte auftreten, und es folgt

Lemma 6.19
Aus den Daten von $N()$ und $N^{-1}()$ lassen sich mit einer $O(m/\log m)$-Prozessor- EREW PRAM *die expliziten Adjazenzlisten aller Knoten $j \in V'$ in $O(\log m)$ Zeit erzeugen. Dieselben Resourcen werden für die Ermittlung der Zeigerarrays $succ()$ und $inv()$ benötigt.*

Kapitel 7

Zusammenfassung und Ausblick

Es folgt noch einmal eine zusammenfassende Tabelle mit den in dieser Arbeit erzielten Ergebnissen. Sie enthält zu jedem Schritt des Separierungsalgorithmus von Fischer und Simon die sequentielle Laufzeit im Vergleich zu den benötigten parallelen Resourcen Laufzeit, Prozessoranzahl und PRAM-Modell.
Man erkennt, daß der gewonnene parallele Algorithmus zur Berechnung eines minimalen konvexen separierenden Polygons im Sinne der in Kapitel 1 gegebenen Definition *optimal* ist:

Schritt	sequentielle Laufzeit	parallele Laufzeit	Prozessoren PRAM-Modell
Berechnung von BOUNDARY(POS)	$O(n' \log n')$	$O(\log n')$	$O(n')$ EREW PRAM
Ermittlung sämtlicher Sichtbarkeitstangenten	$O(m \log n)$	$O(\log n)$	$O(m)$ CREW PRAM
Sortieren der Sichtbarkeitstangenten im Uhrzeigersinn	$O(m \log m)$	$O(\log m)$	$O(m)$ EREW PRAM
Ermittlung des zu POS und NEG assoziierten Circular-Arc-Systems S	$O(m)$	$O(\log m)$	$O(m/\log m)$ EREW PRAM
Berechnung einer minimalen Transversale M in S	$O(m)$	$O(\log m)$	$O(m/\log m)$ EREW PRAM
Ermittlung und Ausgabe der korrespondierenden Zwischentangenten	$O(m)$	$O(\log m)$	$O(m/\log m)$ EREW PRAM
INSGESAMT	$O(N \log N)$	$O(\log N)$	$O(N)$ CREW PRAM

($n' = |\text{POS}|$, $n = |\text{BOUNDARY(POS)}|$, $m = |\text{NEG}|$, $N = n' + m$)

In der Einleitung wurde bereits erwähnt, daß ein minimales konvexes separierendes Polygon nur genau dann existieren kann, wenn HULL(POS) $\cap$ NEG $\neq \emptyset$ gilt, und diese Beziehung wurde dann auch für alle weiteren Betrachtungen vorausgesetzt.

Was passiert aber, wenn diese Bedingung nicht eingehalten wird und sich ein Punkt $q \in$ NEG doch innerhalb von HULL(POS) befindet? Kann der Separierungs-Algorithmus diese Situation erkennen?

Die Lösung ist überraschend einfach und erfordert keinen großen Aufwand:

Betrachtet man in Kapitel 5 den Algorithmus zur Berechnung einer rechten Sichtbarkeitstangente, so stellt man fest, daß dieser Algorithmus in jedem Fall terminiert und einen Punkt p_r ausgibt. Dabei spielt es keine Rolle, ob der Bezugspunkt q innerhalb oder außerhalb des konvexen Polygons liegt.

Nach Lemma 5.3 gilt für einen Stützpunkt p_r der rechten Sichtbarkeitstangente

$$\Delta(q, p_{r-1}, p_r) \leq 0 \quad \wedge \quad \Delta(q, p_r, p_{r+1}) > 0.$$

(Alle Indizes von Punkten aus BOUNDARY(POS) und NEG in diesem Kapitel sind „*modulo n*“ bzw. „*modulo m*“ zu verstehen.)
Angenommen q befindet sich innerhalb von HULL(POS), dann folgt jedoch auf Grund der Sortierung der Eckpunkte für jeden Punkt $p_r \in \{p_0, p_1, \ldots, p_{n-1}\}$

$$\Delta(q, p_{r-1}, p_r) \leq 0 \quad \wedge \quad \Delta(q, p_r, p_{r+1}) \leq 0.$$

Mit anderen Worten: Gilt bei einem für einen Punkt $q \in$ NEG berechneten Stützpunkt p_r der rechten Sichtbarkeitstangente $\Delta(q, p_r, p_{r+1}) \leq 0$, dann muß q innerhalb von HULL(POS) liegen, und das Ausgangsproblem, die Ermittlung eines minimalen konvexen separierenden Polygons, ist nicht lösbar.

