Seminar Ergebnisprüfung: Spotcheckers

Tieu Duyen Chung
227441

9. Dezember 2003

Zusammenfassung

Spotcheckers stellen im Rahmen der Ergebnisüberprüfung ein Konzept dar, das extrem schnell entscheiden kann, ob eine Programmausgabe annähernd korrekt ist. Genauer bedeutet dies, dass der Checker noch nicht einmal die gesamte Ein- und Ausgabe eines Programms betrachtet und trotzdem mit hoher Wahrscheinlichkeit eine richtige Aussage über die annähernde Korrektheit des Ergebnisses machen kann.

1 Einleitung und Motivation

Obwohl Programmkorrektheit von enormer Relevanz ist, reicht die Garantie eines annähernd richtigen Ergebnisses schon oftmals aus. Die zentrale Frage lautet also: Wie kann man mit möglichst geringem Aufwand und möglichst geringer Irrtumswahrscheinlichkeit entscheiden, ob das Programm für beliebige Eingaben eine annähernd korrekte Ausgabe liefert.
Die Idee für *Spotchecker* ist aus dem Alltag bekannt: Um sicherzustellen, dass die meisten Autofahrer nicht zu schnell fahren, werden an bestimmten Stellen Geschwindigkeitskontrollen durchgeführt.
Diese Ausarbeitung präsentiert ein Modell für Spotchecker, das unabhängig vom Verhältnis von Ein- zu Ausgabegröße ist. Im Folgenden wird an einem Beispielproblem für zwei der drei Möglichkeiten ein *Spotchecker* vorgestellt.
In Kapitel 2 wird ein *Spotchecker* formal definiert und die Definition kommentiert. Kapitel 3 und 4 befassen sich mit der Überprüfung von Programmen, bei denen die Ausgabe asymptotisch genauso groß wie die Eingabegröße ist. Hier werden zur Veranschaulichung das Sortierproblem und das "Konvexe-Hülle-Problem" genauer untersucht.
Das fünfte Kapitel stellt die Arbeitsweise eines *Spotcheckers* vor, wenn die Eingabe wesentlich größer als die Ausgabe ist. Dazu betrachten wir das Entscheidungsproblem "Element-Distinctness".
Zum Schluss wird die Problematik noch kurz zusammengefasst.

2 Definition

Definition 1 (ϵ-*Spotchecker*) *Sei f eine Funktion, die von einem Programm P berechnet werden soll. Sei $\Delta(\cdot,\cdot)$ eine Abstandsfunktion. Dann ist $\mathcal{C}$ ein ϵ-Spotchecker für f mit Abstandsfunktion Δ, wenn*

1. *Bei gegebenem Programm P, das f berechnen soll, und gegebenem ϵ soll für jede beliebige Eingabe x gelten: $\mathcal{C}$ gibt mit Wahrscheinlichkeit $\geq 3/4$ (bzgl. der internen Münzwürfe von $\mathcal{C}$) PASS aus, wenn $\Delta(\langle x, P(x)\rangle, \langle x, f(x)\rangle) = 0$ und FAIL, wenn für jede Eingabe y $\Delta(\langle x, P(x)\rangle, \langle y, f(y)\rangle) > \epsilon$*
   (Dies garantiert annähernde Korrektheit.)
2. *Die Laufzeit von $\mathcal{C}$ ist $o(|x| + |f(x)|)$.*
   (Diese Bedingung gewährleistet, dass der Aufwand gering ist, da nicht genug Zeit bleibt um sowohl Ein- als auch Ausgabe zu lesen.)

Man beachte, dass bei dieser Definition der Abstand der Ausgabe zum richtigen Ergebnis mit der Wahrscheinlichkeit FAIL auszugeben korreliert: Je größer die Distanz ist, desto wahrscheinlicher gibt der Checker FAIL aus.
Eine schwächere Bedingung dagegen wäre, bei beliebiger Entfernung zum richtigen Ergebnis mit bestimmter Wahrscheinlichkeit FAIL auszugeben. In diesem Fall ist es möglich, dass die Programmausgabe sehr weit vom richtigen Ergebnis entfernt ist und der Checker trotzdem PASS ausgibt.
Der Checker erfüllt wegen der zweiten Bedingung die "little-Oh"-Eigenschaft, d.h. durch die Begrenzung der Laufzeit muss sich der Checker von jedem Algorithmus, der die Funktion f berechnet, unterscheiden. Weiterhin beobachten wir, dass sich bei $O(\lg 1/\delta)$ Wiederholungen eine Konfidenz von $1-\delta$ ergibt.

