Rheinische Friedrich-Wilhelms-Universität Bonn
Institut für Informatik - Abteilung I

Kleinste farbumspannende Rechtecke

Diplomarbeit

von
Andreas Lotz

Gutachter: Prof. Dr. Rolf Klein
Prof. Dr. Christel Baier

Bonn, den 29. März 2005

„Wer in medias res geht, spart Umschweife und Vorreden.“
Horaz (65-8 v. Chr.)

Inhaltsverzeichnis

1 Einführung 1-1
1.1 Motivation 1-1
1.2 Das formale Problem 1-2
1.3 Thema der Arbeit 1-2

2 Voraussetzungen und Definitionen 2-3
2.1 Voraussetzungen 2-3
2.2 Definitionen 2-3
2.3 Weitere Vorüberlegungen 2-4
2.4 Die Struktur MaxElem 2-6
2.5 Der Sonderfall 2-8

3 Der ursprüngliche Algorithmus 3-9
3.1 Erstes geeignetes Paar 3-9
3.2 Schema des Algorithmus 3-10
3.3 Die Kosten 3-11

4 Der verbesserte Algorithmus 4-16
4.1 Die Idee 4-16
4.2 Erstellung von Farbkonturen 4-17
4.3 Die Liste MaxElem 4-19
4.4 Schema des Algorithmus 4-22
4.5 Die Kosten (1) 4-23
4.6 Initialisierung von MaxElem in O(n) 4-23
4.7 Die Kosten (2) 4-26

5 Beliebige Orientierung 5-27
5.1 Einige Vorüberlegungen 5-27
5.2 Schema des Algorithmus 5-28
5.3 Die Kosten 5-32
5.4 Warum nur $n(n\text{-}1)$ Orientierungen? 5-37

6 Das Java-Applet 6-41
6.1 Die Klassen 6-41
6.2 Relevante Methoden 6-43
6.3 Bedienhinweise 6-45

7 Zusammenfassung und Ausblick 7-48
7.1 Zusammenfassung 7-48
7.2 Ausblick 7-49

Symbolverzeichnis 50

Abbildungsverzeichnis 53

Literatur- und Quellenverzeichnis 54

1 Einführung

1.1 Motivation

Eine Frage, die wahrscheinlich jedem Diplomanden der theoretischen Informatik gestellt wird, ist: "Was macht man dann damit?" Diese Frage kann für das Thema dieser Arbeit durch die Darstellung des folgendenden realitätsnahen Problems beantwortet werden:

Jemand sucht eine Wohnung in der Nähe einer Post, eines Bahnhofs, eines Supermarktes und einer U-Bahn-Haltestelle. Dafür nimmt er sich einen Stadtplan zur Hand und zeichnet die Lage sämtlicher oben erwähnten Einrichtungen jeweils mit unterschiedlichen Farben in diesen ein (freie Wohnung = gelb, Post = rot, ...). Jetzt stellt sich die Frage, von welcher freien Wohnung die Entfernungen zu allen obigen Einrichtungen am geringsten sind.

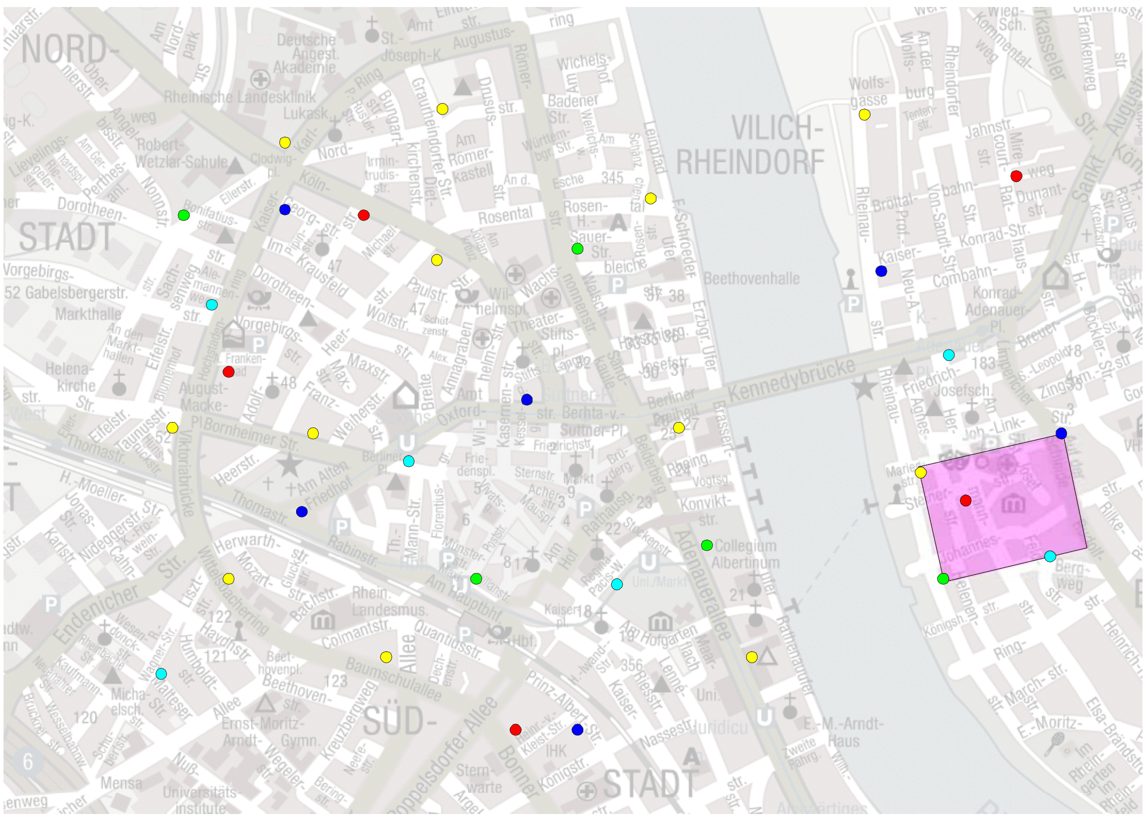


Abb. 1.1.1) Wohnungssuche: ●=Wohnung, ●=Supermarkt, ●=Post, ●=Bahnhof, ●=U-Bahn

Falls die Stadt nur rechtwinklige Straßenverläufe hat, muss man jetzt das flächenmäßig kleinste achsenparallele (straßenverlaufsparallele) Rechteck suchen, das alle Farben enthält.
Bei nicht rechtwinkligen Straßenverläufen böte sich die Suche nach kleinsten flächigen Rechtecken beliebiger Orientierung an (siehe Abbildung 1.1.1).

Ein weiteres, durchaus nicht unrealistisches Problem haben z.Zt. einige Bauherren in Shanghai [2]. Hier wurde bei der Errichtung eines 16stöckigen Hochhauses der Fahrstuhl vergessen. Zur Lösung dieses Problems könnten alle zukünftigen Mieter jeder Etage gefragt werden, wo sie den Aufzug hinhaben wollen. Dafür geben sie jeweils die gewünschten Lagen (möglichst mehrere pro Etage, da sonst $n = k$) des Aufzugschachtes an. Diese Lagen werden nun etagenabhängig mit unterschiedlichen Farben markiert ($k = 16$, siehe Abbildung 1.1.2), das Gebäude von oben betrachtet und das kleinste flächige Rechteck gesucht. In dieses baut man dann den Schacht und ist somit den Wünschen der Mieter der einzelnen Etagen nahe gekommen. Das Beispiel verdeutlicht auch, warum nur achsenparallele Rechtecke gesucht werden, da die übrigen Räume rechtwinklige Ecken haben sollten und der Fahrstuhlschacht nicht aus der Außenwand des Hauses hinausragen darf.

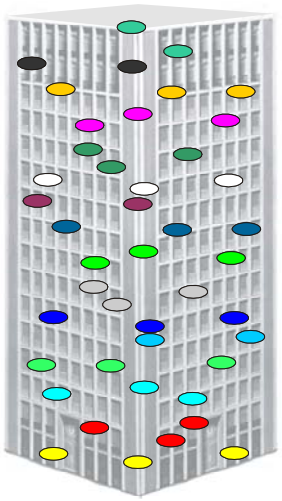


Abb. 1.1.2) Haus mit erwünschten Lagen des Fahrstuhlschachtes

1.2 Das formale Problem

Gegeben sind ***n*** Punkte in der Ebene, die jeweils mit einer von ***k*** Farben eingefärbt sind. Gesucht wird das kleinste achsenparallele Rechteck bzw. das kleinste Rechteck beliebiger Orientierung, das Punkte aller Farben im Inneren oder auf dem Rand des Rechtecks enthält. Das Maß für die Größe kann die Fläche, der Umfang, die Diagonale etc. sein.

1.3 Thema der Arbeit

In dieser Diplomarbeit wird gezeigt, dass die Suche nach kleinsten achsenparallelen farbumspannenden Rechtecken nicht – wie in [1] dargestellt – $O((n-k)\, n \log^2 k)$, sondern nur $O((n-k)(n + (n-k) \log k)$ Zeit benötigt. Hierbei gibt n die Anzahl der Punkte und k die Anzahl der Farben an.

Des Weiteren wird gezeigt, dass die Suche nach kleinsten farbumspannenden Rechtecken beliebiger Orientierung nur $O((n^2-k^2)(n + (n-k) \log k))$ Zeit benötigt.

Zuerst wird hierfür der ursprüngliche Algorithmus aus [1] erklärt, da die beiden neu vorgestellten Verfahren zu einem Großteil auf diesem aufbauen.

Dann werden in Kapitel 4 und 5 die neuen Verfahren vorgestellt und die dafür zusätzlich benötigten Schritte erläutert.

In Kapitel 6 werden die Klassen, Methoden und die Bedienung des von mir programmierten Java-Applets beschrieben, welches diese beiden Algorithmen verwendet.

2 Voraussetzungen und Definitionen

2.1 Voraussetzungen

In dieser Arbeit gelten folgende Prämissen:

- Alle Punkte haben paarweise unterschiedliche X- und Y-Koordinaten.
- Die Anzahl der Farben beträgt mindestens drei, da sich das Problem für $k = 2$ auf die Suche nach der kürzesten Distanz in der geometrischen Realisierung eines vollständigen bipartiten Graphen $K_{i,j}$, mit $i + j = n$, reduziert.
- Es gilt immer $k \leq n$, da es sonst kein Rechteck gibt, dass alle k Farben enthält.
- Die Punktmenge ist nach absteigend sortierten Y-Koordinaten in einem Array $\boldsymbol{p}_{[1,n]}$ gespeichert.
- Die Punkte sind bzgl. der X-Koordinate in aufsteigender Reihenfolge verkettet. Diese Verkettung wird in Abschnitt 4.6 für die Initialisierung der ***MaxElem***-Liste in linearer Zeit benötigt.
- Sowohl die X- als auch die Y-Koordinaten liegen innerhalb des offenen Intervalls]0, 1[. Da nur endliche Punktmengen betrachtet werden, ist das ohne Beschränkung der Allgemeinheit möglich.

Ein Punkt der Punktmenge ***P***, der während der Ausführung der Algorithmen als potenzielle untere Grenze der gesuchten Rechtecke betrachtet wird, heißt $\boldsymbol{p}_{\mathbf{u}}$.
Analog hierzu heißt der obere Begrenzungspunkt $\boldsymbol{p}_{\mathbf{o}}$, der Rechte $\boldsymbol{p}_{\mathbf{r}}$ und der Linke $\boldsymbol{p}_{\mathbf{l}}$.
Die Eigenschaften (Lage, Farbe und Index) der Punkte sind – objektorientiert – durch ***p.x***, ***p.y***, ***p.col*** und ***p.index*** dargestellt.
Punkte mit bestimmter Position im Array werden durch $\boldsymbol{p}_{[\mathbf{index}]}$ bezeichnet. In dieser Ausarbeitung ist das Array – anders als im Quellcode des Applets – nicht „zerobased“, sondern beginnt bei 1 und endet bei n.
Die Punktmenge $\mathbf{R_{ect}(x_{left}, y_{top}, x_{right}, y_{bottom})}$ stellt ein achsenparalleles Rechteck mit den entsprechenden Koordinaten dar.

2.2 Definitionen

Eine Punktmenge P ist **farbumspannend**, falls es für jede der k Farben $\boldsymbol{c}_{\mathbf{i}}$ ($1 \leq i \leq k$) einen Punkt $p \in P$ mit $p.col = c_i$ gibt.

Ein **nicht verkleinerbares achsenparalleles farbumspannendes Rechteck** (im Folgenden nur noch „**nicht verkleinerbares Rechteck“** genannt) wird wegen der paarweisen Unterschiedlichkeit der Koordinaten durch zwei, drei oder vier Randpunkte definiert. Die Menge aller Punkte des Rechtecks ist farbumspannend, die Randpunkte haben paarweise unterschiedliche Farben und keine Punkte im Inneren des Rechtecks haben eine Farbe, die auch ein Randpunkt besitzt.

Beweis für die Minimalität solch eines Rechtecks:
Betrachte hierfür ein Rechteck, das einen Randpunkt hat, dessen Farbe ein weiteres mal in diesem Rechteck vorkommt.
Sei o.B.d.A. der Randpunkt, der die untere Grenze des Rechtecks definiert, solch ein unerwünschter Punkt. Jetzt kann man diese untere Grenze über diesen Punkt hinaus weiter nach oben zum nächsten Punkt schieben und hat trotzdem noch alle Farben in dem jetzt kleineren Rechteck enthalten.
Somit ist das ursprüngliche Rechteck verkleinerbar gewesen. q.e.d.

Ein Punkt $\boldsymbol{p_d}$ (dominierter) wird von einem Punkt $\boldsymbol{p_D}$ (Dominierender) bzgl. eines Punktes $\boldsymbol{p_A}$ (Anker) **dominiert**, wenn nach der Verschiebung dieser drei Punkte um den Vektor $-(p_D.x, p_D.y)^t$ (p_D liegt jetzt im Ursprung) p_A und p_d in jeweils gegenüberliegenden geschlossenen Quadranten des Koordinatensystems liegen.
Der Punkt p_D ist **maximal**, wenn er von keinem anderen Punkt bzgl. eines festen Ankerpunktes p_A dominiert wird (siehe Abbildung 2.2.1).
Diese Dominanzdefinition wird für die Verwaltung der Elemente in der *MaxElem*-Struktur/-Liste mit dem Ankerpunkt $p_A = (0, 1)$ und zur Erstellung der Farbkonturen mit dem Ankerpunkt $p_A = p_u$ benötigt.

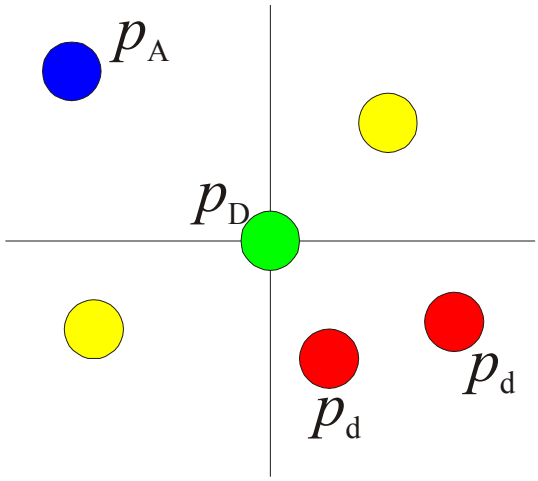


Abb. 2.2.1) ●=Ankerpunkt, ●=dominierender Punkt, ●=nicht dominierte Punkte, ●=dominierte Punkte.

2.3 Weitere Vorüberlegungen

Betrachte nun waagerechte Streifen der Punktmenge, deren oberer Rand durch $p_o.y$ und deren unterer Rand durch $p_u.y$ vorgegeben sind.
Damit in diesem Streifen ein nicht verkleinerbares Rechteck liegen kann, muss gelten:

$$|\{p_{[i]}.col \mid p_o.index \le i \le p_u.index\}| = k$$
(Alle Farben müssen in diesem Streifen vorhanden sein)

Für nicht verkleinerbare Rechtecke, die in solch einem Streifen liegen, gelten folgende Aussagen:

Da p_u und p_o auf dem Rand des Rechtecks liegen müssen, gilt für p_l und p_r:

Eigenschaft1

$$p_l.x \le \min\{p_o.x, p_u.x\}$$
und
$$p_r.x \ge \max\{p_o.x, p_u.x\}$$
(Oberer und unterer Punkt müssen – in X-Richtung – zwischen linkem und rechtem Rand liegen)

Für die Farben der Randpunkte und der inneren Punkte gilt:

Eigenschaft2

$$\{p_u.col, p_o.col, p_l.col, p_r.col\} \cap \{p.col \mid p \in P \wedge p.x \in]p_l.x, p_r.x[\wedge p.y \in]p_u.y, p_o.y[\} = \varnothing$$

(Die Farben der Randpunkte dürfen nicht im Inneren des Rechtecks vorkommen)

Mit den Definitionen

Definition1

$$L(c, p_u, p_o) := \max(\{0\} \cup \{p.x \mid p \in P \wedge p.col = c \wedge p_u.y < p.y < p_o.y \wedge p.x < p_u.x\})$$

und

$$R(c, p_u, p_o) := \min(\{1\} \cup \{p.x \mid p \in P \wedge p.col = c \wedge p_u.y < p.y < p_o.y \wedge p.x > p_u.x\})$$

(Bei gedachter Senkrechte S_u durch p_u liefert $L(c, p_u, p_o)$ die X-Koordinate des nächstgelegenen Punktes der Farbe c auf der linken Seite der Senkrechten und innerhalb des Streifens zwischen p_u und p_o. Hier sei nochmals darauf hingewiesen, dass die Koordinaten der Punkte nicht 0 oder 1 annehmen können und daher Resultate von 0 oder 1 anzeigen, dass die Farbe c nicht auf der entsprechenden Seite vorkommt (siehe Abschnitt 2.1).)

und *Eigenschaft 1* kann man für p_l und p_r auch noch folgende Eigenschaft bestimmen:

Eigenschaft3

$$p_l.x > \max\{ L(p_o.col, p_u, p_o), L(p_u.col, p_u, p_o) \}$$

und

$$p_r.x < \min\{ R(p_o.col, p_u, p_o), R(p_u.col, p_u, p_o) \}$$

(Die X-Koordinate des linken Randpunktes p_l muss rechts von jedem Punkt liegen, der links von S_u und innerhalb des Streifens liegt und die Farbe $p_o.col$ oder die Farbe $p_u.col$ hat (siehe Abbildung 2.3.1).)

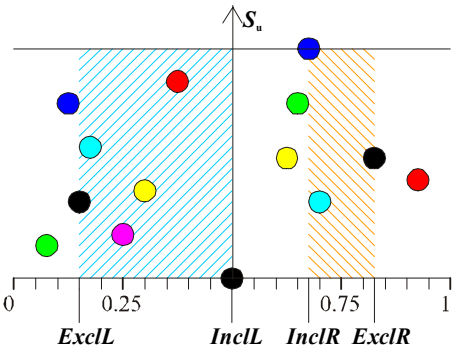


Abb. 2.3.1) Grün kann kein linker Randpunkt sein, da sonst blau und schwarz auf dem Rand und auch im Inneren des Rechtecks vorkämen. Rot kann kein rechter Randpunkt sein, da sonst schwarz auf dem Rand und auch im Inneren des Rechtecks vorkäme.