Also erweitert man den Algorithmus von Fischer und Simon um einen Schritt 2a, in dem für alle Punkte q_j, $j \in \{0, 1, \ldots, m-1\}$, getestet wird, ob $\Delta(q_j, p_{r_j}, p_{r_j+1}) \leq 0$ gilt. Bei positivem Ausgang läßt man den Algorithmus dann mit der Ausgabe „Kein konvexes separierendes Polygon vorhanden!“ terminieren.

Die Implementierung dieses Ergänzungsschrittes auf einer CREW PRAM bereitet keine Schwierigkeiten, und es ergibt sich:

Theorem 7.1
Sind zwei Punktmengen POS *und* NEG *der Ebene mit Gesamtgröße N gegeben, dann kann eine* CREW PRAM *mit $O(N)$ Prozessoren ein minimales konvexes Polygon, das* POS *von* NEG *separiert, in $O(\log N)$ Zeit bei einem Platzbedarf von $O(N)$ ermitteln, bzw. erkennen, daß ein solches Polygon nicht existiert.*

Einen breiten Raum bei der Entwicklung des Transversalen-Algorithmus in Kapitel 6 nehmen die Verfahren ein, die ein beliebiges Circular-Arc-System in ein properes Circular-Arc-System überführen, indem alle überdeckenden Bögen gelöscht werden.

Jeder Punkt $q_i \in$ NEG wird in dem zu POS und NEG assoziierten Circular-Arc-System S durch einen Bogen I_i repräsentiert, und es stellt sich die Frage, ob es überhaupt Punkte q_i und q_j derart geben kann, daß der Bogen I_i den Bogen I_j überdeckt. Oder ist vielleicht jedes zu POS und NEG assoziierte Circular-Arc-System a priori proper ?

Die Antwort ist leicht zu finden:

Damit I_i den Bogen I_j überdeckt, muß der Sweep-Line-Algorithmus aus Kapitel 2 die Sichtbarkeitstangenten von q_i und q_j in der folgenden Reihenfolge entdecken:

1. Rechte Sichtbarkeitstangente von q_i
2. Rechte Sichtbarkeitstangente von q_j
3. Linke Sichtbarkeitstangente von q_j
4. Linke Sichtbarkeitstangente von q_i

Bild 7.1 zeigt zwei Punkte q_i und q_j, auf die diese Situation zutrifft:

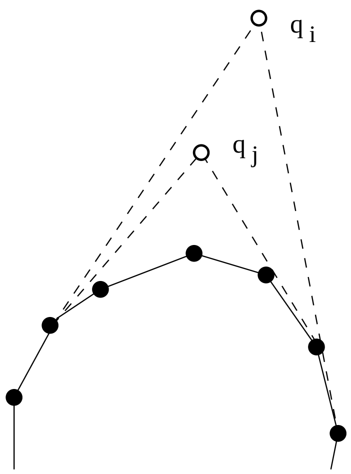


Bild 7.1

Man erkennt:

Lemma 7.2
Für jedes zu POS *und* NEG *assoziierte Circular-Arc-System gilt:*

$$I_i \text{ überdeckt } I_j \iff \left\{ \begin{array}{l} \textit{Der Sichtbarkeitsbereich von } q_i \textit{ enthält} \\ \textit{den Sichtbarkeitsbereich von } q_j. \end{array} \right.$$

Ein Punkt q_i, zu dem ein anderer Punkt q_j existiert, dessen Sichtbarkeitsbereich völlig im Sichtbarkeitsbereich von q_i liegt, soll deshalb *überdeckender Punkt* heißen. Es folgt:

Folgerung 7.3

(i) *Das zu* POS *und* NEG *assoziierte Circular-Arc-System ist genau dann proper, wenn* NEG *keine überdeckenden Punkte enthält.*

(ii) *Sei* $\mathrm{NEG}_{\ddot{U}}$ *die Menge aller überdeckenden Punkte von* NEG. *Dann gilt: Jedes minimale konvexe separierende Polygon bzgl.* POS *und* $\mathrm{NEG} \setminus \mathrm{NEG}_{\ddot{U}}$ *ist auch ein minimales konvexes separierendes Polygon bzgl.* POS *und* NEG.