3 Sortieren

Bei Sortierproblemen hat die Eingabe dieselbe Größenordnung wie die Ausgabe: Das Programm erhält als Eingabe eine Menge der Mächtigkeit n und gibt eine sortierte Liste (ebenfalls der Länge n) zurück.
Die Eingabe des *Spotcheckers* besteht also aus einem Array der Länge n, auf dessen Elementen eine Ordnungsrelation definiert ist. Außerdem sollen alle Elemente unterschiedlich sein. Zusätzlich nehmen wir an, das sowohl der Zugriff auf die einzelnen Elemente als auch der Vergleich zweier Elemente konstante Zeit in Anspruch nimmt.
Im Folgenden wird ein $2\epsilon-$*Spotchecker* vorgestellt, dessen Laufzeit $O(\lg n)$ beträgt. Dazu definieren wir die Abstandsfunktion Δ wie folgt: Seien x, y Listen von Elementen und bezeichne $\rho(u,v)$ die Anzahl der Elemente, die zu einem String u hinzugefügt bzw. aus u gelöscht werden müssen um den String v zu erhalten.

$$\Delta(\langle x, P(x)\rangle, \langle y, f(y)\rangle) = \begin{cases} \infty & x \neq y \text{ oder } |P(x)| \neq |f(y)| \\ \frac{\rho(P(x), f(y))}{|P(x)|} & \text{sonst} \end{cases}$$

Die Idee des Checkers besteht darin das Ergebnis in zwei Schritten zu überprüfen:

1. Test, ob aufsteigende Subsequenz der Länge $(1-\epsilon)|x|$ in $P(x)$ vorhanden
2. Test, ob die Mengen $P(x)$ und x einen Overlap mindestens der Größe $(1-\epsilon)|x|$ besitzen.

Falls $P(x)$ beide Tests besteht, ist der Abstand zum richtigen Ergebnis klein genug ($\Delta(\langle x, P(x)\rangle, \langle y, f(y)\rangle) \leq 2\epsilon$), und wir erhalten einen $2\epsilon-$*Spotchecker*.

3.1 *Spotchecker* für aufsteigende Sequenz

Wir beginnen diesen Abschnitt, indem wir direkt den *Spotchecker* vorstellen:

```
Procedure Sort-Check(A, ε)
repeat O(1/ε) times
    choose i ∈_R [1, n]
    for k ← 0...⌈lg i⌉ do
       repeat O(1) times
          choose j ∈_R [1, 2^k]
          if (A[i − j] > A[i]) then return FAIL
    for k ← 0...⌈lg(n − i)⌉ do
       repeat O(1) times
          choose j ∈_R [1, 2^k]
          if (A[i] > A[i + j]) then return FAIL
return PASS
```

Theorem 1 *Sort-Check(A, ϵ) läuft in $O((1/\epsilon)\lg n)$ und erfüllt folgende Bedingungen:*

1. *Wenn A sortiert ist, gilt Sort-Check(A, ϵ)=PASS.*
2. *Wenn A keine aufsteigende Untersequenz mind. der Länge $(1-\epsilon)n$ hat, dann Pr(Sort-Check(A, ϵ)=FAIL)$\geq 3/4$.*

Um dies zu zeigen definieren wir für jedes Element des Arrays Nachbarschaften.

Definition 2 $\forall i\colon \Gamma^{+}_{(t,t')}(i) = \{A[j] |\ A[j] > A[i],\ j > i,\ t \leq j \leq t'\}$
$\forall i\colon \Gamma^{-}_{(t,t')}(i) = \{A[j] |\ A[j] < A[i],\ j < i,\ t \leq j \leq t'\}$

Γ^{+} und Γ^{-} bezeichnen also die Elemente im Intervall (t, t') , die in Bezug auf $A[i]$ die richtige Position im Array haben.

Definition 3 (schweres Element) $\forall k, 0 \leq k \leq \lg i\colon |\Gamma^{-}_{(i-2^k,i)}(i)| \geq \eta 2^k$ *und* $\forall k, 0 \leq k \leq \lg(n-i)\colon |\Gamma^{+}_{(i,i+2^k)}(i)| \geq \eta 2^k$ *mit* $\eta = 3/4$

Ist ein Element $A[i]$ schwer, so ist es sehr wahrscheinlich, dass dieses Element $A[i]$ an der richtigen Position steht, da die meisten Elemente $A[j]$, die vor $A[i]$ im Array stehen, tatsächlich kleiner und umgekehrt die meisten $A[k]$, die hinter $A[i]$ im Array stehen, wirklich größer als $A[i]$ sind.