Aus *Eigenschaft1* und *Eigenschaft3* können nun die Definitionen

Definition2

$$InclL := \min\{p_o.x, p_u.x\}$$
$$InclR := \max\{p_o.x, p_u.x\}$$
$$ExclL := \max\{ L(p_o.col, p_u, p_o), L(p_u.col, p_u, p_o) \}$$
$$ExclR := \min\{ R(p_o.col, p_u, p_o), R(p_u.col, p_u, p_o) \}$$

erstellt und die Positionen von p_l und p_r durch

Definition3

$$ExclL < p_l.x \leq InclL$$

und

$$InclR \leq p_r.x < ExclR$$

eingeschränkt werden.

2.4 Die Struktur MaxElem

Man betrachte für jede Farbe c den Punkt

$$\boldsymbol{e(c) := (L(c, p_u, p_o), R(c, p_u, p_o))}$$

Im Folgenden wird dieser Punkt *Element* genannt, um eine Verwechselung mit den Punkten der Menge P zu vermeiden.
Sei $\boldsymbol{E}$ die Menge aller Elemente $e(c_i)$, mit $1 \leq i \leq k$. Diese Elemente liegen wegen *Definition1* in dem Bereich $[0, 1] \times [0, 1]$. Es ist nun ein besonderer Zusammenhang zwischen der Lage der Punkte und der Lage der Elemente feststellbar.

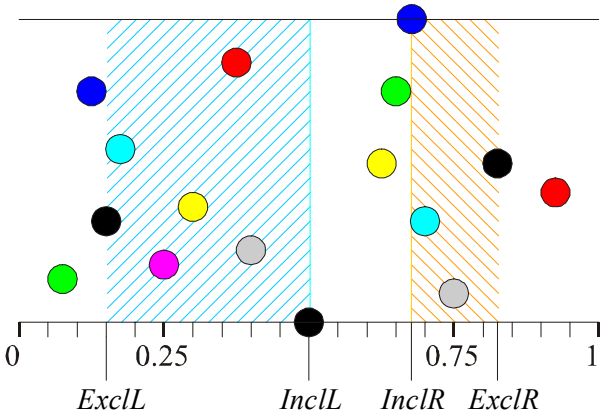


Abb. 2.4.1

Abbildung 2.4.1 zeigt eine Punktkonstellation, in der alle Farben innerhalb des Streifens zwischen p_u und p_o vorkommen. p_u ist der unterste schwarze Punkt und p_o ist der oberste blaue Punkt. Hier sind auch die Grenzen *ExclL*, *InclL*, *InclR*, *ExclR* eingezeichnet. Die dazugehörigen erlaubten Bereiche, in denen p_l und p_r liegen dürfen, sind schraffiert dargestellt. Weiterhin sei hier bemerkt, dass *magenta* nur auf der linken Seite von p_u vorkommt.

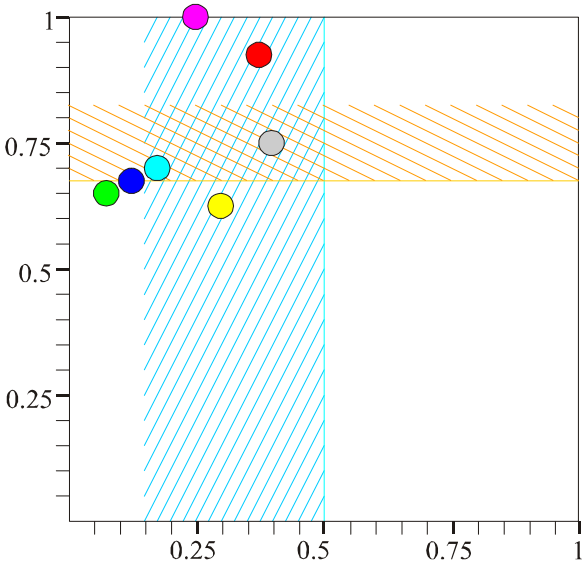


Abb. 2.4.2

In Abbildung 2.4.2 sieht man nun die zu den Punkten dazugehörigen Elemente *e(c)*, die in der *MaxElem*-Struktur verwaltet werden. Es fällt auf, dass es kein Element für schwarz gibt. Das ist auch nicht notwendig, da die *MaxElem*-Struktur nur Farben für potenzielle linke und rechte Randpunkte verwaltet. Weil aber schwarz schon als fester unterer Randpunkt vorgegeben ist, muss er hier nicht weiter betrachtet werden. Der Sonderfall, dass der untere Punkt auch linker oder rechter Randpunkt sein kann, wird später behandelt. Natürlich müssen trotzdem alle schwarzen Punkte innerhalb des Streifens zur Bildung der vier Grenzen (*ExclL*,...) herangezogen werden. Weiterhin sieht man hier, dass *magenta* ganz oben liegt, da in Abbildung 2.4.1 kein Punkt der Farbe magenta rechts von p_u liegt.

Um zu dem „besonderen Zusammenhang“ zu kommen, sind hier – wie in [1] – die folgenden drei Lemmata zu beachten.

Lemma 2.4.1) Wenn zwischen p_u und p_o alle Farben vertreten sind, dann beinhaltet das Rechteck $Rect(p_l.x, p_o.y, p_r.x, p_u.y)$ mit fester Farbe $p_l.col = p_r.col = f$ genau dann alle Farben, außer evtl. $p_o.col$, wenn *e(f)* von keinem anderen Element der Menge *E* bzgl. des Ankerpunktes (0, 1) dominiert wird; also maximal ist.

Beweis „⇐“:
Es gilt: *e(f)* ist maximal.
Sei *c* eine Farbe, die nicht in Rect vorkommt. Dann gilt wegen *Definition1*:
$L(c, p_u, p_o) < L(f, p_u, p_o)$ und $R(c, p_u, p_o) > R(f, p_u, p_o)$. Damit liegt aber *e(c)* links oberhalb von *e(f)* und dominiert demnach auch *e(f)*, was der Maximalität von *e(f)* widerspricht.
Beweis „⇒“:
Es gilt: Das Rechteck $Rect(p_l.x, p_o.y, p_r.x, p_u.y)$ mit fester Farbe $p_l.col = p_r.col = f$ beinhaltet alle Farben, außer evtl. $p_o.col$.
Deshalb gilt: Jede Farbe *c*, außer *f* selber, liegt entweder in dem vertikalen Streifen zwischen p_l und p_u oder in dem vertikalen Streifen zwischen p_u und p_r.
Daraus folgt: $L(c, p_u, p_o) > L(f, p_u, p_o) \vee R(c, p_u, p_o) < R(f, p_u, p_o)$
Also liegen alle Elemente *e(c)* rechts oder unterhalb von *e(f)*.
Damit ist *e(f)* maximal. q.e.d. (Lemma 2.4.1)

Lemma 2.4.2) Wenn das Rechteck $Rect(p_l.x, p_o.y, p_r.x, p_u.y)$ mit $p_l \neq p_u$, $p_l \neq p_o$, $p_r \neq p_u$ und $p_r \neq p_o$ nicht verkleinerbar ist, dann sind $e(p_r.col)$ und $e(p_l.col)$ zwei benachbarte Elemente in einer nach X-Koordinate sortierten Liste aller maximalen Elemente in *E*.

Beweis der Maximalität von $e(p_r.col)$ und $e(p_l.col)$ in *E*:
Falls $e(p_l.col)$ nicht maximal ist, dann gibt es einen Punkt p_b mit der Farbe *c*, so dass *e(c)* links oberhalb von $e(p_l.col)$ liegt.
Somit gilt: $L(c, p_u, p_o) < L(p_l.col, p_u, p_o)$ und $R(c, p_u, p_o) > R(p_l.col, p_u, p_o)$.
Daraus folgt: $p_b.x \notin [L(p_l.col, p_u, p_o), R(p_l.col, p_u, p_o)]$.
Weil aber $[p_l.x, p_r.x]$ Teilmenge von $[L(p_l.col, p_u, p_o), R(p_l.col, p_u, p_o)]$ ist, liegt p_b mit der Farbe *c* nicht in dem Rechteck. Somit enthält $Rect(p_l.x, p_o.y, p_r.x, p_u.y)$ nicht alle Farben und ist demnach nicht nicht verkleinerbar – d.h. nicht, dass es jetzt verkleinerbar ist, sondern dass es nicht mehr farbumspannend ist (siehe Definition in 2.2) – was ein Widerspruch zur Implikationsvoraussetzung ist. Der Beweis für die Maximalität von $e(p_r.col)$ ist analog.

Beweis, dass $e(p_r.col)$ und $e(p_l.col)$ in *E* aufeinanderfolgend sind:
Angenommen, es gibt zwei Punkte p_{bl} und p_{br} mit der Farbe *c* und $p_{bl}.x = L(c, p_u, p_o)$ und $p_{br}.x = R(c, p_u, p_o)$, so dass *e(c)* in *E* maximal ist und zwischen $e(p_r.col)$ und $e(p_l.col)$ liegt.
Dann gilt: $L(p_r.col, p_u, p_o) < L(c, p_u, p_o) < L(p_l.col, p_u, p_o)$ und $R(p_r.col, p_u, p_o) < R(c, p_u, p_o) < R(p_l.col, p_u, p_o)$. Demnach liegt p_{bl} links von p_l und p_{br} liegt rechts von p_r. Also ist die Farbe *c* nicht in dem Rechteck $Rect(p_l.x, p_o.y, p_r.x, p_u.y)$ enthalten, was ein Widerspruch ist. q.e.d. (Lemma 2.4.2)

Lemma 2.4.3) Seien in dem horizontalen Streifen zwischen p_u und p_o alle Farben enthalten. Wenn $e(p_r.col)$ und $e(p_l.col)$ aufeinanderfolgende maximale Elemente in *E* sind, dann ist das Rechteck $Rect(p_l.x, p_o.y, p_r.x, p_u.y)$ nicht verkleinerbar.

Beweis:
Es gilt: $L(p_r.col, p_u, p_o) < L(p_l.col, p_u, p_o)$. Wenn die linke Seite des Rechtecks R_{ect} weiter nach links bis zu $L(p_r.col, p_u, p_o)$ verschoben wird, dann sind nach *Lemma 2.4.1* immer noch alle Farben in diesem Rechteck enthalten.
Nun wird die linke Seite des Rechtecks so lange punktweise wieder nach rechts verschoben, bis ein nicht verkleinerbares Rechteck gefunden wurde. Sei p ein Punkt, der die linke Seite solch eines gesuchten Rechtecks definiert. Dann gilt nach *Lemma 2.4.2*, dass $e(p_r.col)$ und $e(p.col)$ aufeinanderfolgend und maximal in E sind. Weil aber nach Voraussetzung $e(p_l.col)$ das auf $e(p_r.col)$ folgende maximale Element in E ist, gilt $e(p_l.col) = e(p.col)$ und $p = p_l$. q.e.d. (Lemma 2.4.3)

Wenn man sich jetzt nochmal das Beispiel in Abbildung 2.4.1 anschaut und *Lemma 2.4.3* anwendet, ist wegen Abbildung 2.4.2 auf Anhieb feststellbar, dass es nicht verkleinerbare Rechtecke zwischen *türkis* und *blau* und zwischen *magenta* und *türkis* gibt. Dabei ist zu beachten, dass bei einem gültigen Elementpaar das linke Element den rechten Randpunkt p_r und das rechte Element den linken Randpunkt p_l definiert. Wegen *Definition3* wird das Rechteck zwischen *blau* und *grün* nicht betrachtet.

2.5 Der Sonderfall

Wie in Abschnitt 2.4 bereits erwähnt, wurde bisher noch nicht die Möglichkeit betrachtet, dass der untere Randpunkt p_u auch gleichzeitig linker (p_l) oder rechter Randpunkt (p_r) sein kann. Zur Berücksichtigung dieses Falles werden in E noch die Elemente $\boldsymbol{e_r}(p_u.col) = (0, p_u.x)$ und $\boldsymbol{e_l}(p_u.col) = (p_u.x, 1)$ eingefügt. Falls nun e_r maximal ist, dann hat e_r keinen maximalen Vorgänger und definiert daher auch keine Farbe eines linken Randpunktes. Analog definiert e_l keine Farbe eines rechten Randpunktes. Weiterhin sei gesagt, dass e_r keine anderen Elemente dominieren kann, da wegen $p_u.x < \min\{e.y \mid e \in E\} = \min\{R(c_i, p_u, p_o) \mid 1 \le i \le k\}$ kein Element unterhalb von e_r liegen kann. Analoges gilt für e_l.

3 Der ursprüngliche Algorithmus

Der Algorithmus sowie die meisten Überlegungen aus Kapitel 2 wurden 2001 von Manuel ABELLANAS, Ferran HURTADO, Christian ICKING, Rolf KLEIN, Elmar LANGETEPE, LIHONG Ma, Belén PALOP und Vera SACRISTÁN mit dem Paper **„***Smallest Color-Spanning Objects*" veröffentlicht.
In diesem Kapitel wird die Arbeitsweise dieses Verfahrens beschrieben und erläutert, wie es zu den Kosten in Höhe von $O((n-k)\, n \log^2 k)$ kommt.

3.1 Erstes geeignetes Paar

Im Folgenden wird davon ausgegangen, dass die maximalen Elemente in einem balancierten Binärbaum und durch eine verkettete und nach X-Koordinate (und demnach auch nach Y-Koordinate) sortierten Liste gespeichert sind, wobei ***e(c).Next*** das rechts von *e(c)* gelegene und analog ***e(c).Previous*** das links von e*(c)* gelegene maximale Element bezeichnet.

Der Algorithmus hat zu Beginn von *Schleife2* (siehe Abschnitt 3.2) jeweils das erste geeignete Paar $e_R(c_R)$, $e_L(c_L)$ von aufeinanderfolgenden maximalen Elementen in der Struktur *MaxElem* zu suchen (d.h.: $e_R(c_R).Next = e_L(c_L)$ und $e_R(c_R) = e_L(c_L).Previous$). Hierbei muss gelten:

$$\mathbf{In}clR \le e_R(c_R).y < ExclR$$
und
$$ExclL < e_L(c_L).x \le InclL.$$

(Im Gegensatz zu Abschnitt 2.4 werden die gesuchten Elemente nicht mehr $e(p_r.col)$ und $e(p_l.col)$ genannt, da die entsprechenden Randpunkte noch nicht bekannt sind und erst durch die Elemente $e_R(c_R)$ und $e_L(c_L)$ auf diese Punkte geschlossen werden kann.)

Für diese Suche wird folgende Funktion verwendet:

Die Funktion ***Suche das erste geeignete Elementpaar $e_R(c_R)$, $e_L(c_L)$***:

- Suche in *MaxElem* das – von links gesehen – erste maximale Element $e_R(c_R)$, mit $InclR \le e_R(c_R).y < ExclR$. (*Suche1*)
  Verlasse die Funktion ohne Rückgabewert, falls solch ein Element nicht existiert.
- Sei $e_L(c_L) = e_R(c_R).Next$.
  Verlasse die Funktion ohne Rückgabewert, falls solch ein Element nicht existiert.
- Wenn nicht $ExclL < e_L(c_L).x \le InclL$ gilt, dann:
  - Suche in *MaxElem* das – von links gesehen – erste maximale Element $e_L(c_L)$, mit $ExclL < e_L(c_L).x \le InclL$. (*Suche2*)
    Verlasse die Funktion ohne Rückgabewert, falls solch ein Element nicht existiert.
  - Sei $e_R(c_R) = e_L(c_L).Previous$.
    Verlasse die Funktion ohne Rückgabewert, falls solch ein Element nicht existiert.
  - Wenn nicht $InclR \le e_R(c_R).y < ExclR$ gilt, dann verlasse die Funktion ohne Rückgabewert.
- Verlasse die Funktion mit dem Rückgabewert $e_R(c_R)$, $e_L(c_L)$.

Ende der Funktion.

Es bleib noch zu zeigen, dass nur diese zwei Paare gesucht werden müssen und man bei Nichtfinden davon ausgehen kann, dass es kein geeignetes Paar gibt:

Beweis:
Angenommen, es gäbe ein geeignetes Paar $e_R(c_R)$, $e_L(c_L)$, obwohl die Suche keines gefunden hat. Dann laufe von $e_R(c_R)$ aus so lange an den maximalen Elementen entlang nach links und betrachte jeweils zwei aufeinanderfolgende maximale Elemente $e_R(c_R)$ und $e_L(c_L)$, bis einer der folgenden Fälle eintritt:

- Das z.Zt. betrachtete Paar $e_R(c_R)$, $e_L(c_L)$ ist nicht geeignet:
  Dann gilt entweder $\neg(InclR \le e_R(c_R).y < ExclR)$ oder $\neg(ExclL < e_L(c_L).x \le InclL)$
  Wenn $\neg(InclR \le e_R(c_R).y < ExclR)$ gilt, dann wäre aber in *Suche1* das Element $e_R(c_R).Next$ gefunden worden, was zu einem Widerspruch zu obiger Annahme führt.
  Analoges gilt für $e_L(c_L)$.
- Die Liste der maximalen Elemente endet:
  Hier hätten auf jeden Fall die – von links gesehen – ersten beiden maximalen Elemente $e_R(c_R)$ und $e_L(c_L)$ gefunden werden müssen, da für diese natürlich gilt:
  $InclR \le e_R(c_R).y < ExclR$ und $ExclL < e_L(c_L).x \le InclL$. q.e.d.

(Diese Funktion wird auch in Kapitel 4 „Der verbesserte Algorithmus“ verwendet. Der einzige Unterschied wird dort sein, dass dann auf eine vereinfachte *MaxElem*-Struktur zugegriffen wird. Das ändert aber nichts an der benötigten Laufzeit O(log *k*) und auch nichts an dem Schema der Suche, so dass diese Prozedur in Kapitel 4 nicht noch einmal erläutert wird.)

3.2 Schema des Algorithmus

Setze die Größe des *bisher kleinsten gefundenen Rechtecks* auf unendlich.

Der Algorithmus durchläuft nun alle potenziellen unteren Begrenzungspunkte p_u der zu suchenden Rechtecke. Hierbei fängt er bei dem *k*-obersten Punkt $p_{[k]}$ an und läuft bis zum untersten Punkt $p_{[n]}$ durch; also eine Schleife von *k* bis *n* (*Schleife1*). Punkte, die oberhalb von $p_{[k]}$ liegen, können keine untere Grenze definieren, da oberhalb dieser Punkte noch nicht alle Farben vorhanden sein können.

> Innerhalb von *Schleife1* wird zuerst die Struktur *MaxElem* initialisiert. Hierfür werden die *k+1* Elemente der Menge
> $E = \{e_r(p_u.col), e_l(p_u.col)\} \cup \{e(c_i) = (0, 1) \mid 1 \le i \le k \wedge c_i \ne p_u.col\}$ in die Struktur eingefügt.
>
> Laufe von $p_{[pu.index - 1]}$ bis zum obersten Punkt $p_{[1]}$ und betrachte die so besuchten Punkte als potenzielle obere Grenze p_o (*Schleife2*).
>
> Innerhalb von *Schleife2* werden folgende Überlegungen gemacht:

i. Wenn $p_u.col \neq p_o.col$ ist, dann aktualisiere das Element $e(p_o.col)$ ggf. wie folgt:
   i.i Wenn $p_o.x < p_u.x$ und $p_o.x > e(p_o.col).x$, dann sei $e(p_o.col).x = p_o.x$.
   i.ii Wenn $p_o.x > p_u.x$ und $p_o.x < e(p_o.col).y$, dann sei $e(p_o.col).y = p_o.x$.
   Man beachte, dass hier vorher nicht maximale Elemente auf einmal maximal werden können und man daher auch die Lage der nicht maximalen Elemente verwalten muss (siehe hierzu Abschnitt 4.1).

ii. Wenn es in der *MaxElem*-Struktur ein Element $e(c_i)$ mit den Koordinaten (0, 1) gibt, also die Farbe c_i noch nicht in dem waagerechten Streifen zwischen p_u und p_o vorkommt, oder wenn $p_u.col = p_o.col$ ist, dann gehe zum Ende von *Schleife2*.

iii. *Suche das erste geeignete Elementpaar* $e_R(c_R)$, $e_L(c_L)$. (Siehe Abschnitt 3.1)
   Falls solch ein Paar nicht existiert, gehe zum Ende von *Schleife2*.

iv. Falls das Rechteck $R_{ect}(e_L(c_L).x, p_o.y, e_R(c_R).y, p_u.y)$ eine kleinere Fläche (Umfang, Diagonale...) als das *bisher kleinste gefundene Rechteck* hat, dann merke dieses als *bisher kleinstes gefundenes Rechteck*.

v. Setze $e_R(c_R) := e_L(c_L)$.
   Gehe zum Ende von *Schleife2*, falls nicht $InclR \leq e_R(c_R).y < ExclR$ gilt.

vi. Sei $e_L(c_L) = e_R(c_R).Next$.
   Wenn solch ein Element existiert und $ExclL < e_L(c_L).x \leq InclL$ gilt,
   dann gehe zu Schritt iv.