Interessant in diesem Zusammenhang ist der folgende Satz, der den Abschluß dieser Diplomarbeit bildet. Er besagt, daß unter der Zusatzvoraussetzung $\mathrm{NEG}_{\ddot{U}} = \emptyset$ bereits eine EREW PRAM ausreicht, um ein separierendes Lösungspolygon effizient zu berechnen:

Satz 7.4
Sind zwei Punktmengen POS *und* NEG *der Ebene mit Gesamtgröße N gegeben, und ist das zu* POS *und* NEG *assoziierte Circular-Arc-System proper, dann kann eine* EREW PRAM *mit $O(N)$ Prozessoren ein minimales konvexes Polygon, das* POS *von* NEG *separiert, in $O(\log N)$ Zeit bei einem Platzbedarf von $O(N)$ ermitteln.*

Beweis:
Betrachtet man die Tabelle auf Seite 75 und Folgerung 7.3, dann erkennt man, daß es als Beweis offenbar ausreicht, einen Algorithmus anzugeben, der unter der Voraussetzung, daß keine überdeckenden Punkte existieren, die Sichtbarkeitstangenten aller Punkte $q \in \mathrm{NEG}$ auf einer $O(N)$-Prozessor EREW PRAM in logarithmischer Zeit ermittelt.

Sind die Punkte aus NEG im Uhrzeigersinn um einen inneren Punkt p^* von HULL(POS) sortiert, und existieren keine überdeckenden Punkte, so kann man folgende wichtige Beobachtung machen:

Hat ein Punkt q_i als Stützpunkt der rechten Sichtbarkeitstangente den Punkt p_{r_i} und q_k als Stützpunkt der rechten Sichtbarkeitstangente den Punkt p_{r_k}, dann haben alle Punkte q_j zwischen q_i und q_k einen Punkt p_{r_j} als Stützpunkt der rechten Sichtbarkeitstangente, der zwischen p_{r_i} und p_{r_k} liegt. Dieselbe Beziehung gilt auch für Stützpunkte der linken Sichtbarkeitstangenten.

Formaler läßt sich diese Beobachtung schreiben als:

Für alle $i,\, k \in \{0, 1, \ldots, m-1\}$ gilt:

$$q_j \in \{q_i, q_{i+1}, \ldots, q_k\} \implies \begin{cases} & p_{r_j} \in \{p_{r_i}, p_{r_i+1}, \ldots, p_{r_k}\} \\ \wedge & p_{l_j} \in \{p_{l_i}, p_{l_i+1}, \ldots, p_{l_k}\} \end{cases} \quad (\star)$$

Beweis:
Gezeigt wird nur die Behauptung über die Stützpunkte rechter Sichtbarkeitstangenten. Der Beweis für Stützpunkte linker Sichtbarkeitstangenten verläuft völlig symmetrisch.

Seien $i,\, k \in \{0, 1, \ldots, m-1\}$ beliebig aber fest gewählt.
Bezeichne S^* den Sektor der Ebene, in dem $q_j \in \{q_i, q_{i+1}, \ldots, q_k\}$ auf Grund der Sortierung von NEG liegen muß. Weiterhin sei der Sektor S ähnlich definiert wie im Beweis zu Lemma 5.4 als der Bereich der Ebene, in dem ein Punkt s liegen muß, damit der Stützpunkt seiner rechten Sichtbarkeitstangente in der Menge $\{p_{r_i}, p_{r_i+1}, \ldots, p_{r_k}\}$ liegt.
Angenommen, es gilt $q_j \in \{q_i, q_{i+1}, \ldots, q_k\}$ aber nicht $p_{r_j} \in \{p_{r_i}, p_{r_i+1}, \ldots, p_{r_k}\}$, dann liegt q_j im Bereich $S^* \setminus S$ der Ebene.
Nach Konstruktion überdeckt q_j in diesem Fall aber den Punkt q_i im Widerspruch zur Voraussetzung, daß die Menge NEG keine überdeckenden Punkte enthält. Also gilt doch $p_{r_j} \in \{p_{r_i}, p_{r_i+1}, \ldots, p_{r_k}\}$.