Lemma 1 *Seien $A[i], A[j]$ schwere Elemente mit $i < j$; $i, j < n$. Dann gilt $A[i] < A[j]$.*

Dieses Lemma besagt, dass schwere Elemente immer in richtiger Reihenfolge zu einander stehen.

Beweis (Lemma 1)
Zeige: Es existiert ein k mit $A[i] < A[k]$ und $A[k] < A[j]$, dann folgt die Behauptung.
Wähle m so, dass $2^m \leq (j-i)$, aber $(2m+1) \geq (j-i)$.
Sei $l = (j-i) - 2^m$ und $I = [j-2^m, i+2^m]$.
Dann gilt: $|I| = (i+2^m) - (j-2^m) + 1 = 2^m - l + 1$.
Wir betrachten in I die Mengen $G := \{A[k] | A[k] > A[i]\}$ und $K := \{A[k] | A[k] < A[j]\}$.
Wenn $|G| + |K| > |I|$, dann gilt wegen des Schubfachprinzips: Es existiert mindestens ein k mit $A[k] \in I$ mit $A[i] < A[k] < A[j]$.
Weil $A[i]$ laut Voraussetzung ein schweres Element ist gilt: $|G| \geq \eta 2^m - ((j-2^m) - i) = \eta 2^m - l$
Weil auch $A[j]$ ein schweres Element ist, gilt: $|K| \geq \eta 2^m - (j - (i+2^m)) = \eta 2^m - l$
Also ergibt sich für die Summe der Mächtigkeiten von G und K: $|G| + |K| = (\eta 2^m - l) + (\eta 2^m - l) \geq |I| = 2^m - l + 1$, falls $l \leq 2^{m-1}$
Falls $l > 2^{m-1}$: Führe Beweis mit $m+1$ statt mit m.

□

Theorem 2 ist dann eine unmittelbare Folgerung aus dem Lemma 1:

Theorem 2 *Sind $(1-c)n$ schwere Elemente in A enthalten, so besitzt A eine aufsteigende Untersequenz mindestens der Länge $(1-c)n$.*

Mit Hilfe dieser Lemmata lässt sich nun Theorem 1 beweisen:
Beweis (Theorem 1)
Der erste Teil von Theorem 1 gilt, weil wenn `Sort-Check(`A, ϵ`)` FAIL ausgibt, stehen mindestens zwei Element an der falschen Position im Array, d.h. A ist nicht sortiert.
Der Beweis des zweiten Teils von Theorem 1 soll nur skizziert werden.
Da wir dies mit Hilfe der Chernoff-Schranken zeigen, soll zunächst deren Definition angeführt werden:

Definition 4 (Chernoff-Schranken) *Geg.: Bernoulli-Experiment mit Trefferwahrscheinlichkeit p.*
Die Zufallsvariable Y_n zähle die Treffer. Für den Erwartungswert erhält man $E[Y_n] = np$. Die Chernoff-Schranken lauten dann:

1. $P(Y_n \geq (1+\epsilon)np \leq e^{-\epsilon^{2np/3}} 0 \leq \epsilon \leq 1$
2. $P(Y_n \geq (1-\epsilon)np \leq e^{-\epsilon^{2np/2}} 0 \leq \epsilon \leq 1$

Wir möchten zeigen, dass, wenn A keine aufsteigende Untersequenz $\geq (1-c)n$ besitzt, `Sort-Check(`A, ϵ`)` mit Wahrscheinlichkeit $\geq 3/4$ FAIL ausgibt. Dazu schränken wir die Irrtumswahrscheinlichtkeit von `Sort-Check(`A, ϵ`)` genügend ein. `Sort-Check(`A, ϵ`)` irrt sich, wenn A weniger als cn schwere Elemente (also mehr als cn leichte Elemente) besitzt, dies aber nicht erkennt. Dieser Fall tritt ein, wenn

1. nur schwere Elemente gezogen werden.
2. ein leichtes Element für schweres Element gehalten wird.