Ende von *Schleife2*.

Ende von *Schleife1*.

Gib das *bisher kleinste gefundene Rechteck* aus.

Ende der Suche.

3.3 Die Kosten

Für die Kosten ist vor allem interessant, wie die k+1 Elemente in der Struktur *MaxElem* verwaltet werden. Hierfür wird ein Algorithmus von OVERMARS und LEEUWEN [3] verwendet. Dieser speichert die Elemente nach Y-Koordinate sortiert in einem balancierten Binärbaum und speichert gleichzeitig eine verkettete Liste der maximalen Elemente bzw. die so genannte „*m-contour*“, also die Liniensegmente, die man erhält, wenn – ausgehend von den maximalen Elementen – Linien nach unten und nach rechts gezeichnet und diese dann an den Schnittpunkten abgeschnitten weden. Hier wurde der Begriff der Maximalität durch eine horizontale Spiegelung leicht abgewandelt (siehe Abbildung 3.3.1). Der Ankerpunkt der dazugehörigen Dominanzrelation liegt

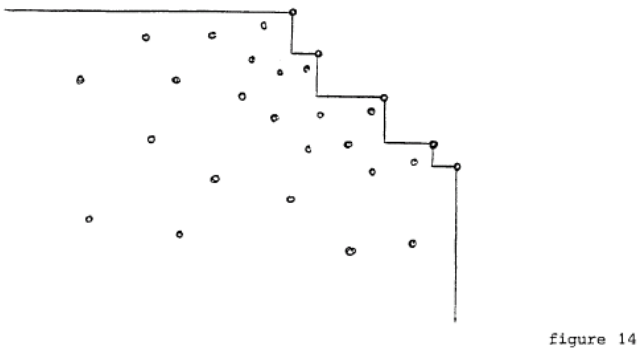


Abb. 3.3.1) m-contour aus [3]

bei $(+\infty, +\infty)$ In Anlehnung an den Begriff „*m-contour*“ wird ab Kapitel 4 auch der Begriff *Farbkontur* verwendet.
Für die Verwaltung des Binärbaumes und der verketteten Liste werden pro Einfügung und Löschung jeweils $O(\log^2 k)$ Schritte benötigt. Für die Suche nach maximalen Elementen in der Liste benötigt man $O(\log k)$ Zeit. Benachbarte maximale Elemente werden dann wegen der Verkettung in konstanter Zeit gefunden.

Insgesamt ergeben sich folgende Kosten:

Schleife1 (mit Laufindex i) _____ $O(n-k)$
- Initialisierung von *MaxElem* _____ $O(k \log^2 k)$
- *Schleife2 (mit Laufindex j)* _____ $O(n)$
  - i) Aktualisierung (ggf. durch Löschung und Einfügung) _____ $O(\log^2 k)$
  - ii) Abfrage: Alle Farben vorhanden? _____ $O(1)$
  - iii) Suche erstes geeignetes Elementpaar _____ $O(\log k)$
  - iv – vi) Prüfung der gefundenen Rechtecke _____ $O(\boldsymbol{a_{i,j}})$

Auffällig sind hier die Kosten $O(a_{i,j})$ für die Schritte iv bis vi. Die Summe aller $a_{i,j}$ ist die Anzahl aller nicht verkleinerbaren Rechtecke bei n Punkten und k Farben.
Die Frage ist jetzt, wie groß diese Summe ist.

Lemma 3.3.1) Es gibt $\Theta((n-k)^2)$ nicht verkleinerbare achsenparallele Rechtecke.

Beweis:

Definition: Gegeben sei ein nicht verkleinerbares Rechteck $\text{Rect}(p_l.x, p_o.y, p_r.x, p_u.y)$, $k \geq 3$.
Das zu Rect *erweiterte Rechteck* $\mathbf{R_{ex}}(p_l.x, \mathit{ExtBorder}, p_r.x, p_u.y)$ wird wie folgt erstellt:

Die obere Kante von Rect wird so lange nach oben verschoben, bis einer der folgenden Fälle eintritt (siehe Abbildung 3.3.2):

- Die Kante erreicht einen Punkt der Farbe $p_u.col$. (Typ 1)
- Die Kante erreicht einen Punkt der Farbe $p_l.col$ oder $p_r.col$. (Typ 2)
- Es wird kein Punkt der Farbe $p_l.col$, $p_r.col$ oder $p_u.col$ erreicht ($\mathit{ExtBorder} = \infty$). (Typ 3)

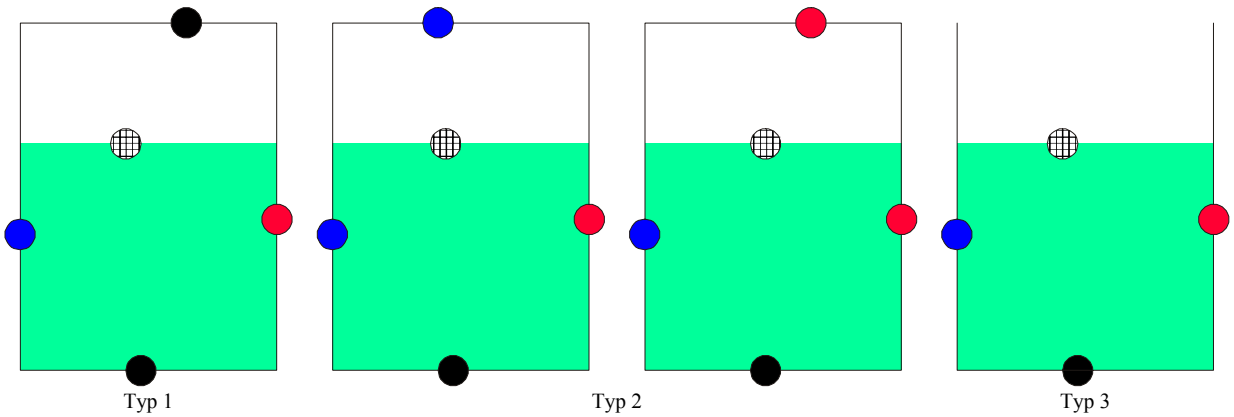


Abb. 3.3.2) Erweiterte Rechtecke. Das nicht verkleinerbare Rechteck ist grün dargestellt.

Das zu einem nicht verkleinerbaren Rechteck R_{ect} zugehörige erweiterte Rechteck R_{ex} ist somit eindeutig bestimmt. Andererseits läßt sich aus einem erweiterten Rechteck R_{ex} – durch Absenkung der oberen Kante – das nicht verkleinerbare Rechteck eindeutig bestimmen.
Es existiert also zu jedem nicht verkleinerbaren Rechteck genau ein erweitertes Rechteck. Die Abbildung

$$f_P: \{\text{nicht verkleinerbare Rechtecke}\} \leftrightarrow \{\text{erweiterte Rechtecke}\}$$

– bzgl. der Punktmenge P – ist daher bijektiv und für den Beweis genügt es zu zeigen, dass es nur $\Theta((n-k)^2)$ erweiterte Rechtecke gibt.

Zuerst wird die obere Schranke $O((n-k)^2)$ bewiesen. Hierzu genügt es zu zeigen, dass es jeweils nur $O((n-k)^2)$ Rechtecke der drei Typen gibt.

Typ 1: Fixiere einen beliebigen Punkt $p_u \in p_{[k, n]}$, der den unteren Randpunkt der erweiterten Rechtecke repräsentiert. Nun sei $\{q_i\}$ die rechte Farbkontur der Farbe $p_u.col$ (siehe Abschnitt 4.2). Es gilt:

- $q_{i+1} = q_i.NextContourPointOfSameColorBelow$ (siehe Abschnitt 4.2)
- $q_i.x < q_{i+1}.x$ und $q_i.y > q_{i+1}.y$
- $p_o \in \{q_i\}$ ist oberer Randpunkt eines erweiterten Rechtecks

Nun sei $r_{i,j}$ ein rechter Randpunkt eines erweiterten Rechtecks, das q_i als oberen Randpunkt hat (siehe Abbildung 3.3.3). Dann gilt:

- $r_{i,j}.x < q_{i+1}.x$, da sonst q_{i+1} im Inneren des erweiterten Rechtecks liegen würde, was aber nach obiger Konstruktion nicht möglich ist.
- $r_{i,j}.x > q_i.x$, da der linke Randpunkt immer links vom oberen Randpunkt liegt.
- Alle $r_{i,j}$ liegen unterhalb der Farbkontur $\{q_i\}$.

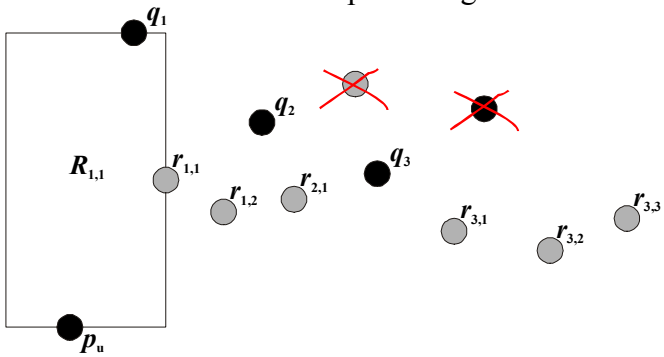


Abb. 3.3.3) Lage von möglichen oberen und rechten Randpunkten eines erweiterten Rechtecks vom Typ 1 bei fixiertem unteren Randpunkt p_u.

Sei $R_{i,j}(p_l.x, q_i.y, r_{i,j}.x, p_u.y)$ das erweiterte Rechteck mit festem unteren, rechten und oberen Randpunkt.
Der linke Randpunkt ist daher eindeutig bestimmt.
Beweis: Gäbe es hier zwei unterschiedliche Randpunkte $l_{i,j,1}$ und $l_{i,j,2}$, und wäre o.B.d.A. $l_{i,j,1}.x < l_{i,j,2}.x$, dann würde entweder das Rechteck mit $l_{i,j,2}$ als linkem Randpunkt nicht alle Farben enthalten, oder die Farbe $l_{i,j,1}.col$ würde auch im Inneren des Rechtecks vorkommen, das $l_{i,j,1}$ als linken Randpunkt hat. q.e.d.

Insbesondere gilt: Bei fixiertem unteren Randpunkt bildet jeder Punkt $r_{i,j}$ für höchstens ein erweitertes Rechteck einen rechten Randpunkt. Da oberhalb von p_u höchstens $n - k + 1$ Punkte als rechter Randpunkt in Frage kommen (links von $r_{i,j}$ müssen alle k Farben vorhanden sein), existieren für jeden unteren Randpunkt p_u auch nur $O(n-k)$ viele erweiterte Rechtecke vom Typ 1. Weil nicht $p_u \in p_{[1, k-1]}$ gelten kann, da sonst nicht alle Farben oberhalb von p_u liegen, existieren insgesamt $O((n-k)^2)$ viele erweiterte Rechtecke vom Typ 1.

Typ 2: Analog zu Rechtecken vom Typ 1 wird hier der linke (bzw. rechte) Randpunkt p_l fixiert. Dann werden die Rechtecke $R_{i,j}$ vom Typ 2 betrachtet, die einen Punkt q_i, mit $q_i.col = p_l.col$, als oberen Randpunkt haben. Auch hier sind die rechten Randpunkte $r_{i,j}$ paarweise verschieden und der untere Randpunkt ist bei festgelegtem linken, oberen und rechten Randpunkt eindeutig bestimmt.

Typ 3: Fixiere den unteren Randpunkt p_u. Dann gibt es nur $n - k + 1$ mögliche linke Randpunkte p_l. Zu jedem festgelegten unteren und linken Randpunkt ist der rechte nun wieder eindeutig bestimmt.

Weitere Typen (siehe Abbildung 3.3.4): Sowohl bei Typ 4 als auch bei Typ 5 ist allein durch die Fixierung des unteren Randpunktes gleichzeitig auch der linke Randpunkt bestimmt. Daher ist für gegebenes q_i auch sofort der rechte Randpunkt eindeutig bestimmt und es existieren pro p_u höchstens $|\{q_i\}|$ erweiterte Rechtecke, wobei hier die q_i ausgeschlossen werden, die nicht alle k Farben links von sich liegen haben. q.e.d. (obere Schranke)

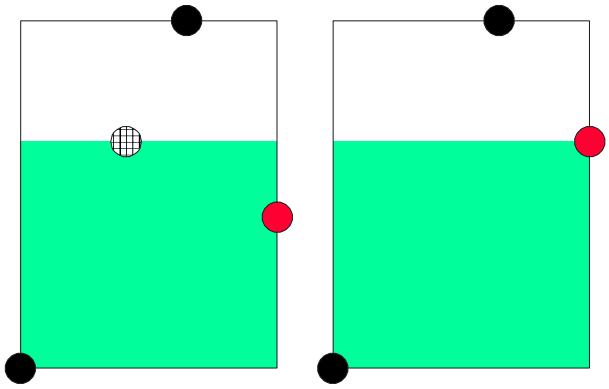


Abb. 3.3.4) Erweiterte Rechtecke mit $p_u = p_l$

Beweis der unteren Schranke $\Omega((n\text{-}k)^2)$ durch Angabe eines Beispiels (Abbildung 3.3.5), das diese Grenze tatsächlich erreicht. q.e.d. (Lemma 3.3.1)

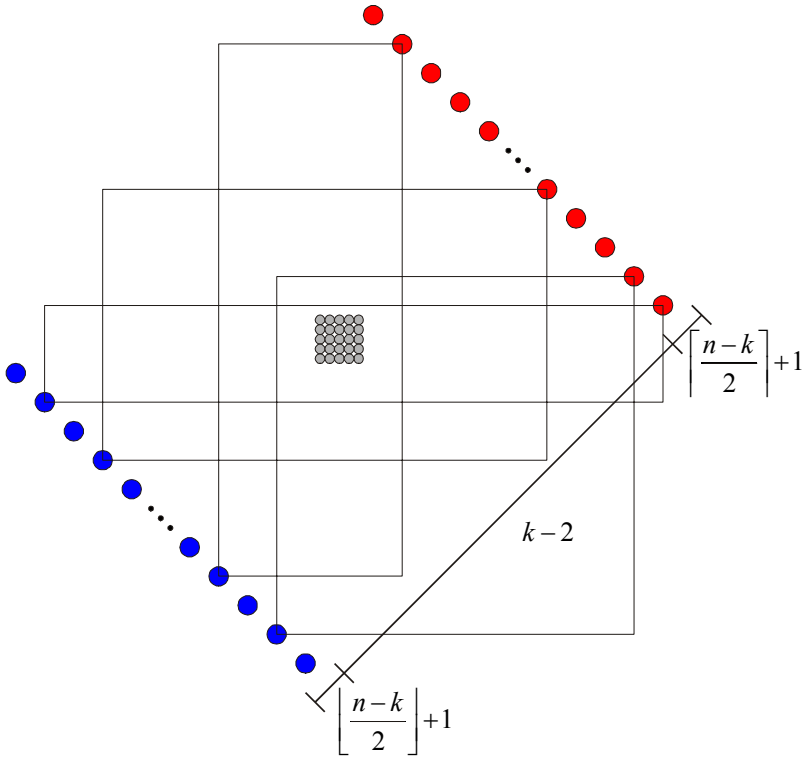


Abb. 3.3.5) Enthält $\Omega((n\text{-}k)^2)$ nicht verkleinerbare achsenparallele Rechtecke

Wegen *Lemma 3.3.1* wird für die Prüfung aller nicht verkleinerbaren Rechtecke insgesamt nicht mehr als $\Theta((n-k)^2)$ Zeit benötigt, was gegenüber den restlichen Kosten irrelevant ist, da pro Durchlauf von *Schleife2* im Durchschnitt nur konstante Kosten für diese Überprüfung benötigt werden.

Somit sind die Kosten für Schritt i) in *Schleife2* die größten und man erhält Gesamtkosten in Höhe von $\mathbf{O((n-k)\ n\ \log^2 k)}$.

4 Der verbesserte Algorithmus

4.1 Die Idee

Der Nachteil des ursprünglichen Algorithmus ist, dass man bei jedem Durchlauf von *Schleife2* mit p_o immer weiter nach oben wandert, und somit die Anzahl der Punkte innerhalb des horizontalen Streifens zwischen p_u und p_o immer größer wird. Das hat leider zur Folge, dass die Werte $L(c, p_u, p_o)$ immer größer und die Werte $R(c, p_u, p_o)$ immer kleiner werden. Dadurch wandern die Elemente in $E = \{e(c_i) := (L(c_i, p_u, p_o), R(c_i, p_u, p_o)) \mid 1 \leq i \leq k\}$ immer weiter nach rechts unten.
Das hat den entscheidenden Nachteil, das Elemente, die nicht maximal sind, auf einmal maximal werden können, obwohl sie nicht verschoben wurden. Betrachte hierfür das Beispiel in folgenden Abbildungen:
In Abbildung 4.1.1 beachte man, dass *magenta* nicht auf der rechten Seite von p_u vorkommt und demnach in *E e(magenta)* ganz oben liegt und unter anderem *rot* als auch *schwarz* dominiert. In Abbildung 4.1.2 gibt es nun eine neue obere Grenze, *magenta* kommt auf der rechten Seite von p_u vor und der Wert
$R(magenta, p_u, p_o)$ wird kleiner. Dadurch wird *e(magenta)* nach unten verschoben und die Elemente *e(rot)* und *e(schwarz)* werden maximal, obwohl sie nicht verschoben wurden. Das heißt, man muss für jeden neuen oberen Randpunkt p_o, für den gilt:

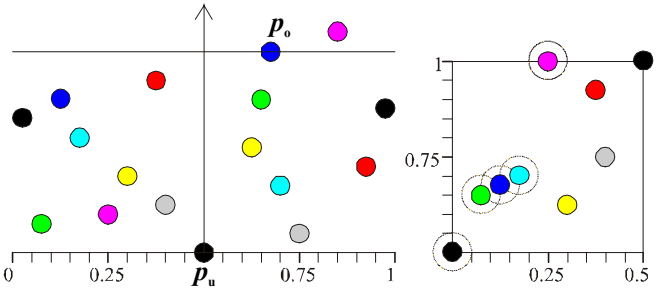


Abb. 4.1.1) Punktmenge *P* und dazugehörige Elementmenge *E*. Maximale Elemente in *E* sind durch einen Extrakreis gekennzeichnet.