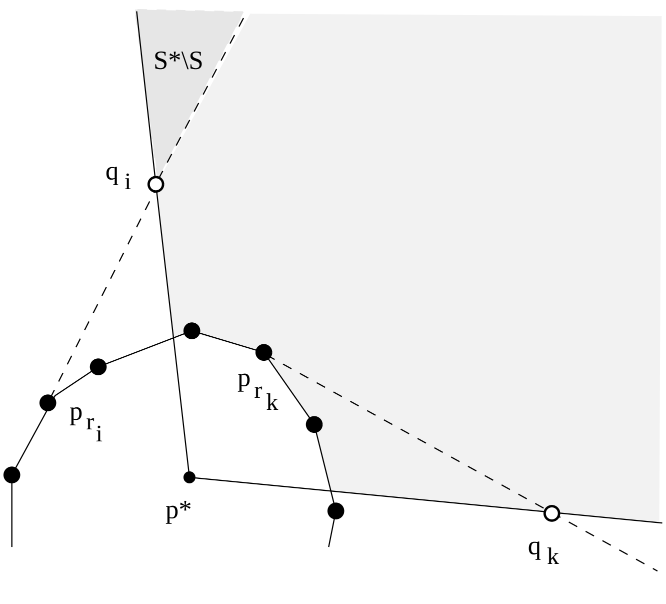


Bild 7.2: *Skizze zum Beweis von Beobachtung* $(\star)$ *. Der schraffierte Bereich stellt den Sektor* S^* *dar, in dem* q_j *liegen muß.* □

Enthält NEG keine überdeckenden Punkte, überträgt sich also quasi die Sortierung von NEG auf die Sortierung der Stützpunkte.

Ausnutzen läßt sich diese „Ordnungstreue“ wie folgt:

Angenommen, nicht alle Punkte aus NEG haben denselben Punkt als Stützpunkt der rechten Sichtbarkeitstangente, und es gilt $p_{r_0} = p_0 \neq p_{r_{m-1}}$. (Durch eine entsprechende Indizierung der Punkte aus BOUNDARY(POS) und NEG läßt sich die zweite Voraussetzung immer erreichen. Dazu später mehr.)

Dann erzeugt man ein Array A der Größe $n + m$, in das man die Punkte $q_0, q_1, \ldots, q_{m-1}$ und $p_0, p_1, \ldots, p_{n-1}$ nach folgenden Kriterien einsortiert:

1. Sortiere q_i genau dann vor q_j ein, wenn $i < j$.
2. Sortiere p_i genau dann vor p_j ein, wenn $i < j$.
3. Sortiere p_i genau dann vor q_j ein, wenn $i \leq r_j$.
   (Diese Entscheidung kann mit Lemma 5.4 in konstanter Zeit getroffen werden, denn $i \leq r_j$ ist äquivalent mit der Aussage $p_{r_j} \in \{p_i, p_{i+1}, \ldots, p_{n-1}\}$.)

Man beachte, daß auf Grund von Beobachtung ($\star$) die Sortierkriterien (insbesondere Kriterium 3.) widerspruchsfrei sind, und außerdem für alle Punkte aus BOUNDARY(POS) oder NEG ihre Positionen in A durch diese Kriterien eindeutig festgelegt sind. A hat die Struktur

$$A = (A_0, q_0, A_1, q_1, \ldots, A_{m-1}, q_{m-1}),$$

wobei jeder Block A_k entweder leer ist oder eine aufsteigend sortierte Folge von Punkten aus BOUNDARY(POS) enthält, so daß gilt:

$$(A_0, A_1, \ldots, A_{m-1}) = (p_0, p_1, \ldots, p_{n-1}).$$

Beispiel: Betrachtet man die Punktmengen BOUNDARY(POS) und NEG in Bild 2.4, so stellt man fest, daß keine überdeckenden Punkte existieren (CAS_1 ist ja auch proper) und $p_{r_0} = p_0 \neq p_{r_{m-1}}$ gilt. Außerdem sind die Punkte aus NEG im Uhrzeigersinn um einen inneren Punkt von HULL(POS) sortiert, so daß alle Voraussetzungen für die Erzeugung des Arrays A erfüllt sind. A ergibt sich als

$$A = (p_0, q_0, p_1, q_1, p_2, p_3, q_2, p_4, q_3, p_5, p_6, q_4, p_7, q_5).$$

Warum die vielleicht etwas unmotiviert wirkende Konstruktion von A überhaupt gemacht wurde, zeigt sich an dieser Stelle:

Aus A läßt sich nämlich zu jedem Punkt $q_j \in$ NEG der Stützpunkt p_{r_j} der rechten Sichtbarkeitstangente einfach ablesen:
p_{r_j} ist der letzte Punkt aus BOUNDARY(POS), der in A vor q_j einsortiert ist. Insbesondere ist r_j der maximale Index aller vor q_j einsortierten Punkte aus BOUNDARY(POS).