Beide Wahrscheinlichkeiten lassen sich mit Hilfe der Chernoff-Schranken einschränken. Fall 1. lässt sich wie folgt als Bernoulli Experiment modellieren: Wir ziehen schwere Elemente mit bestimmter Trefferwahrscheinlichkeit $p < 1-c$. In den $O((1/\epsilon) \lg n)$

ziehen wir ausschließlich schwere Elemente, obwohl der Erwartungswert uns weniger schwere Elemente angibt. Die Anwendung der 1. Chernoff-Schranke zeigt nun, dass die Wahrscheinlichkeit für diese Abweichung klein genug ist. Fall 2. tritt nur dann ein, wenn der Checker für ein k übersieht, dass $|\Gamma^-_{(i-2^k,i)}(i)|$ oder $|\Gamma^+_{(i,i+2^k)}(i)|$ zu klein ist. Wiederum lässt sich ein entsprechendes Bernoulli Experiment modellieren und durch Anwendung der Chernoff-Schranken kann man erneut zeigen, dass auch diesmal die Irrtumswahrscheinlichkeit klein genug ist.
Insgesamt gilt daher: Wenn $\rho >$ größer als ϵn ist (dann ist der Abstand Δ ebenfalls größer als ϵ), dann gibt `Sort-Check`(A, ϵ) FAIL mit Wahrscheinlichkeit $\geq 3/4$ aus.

□

3.2 *Spotchecker* für Overlap

Nachdem wir im vorherigen Abschnitt einen $\epsilon-$*Spotchecker*für aufsteigende Untersequenzen nachgewiesen haben, bleibt zu zeigen: Für den Test auf einen ausreichend großen Overlap existiert ebenfalls ein $\epsilon-$*Spotchecker*.

Lemma 2 *Seien A, B zwei Listen mit $|A| = |B| = n$.A sortiert und aus unterschiedlichen Elementen bestehend. Dann existiert ein Checker*

- *mit Laufzeit $O(\lg n)$.*
- *für den gilt, wenn A sortiert ist und $|A \cap B| = n$, dann liefert er mit hoher Wahrscheinlichkeit die Ausgabe PASS.*
- *und falls $|A \cap B| \leq \epsilon n$, dann liefert er mit hoher Wahrscheinlichkeit die Ausgabe FAIL.*

Beweis (Lemma 2) Sei A sortiert.

```
choose b ∈ B randomly
search b in A via binary search
if b ∉ A return FAIL
else return PASS.
```

ist der gewünschte Checker. Jeder Test hat Komplexität $O(\lg n)$ und da nur eine konstante Anzahl an Tests notwendig ist, ergibt sich die Behauptung.

□

Wir erhalten also durch die Hintereinanderausführung der in Kapitel 3.1 und 3.2 vorgestellten $\epsilon-$*Spotchecker* für die Teilprobleme einen $2\epsilon-$*Spotchecker* für das Sortierproblem.

□

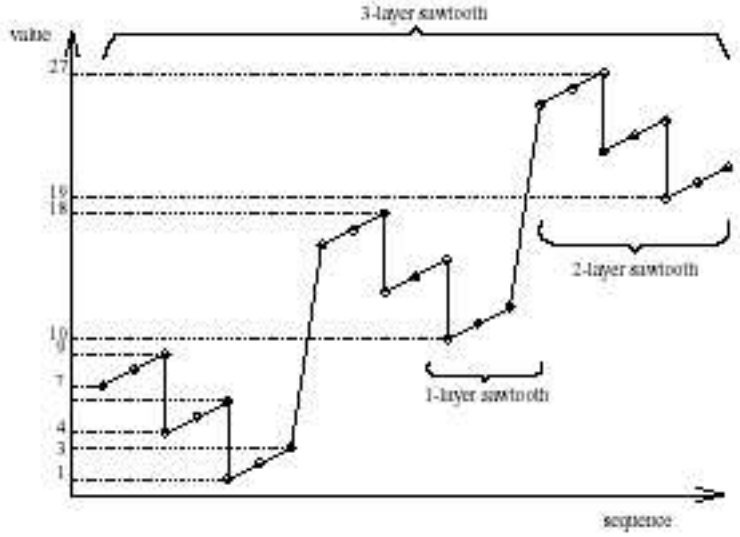

Figure 1: 3-layer saw-tooth: an $lst_3(3,3,3)$ sequence.