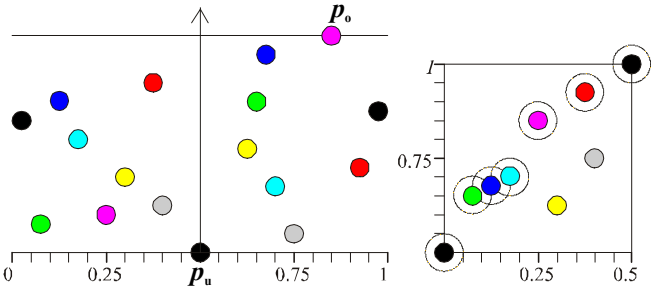


Abb. 4.1.2) *e(magenta)* wird nach unten verschoben. *e(rot)* und *e(schwarz)* werden maximal.

$$(p_o.x < p_u.x \wedge p_o.x > e(p_o.col).x) \vee (p_o.x > p_u.x \wedge p_o.x < e(p_o.col).y)$$

nicht nur das Element $e(p_o.col)$ verschieben, sondern auch jedes mal überprüfen, ob vorher dominierte Elemente maximal geworden sind.

Meine Idee ist nun, die Elemente in *E* nicht nach unten rechts wandern zu lassen, sondern nach oben links. Dadurch können dominierte Elemente erst dann wieder maximal werden, wenn sie selber verschoben werden. Dazu muss der Algorithmus in *Schleife2* nicht von $p_{[pu.index - 1]}$ bis $p_{[1]}$ laufen, sondern von $p_{[1]}$ bis $p_{[pu.index - k+1]}$. Damit aber für jede obere Schranke p_o die korrekten Werte $L(c_i, p_u, p_o)$ und $R(c_i, p_u, p_o)$ aller Farben c_i bekannt sind, bedarf es noch der folgenden Vorarbeit.

4.2 Erstellung von Farbkonturen

Definition eines *linken Farbkonturpunktes* bei gegebener Punktmenge P und fixiertem unteren Randpunkt p_u:
Sei $\boldsymbol{P_L} := \{p \in P \mid p.x < p_u.x \text{ und } p.y > p_u.y\}$.
Weiterhin sei $\boldsymbol{P_L(c)} := \{p \in P_L \mid p.col = c\}$.
Dann ist $p \in P_L$ genau dann linker Farbkonturpunkt,
wenn gilt: $\forall\ q \in P_L(p.col) : q.x < p.x \vee q.y > p.y$. (Siehe Abbildung 4.2.1)

Die Definition eines *rechten Farbkonturpunktes* ist analog.

Eine *rechte Farbkontur* der Farbe c ist nun eine nach Y-Koordinate absteigend sortierte verkettete Liste von rechten Farbkonturpunkten der Farbe c. Linke Farbkonturen sind analog definiert.

Es sei noch erwähnt, dass die Punkte p neben den in Abschnitt 2.1 genannten Eigenschaften (Lage, Farbe und Index) jetzt auch noch die Folgenden haben:

p.NextContourPointBelow: Zeigt – unabhängig von Farbe oder Seite – auf den nächsten Farbkonturpunkt unterhalb von p.

p.NextContourPointOfSameColorBelow: Zeigt auf den nächsten Farbkonturpunkt, der unterhalb von p und bzgl. p_u auf der gleichen Seite wie p liegt und die gleiche Farbe wie p hat.

Die Punkte p, für die entweder $p.x = L(p.col, p_u, p_o)$ oder $p.x = R(p.col, p_u, p_o)$ gilt, werden in dem zweidimensionalen Array ***CurContourPoints*[1, 2][1, *k*]** gespeichert, wobei in *CurContourPoints*[1] die Punkte auf der linken Seite von p_u enthalten sind, also die Punkte, für die $p.x = L(p.col, p_u, p_o)$ gilt, und in *CurContourPoints*[2] die Punkte der rechten Seite. *CurContourPoints* zeigt somit auch auf die Anfänge der verketteten Listen der Farbkonturen. Die k Farben sind von 1 bis k durchnummeriert und haben die entsprechende Nummer auch als Eigenschaft ***index***. Somit wird der Punkt p, für den $p.x = R(p.col, p_u, p_o)$ gilt, also oberster rechter Farbkonturpunkt der Farbe *p.col* ist, in *CurContourPoints*[2][*p.col.index*] gespeichert. Hier sei nochmals darauf hingewiesen, dass alle Punkte paarweise unterschiedliche X- und Y-Koordinaten haben, so dass die aktuellen Konturpunkte immer eindeutig bestimmt sind. Das Array *CurContourPoints* wird ab jetzt ***CCP*** genannt. In der Variable ***CurUpperBound*** wird der zuletzt gefundenen Konturpunkt gespeichert und mit ***null*** ist der leere Zeiger und nicht der Zahlenwert 0 gemeint.

Die Prozedur ***Erstellung der Farbkonturen***:

Initialisiere *CPP*:
CCP[1][1, *k*].*x* = 0
CCP[2][1, *k*].*x* = 1
CCP[1, 2][1, *k*].*NextContourPointBelow* = *null*
CCP[1, 2][1, *k*].*NextContourPointOfSameColorBelow* = *null*
(gemeint ist hier das Setzen der Eigenschaften für jeden Punkt der angegebenen Arrays, da das Array selber nicht Eigenschaften wie .x haben kann)

CurUpperBound = *null*

Laufe in einer Schleife von $p_{[pu.index-1]}$ bis $p_{[1]}$ mit Laufindex i.
  Wenn $p_{[i]}.x < p_u.x$: // $p_{[i]}$ liegt links von p_u
    Wenn $CCP[1][p_{[i]}.col.index].x < p_{[i]}.x$:
      $p_{[i]}.NextContourPointBelow = CurUpperBound$
      $p_{[i]}.NextContourPointOfSameColorBelow = CCP[1][p_{[i]}.col.index]$
      $CCP[1][p_{[i]}.col.index] = p_{[i]}$
      $CurUpperBound = p_{[i]}$
  Sonst: // $p_{[i]}$ liegt rechts von p_u
    Wenn $CCP[2][p_{[i]}.col.index].x > p_{[i]}.x$:
      $p_{[i]}.NextContourPointBelow = CurUpperBound$
      $p_{[i]}.NextContourPointOfSameColorBelow = CCP[2][p_{[i]}.col.index]$
      $CCP[2][p_{[i]}.col.index] = p_{[i]}$
      $CurUpperBound = p_{[i]}$
Ende der Prozedur

Neben dem Vorteil, dass jetzt von $p_{[1]}$ bis $p_{[pu.index-k+1]}$ gewandert werden kann und für jede obere Schranke p_o die korrekten Werte $L(c_i, p_u, p_o)$ und $R(c_i, p_u, p_o)$ aller Farben c_i bekannt sind, existiert noch ein weiterer Vorteil:

Betrachte hierfür Abbildung 4.2.1 und nehme einen Punkt, der nicht auf einer Farbkontur (🐝) und links von p_u liegt. Solch ein Punkt kann nicht linker Randpunkt p_l eines nicht verkleinerbaren Rechtecks sein, da es immer einen Punkt der gleichen Farbe gibt, der im Inneren solch eines Rechtecks liegen würde. Aus dem gleichen Grund können solche Punkte auch kein oberer Randpunkt p_o sein. Für die rechte Seite gilt eine analoge Argumentation. Daher verringert sich die Anzahl der zu überprüfenden potenziellen Randpunkte unter Umständen erheblich, was aber leider keine Auswirkung auf die Gesamtkosten des Algorithmus in der O-Notation hat.

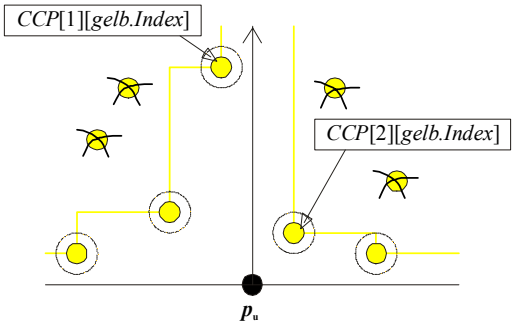


Abb. 4.2.1) Farbkonturen für *gelb*.

Wie man in Abbildung 4.2.1 sieht, kommt die Farbe *schwarz* nur als unterer Randpunkt p_u vor. Durch Erweiterung des obigen Beispiels durch Hinzunahme von schwarzen Punkten oberhalb von p_u kann man die Anzahl der gelben Konturpunkte noch weiter reduzieren: Man betrachte sich die jetzt gestrichenen Punkte (🐝) in Abbildung 4.2.2. Diese wären eigentlich Konturpunkte bzgl. der eigenen Farbe. Weil diese aber von schwarzen Punkten – bzgl. p_u – dominiert werden, können sie ebenfalls keine Randpunkte sein, da sonst im Inneren solch eines Rechtecks *schwarz* nochmal vorkommen würde.

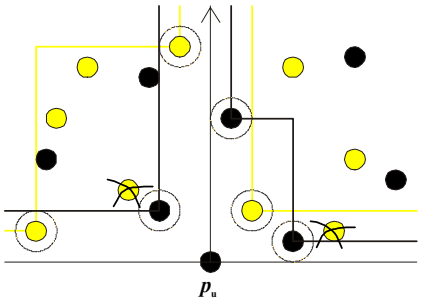


Abb. 4.2.2) Farbkonturen für *gelb* unter Berücksichtigung, dass von *schwarz* dominierte Punkte – bzgl. p_u – auch keine Randpunkte sein können.

Leider hat die Streichung dieser Punkte einen bedenklichen Nachteil. Man betrachte hierfür den Verlauf des oberen Randpunktes p_o und die dementsprechenden Werte $L(c_i, p_u, p_o)$ und

R(c_i, p_u, p_o) (siehe Abbildung 4.2.3). Wenn der oberste gelbe Farbkonturpunkt p_A auch oberer Randpunkt p_o ist, dann müsste eigentlich p_B für die Bildung von L(*gelb*, p_u, p_o) herangezogen werden. Es gilt aber:
L(*gelb*, p_u, p_o) = p_C.*x*, weil *CPP*[1][*gelb.index*] jetzt auf p_C zeigt. Das ist natürlich ein Widerspruch zur ursprünglichen Definition von L(*c*, p_u, p_o). Man frage sich jetzt aber, wann der Wert L(*gelb*, p_u, p_o) wieder relevant wird. Das geschieht erst, wenn die obere Schranke unterhalb von p_s liegt, da der Wert *ExclL* es bis dahin sowieso verbietet, sich Punkte links von p_s anzuschauen. Weil die obere Schranke dann aber schon vorher p_B überschritten haben muss, ist der Wert L(*gelb*, p_u, p_o) = p_C.*x* wieder korrekt.

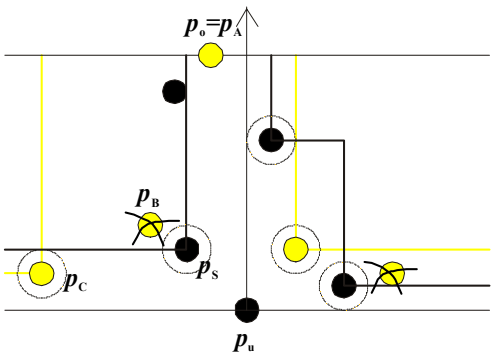


Abb. 4.2.3) *ExclL* wird's richten.

Somit kann man die *Erstellung der Farbkonturen* noch um eine Bedingung erweitern und erhält die Schleife:

Laufe in einer Schleife von $p_{[pu.index - 1]}$ bis $p_{[1]}$ mit Laufindex *i*.
Wenn $p_{[i]}.x < p_u.x$: // $p_{[i]}$ liegt links von p_u
Wenn *CCP*[1][$p_{[i]}$.*col.index*].*x* < $p_{[i]}.x$ und *CCP*[1][p_u.*col.index*].x < $p_{[i]}.x$:
$p_{[i]}$.*NextContourPointBelow* = *CurUpperBound*
$p_{[i]}$.*NextContourPointOfSameColorBelow* = *CCP*[1][$p_{[i]}$.*col.index*]
CCP[1][$p_{[i]}$.*col.index*] = $p_{[i]}$
CurUpperBound = $p_{[i]}$
Sonst: // $p_{[i]}$ liegt rechts von p_u
Wenn *CCP*[2][$p_{[i]}$.*col.index*].*x* > $p_{[i]}.x$ und *CCP*[2][p_u.*col.index*].x > $p_{[i]}.x$:
$p_{[i]}$.*NextContourPointBelow* = *CurUpperBound*
$p_{[i]}$.*NextContourPointOfSameColorBelow* = *CCP*[2][$p_{[i]}$.*col.index*]
CCP[2][$p_{[i]}$.*col.index*] = $p_{[i]}$
CurUpperBound = $p_{[i]}$

Die Elemente $e(c_i)$ der Farbe c_i sind jetzt definiert durch:

$$e(c_i) := (CCP[1][c_i.index].x, CCP[2][c_i.index].x)$$

4.3 Die Liste MaxElem

Wie bereits angesprochen, hat der ursprüngliche Algorithmus das Manko, immer alle *k* Elemente $e(c_i)$ verwalten zu müssen, insbesondere solche, die nicht maximal sind, da diese durch eine Verschiebung anderer Elemente nach unten oder rechts maximal werden können. Dieses Problem hat der verbesserte Algorithmus nicht mehr, da durch die ausschließliche Betrachtung der Farbkonturpunkte – von oben nach unten verlaufend – die Elemente in *E* immer weiter nach links oben wandern.
Die obige Behauptung wird nun durch die folgenden zwei Lemmata bewiesen:

Lemma 4.3.1) Durch die in Y-Position absteigende und ausschließliche Betrachtung von Farbkonturpunkten wandern die Elemente $e(c_i)$ in E immer weiter nach links oben.

Beweis: Durch die Konstruktion der Farbkonturpunktlisten gilt immer:
$CCP[1][c_i].x > CCP[1][c_i].NextContourPointOfSameColorBelow.x$ und
$CCP[2][c_i].x < CCP[2][c_i].NextContourPointOfSameColorBelow.x$.
Betrachte nun die innere Schleife (*Schleife2*) des verbesserten Algorithmus (siehe Abschnitt 4.4): Sei o.B.d.A. $CCP[1][c_i]$, der Punkt, der gerade als oberer Randpunkt p_o betrachtet wird ($p_o = CCP[1][c_i]$). Der Algorithmus wandert zu Beginn der Schleife die linke Farbkontur der Farbe c_i einen Punkt weiter nach unten:
$CCP[1][c_i] := CCP[1][c_i].NextContourPointOfSameColorBelow$.
So ist während des Schleifendurchlaufs der korrekte Wert $L(c_i, p_u, p_o) = CCP[1][c_i].x$ für die Berechnung von *ExclL* gegeben. Am Ende der Schleife wird $e(c_i)$ durch $e(c_i).x := CCP[1][c_i].x$ verschoben, wobei der neue Wert $CCP[1][c_i].x$ kleiner ist, als der vorherige Wert $e(c_i).x$. Dadurch wandert $e(c_i)$ nach links. Analoges gilt, wenn p_o ein rechter Farbkonturpunkt ist. Hier wandert $e(c_i)$ nach oben. q.e.d. (Lemma 4.3.1)

Lemma 4.3.2) Einmal dominierte Elemente $\boldsymbol{e_d(c_i)}$ können erst dann wieder maximal (dominant) werden, wenn sie selbst verschoben werden.

Beweis: Sei $e_d(c_i)$ ein Element, dass von $\boldsymbol{e_D(c_j)}$ dominiert wird.
Dann gilt: $e_d(c_i).x > e_D(c_j).x$ und $e_d(c_i).y < e_D(c_j).y$.
Angenommen, $e_d(c_i)$ wird maximal, obwohl es nicht verschoben wird.
Dann müsste gelten: $e_d(c_i).x < e_D(c_j).x_{neu}$ oder $e_d(c_i).y > e_D(c_j).y_{neu}$. Weil aber nach *Lemma 4.3.1* die Werte $e(c_i).x$ immer kleiner werden und die Werte $e(c_i).y$ immer größer werden, kann die obige Annahme nie vorkommen. q.e.d. (Lemma 4.3.2)

Wegen *Lemma 4.3.2* müssen in der Struktur *MaxElem* nicht mehr alle Elemente aus E verwaltet werden, sondern nur noch solche, die maximal sind. Maximale Elemente haben aber die schöne Eigenschaft, dass eine nach X-Koordinate sortierte Liste dieser Elemente auch gleichzeitig nach Y-Koordinate sortiert ist. Man erhält so eine streng monoton steigende Folge von Elementen (streng, wegen paarweise unterschiedlichen X-Koordinaten der Punkte in P). Diese kann in einem balancierten Binärbaum gespeichert werden.

Beim Verschieben eines Elements $e(c_i)$ müssen nun folgende Fälle beachtet werden (hierbei wird das alte Element $e_{alt}(c_i)$ und das neue Element $e_{neu}(c_i)$ genannt):

- Fall a) Sowohl $e_{alt}(c_i)$ und $e_{neu}(c_i)$ sind nicht maximal: Die Liste wird nicht verändert.
- Fall b) $e_{alt}(c_i)$ ist nicht maximal und $e_{neu}(c_i)$ ist maximal: Füge $e_{neu}(c_i)$ an korrekter X-Position in *MaxElem* ein. Lösche dann solange nachfolgende Elemente in der Liste, bis das Folgeelement nicht von $e_{neu}(c_i)$ dominiert wird.
- Fall c) $e_{alt}(c_i)$ ist maximal und somit auch $e_{neu}(c_i)$: Lösche $e_{alt}(c_i)$ und gehe zu Fall b).

Es bleibt jetzt noch zu klären, wie hoch die Kosten für die obigen Operationen sind:

- Fall a) Die Suche benötigt $O(\log k)$ Zeit, da für jede der k Farben ein Element in *MaxElem* gespeichert sein kann. Das kommt genau dann vor, wenn die nach X-Koordinate sortierte

Reihenfolge der aktuellen Farbkonturpunkte auf der linken Seite von p_u genau der Reihenfolge der aktuellen Farbkonturpunkte auf der rechten Seite entspricht.

- Fall b und Fall c) Das Einfügen bzw. das Löschen und Einfügen eines Elements benötigt O(log k) Zeit. Die Frage ist nur: Wie viele neu dominierte Elemente sind zu löschen? Die *Schleife2* wird $p_u.index$ mal durchlaufen (genauer: $i \in [1, p_u.index - k + 1]$). D.h., es können nicht mehr als O(n) Einfügungen stattfinden. Da ein Element auch nur höchstens einmal gelöscht werden kann, können pro Durchlauf von *Schleife1* auch nicht mehr als O(n) Löschungen vorgenommen werden. Damit hat man pro Durchlauf von *Schleife1* Gesamtkosten in Höhe von O(n log k) für sämtliche Löschungen und demnach im Durchschnitt auch nicht mehr als O(log k) Kosten für Löschungen pro Durchlauf von *Schleife2*.

Insgesamt wurde gezeigt, dass nur die maximalen Elemente aus E verwaltet werden müssen, und pro Operation auf der dazugehörigen Liste *MaxElem* auch nicht mehr als O(log k) Zeit benötigt wird.

4.4 Schema des Algorithmus

(Änderungen bzgl. des ursprünglichen Algorithmus' sind blau gekennzeichnet.)

Setze die Größe des *bisher kleinsten gefundenen Rechtecks* auf unendlich.

Der Algorithmus durchläuft alle potenziellen unteren Begrenzungspunkte p_u der zu suchenden Rechtecke. Hierbei fängt er bei dem k-obersten Punkt $p_{[k]}$ an und läuft bis zum untersten Punkt $p_{[n]}$ durch (*Schleife1*).