Beispiel: Aus dem Array A von oben liest man für die Punkte $q_0, q_1, \ldots, q_5$ die folgenden korrekten Stützpunkte ab:
$p_{r_0} = p_0, \quad p_{r_1} = p_1, \quad p_{r_2} = p_3, \quad p_{r_3} = p_4, \quad p_{r_4} = p_6, \quad p_{r_5} = p_7$

Hier nun der komplette Algorithmus, mit dem man unter der Voraussetzung, daß keine überdeckenden Punkte existieren, alle rechten Sichtbarkeitstangenten berechnen kann:

Schritt 1: Berechne einen inneren Punkt p^* von HULL(POS).

Schritt 2: Sortiere die Punkte $p_0, p_1, \ldots, p_{n-1}$ im Uhrzeigersinn um p^*.

Schritt 3: Dieser Schritt schafft die Voraussetzung $p_{r_0} = p_0 \neq p_{r_{m-1}}$:

- Berechne p_{r_0} wie in Kapitel 5 beschrieben.
- Damit $p_{r_0} = p_0$ gilt und die Punkte aus BOUNDARY(POS) weiterhin im Uhrzeigersinn sortiert sind, führe eine zyklische Vertauschung durch, indem jeder Punkt p_i durch den Punkt p_{i-r_0} ersetzt wird.
- Nun muß noch $p_0 \neq p_{r_{m-1}}$ sichergestellt werden:
  Teste deshalb mit Lemma 5.3, für welche Punkte $q_j \in$ NEG ebenfalls $p_{r_j} = p_0$ gilt. Auf Grund von Beobachtung $(\star)$ ist dies ein zusammenhängender Block $q_k, q_{k+1}, \ldots, q_{m-1}, q_0, q_1, \ldots, q_{k'}$.
  Für k gilt $p_{r_k} = p_0 \neq p_{r_{k-1}}$. Also leistet eine zyklische Verschiebung der Form $q_j \rightarrow q_{j-k}$ für alle Punkte aus NEG das Gewünschte.
  (Existiert ein solches k <u>nicht</u>, gilt $p_{r_0} = p_{r_1} = \ldots = p_{r_{m-1}}$, d.h. man hat alle Stützpunkte bereits ermittelt und kann abbrechen.)

Schritt 4: Jetzt wird das Array A ermittelt:
Definiere eine totale Ordnungsrelation "$\ll$" auf den Punkten von BOUNDARY(POS) und NEG durch

$$\begin{array}{rl} & (q_i \ll q_j \;:\Longleftrightarrow\; i < j) \\ \wedge & (p_i \ll p_j \;:\Longleftrightarrow\; i < j) \\ \wedge & (p_i \ll p_j \;:\Longleftrightarrow\; i \leq r_j) \quad \text{(verwende hierfür Lemma 5.4).} \end{array}$$

Sortiere nun die Punkte aus BOUNDARY(POS) und NEG bzgl. "$\ll$", und es ergibt sich das gewünschte Array A.

Schritt 5: Bestimme für jeden Punkt $q_j \in$ NEG den Stützpunkt seiner rechten Sichtbarkeitstangente p_{r_j}, wobei gilt:
$r_j = \max\{\, i \mid p_i \text{ ist in } A \text{ vor } q_j \text{ einsortiert}\}$.

Dieser Algorithmus läßt sich auf einer $O(n+m)$-Prozessor-EREW PRAM mit Hilfe der bekannten Standardtechniken mehr oder weniger „straight-forward" implementieren und erzielt eine Laufzeit von $O(\log(n+m))$. (In Schritt 1 kann p^* beispielsweise als der Mittelpunkt des Dreiecks mit den Eckpunkten p_0, p_1, p_2 gewählt werden. Für die Schritte 2 und 4 verwende man Pipelined-Parallel-Mergesort. Schritt 5 ist eine Aufgabe für Parallel-Prefix-Computation.)

Daß es zu keinen Schreibkonflikten kommt, ist klar.
Um Lesekonflikten bei der Anwendung der Lemmata 5.3 und 5.4 vorzubeugen, genügt es, in einem Preprocessing-Schritt die Datenstruktur jedes Punktes $p_i \in$ BOUNDARY(POS) um die Koordinaten seiner Nachbarpunkte p_{i-1} und p_{i+1} zu erweitern.
„Broadcastet" man zusätzlich in Schritt 3 die Daten von p_{r_0} und k und in Schritt 4 die Daten von p_{n-1}, so kommt es bei der parallelen Ausführung dieses Algorithmus an keiner Stelle zu Lesekonflikten.