3.3 Untere Schranke

Im Folgenden wollen wir nun die Idee für den Beweis liefern, dass die Komplexität unseres *Spotcheckers* ($O(\lg n)$) eine untere Schranke für sämtliche *Spotchecker* darstellt, die das Sortierproblem überprüfen. Die Behauptung lautet also: Es existiert kein *Spotchecker* mit Laufzeit $o(\lg n)$. Der Beweis wird indirekt geführt, indem wir annehmen, dass es einen *Spotchecker* mit Laufzeit $o(\lg n)$ für das Sortierproblem gäbe. Dann zeigen wir, dass die Irrtumswahrscheinlichkeit von $\mathcal{C}$ zu groß ist:
1. Es existiert eine komplett sortierte Eingabe y ($\Delta = 0$), für die der Checker FAIL ausgibt oder 2. der Checker gibt PASS aus, obwohl keine aufsteigende Sequenz der Länge $\Omega(n)$ existiert ($\Delta > \epsilon$). Die Wahrscheinlichkeit für den 1.Fall oder den 2.Fall muss größer als 1/4 sein. Wir zeigen, dass die Wahrscheinlichkeit für den 2.Fall schon zu groß ist und schlussfolgern daraus, dass $\mathcal{C}$ gar keinen *Spotchecker* darstellt.
In Ref[1] wird eine solche Eingabe y, dem sog. 3-Schicht-Sägezahn (engl: 3-layer sawtooth) (s. Figure 1), konstruiert. Dass diese Sequenz unsortiert ist, kann der *Spotchecker* nur daran erkennen, wenn er ein $A[i]$ aus einem 1-Schicht-Sägezahn (1-layer sawtooth) zieht und $A[j]$ aus dem benachbarten 1-Schicht-Sägezahn zieht. Zugleich muss gelten, dass beide Elemente aus demselben 2-Schicht-Sägezahn stammen. Die Sequenz wurde aber so modelliert, das die Wahrscheinlichkeit hierfür zu gering ist.

□

4 Konvexe Hülle

Als nächstes Beispiel eines Problems, bei dem die Eingabegröße in etwa genauso groß wie die Ausgabegröße ist, untersuchen wir das "Konvexe Hülle-Problem".
Hierbei erhält das Programm als Eingabe eine Menge M von n Punkten in der euklidischen Ebene. Die Ausgabe besteht aus $\langle x_0, x_1, ..., x_k \rangle$, einer Sequenz von Zeigern auf $k+1$ Punkte in M, die gegen Uhrzeigerrichtung eine konvexe Hülle ergeben sollen.
Die Abstandsfunktion für den *Spotchecker* definieren wir wie folgt:
Sei f die Funktion, die die konvexe Hülle einer Punktmenge korrekt bestimmt.

$$\Delta(\langle x, P(x)\rangle, \langle y, f(y)\rangle) = \begin{cases} \infty & y \neq P(x) \\ max\{d_{con}, d_{hull}\} & \text{sonst} \end{cases}$$

wobei d_{con} der minimale Anteil an Punkten in x ist, deren Entfernung die Programmausgabe $P(x)$ zu dem konvexen Polygon $f(y)$ macht und d_{hull} den Anteil an Punkten in x darstellt, der entfernt werden muss, damit $f(y)$ die Hüllenbedingung erfüllt.
Wir zeigen folgendes Theorem:

Theorem 3 *Es existiert ein $\epsilon-$Spotchecker für das "Konvexe-Hülle-Problem" mit Komplexität $O(\lg n)$.*

Wieder arbeitet unser *Spotchecker* in zwei Phasen:

1. Test auf Konvexität: Wir tolerieren eine Abweichung von ϵk Elementen.
2. Test, ob ausreichend Punkte in dem zurückgelieferten konvexen Polygon liegen (Hüllenbedingung). Auch hier ist es akzeptabel, wenn ϵk Elemente außerhalb der vermeintlichen Hülle liegen.

4.1 *Spotchecker* für Konvexität

Zunächst definieren wir eine Relation $\mathcal{R}$ auf den Winkeln von Kanten, die uns einen Anhaltspunkt für Konvexität liefert. Der Winkel einer Kante soll folgendermaßen bestimmt werden:
Sei CH eine Menge von $k+1$ Kanten, $e_i = (x_i, x_{i+1 \bmod k+1})$. Dann soll für den Winkel der ersten Kante gelten: $\angle e_0 = 0$. Als $\angle e_i$ bezeichnen wir den zwischen e_0 und e_i eingeschlossenen Winkel, wobei $\angle e_i$ im Intervall $[0, 2\pi)$ liegen soll.

Definition 5 $0 \leq i, j \leq k$: $e_i \mathcal{R} e_j \Leftrightarrow$

1. $i < j$
2. $x_{i+1} = x_j$ *und* $0 < \angle e_j - \angle e_i < \pi$
   oder $0 < \angle e_j - \angle(x_{i+1}, x_j) < \pi$ *und* $0 < \angle(x_{i+1}, x_j) - \angle e_i < \pi$

$e_k \mathcal{R} e_0$, *wenn* $\angle e_k > \pi$

Wir beobachten, dass in einem konvexen Polygon sämtliche aufeinanderfolgende Kanten über $\mathcal{R}$ miteinander in Relation stehen. Das nächste Lemma zeigt, dass auch die Umkehrung gilt.