> *Erstellung der Farbkonturen* // Siehe Abschnitt 4.2
>
> Initialisierung von *MaxElem*: Füge die $k+1$ Elemente der Menge
> $E = \{e_r(p_u.col), e_l(p_u.col)\} \cup \{e(c_i) = (0, 1) \mid i \in [1, k] \wedge c_i \neq p_u.col\}$ in die Liste ein.
>
> $p_o = CurUpperBound$ // Oberster Farbkonturpunkt ist durch die Erstellung
> // der Farbkonturen bekannt
>
> Solange $\neg(\exists\ e(c_i) \in E \mid e(c_i) = (0, 1))$ (*Schleife2*): // Alle Farben sind im Streifen
> // zwischen p_u und p_o enthalten. Daher wird diese Schleife spätestens bei $p_{[pu.index - k + 1]}$
> // beendet.
>
> i. Wenn $p_o.x < p_u.x$, dann sei // Linke Farbkontur weiterwandern
> $CCP[1][p_o.col] = CCP[1][p_o.col].NextContourPointOfSameColorBelow$,
> sonst : // Rechte Farbkontur weiterwandern
> $CCP[2][p_o.col] = CCP[2][p_o.col].NextContourPointOfSameColorBelow$
>
> ii. *Suche das erste geeignete Elementpaar* $e_R(c_R)$, $e_L(c_L)$. // Siehe Abschnitt 3.1
> Falls solch ein Paar nicht existiert, gehe zu Schritt vi.
>
> iii. Falls das Rechteck $Rect(e_L(c_L).x, p_o.y, e_R(c_R).y, p_u.y)$ eine kleinere Fläche (Umfang, Diagonale...) als das *bisher kleinste gefundene Rechteck* hat, dann merke dieses als *bisher kleinstes gefundenes Rechteck*.
>
> iv. Setze $e_R(c_R) := e_L(c_L)$.
> Gehe zu Schritt vi, falls nicht $InclR \leq e_R(c_R).y < ExclR$ gilt.
>
> v. Sei $e_L(c_L) = e_R(c_R).Next$.
> Wenn solch ein Element existiert und $ExclL < e_L(c_L).x \leq InclL$ gilt,
> dann gehe zu Schritt iii.
>
> vi. Aktualisiere ggf. die Liste *MaxElem* durch das Element
> $e_{neu}(p_o.col) = (CCP[1][p_o.col].x, CCP[2][p_o.col].x)$. // Siehe Abschnitt 4.3
> $p_o = p_o.NextContourPointBelow$
>
> Ende von *Schleife2*.

Ende von *Schleife1*.

Gib das *bisher kleinste gefundene Rechteck* aus.

Ende der Suche.

4.5 Die Kosten (1)

Insgesamt ergeben sich für den verbesserten Algorithmus folgende Kosten:

Schleife1 (mit Laufindex i) ____ O(n-k)
- Erstellung der Farbkonturen ____ O(n)
- Initialisierung von *MaxElem* ____ O(k log k)
- *Schleife2 (mit Laufindex j)* ____ O(n-k)
  - i) Kontur weiterwandern ____ O(1)
  - ii) Suche erstes geeignetes Elementpaar ____ O(log k)
  - iii – v) Prüfung der gefundenen Rechtecke ____ O($a_{i,j}$)
  - vi) Aktualisierung der Liste *MaxElem* ____ O(log k)

Wie in Abschnitt 3.3, sind die Kosten für die Prüfung der gefundenen Rechtecke in Schritt iii – v irrelevant. Daher erhält man Gesamtkosten in Höhe von **O((*n*-*k*) *n* log *k*)**.

In den folgenden zwei Abschnitten wird gezeigt, wie die Gesamtkosten noch weiter gesenkt werden können.

4.6 Initialisierung von MaxElem in O(n)

An dieser Stelle sei nochmals daran erinnert, dass die Punkte bzgl. der X-Koordinate in aufsteigender Reihenfolge verkettet sind (siehe Abschnitt 2.1). Die Verkettung wird durch die zusätzlichen Punkteigenschaften ***Next*** und ***Previous*** dargestellt, wobei gilt:
$p.Previous.x < p.x < p.Next.x$.

Weiterhin sei bemerkt, dass die Farbkonturen zum Zeitpunkt der Initialisierung der *MaxElem*-Liste schon bekannt sind.

Ziel ist es, in O(n) Zeit die sortierte Liste der Elemente zu finden, die nach der Initialisierung in *MaxElem* übrig bleiben, damit dann in O(k) Zeit die balancierten Binärbäume der nach X- als auch Y-Koordinate sortierten Elemente erstellt werden können. Der schematische Ablauf der Erstellung dieser Liste sieht wie folgt aus:

Die Prozedur ***Initialisiere MaxElem***

Erstelle ein Array $Q[1, k]$ aus folgenden Objekten:
- $Q[i].bolFound$ = false
- $Q[i].x = 0$
- $Q[i].y = 1$
- $Q[i].Next = null$
- $Q[i].Previous = null$

Erstelle weiterhin ein Array $W[1, k + 1]$ von Punkten.

```
p = p_u
LastFoundPoint = null
p_uColMinY = 1          // Sorgt dafür, dass die gefundenen Punkte nicht von Punkten
                        // der Farbe p_u.col dominiert werden
Solange p.Previous ≠ null (SchleifeA)
        p = p.Previous
        Wenn p.y > p_u.y und p.y < p_uColMinY
                Wenn p.col ≠ p_u.col und Q[p.col.index].bolFound = false
                        Q[p.col.index].bolFound = true
                        Q[p.col.index].x = p.x
                        Q[p.col.index].Next = LastFoundPoint
                        Q[p.col.index].col = p.col
                        Wenn LastFoundPoint ≠ null
                                Q[p.col.index].Next.Previous = Q[p.col.index]
                        LastFoundPoint = Q[p.col.index]
                Wenn p.col = p_u.col
                        p_uColMinY = p.y
Ende von SchleifeA

Sei MaxY ein Punkt mit
        MaxY.x = 0
        MaxY.y = p_u.x
        MaxY.col = p_u.col

For m = 1 to k (SchleifeB):
        Wenn m ≠ p_u.col.index und CCP[2][m].x > MaxY.y und Q[m].bolFound = false
                MaxY.y = CCP[2][m].x
                MaxY.col = c_m
Ende von SchleifeB

CountElements = 1
W[CountElements] = MaxY

Solange LastFoundPoint ≠ null (SchleifeC)
        Wenn CCP[2][LastFoundPoint.col.index].x > W[CountElements].y
                CountElements = CountElements + 1
                W[CountElements].x = LastFoundPoint.x
                W[CountElements].col = LastFoundPoint.col
                W[CountElements].y = CCP[2][LastFoundPoint.col.index].x
        LastFoundPoint = LastFoundPoint.Next
Ende von SchleifeC

Wenn W[CountElements].y < 1
        CountElements = CountElements + 1
        W[CountElements].x = p_u.x
        W[CountElements].col = p_u.col
        W[CountElements].y = 1
```

Erstelle die balancierten Binärbäume mit den nach X- und Y-Koordinate sortierten Punkten des Arrays $W[1, CountElements]$

Ende der Prozedur

Hier sei noch einmal in verständlicheren Worten erklärt, was die obige Prozedur leistet:

SchleifeA: Laufe vom unteren Grenzpunkt p_u nach links und suche das erste Vorkommen jeder Farbe, die oberhalb von p_u liegt und nicht von Punkten der Farbe $p_u.col$ dominiert wird, und speicher die X-Positionen und die direkten Nachfolger (und vom Nachfolger den jetzt bekannten Vorgänger) in einem Array Q, welches nach Farben sortiert ist (siehe Abbildung 4.6.1). Die Kosten für *SchleifeA* liegen in $O(n)$.

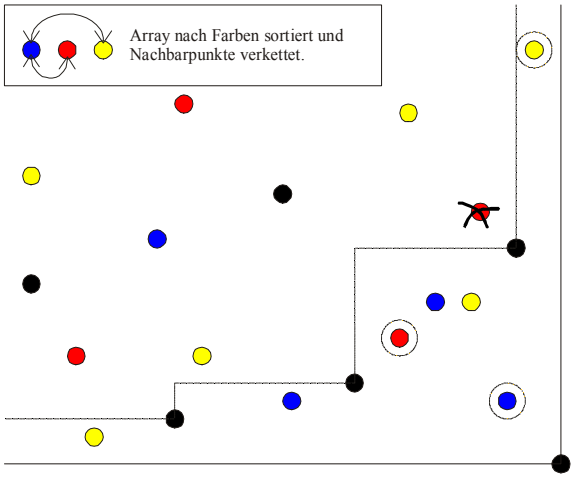


Abb. 4.6.1) *SchleifeA* sucht oberhalb und links von p_u und unterhalb der schwarzen Farbkontur nach den jeweils am weitesten rechts liegenden Vorkommen aller Farben.

SchleifeB: Suche jetzt in den rechten Farbkonturen den höchsten X-Wert der Farben, die nicht auf der linken Seite gefunden wurden. Das entsprechende Element mit der – aus dem X-Wert des Punktes resultierenden – höchsten Y-Koordinate ist das Erste in der *MaxElem* - Liste. Falls alle Farben auf der linken Seite vorkommen, wird $(0, p_u.x)$ das erste Element. Die Kosten für *SchleifeB* liegen in $O(k)$.

SchleifeC: Untersuche dann die jeweils ersten Punkte der rechten Farbkonturen in der umgekehrten Reihenfolge, wie sie in *SchleifeA* gefunden wurden. Überprüfe, ob die X-Koordinate des Farbkonturpunktes höher ist, als die Y-Koordinate des zuletzt eingefügten Elements $W[CountElements]$. Wenn ja, dann füge das neue Element in W ein. Falls die entsprechende Farbe auf der rechten Seite nicht vorkommt, dann wird die Y-Koordinate auf 1 gesetzt, das Element wird eingefügt und die Erstellung der Liste ist somit fertig (siehe Abbildung 4.6.2). Die Kosten für *SchleifeC* liegen in $O(k)$.

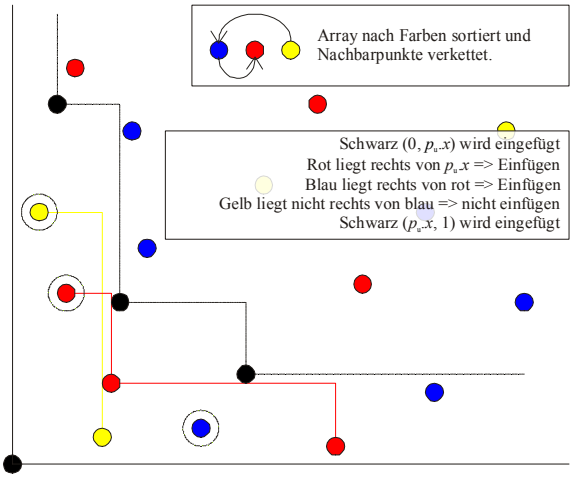


Abb. 4.6.2) *SchleifeC* überprüft jeweils den ersten Punkt jeder rechten Farbkontur in der umgekehrten Reihenfolge, wie sie in *SchleifeA* gefunden wurden.

Ganz zum Schluss überprüfe man noch, ob $e_l = (p_u.x, 1)$ sich nach den Kriterien in *SchleifeC* auch einfügen lässt, und füge es dementsprechend ein.

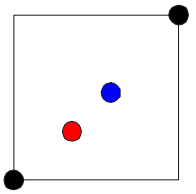


Abb. 4.6.3) Die Liste *MaxElem*.

Erstelle jetzt in $O(k)$ Zeit die balancierten Binärbäume mit den nach X- und Y-Koordinaten sortierten Elementen des Arrays $W[1, CountElements]$. Man beachte, dass $CountElements \leq k+1$ ist, da dominierte Elemente natürlich nicht in *MaxElem* eingefügt werden.

4.7 Die Kosten (2)

Insgesamt ergeben sich für den nochmals verbesserten Algorithmus folgende Kosten:

Schleife1 (mit Laufindex i) _____ O(*n-k*)
- Erstellung der Farbkonturen _____ O(*n*)
- Initialisierung von *MaxElem* _____ O(*n*)
- *Schleife2 (mit Laufindex j)* _____ O(*n-k*)
  - i) Kontur weiterwandern _____ O(1)
  - ii) Suche erstes geeignetes Elementpaar _____ O(log *k*)
  - iii – v) Prüfung der gefundenen Rechtecke _____ O($a_{i,j}$)
  - vi) Aktualisierung der Liste *MaxElem* _____ O(log *k*)

Man erhält somit Gesamtkosten in Höhe von $\mathbf{O((n-k)(n + (n-k) \log k))}$.

Es stellt sich die Frage, wann $n \leq (n-k) \log k$ gilt:

$$
\begin{aligned}
& n \leq (n-k)\log k \Leftrightarrow \\
& n \leq n\log k - k\log k \Leftrightarrow \\
& n - n\log k \leq -k\log k \Leftrightarrow \\
& n(1-\log k) \leq -k\log k \Leftrightarrow \\
& n \geq -\frac{k\log k}{1-\log k} \Leftrightarrow \\
& n \geq k\frac{\log k}{\log k - 1}
\end{aligned}
$$

Damit hat man

für $n \geq k\frac{\log k}{\log k - 1}$ Kosten in Höhe von $\mathbf{O((n-k)^2 \log k)}$

und für $n \leq k\frac{\log k}{\log k - 1}$ Kosten in Höhe von $\mathbf{O((n-k)n)}$.

Eine graphische Veranschaulichung des Verhältnisses zwischen *n* und *k* stellt Abbildung 4.7.1 dar: Das Diagramm zeigt in X-Richtung log *k* und in Y-Richtung den Faktor, um den *n* mindestens größer als *k* sein muss, damit die Kosten O($(n-k)^2 \log k$) erreicht werden.

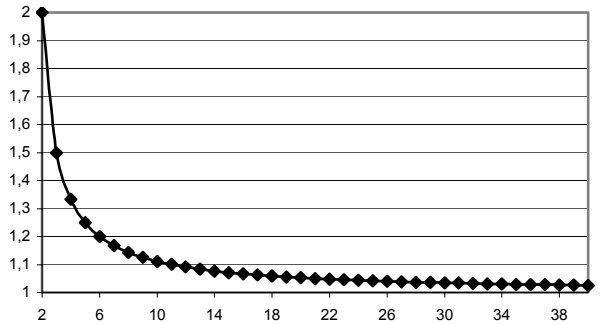


Abb. 4.7.1) Das Verhältnis zwischen *n* und *k*.

5 Beliebige Orientierung

In diesem Kapitel wird erläutert, wie obiger Algorithmus dazu verwendet werden kann, kleinste farbumspannende Rechtecke beliebiger Orientierung zu suchen. Die Schwierigkeit dieses Problems liegt darin, für jede, der $n(n\text{-}1)$ vielen Orientierungen die Voraussetzungen aus Abschnitt 2.1 sicherzustellen, damit man obigen Algorithmus überhaupt verwenden kann.

5.1 Einige Vorüberlegungen

Die Voraussetzungen aus Abschnitt 2.1 fordern vor allem $n(n\text{-}1)$ viele Sortierungen der Punktmenge. (Die Frage, warum nur die Winkel betrachtet werden müssen, die durch zwei Punkte gebildet werden, wird in Abschnitt 5.4 geklärt.)
Bei naiver Implementierung würde man dafür $O(n^3 \log n)$ Zeit benötigen. Damit hätte auch der gesamte Algorithmus mindestens diese Kosten.

Die Idee ist nun, die Punktmenge im Uhrzeigersinn rotieren zu lassen. Wenn während dieser Rotation zwei Punkte die gleiche Y-Koordinate haben, dann betrachte man diese Punkte als potenzielle untere Randpunkte $\boldsymbol{p_{u1}}$ und $\boldsymbol{p_{u2}}$ eines nicht verkleinerbaren Rechtecks und verwende jetzt eine leicht abgeänderte Version des Inneren von *Schleife1* des in Abschnitt 4.4 und 4.7 vorgestellten Algorithmus. Danach werden die Punkte p_{u1} und p_{u2} in dem nach Y-Koordinaten sortierten Array $p_{[1, n]}$ vertauscht.
Weiterhin werden in der nach X-Koordinate sortierten Verkettung die Punkte vertauscht, die während der Rotation die gleiche X-Koordinate haben.
So ist für alle $n(n\text{-}1)$ Orientierungen eine korrekte Sortierung beider Koordinaten sichergestellt.
Formal gesehen betrachte man sich folgende Überlegungen:

Definition 5.1.1) Eine *Y-Orientierung* $\alpha_{i,j}$ zwischen zwei Punkten p_i und p_j wird wie folgt berechnet:
Zeichne einen waagerechten Strahl von p_i nach rechts und rotiere diesen Strahl so lange gegen den Uhrzeigersinn um p_i, bis er p_j berührt. $\alpha_{i,j}$ entspricht dann dem Rotationswinkel (siehe Abbildung 5.1.1).

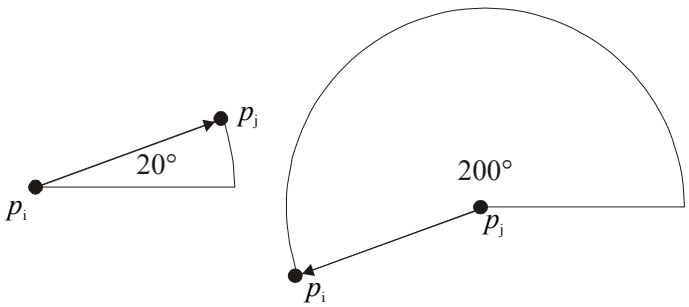


Abb. 5.1.1) Die Y-Orientierungen $\alpha_{i,j}$ und $\alpha_{j,i}$

Definition 5.1.2) Eine *X-Orientierung* $\xi_{i,j}$ zwischen zwei Punkten p_i und p_j wird wie folgt berechnet:
Zeichne einen senkrechten Strahl von p_i nach oben und rotiere diesen Strahl so lange gegen den Uhrzeigersinn um p_i, bis er p_j berührt. $\xi_{i,j}$ entspricht dann dem Rotationswinkel.

Lemma 5.1.1) In einem nach Y-Koordinaten absteigend sortierten Punkt-Array $p_{[1, n]}$ liegt die kleinste aller $n(n\text{-}1)$ möglichen Y-Orientierungen zwischen zwei benachbarten Punkten $p_{[i]}$ und $p_{[i-1]}$ des Arrays.

Beweis: Angenommen, es gäbe zwei Punkte $p_{[i]}$ und $p_{[i-m]}$, mit $m > 1$, deren Y-Orientierung $\alpha_{[i],[i-m]}$ die kleinste ist. Dann gilt aber:

Entweder ist $\alpha_{[i],[i-1]}$ oder $\alpha_{[i-1],[i-m]}$ kleiner als $\alpha_{[i],[i-m]}$. Das ist ein Widerspruch zu obiger Annahme (siehe Abbildung 5.1.2). q.e.d.
(Lemma 5.1.1)

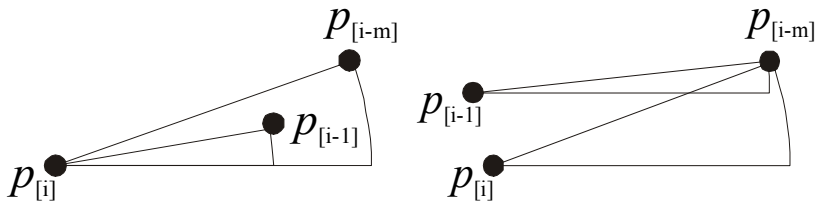


Abb. 5.1.2) Die kleinste Y-Orientierung liegt zwischen zwei aufeinanderfolgenden Punkten.