Da Beobachtung $(\star)$ auch auf die Stützpunkte aller linken Sichtbarkeitstangenten zutrifft, ermittelt ein analoger Algorithmus sämtliche linken Sichtbarkeitstangenten. Insgesamt ergibt sich die Behauptung. $\square$

Schlußbemerkung:
Zur Zeit ist es nur möglich, überdeckende Punkte dadurch zu entdecken, daß man auf einer CREW PRAM ihre Sichtbarkeitstangenten ermittelt und anschließend überprüft, ob ihre Bögen im assoziierten Circular-Arc-System überlappend sind.
Gelänge es, überdeckende Punkte mittels irgendwelcher anderer Methoden auf einer EREW PRAM mit $O(N)$ Prozessoren in $O(\log N)$ Zeit zu finden, könnte man diese Punkte aus NEG löschen und anschließend den obigen Algorithmus zur Berechnung der Sichtbarkeitstangenten anwenden.
Ein minimales konvexes separierendes Polygon bzgl. der verkleinerten Punktmengen wäre immer noch eine Lösung des Ausgangsproblems (vgl. Folgerung 7.3).
Auf diese Weise hätte man automatisch einen optimalen parallelen Separierungsalgorithmus entwickelt, der zusätzlich auf dem schwächsten aller PRAM-Modelle, der EREW PRAM, lauffähig ist.

Literaturverzeichnis

[AG] M.J. Atallah, M.T. Goodrich. Efficient Parallel Solutions to Geometric Problems, *IEEE Transactions on Computers,* S. 411-417, 1985

[CG] R. Cole, M.T. Goodrich. Optimal Parallel Algorithms for Point-Set and Polygon Problems, *Algorithmica 7,* S. 3-23, 1992

[Co] R. Cole. Parallel Merge Sort, *SIAM Journal on Computing 3,* S. 770-785, 1988

[CV] R. Cole, U. Vishkin. Approximate Parallel Scheduling. Part I: The Basic Technique with Applications to Optimal Parallel List Ranking in Logarithmic Time, *SIAM Journal on Computing 17,* S. 128-142, 1988

[EP] H. Edelsbrunner, F.P. Preparata. Minimum Polygon Separation, *Information and Computation 77,* S. 218-232, 1988

[FS] P. Fischer, H.U. Simon. Two-Dimensional Separation Problems, *Technischer Bericht,* Saarbrücken 1990

P. Fischer, H.U. Simon. Separation Problems and Circular Arc Systems, in: *Proceedings 16th International Workshop WG '90, Berlin, June 1990, Springer-Verlag,* S. 251-259, 1990

[Go] M.T. Goodrich. Finding the Convex Hull of a Sorted Point Set in Parallel, *Information Processing Letters 26,* S. 173-179, 1987/1988

[KRS] C.P. Kruskal, L. Rudolph, M. Snir. The Power of Parallel Prefix, *IEEE Transactions on Computers,* S. 180-185, 1985

[MS] R. Miller, Q. Scout. Efficient Parallel Convex Hull Algorithms, *IEEE Transactions on Computers 37,* S. 1605-1619, 1988

[PS] F.P. Preparata, M.I. Shamos. Computional Geometry (An Introduction), *Lehrbuch, Springer-Verlag New-York,* 1985

[RR] A.S. RAO, C.P. RANGAN. Optimal Parallel Algorithms on Circular-Arc Graphs, *Information Processing Letters 33,* S. 147-156, 1989/1990

[TV] R.E. TARJAN, U. VISHKIN. An Efficient Parallel Biconnectivity Algorithm, *SIAM Journal of Computation 14,* S. 862-874, 1985

Grundlagen bezogen aus:

[ACG] A. AGGARWAL, B. CHAZELLE, L. GUIBAS, C.O DUNLAING, C. YaS. Parallel Computional Geometry, *Algorithmica 3,* S. 293-327, 1988

[Ja] J. JAJA. An Introduction to Parallel Algorithms, *Addison-Wesley Publishing Company, Inc.,* USA 1992

[KR] R.M. KARP, V. RAMACHANDRAN. Parallel Algorithms for Shared-Memory Machines, *Handbook of Theoretical Computer Science, Part A,* S. 871-941, 1988

[We] I. WEGENER. Effiziente Algorithmen für grundlegende Funktionen, *Teubner Verlag,* Stuttgart 1989