Lemma 3 *Sei $S = \langle e_0, ..., e_l \rangle$ eine Sequenz von Kanten mit $e_i \mathcal{R} e_{i+1}$ $\forall i$ mit $0 \leq i \leq l$. Sei C das Polygon, das sich ergibt, wenn man alle e_i aus S ggf. durch Einfügen neuer Kanten verbindet. Dann gilt:*

1. *C besitzt keine Schleifen.*
2. *C ist konvex.*

Beweis (Lemma 3)
Wenn $\mathcal{C}$ keine Schleifen besäße, ist $\mathcal{C}$ ein einfaches Polygon, in dem alle Winkel der Kanten in aufsteigender Reihenfolge sortiert sind. Wegen der Geschlossenheit des Polygons kann der Winkelunterschied zwischen zwei aufeinander folgenden Kanten nicht größer als π sein. Daher sind alle inneren Winkel von $\mathcal{C}$ kleiner als π und $\mathcal{C}$ ist konvex.
Zeigen wir nun Lemma 3.1.: Wir führen den Beweis indirekt, indem wir sämtliche Fälle, in denen $\mathcal{C}$ mindestens eine Schleife besitzt zum Widerspruch führen. Wenn $\mathcal{C}$ eine Schleife besitzt, dann muss $\mathcal{C}$ wie in Figure 2 abgebildet aussehen.

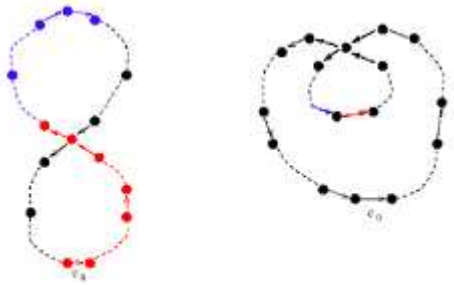

Figure 2: Looping closed curves.

Im ersten Fall (linke Seite von Figure 2) beobachten wir, dass in $\mathcal{C}$ ein Richtungswechsel vollzogen wird: Das Polygon startet mit einer Linkskurve (rot markiert), die nach dem Schnittpunkt x in eine Rechtskurve (blau markiert) übergeht. Sei e_{right} die erste Kante in der Rechtskurve und $e_{right-1}$ die vorhergehende Kante. Dann gilt: $\angle e_{right} - \angle e_{right-1} < 0$, da sonst kein Wechsel von Links- in Rechtskurve möglich wäre. Dies verstößt aber gegen Bedingung 1.2 der Definition von $\mathcal{R}$, d.h. $e_{right-1} \not\mathcal{R} e_{right}$. Das ist aber ein Widerspruch zur Annahme, dass alle aufeinanderfolgenden Kanten miteinander über $\mathcal{R}$ in Relation stehen.
Im zweiten Fall besitzt das Polygon keinen Richtungswechsel.(rechte Seite von Figure 2) $\mathcal{C}$ besteht aus zwei ineinander geschachtelten Polygonen. Sei $\mathcal{C}'$ das im Innern liegende Teilpolygon. Sei e_{first} die erste Kante in $\mathcal{C}'$, deren Winkel zwischen 0 und π liegt (rote Kante) (und $e_{first-1}$ (also die vorhergehende Kante,in Figure 2 rechts blau markierte Kante) habe einen Winkel $> \pi$. Ein solches Paar $(e_{first}, e_{first-1})$ muss es geben, da $\mathcal{C}$ sonst keine Schleife haben könnte. Dann gilt $\angle e_{first} - \angle e_{first-1} < 0$, also $e_{first} \not\mathcal{R} e_{first-1}$. Dies ist ein Widerspruch zur Annahme. Da wir beide Fälle, in denen $\mathcal{C}$ eine Schleife haben könnte zum Widerspruch geführt haben, folgt der erste Teil unseres Lemmas.