Analoges gilt für X-Orientierungen in einer nach X-Koordinate sortierten und verketteten Punkt-Liste.
Dank dieser Eigenschaften kann der Algorithmus wie folgt definieret werden.

5.2 Schema des Algorithmus

1. Füge die n Punkte nach Y-Koordinate absteigend sortiert in das Array $p_{[1, n]}$ ein und verkette sie nach sortierten X-Koordinaten.
2. Erstelle eine in einem AVL-Baum $\mathfrak{Y}$ gespeicherte sortierte Liste aller n-1 Y-Orientierungen $\alpha_{[i],[i-1]}$ der Punktpaare $p_{[i]}$, $p_{[i-1]}$ mit $1 < i \leq n$.
3. Erstelle eine in einem AVL-Baum $\mathfrak{X}$ gespeicherte sortierte Liste aller n-1 X-Orientierungen $\xi_{j,j-1}$ der jeweils benachbarten Punkte p_j und $p_j.Previous$.
4. $\alpha_{alt} = 0$
5. Durchlaufe die folgende Schleife $2n(n\text{-}1)$ mal. // Die Frage, warum nur die Winkel // betrachtet werden müssen, die durch zwei Punkte gebildet werden, wird in // Abschnitt 5.4 geklärt.
   a. Suche die kleinste Y-Orientierung $\alpha_{[i],[i-1]}$ und die kleinste X-Orientierung $\xi_{j,j-1}$.
   b. Wenn $\alpha_{[i],[i-1]} < \xi_{j,j-1}$, dann
      i. Wenn $p_{[i]}.col = p_{[i-1]}.col$, dann gehe zu 5.b.vii.
      ii. Wenn $i < k$, dann gehe zu 5.b.vii.
      iii. Rotiere die Punktmenge im Uhrzeigersinn um den Winkel $\alpha_{[i],[i-1]} - \alpha_{alt}$.
      iv. $\alpha_{alt} = \alpha_{[i],[i-1]}$.
      v. Betrachte die Punkte $p_{[i]}$ und $p_{[i-1]}$ als untere Randpunkte p_{u1} und p_{u2}.
      vi. Verwende eine leicht geänderte Version des Inneren von *Schleife1* des in Abschnitt 4.4 und 4.7 vorgestellten Algorithmus (s.u.).
      vii. Lösche in $\mathfrak{Y}$ die Orientierungen $\alpha_{[i+1],[i]}$, $\alpha_{[i],[i-1]}$ und $\alpha_{[i-1],[i-2]}$.
      viii. Vertausche die Punkte $p_{[i]}$ und $p_{[i-1]}$ in dem nach Y-Koordinate absteigend sortierten Array $p_{[1, n]}$.
      ix. Füge die neuen Orientierungen $\alpha_{[i+1],[i]} + \alpha_{alt}$, $\alpha_{[i],[i-1]} + \alpha_{alt}$ und $\alpha_{[i-1],[i-2]} + \alpha_{alt}$ in $\mathfrak{Y}$ ein, sofern sie kleiner als 2π ,bzw. 360° sind.
   c. Sonst
      i. Lösche in $\mathfrak{X}$ die Orientierungen $\xi_{j-1,j-2}$, $\xi_{j,j-1}$ und $\xi_{j+1,j}$.
      ii. Vertausche die Punkte p_j und p_{j-1} in der nach X-Koordinate sortierten Verkettung.

iii. Füge die neuen Orientierungen $\xi_{j-1,j-2} + \alpha_{alt}$, $\xi_{j,j-1} + \alpha_{alt}$ und $\xi_{j+1,j} + \alpha_{alt}$ in $\mathfrak{X}$ ein, sofern sie kleiner als 2π ,bzw. 360° sind.

6. Drehe die Punktmenge um den Winkel $2\pi - \alpha_{alt}$ im Uhrzeigersinn und gib die kleinsten Rechtecke mit den entsprechenden Orientierungen aus.

In der leicht geänderten Version des Inneren von *Schleife1* des in Abschnitt 4.4 und 4.7 vorgestellten Algorithmus in Schritt 5.b.vi muss Folgendes beachtet werden:

- Am Ende von Abschnitt 4.2 wurde erläutert, dass Punkte der Farbe des unteren Randpunktes p_u als weitere Einschränkung für die Erstellung der Farbkonturen herangezogen werden dürfen. Da es nun zwei untere Randpunkte gibt, können auch Punkte beider Farben zur Einschränkung der Konturen herangezogen werden, da beide Farben nicht ein weiteres mal im Rechteck vorkommen dürfen (Abbildung 5.2.1).

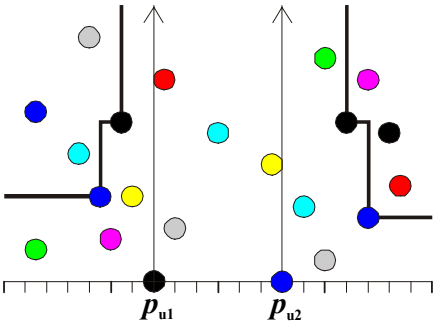


Abb. 5.2.1) Einschränkung durch zwei Farben

- Für die Punkte oberhalb von p_{u1} und p_{u2} gibt es jetzt neben den Lagen „links von p_{u1}“ und „rechts von p_{u2}“ auch noch die dritte Lage „zwischen p_{u1} und p_{u2}“. Punkte $\boldsymbol{p_m}$ in solch einer Lage haben weitere einschränkende Eigenschaften:

  - Wenn $p_m.col = p_{u1}.col$ oder $p_m.col = p_{u2}.col$, dann beende die Farbkonturerstellung, weil die gesuchten Rechtecke nicht höher sein können, da sonst ein Punkt mit der Farbe eines unteren Randpunktes im Inneren eines solchen Rechtecks liegen würde. Das hat den großen Vorteil, dass man u.U. nicht bis ganz nach oben zu $p_{[1]}$ durchlaufen muss, sondern schon viel eher aufhören darf (Abbildung 5.2.2).

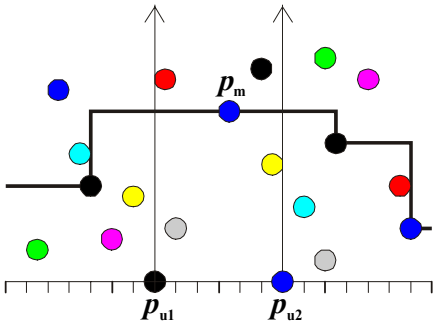


Abb. 5.2.2) Vorzeitige Beendigung der Farbkonturerstellung

  - Falls $p_m.col \neq p_{u1}.col$ und $p_m.col \neq p_{u2}.col$, dann kann die Erstellung der Farbkontur der Farbe $p_m.col$ beendet werden. Hierbei werden dann die linke und rechte Kontur dieser Farbe in p_m verknüpft (Abbildung 5.2.3).

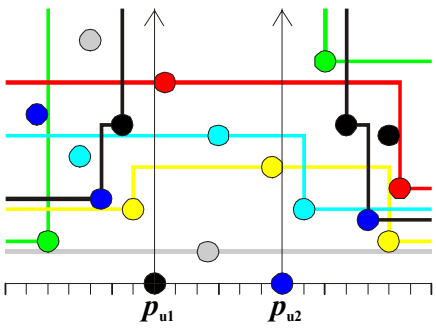


Abb. 5.2.3) Verknüpfung einiger Farbkonturen

- Der in Abschnitt 2.5 beschriebene Sonderfall bei der Initialisierung von *MaxElem* wird nun durch Einfügung der Elemente $e_r(p_{u2}.col) = (0, p_{u2}.x)$ und $e_l(p_{u1}.col) = (p_{u1}.x, 1)$ abgeändert, damit jetzt auch Rechtecke gefunden werden können, die p_{u1} als linken unteren und p_{u2} als rechten unteren Eckpunkt haben (Abbildung 5.2.4 und 5.2.5).

- Farbkonturpunkte p_m in mittlerer Lage müssen nicht in *MaxElem* eingefügt werden, da sie auf jeden Fall dominiert werden. Diesen Umstand verdanken wir den besonderen Koordinaten dieser Elemente $e(p_m.col) = (p_m.x, p_m.x)$ und dem Vorhandensein von $e_r(p_{u2}.col)$ und $e_l(p_{u1}.col)$ bzw. durch noch weiter links bzw. höher liegenden Elementen (Abbildung 5.2.4).

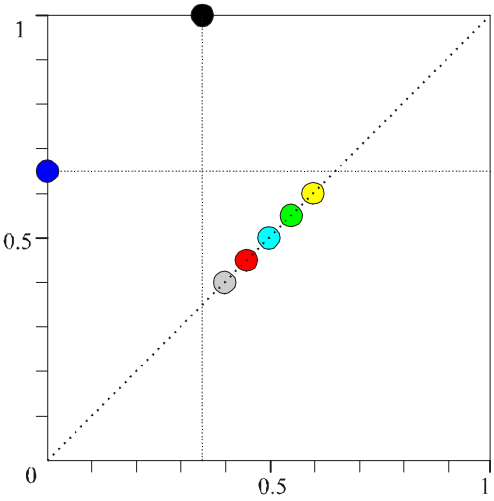


Abb. 5.2.4) Die Elemente $e(p_m.col)$ liegen immer unterhalb von $e_r(p_{u2}.col)$ und rechts von $e_l(p_{u1}.col)$

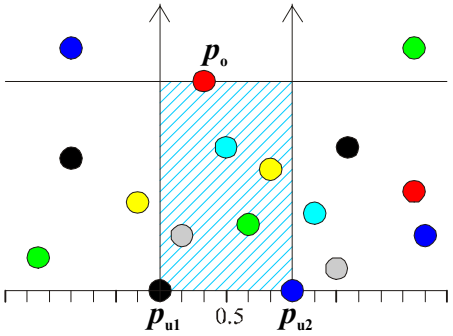


Abb. 5.2.5) Die beiden unteren Randpunkte können gleichzeitig Eckpunkte sein

- Bei der Initialisierung von *MaxElem* in linearer Zeit muss natürlich auch berücksichtigt werden, dass zwei untere Randpunkte vorhanden sind. Die Änderungen bzgl. der alten Prozedur sehen wie folgt aus:

Die Prozedur ***Initialisiere MaxElem2***
(Änderungen bzgl. der ursprünglichen Prozedur sind blau gekennzeichnet.)

Erstelle die Arrays $Q[1, k]$ und $W[1, k + 1]$ wie gehabt.

$p = p_{u1}$
LastFoundPoint = *null*
$p_u ColMinY = p_o.y$ // Sorgt dafür, dass die gefundenen Punkte nicht von Punkten
// der Farbe $p_{u1}.col$ oder $p_{u2}.col$ dominiert werden

For m = 1 to k (*SchleifeM*): // Die Farben mittlerer Konturpunkte müssen
Wenn $CCP[0][m] = CCP[1][m]$ // zwar gefunden werden, brauchen aber sonst
$Q[m].bolFound$ = true // nicht weiter beachtet werden

Solange $p.Previous \neq null$ (*SchleifeA*)
$p = p.Previous$

Wenn $p.y > p_u.y$ und $p.y \leq p_u ColMinY$
Wenn $p.col \neq p_{u1}.col$ und $p.col \neq p_{u2}.col$ und
$Q[p.col.index].bolFound$ = false
$Q[p.col.index].bolFound$ = true
$Q[p.col.index].x = p.x$
$Q[p.col.index].Next = LastFoundPoint$
$Q[p.col.index].col = p.col$
Wenn $LastFoundPoint \neq null$
$Q[p.col.index].Next.Previous = Q[p.col.index]$
$LastFoundPoint = Q[p.col.index]$
Wenn $p.col = p_{u1}.col$ oder $p.col = p_{u2}.col$
$p_u ColMinY = p.y$
Ende von *SchleifeA*

Sei *MaxY* ein Punkt mit
$MaxY.x = 0$
$MaxY.y = p_{u2}.x$
$MaxY.col = p_{u2}.col$

For m = 1 to k (*SchleifeB*):
Wenn $m \neq p_{u1}.col.index$ und $m \neq p_{u2}.col.index$ und $CCP[2][m].x > MaxY.y$ und
$Q[m].bolFound$ = false
$MaxY.y = CCP[2][m].x$
$MaxY.col = c_m$
Ende von *SchleifeB*

$CountElements$ = 1
$W[CountElements] = MaxY$

Durchlaufe *SchleifeC* wie gehabt

Wenn $W[CountElements].y < 1$
$CountElements = CountElements + 1$
$W[CountElements].x = p_{u1}.x$
$W[CountElements].col = p_{u1}.col$
$W[CountElements].y = 1$

Erstelle die balancierten Binärbäume mit den nach X- und Y-Koordinate sortierten Punkten des Arrays $W[1, CountElements]$

Ende der Prozedur

5.3 Die Kosten

Es ergeben sich daraus die zu erwartenden Kosten:

- O(n log n) für die Erstellung des nach Y-Koordinate sortierten und nach X-Koordinate verketteten Arrays $p_{[1, n]}$ und den AVL-Bäumen $\mathfrak{X}$ und $\mathfrak{Y}$.
- $2n(n\text{-}1)$ Schleifendurchläufe mit den jeweiligen Kosten:
  - Suche der kleinsten Orientierungen in O(*1*).
  - Rotation der Punkte in O(n).
  - Erstellung der Farbkonturen in O(n).
  - Initialisierung von *MaxElem* in O(n).
  - O((n-k) log k) für den inneren Schleifendurchlauf von $p_{[0]}$ bis $p_{[i\text{-}k+1]}$.
  - Löschen und Einfügen von Orientierungen in O(log n)

Daraus ergeben sich für die Berechnung kleinster farbumspannender Rechtecke beliebiger Orientierung Gesamtkosten in Höhe von $O(n^2(n + (n\text{-}k) \log k))$.

In dieser Abschätzung wurden noch nicht die in dem Schema enthaltenen Einschränkungen aus den Schritten 5.b.i und 5.b.ii berücksichtigt. Es bleibt also die Frage, wie oft zumindest eine dieser beiden Bedingungen wirksam wird und man demnach für diese Orientierungen keine Kosten in Höhe von $O((n\text{-}k) \log k)$ hat.

Zu 5.b.i) Wie oft haben die Punkte p_{u1} und p_{u2} die selbe Farbe?
Die Wahrscheinlichkeit, dass bei k Farben die beiden Punkte die selbe Farbe haben, ist bei absoluter Gleichverteilung der Farben $1/k$ (n/k Punkte pro Farbe). Sobald es Farben gibt, die mehr als n/k Punkte einfärben, erhöht sich diese Wahrscheinlichkeit wegen folgender einfachen Rechnung:
Sei $q=n/k$. Dann gilt: $2q^2 < (q\text{-}1)^2 + (q+1)^2 = 2q^2 + 2$.
Also fallen für mindestens $n(n\text{-}1)/k$ Y-Orientierungen keine Kosten in Höhe von $O((n\text{-}k) \log k)$ an.

Somit reduzieren sich die Gesamtkosten auf $O\left(n^2 \frac{k-1}{k}\left(n + \left(n-k\right)\log k\right)\right)$.

Der Faktor $(k\text{-}1)/k$ ist zwar O-Notations-irrelevant und könnte eigentlich weggelassen werden, aber Abbildung 5.3.2 zeigt, dass gerade für kleine k hierdurch eine nicht unerhebliche Zeiteinsparung von bis zu 33% erzielt wird.

Zu 5.b.ii) Wie oft liegt $p_{[i+1]}$ so weit oben, so dass oberhalb von $p_{[i+1]}$ nicht mehr alle k-2 Farben vorhanden sein können?
Die Wahrscheinlichkeit, dass $p_{[i+1]}$ zu weit oben liegt, ist für allgemeine Punktkonstellationen WK($i < k$) = (k-2)/(n-1). Eine naheliegende Vermutung ist, dass immer nur $n(n\text{-}1)(n\text{-}k+1)/(n\text{-}1)$ viele Orientierungen überprüfen werden müssen und demnach O(nk) viele Orientierungen wegfallen. Leider zeigt das Beispiel in Abbildung 5.3.1, dass zumindest für k=3 und beliebige n im worst case nur drei Orientierungen ($\alpha_{1,3}$, $\alpha_{3,2}$ und $\alpha_{2,1}$) nicht zu überprüfen sind. Dass aber die Anzahl der Farben für die Kostenabschätzung nicht unerheblich ist, zeigt das Folgende:

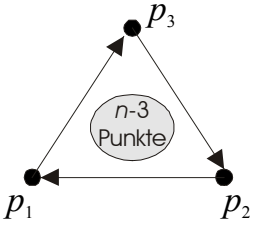


Abb. 5.3.1) Für beliebige n und k=3 fallen nur drei Orientierungen weg

Definition: „*oberhalb* einer Orientierung $\alpha_{i,j}$“ bezeichne Lagen in der Halbebene, die, von p_i aus nach p_j gesehen, links liegt. Der Grund für diese Bezeichnung liegt darin, dass alles, was *oberhalb* dieser Orientierung liegt, tatsächlich über der Geraden – die durch p_i und p_j definiert wird – liegt, wenn p_i und p_j als untere Randpunkte p_{u1} und p_{u2} betrachtet werden.

Lemma 5.3.1) Bei n Punkten und k Farben müssen insgesamt nur $O(n^2\text{-}k^2)$ viele Orientierungen überprüft werden.

Beweis:
Betrachte einen der k-2 oberen Punkte $p_{[1]}$ bis $p_{[k-2]}$ in P, also $p_{[i]}$, mit $1 \le i \le k$-2.
Zeichne eine waagerechte Gerade $\boldsymbol{g_i}$ durch $p_{[i]}$.
Es gilt: In der geschlossenen und oberhalb von g_i liegenden Halbebene $\boldsymbol{H_i}$ liegen i Punkte.
Rotiere nun H_i so lange um $p_{[i]}$ im Uhrzeigersinn, bis nach und nach k-1 Punkte in H_i liegen. Die so neu in H_i hinzugekommenen Punkte heißen $\boldsymbol{p_{[i],i+1}}$ bis $\boldsymbol{p_{[i],k-1}}$.Hierbei ist zu beachten, dass diese Punkte, wenn sie neu hinzukommen, nur auf dem Strahl $\boldsymbol{s_{ir}} \subset g_i$ liegen, der ursprünglich rechts von $p_{[i]}$ lag. Wenn durch die Rotation zwischenzeitlich m Punkte auf dem ursprünglich linken Strahl $\boldsymbol{s_{il}}$ liegen und durch weitere Rotation nicht mehr in H_i liegen, dann wird die Rotation beendet, wenn k-1-m Punkte in H_i liegen. $p_{[i]}$ bildet mit jedem dieser k-i-1 Punkte $p_{[i],i+1}$ bis $p_{[i],k-1}$ eine Orientierung, die nicht überprüft werden muss, da *oberhalb* dieser Orientierung nicht genug Punkte liegen können. Das ergibt für alle k-2 oberen Punkte $p_{[1]}$ bis $p_{[k-2]}$:

$$\sum_{i=1}^{k-2} k-i-1 = \frac{(k-2)(k-1)}{2} \text{ nicht zu untersuchende Orientierungen.}$$

Zusammen mit der gleichen Betrachtung für die k-2 untersten Punkte $p_{[n]}$ bis $p_{[n-k+3]}$ gibt es $(k-2)(k-1)$ nicht zu überprüfende Orientierungen. Beachte hierbei, dass keine Orientierung doppelt gezählt werden kann, da bei der Betrachtung der oberen Punkte die jeweils zu einer wegfallenden Orientierung gehörende Halbebene nach *rechts hin offen* ist
(es gilt: $\forall\, x \in \mathbb{R} \wedge x > p_{[i]}.x$: $(x, p_{[i]}.y) \in H_i$) und bei der Betrachtung der unteren Punkte die Halbebene immer nach *links hin offen* ist. Somit müssen höchstens

$$n(n-1)-(k-2)(k-1) \in O(n^2-k^2)$$

Orientierungen überprüft werden . q.e.d. (Lemma 5.3.1)

Die Anwendung von *Lemma 5.3.1* ergibt die Gesamtkosten

$$O\left(\left(n^2-k^2\right)\left(n+(n-k)\log k\right)\right)$$

für die Suche nach kleinsten farbumspannenden Rechtecken beliebiger Orientierung.