□

Die folgende Prozedur stellt den gewünschten *Spotchecker* zur Überprüfung auf Konvexität dar:

```
Procedure Convex-Check(CH, ε)
run Sort-Check(CH, ε/2), replacing < with R
if e_k or e_0 is not heavy return FAIL
if ∠e_k ≤ π return FAIL
return PASS
```

Theorem 4 *Wenn CH* `Convex-Check`*(CH, ϵ) besteht, dann kann CH konvex gemacht werden, indem $\leq \epsilon k$ Knoten entfernt werden.*

Beweis (Theorem 4)
Wenn der Test bestanden wird, existieren zwei disjunkte Subsequenzen von CH (s.Figure 4)

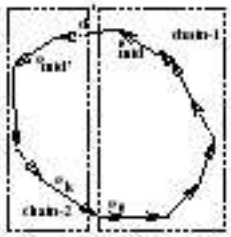

Figure 4: The two chains.

mit Gesamtlänge $\geq (1-\epsilon)k$ (Dies liegt daran, dass `Sort-Check` mit Parameter $\delta/2$ aufgerufen wird). Es müssen also höchstens k Kanten entfernt werden und dementsprechend höchstens k Knoten.
`Convex-Check` ist damit ein $\epsilon-$*Spotchecker.*
Da `Sort-Check` nur dann angewendet werden kann, wenn eine transitive Relation vorhanden ist, zeigen wir im folgenden Lemma die eingeschränkte Transitivität der $\mathcal{R}$ Relation.

Lemma 4 (eingeschränkte Transitivität) *Falls $e_i \mathcal{R} e_j$ und $e_j \mathcal{R} e_k$ mit $\angle e_k - \angle e_i < \pi$ gilt, dann folgt $e_i \mathcal{R} e_k$.*

Beweis (Lemma 4)
Wir betrachten nur den Fall $x_{i+1} \neq x_j$ und $x_{j+1} \neq x_k$. Setze $e{:=}(x_{i+1}, x_j)$, $e'{:=}(x_{j+1}, x_k)$. Dann gilt $\angle e_i < \angle e < \angle e_j < \angle e' < \angle e_k$.

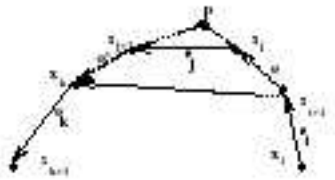

Figure 3: Transitivity under restricted conditions.

Aus $\angle e' - \angle e < \pi$ folgt, dass ein Schnittpunkt p der beiden Kanten existiert, so dass x_{i+1}, p, x_k ein Dreieck bilden. (s. Figure3). Damit gilt: $\angle e < \angle(x_{i+1}, x_k) < \angle e'$, d.h. $e_i \mathcal{R}\ e_k$

□

Lemma 5 *Wenn* `Convex-Check`(CH, ϵ) *=PASS, dann* $e_{mid} \mathcal{R} e_{mid'}$.

Beweis
Wir führen den Beweis indirekt. Dazu nehmen wir $e_{mid}\ \not\mathcal{R} e_{mid'}$ an. e_{mid} und $e_{mid'}$ können nicht adjazent sein, weil beide schwere Elemente sind. Dann muss ein e_r existieren mit $e_{mid} \mathcal{R} e_{mid'}$, wobei e_r kein schweres Element sein darf. (Sonst würde $e_{mid}\ \not\mathcal{R} e_{mid'}$ aufgrund der eingeschränkten Transitivität gelten.) Daher gilt $\angle e_{mid'} - \angle e_{mid} > \pi$.
Setze d:=$(x_{mid+1}, x_{mid'})$. Wegen $e_{mid}\ \not\mathcal{R} e_{mid'}$ muss einer der beiden folgenden Fälle gelten:

1. $\angle d - \angle e_{mid} > \pi$ oder
2. $\angle e_{mid'} - \angle d > \pi$

Nimm o.B.d.A. (2.) an.
Für die Folge $e_0, ..., e_{mid}, d, e_{mid'}, ..., e_k$ gilt $e_i \mathcal{R} e_j$. Daher liegt ein konvexes Polygon mit aufsteigende Winkelfolge vorliegen. Weil aber nun $\angle e_{mid'} - \angle d > \pi$ gelten soll, muss $e_{mid'}$ größer als π sein. Für alle nachfolgenden e_j (also $j > mid'$) muss $\angle e_j$ größer π sein, was einen Widerspruch zur Geschlossenheit des Polygons darstellt.