Damit wurden sowohl für große als auch kleine k erhebliche Zeiteinsparungen gewonnen. Abbildung 5.3.2 zeigt den prozentualen Anteil der Orientierungen, die nicht Bedingung 5.b.i und 5.b.ii erfüllen (mit logarithmischer Skalierung von n und k), also den Funktionsgraphen

$$z(n,k) = 100\,\frac{\left(n^2-k^2\right)\frac{k-1}{k}}{n(n-1)}, \text{ für } 3 \le k \le n.$$

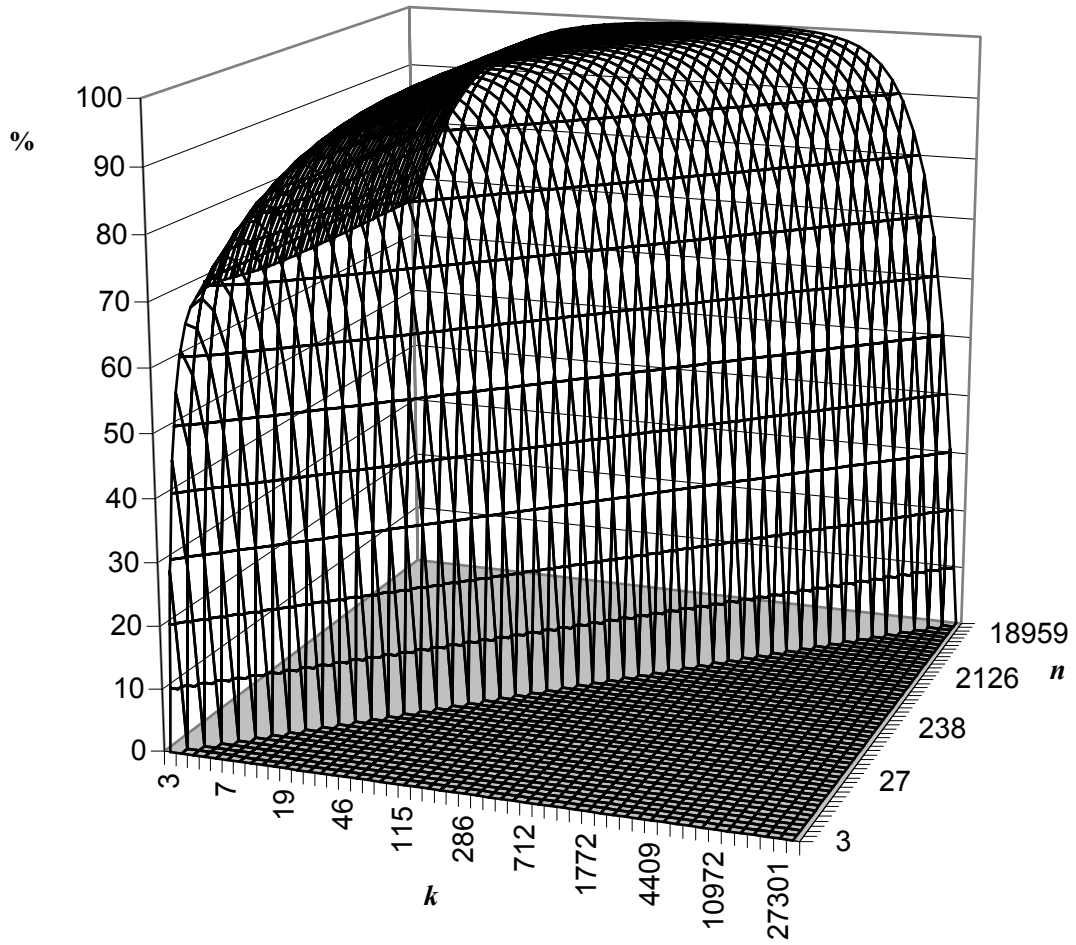


Abb. 5.3.2) Prozentualer Anteil der $n(n\text{-}1)$ Orientierungen, die tatsächlich untersucht werden müssen.

Bei obiger Kostenberechnung wurde noch nicht beachtet, wie viele nicht verkleinerbare Rechtecke beliebiger Orientierung höchstens existieren. Hier wurde davon ausgegangen, das diese Anzahl gegenüber den restlichen Kosten unerheblich ist. Gäbe es aber $O(n^3)$ gesuchte Rechtecke, dann wäre die ganze Berechnung hinfällig. Daher bleibt zu zeigen, dass pro Schleifendurchlauf in Schritt 5.b.vi im Durchschnitt nur konstant viele Kandidaten gefunden werden.

Lemma 5.3.2) Es gibt $O((n^2\text{-}k^2)(n\text{-}k))$ nicht verkleinerbare Rechtecke beliebiger Orientierung.

Beweis:

Der Beweis wird – abgesehen von Typ 2-Rechtecken – analog zu dem von *Lemma 3.3.1* geführt. Der Unterschied besteht darin, dass hier nicht ein beliebiger unterer Punkt, sondern eine beliebige Orientierung fixiert wird. Zu beachten ist dann, dass bei Rechtecken vom Typ 1 zwei Farben als obere Begrenzung möglich sind und dass bei der Erweiterung der dazugehörigen Rechtecke die Punktmenge als – der Orientierung entsprechend – rotiert betrachtet wird.

Abbildung 5.3.3 zeigt die unterschiedlichen Typen erweiterter Rechteckte.

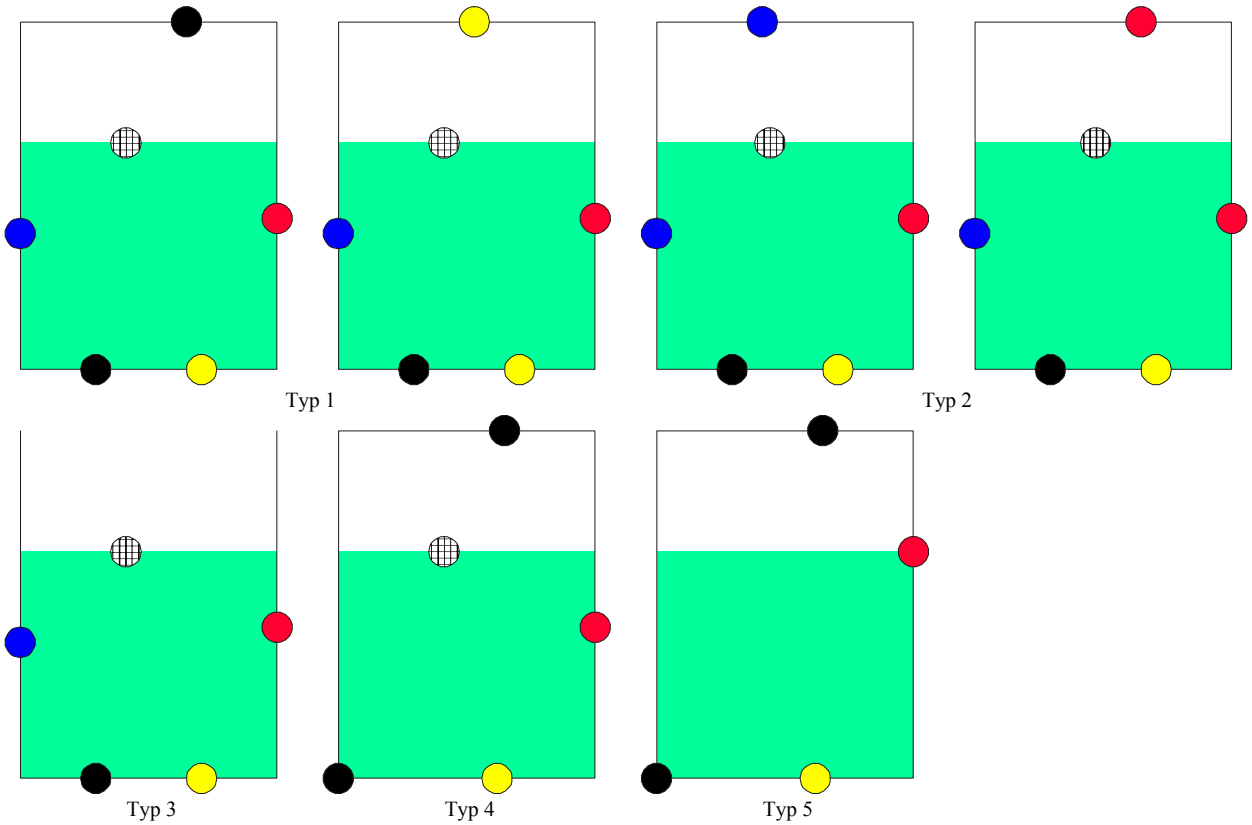


Abb. 5.3.3) Erweiterte Rechtecke beliebiger Orientierung.

Auch hier ist die Abbildung zwischen nicht verkleinerbarem Rechteck und erweitertem Rechteck bijektiv und es genügt zu zeigen, dass es nur $O((n^2-k^2)(n-k))$ viele erweiterte Rechtecke gibt. Hierzu genügt es zu zeigen, dass es pro Orientierung jeweils nur $O(n-k)$ Rechtecke der unterschiedlichen Typen gibt.

Typ 1:
Fixiere eine beliebige Y-Orientierung $\alpha_{u1,u2}$, wobei die dazugehörigen Punkte p_{u1} und p_{u2} die unteren Randpunkte der erweiterten Rechtecke repräsentieren. Nun sei $\{q_i\}$ die rechte *Zweifarbkontur* der Farben $p_{u1}.col$ und $p_{u2}.col$ (siehe Abbildung 5.2.1 und 5.2.2).
Es gilt:

- $q_{i+1} = q_i.NextContourPointOfSameColorBelow$
- $q_i.x < q_{i+1}.x$ und $q_i.y > q_{i+1}.y$
- $p_o \in \{q_i\}$ ist oberer Randpunkt eines erweiterten Rechtecks

Nun sei $r_{i,j}$ ein rechter Randpunkt eines erweiterten Rechtecks, das q_i als oberen Randpunkt hat (siehe Abbildung 5.3.4). Dann gilt:

- $r_{i,j}.x < q_{i+1}.x$, da sonst q_{i+1} im Inneren des erweiterten Rechtecks liegen würde, was aber nach obiger Konstruktion nicht möglich ist.
- $r_{i,j}.x > q_i.x$, da der linke Randpunkt immer links vom oberen Randpunkt liegt.
- Alle $r_{i,j}$ liegen unterhalb der *Zweifarbkontur* $\{q_i\}$.

Sei $R_{i,j}(p_l.x, q_i.y, r_{i,j}.x, p_{u1}.y)$ das erweiterte Rechteck mit festem unteren, rechten und oberen Randpunkt.
Der linke Randpunkt ist daher eindeutig bestimmt.
Beweis: Gäbe es hier zwei unterschiedliche Randpunkte $l_{i,j,1}$ und $l_{i,j,2}$, und wäre o.B.d.A. $l_{i,j,1}.x < l_{i,j,2}.x$, dann würde entweder das Rechteck mit $l_{i,j,2}$ als linkem Randpunkt nicht alle Farben enthalten, oder die Farbe $l_{i,j,1}.col$ würde auch im Inneren des Rechtecks vorkommen, das $l_{i,j,1}$ als linken Randpunkt hat. q.e.d.

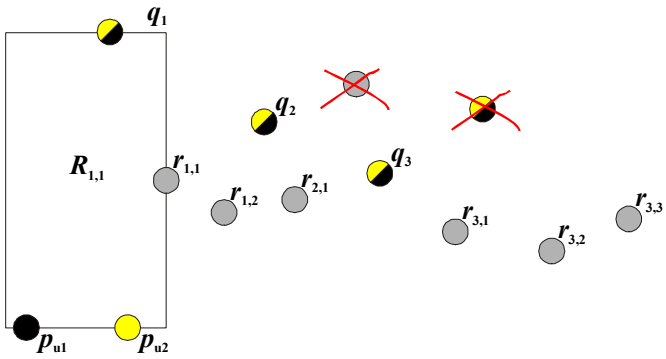


Abb. 5.3.4) Lage von möglichen oberen und rechten Randpunkten eines erweiterten Rechtecks vom Typ 1 bei fixierter Orientierung.

Insbesondere gilt: Bei fixierter Orientierung bildet jeder Punkt $r_{i,j}$ für höchstens ein erweitertes Rechteck einen rechten Randpunkt. Da oberhalb von p_{u1} und p_{u2} höchstens $n - k + 1$ Punkte als rechter Randpunkt in Frage kommen (links von $r_{i,j}$ müssen alle k Farben vorhanden sein), existieren für jede Orientierung $\alpha_{u1,u2}$ auch nur $O(n\text{-}k)$ viele erweiterte Rechtecke vom Typ 1. Weil es wegen *Lemma 5.3.1* nur $O(n^2\text{-}k^2)$ zu überprüfende Orientierungen gibt, existieren insgesamt $O((n^2\text{-}k^2)(n\text{-}k))$ viele erweiterte Rechtecke vom Typ 1.

Typ 2:
Im Gegensatz zu allen anderen Typen ist hier eine zu *Lemma 3.3.1* analoge Beweisführung nicht möglich. Ein beliebiger fixierter linker Randpunkt ließe noch nicht auf die dazugehörigen „oberen“ Randpunkte schließen, da hier noch nicht geklärt ist, wo „oben“ ist. Abbildung 5.3.5 verdeutlicht diese Orientierungslosigkeit, die selbst dann gegeben ist, wenn ein „oberer“ Punkt q_i vorhanden wäre.

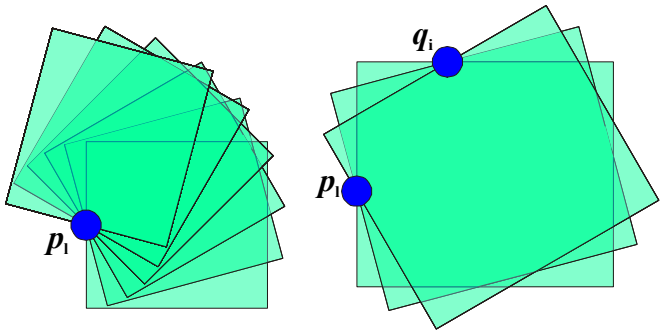


Abb. 5.3.5) Orientierungslosigkeit.

Es wird also auch hier von einer festen Y-Orientierung $\alpha_{u1,u2}$ ausgegangen.

Sei c eine festgelegte Farbe mit $p_{u1}.col \neq c \neq p_{u2}.col$ und sei $\{\boldsymbol{f}_{lci}\}$ die linke Farbkontur und $\{\boldsymbol{f}_{rci}\}$ die rechte Farbkontur inklusive einem möglichen mittig liegenden *Farbkonturabschlusspunkt* p_m (siehe Abbildung 5.2.3) der Farbe c.

Abbildung 5.3.6 zeigt die Lagen dieser Farbkonturpunkte. Weiterhin ist hier zu sehen, dass jeder Punkt aus $\{f_{lci}\} \cup \{f_{rci}\}$ höchstens einmal oberer Randpunkt sein kann. Daher existieren für eine Orientierung $\alpha_{u1,u2}$ bei festgelegter Farbe $O(\{f_{lci}\} \cup \{f_{rci}\})$ viele erweiterte Rechtecke vom Typ 2. Für alle Farben ergäbe das $O(n)$ viele Rechtecke, da jeder Punkt *oberhalb* von $\alpha_{u1,u2}$ gezählt wurde. Die k-3 Punkte, die direkt *oberhalb* von $\alpha_{u1,u2}$ liegen, können aber kein oberer Randpunkt sein, da in dem entsprechenden Streifen nicht alle Farben vorhanden sind.
Somit existieren auch nur $O(n\text{-}k)$ erweiterte Rechtecke vom Typ 2 für eine beliebige fixierte Orientierung.

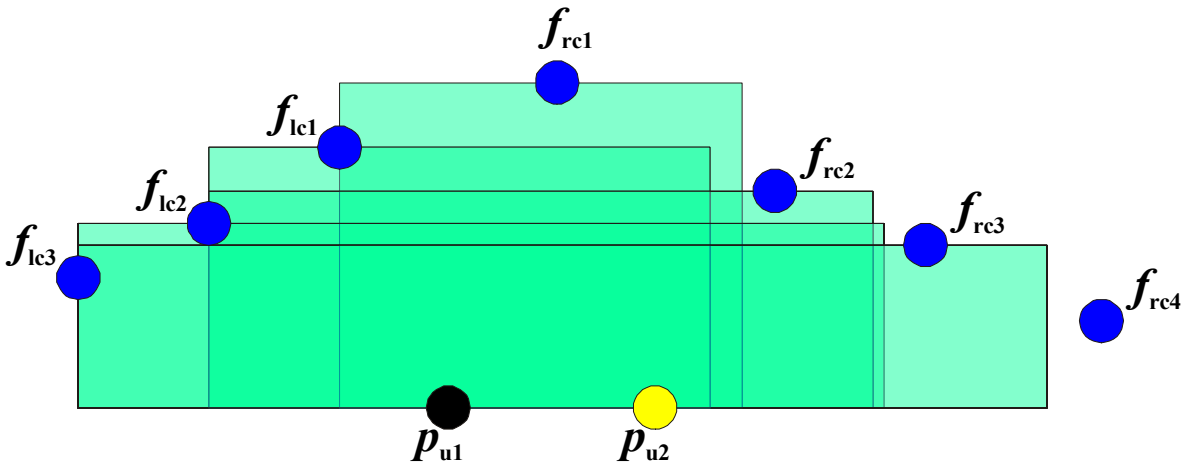


Abb. 5.3.6) Lage von möglichen erweiterten Rechtecken vom Typ 2 bei fixierter Orientierung und festgelegter Farbe.

Typ 3:
Fixiere eine beliebige Orientierung. Dann gibt es nur $n - k + 1$ mögliche linke Randpunkte p_l. Zu jedem festgelegten linken Randpunkt und festen unteren Randpunkten ist der rechte nun wieder eindeutig bestimmt.

Typ 4 und Typ 5:
Sowohl bei Typ 4 als auch bei Typ 5 ist allein durch die Fixierung der unteren Randpunkte gleichzeitig auch der linke Randpunkt bestimmt. Daher ist für gegebenes q_i auch sofort der rechte Randpunkt eindeutig bestimmt und es existieren pro Orientierung höchstens $|\{q_i\}|$ erweiterte Rechtecke, wobei hier die q_i ausgeschlossen werden, die nicht alle k Farben links von sich liegen haben. q.e.d. (Lemma 5.3.2)

5.4 Warum nur $n(n\text{-}1)$ Orientierungen?

Im obigen Algorithmus wurde davon ausgegangen, dass die nicht verkleinerbaren Rechtecke auf einer ihrer vier Kanten zwei Punkte liegen haben und somit die Orientierung festgelegt wird. Was wäre aber, wenn das kleinste Rechteck auf jeder Kante nur einen Punkt hat (siehe Abbildung 5.4.1).

Hier gilt:

$$\beta \notin \{\alpha_{i,j} \mid p_i, p_j \in P\}$$

Obiger Algorithmus könnte dieses Rechteck nicht finden.