□

Als nächstes stellen wir einen *Spotchecker* zur überprüfung der Hüllenbedingung vor. Die Idee besteht darin $O(1/\epsilon)$ Knoten zu wählen und für jeden dieser Knoten zu testen, ob er im konvexen Polygon liegt. Der Test funktioniert dabei wie folgt: Wenn $v \in CH$ innerhalb des konvexen Polygons liegt, dann existiert ein Dreieck y, y', y'', wobei y' und y'' adjazent sind und v im Dreieck liegt. Dann gilt: $\angle(y, y') \leq \angle(v, y) \leq \angle(y'', y)$. Suche (y', y'') mittels binärer Suche in $O(\lg n)$. Mit Hilfe der Chernoff-Schranken lässt sich zeigen, dass die Irrtumswahrscheinlichkeit klein genug ist. Problematisch ist jedoch, dass die binäre Suche auf dem konvexen Polygon, das sich aus CH ergibt, durchgeführt werden müsste.

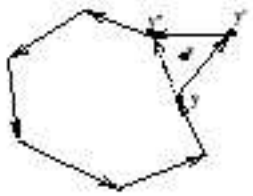

Figure 5: Potential problem caused by vertex out of sequence in CH.

In CH ist möglicherweise die Winkelfolge nur annähernd aufsteigend sortiert, so dass CH nicht unbedingt konvex sein muss. Die binäre Suche könnte daher entweder ein falsch positives oder ein falsch negatives Ergebnis liefern. Nur der zweite Fall ist

relevant (der erste wird nämlich durch eine falsche Reihenfolge der Winkel der Kanten verursacht); dieser Fall wird in Figure 5 verdeutlicht. Wir sehen, dass hierbei die Kante (y', y'') nicht in dem konvexen Polygon liegt. Dieses Problem lässt sich dadurch beheben, indem man zusätzlich in $O(\lg k)$ überprüft, ob (y', y'') ein schwere Kante ist.
Wenn v nicht in dem konvexen Polygon liegt, dann soll der Checker FAIL ausgeben. Für jeden Test ergibt sich also ein Aufwand von $O(\lg k)$, und da nur eine konstante Zahl an Tests erforderlich ist, folgt die Behauptung.

5 Element-Distinctness

In diesem Kapitel soll die Idee für den Begriff der Nähe zum korrekten Ergebnis vermittelt werden, wenn die Programmausgabe asymptotisch kleiner als die Programmeingabe ist. Dazu betrachten wir das *Element-Distinctness*-Problem. Hierbei erhält das Programm eine Menge A mit $|A| = n$ als Eingabe und soll entscheiden, ob alle Elemente unterschiedlich sind. Es genügt, wenn A mehr als $(1 - \epsilon)n$ unterschiedliche Elemente hat.
Sei f die Funktion, die das *Element-Distinctness*-Problem korrekt entscheiden kann mit

$$f(A) = \begin{cases} 0 & \text{wenn } A \text{ nicht } n \text{ unterschiedl El besitzt} \\ 1 & \text{sonst} \end{cases}$$

Dann definieren wir die Abstandsfunktion wie folgt:

$$\Delta(\langle x, P(x)\rangle, \langle y, f(y)\rangle) = \begin{cases} \infty & P(x) \neq f(y) \\ \frac{\rho(x,y)}{|x|} & \text{sonst} \end{cases}$$

mit $\rho(x, y) =$ Anz. El., die zu x hinzugefügt bzw. aus x gelöscht werden müssen um y zu erhalten.

Diese Definition von Δ besagt also: Wenn das Programm von einer Eingabe fälschlicherweise behauptet, alle Elemente seien unterschiedlich, darf der *Spotchecker* trotzdem PASS sagen, falls es eine andere Eingabe y gibt, die nah genug an der eigentlichen Eingabe x liegt, für die die Programmausgabe zutreffen würde.
Ohne Beweis wird nun ein $\epsilon-$*Spotchecker* mit Laufzeit $\tilde{O}(\sqrt{(\epsilon n)})$ gezeigt:

```
Procedure Element-Distinctness-Check(A, ε)
choose random √(εn) elements X from A
if X has any repeated elements return FAIL
return PASS
```

6 Zusammenfassung

In dieser Ausarbeitung wurde das Konzept der *Spotchecker* vorgestellt. Diese können eingesetzt werden, wenn es nicht unbedingt erforderlich ist, dass die Programmausgabe hundertprozentig richtig ist. Sie garantieren aber, dass die Ausgabe nah genug am korrekten Ergebnis ist. Der enorme Vorteil von *Spotcheckern* besteht in ihrer Geschwindigkeit: Ihre Laufzeit beträgt o(Eingabe + Ausgabe).

Literatur

[1] Ergün, Kannan, Kumar, Rubinfeld, Viswanathan: Spot-Checkers