Es bleibt also zu zeigen, dass diese Annahme nie zutreffen kann.

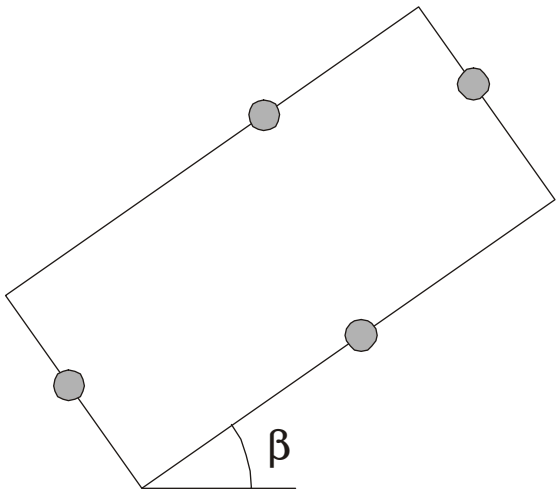


Abb. 5.4.1) Rechteck mit unbekannter Orientierung.

Lemma 5.4.1) Für die Suche nach kleinsten farbumspannenden Rechtecken beliebiger Orientierung müssen nur die Orientierungen $\alpha_{i,j}$ beachtet werden, die jeweils von einem Punktpaar $p_i, p_j \in P$, mit $p_i \neq p_j$, gebildet werden.

Beweis:
Nehme eine beliebige Punktmenge $Q \subseteq P$ mit $|Q| > 2$.

Bilde die konvexe Hülle C von Q.

Sei R ein Rechteck, dass um die so erhaltene Punktmenge C gespannt wird und seien k_1, k_2, k_3 und k_4 die zu dem Rechteck gehöhrenden Kanten.
Es gilt:

- $C \subset R$ // Das Rechteck umschließt die Hülle
- $k_1 \cup k_2 \cup k_3 \cup k_4 = \partial R$
- $k_i \cap \partial C \neq \varnothing$ // Jede Kante berührt die Hülle in mindestens einem Punkt

Rotiere nun R um 180° und betrachte dabei sowohl die Höhe, als auch die Breite des Rechtecks. In Abhängigkeit vom Rotationswinkel β erhält man so zwei Funktionen f_H für die Höhe und f_B für die Breite (siehe Abbildung 5.4.6). Es stellt sich nun die Frage, wo diese Funktionen ihre lokalen Minima haben. Betrachte hierfür die Abbildungen 5.4.2 und 5.4.3.

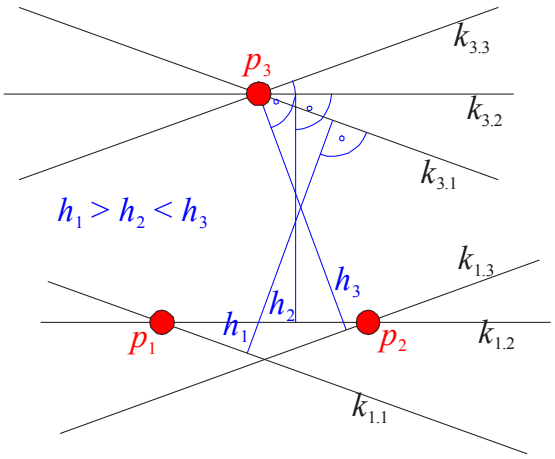


Abb. 5.4.2) Ein lokales Minimum für die Höhe entsteht, wenn das Liniensegment L(p_1, p_2) in k_1 liegt.

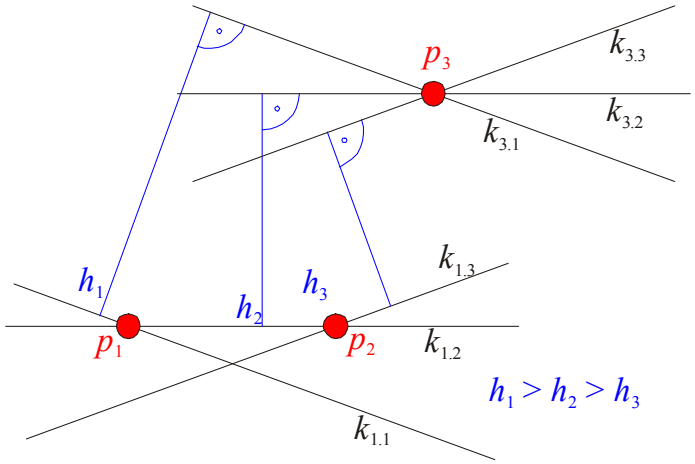


Abb. 5.4.3) Hier entsteht kein lokales Minimum. Die Funktion f_H ist aber auch hier an der Stelle nicht differenzierbar, wo das Liniensegment L(p_1, p_2) in k_1 liegt

Hier wird jeweils das Rechteck um die drei Punkte p_1, p_2 und p_3 rotiert. Die Abbildung erfolgt in 20°-Schritten und zeigt jeweils die obere Kante $k_{3,i}$ und die untere Kante $k_{1,i}$ und die dazugehörige Höhe h_i. Es ist klar, dass die Funktion f_H an der Stelle, wo das Liniensegment L(p_1, p_2) in k_1 liegt, nicht differenzierbar ist. Im linken Beispiel entsteht hier sogar ein lokales Minimum.
Im Folgenden wird nun bewiesen, dass nur an diesen Stellen ein Minimum entstehen kann. Hierfür wird gezeigt, dass die Funktion f_H zwischen diesen Stellen rechtsgekrümmt ist. Es sei bemerkt, dass die Kanten k_1 und k_3 dann jeweils nur einen Punkt der konvexen Hülle berühren. Daher genügt es, das Folgende zu zeigen:

Betrachte die Punkte $p_1 = (0, 0)$ und $p_2 = (0, 2)$ und lege durch p_2 eine Gerade g. Rotiere diese Gerade (mit Rotationswinkel β) in p_2 und messe den Abstand $d(\beta)$ zwischen p_1 und g.

Wegen des Satzes des Thales muss für den Abstand der Betrag des Vektors

$$(0, 1)^t + (\cos 2\beta, \sin 2\beta)^t$$

berechnet werden (siehe Abbildung 5.4.4 und 5.4.5).

Also $d(\beta) = \left\| \begin{pmatrix} 0 \\ 1 \end{pmatrix} + \begin{pmatrix} \cos 2\beta \\ \sin 2\beta \end{pmatrix} \right\| = \sqrt{\sin^2 2\beta + \cos^2 2\beta + 2\sin 2\beta + 1} = \sqrt{2 + 2\sin 2\beta}$.

Damit diese Funktion rechtsgekrümmt ist, muß die zweite Ableitung negativ sein:

$$d'(\beta) = \frac{\cos(2\beta)}{\sqrt{2 + 2\sin(2\beta)}}$$

$$d''(\beta) = \frac{-\sin(2\beta) - \dfrac{\cos^2(2\beta)}{2 + 2\sin(2\beta)}}{\sqrt{2 + 2\sin(2\beta)}}$$

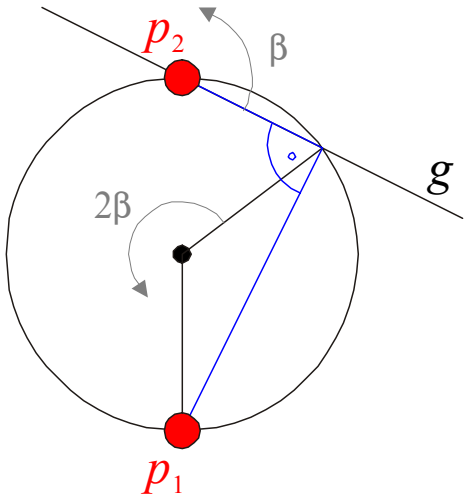


Abb. 5.4.4) Abstandsmessung zwischen p_1 und g.

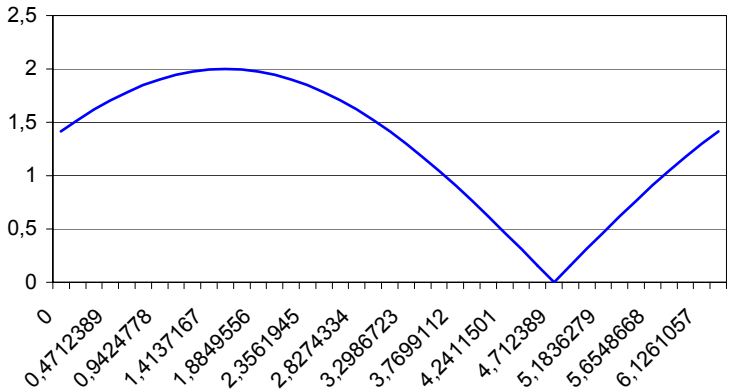


Abb. 5.4.5) Betrag des Vektors $(0, 1)^t + (\cos 2\beta, \sin 2\beta)^t$.

Es bleibt zu zeigen:

$$-\sin(2\beta) - \frac{\cos^2(2\beta)}{2 + 2\sin(2\beta)} < 0 \Leftrightarrow$$

$$-\frac{\cos^2(2\beta)}{2 + 2\sin(2\beta)} < \sin(2\beta) \Leftrightarrow \mid \sin(2\beta) \neq -1$$

$$-\cos^2(2\beta) < 2\sin^2(2\beta) + 2\sin(2\beta) \Leftrightarrow$$

$$0 < \sin^2(2\beta) + \cos^2(2\beta) + \sin^2(2\beta) + 2\sin(2\beta) \Leftrightarrow$$

$$0 < 1 + \sin^2(2\beta) + 2\sin(2\beta) = (\sin(2\beta) + 1)^2$$

Somit ist für $\sin(2\beta) \neq -1$ die obige Funktion rechtsgekrümmt. Daher können in f_H nur an den nicht differenzierbaren Stellen Minima auftreten. Analoges gilt für f_B.

Zur Berechnung des Umfangs solch eines Rechtecks werden nun die Funktionen addiert:

$$f_U = f_H + f_B.$$

Auch hier ist die zweite Ableitung negativ. Somit erreicht f_U nur dann lokale Minima, wenn gilt:

$$\exists\ 1 \leq i \leq 4: |k_i \cap \partial C| = \infty$$

(Mindestens eine Kante des konvexen Polygons liegt auf dem Rand des Rechtecks).

Eine analoge Argumentation gilt für die Berechnung der Fläche. q.e.d. (Lemma 5.4.1)

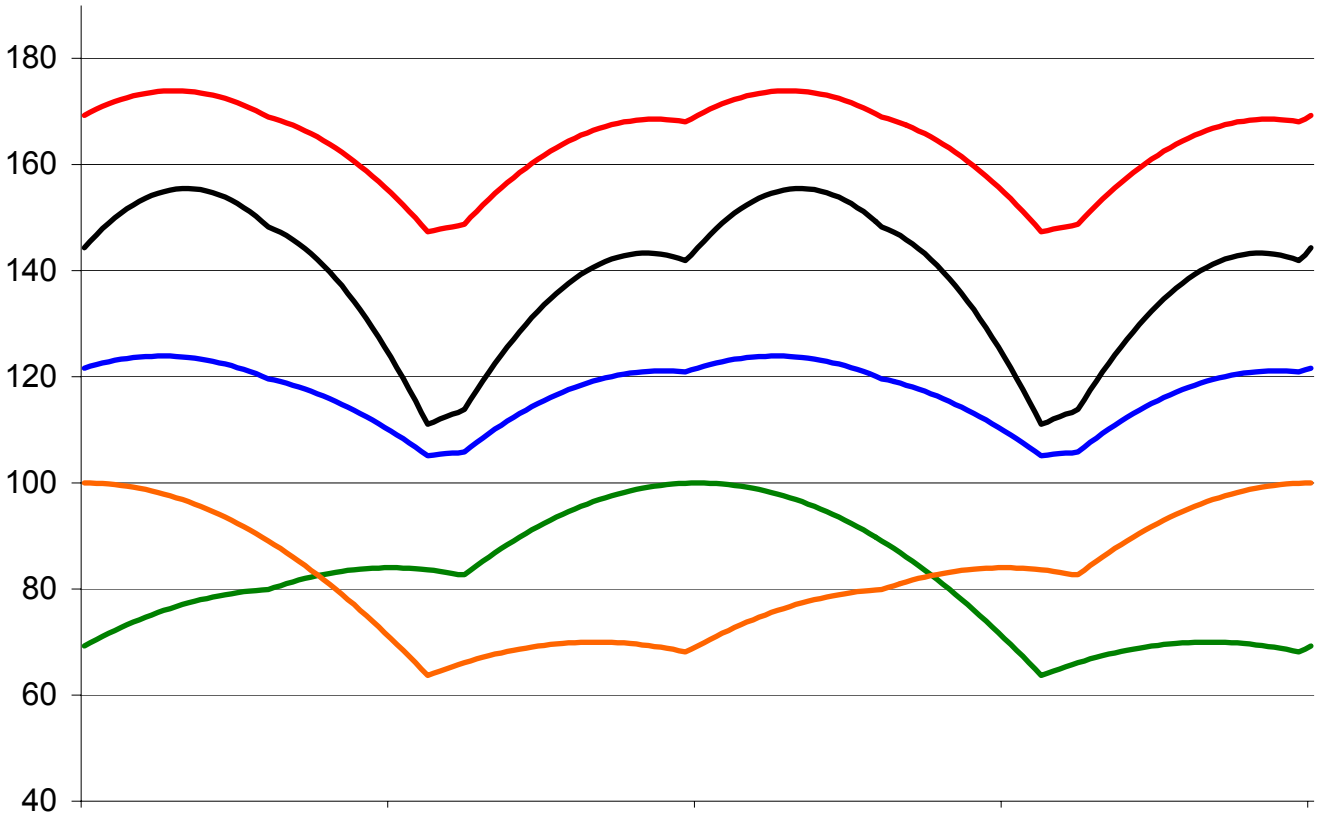


Abb. 5.4.6) Maße eines rotierten Rechtecks

Abbildung 5.4.6 zeigt die Funktionen:

- Höhe: f_H (orange)
- Breite: f_B (grün)
- Umfang: $f_U = f_B + f_H$ (rot)
- Fläche: $f_A = f_B \cdot f_H$ (schwarz) // um den Faktor 0,02 skaliert
- Diagonale: $f_D = \mathrm{Sqrt}(f_B^2 + f_H^2)$ (blau)

6 Das Java-Applet

Das Applet für die Suche nach kleinsten farbumspannenden Rechtecken beliebiger oder fester Orientierung wurde mit *NetBeans IDE 3.6* geschrieben und mit *Java 2 SDK, Standard Edition, v. 1.4.2* kompiliert. Die entsprechenden Setups findet man als Bundle unter
http://java.sun.com/j2se/1.4.2/download-netbeans.html.

Das Applet, die dazugehörigen Quellcodes und diese Ausarbeitung als PDF-File findet man unter
http://www.rechtecke.de.vu/.

Da der Quellcode des Applets ca. 3100 Zeilen lang ist, werden in den folgenden beiden Abschnitten nur die Klassen-Struktur, die Aufgabe der einzelnen Klassen und die wichtigsten Methoden erläutert.

6.1 Die Klassen

`public class frmMain extends … JFrame`

Beinhaltet außer den Methoden für die Verwaltung von *MaxElem* alles Relevante für die Berechnung, Ausgabe und Visualisierung der gesuchten Rechtecke.
Innerhalb von `frmMain` werden auch noch folgende Klassen definiert:

`class MyZoomFrame extends … JFrame …`

Ermöglicht eine detailliertere Ansicht der Punktmenge. Sehr nützlich bei vielen Punkten ($n > 10\,000$) oder beliebiger Orientierung und dementsprechend kleinen gefundenen Rechtecken. Innerhalb von `MyZoomFrame` wird noch folgende Klasse verwendet:

`class MyZoomPanel extends javax.swing.Jpanel`

Zeichnet den aktuellen Zoomausschnitt.

`class MyPanel extends … JPanel`

Regelt die Ausgabe der Punkte und Visualisierung der Berechnung.

`class MyMaxElemPanel extends … JPanel`

Regelt die Ausgabe der maximalen Elemente während der Visualisierung.

`class MyColor extends java.awt.Color`

Hat noch die zusätzlichen Eigenschaften `Name` und `OriginalIndex`. Wird für die Farbverwaltung des Punkteditors verwendet.

`class trdSearchCls extends Thread`

Die beiden Thread-Klassen werden für die Visualisierung benötigt und ermöglichen es, ein zeitnahes Neuzeichnen der Ausgabepanels `MyPanel` und `MyMaxElemPanel` zu erzwingen, ohne die Berechnungen zwischendurch komplett abzubrechen. Die – ebenfalls für eine zwischenzeitliche Ausgabe erstellte und bei Sun erhältliche – Klasse `SwingWorker` benötigt zwar keinen zwischenzeitlichen Abbruch der Berechnung, lässt aber auch keine zeitlich genaue Steuerung der Ausgabe zu.

`class trdPaintCls extends Thread`

Dieser Thread wartet, bis `trdSearchCls` pausiert, um dann die Ausgabepanels neu zu zeichnen. Danach meldet dieser Thread dem pausierenden Thread

trdSearchCls, dass die Neuzeichnung beendet ist. Das Zusammenspiel dieser beiden Threads wird auch von der Methode *frmMain.**MyRepaint*** geregelt.

class *MyXKoordComparator* implements Comparator und
class *MyDescYKoordComparator* implements Comparator
Die beiden Comparator-Klassen werden für die Erstellung des nach X-Koordinaten verketteten und nach Y-Koordinaten sortierten Punkt-Arrays ***PTS***[] benötigt.

class *MyOrientComparator* implements Comparator
Wird für die sortierten Mengen benötigt, die die Aufgabe der AVL-Bäume $\mathfrak{X}$ und $\mathfrak{Y}$ (siehe Abschnitt 5.2) übernehmen.

class *MyOrient*
Wird von Methoden für die Übergebung von Orientierungen – also Winkel, p1 und p2 – verwendet.

(Ende frmMain)

public class *MyPoint*
Wird zur Speicherung der Punkte und deren Verkettung bzgl. der X-Position – durch die Eigenschaften *Next* und *Previous* – und der Stapelung der Farbkonturpunkte – durch die Eigenschaften *NextContourPointBelow* und *NextContourPointOfSameColorBelow* – verwendet. Die Eigenschaft *bolFound* wird für die Initialisierung der *MaxElem*-Liste verwendet.
Die Klasse wird auch als Blatt in den AVL-Bäumen zur Speicherung der maximalen Elemente verwendet.

public class *MyRect*
Enthält Eigenschaften für Größe und Lage von Rechtecken und ermöglicht durch die Eigenschaft *OlderRect* eine Stapelung dieser Rechtecke.

public class *frmMyPointEditor* extends … JFrame
Ein Editor zur manuellen Eingabe der Punkte.

public class *MyMaxElem*
Verwaltet die Liste der maximalen Elemente. Hierfür werden zwei balancierte Binärbäume erstellt, die eine optimale Suche sowohl nach X- als auch nach Y-Koordinate sicherstellen. Innerhalb von *MyMaxElem* werden hierfür folgende Klassen bereitgestellt:

class *AVLNode*
Innere Knoten der beiden AVL-Bäume für die sortierte Speicherung der X- und Y-Koordinaten.

class *MyStack*
Speichert die X- bzw. Y-Koordinaten der zu löschenden/dominierten maximalen Elemente nach Einfügung eines neuen, dominanten/maximalen Elements.

public class *Start* extends JApplet
Zum Einfügen in eine HTML-Seite. Enthält einen Button zum Starten des Applets.

Weiterhin werden die von NetBeans bereitgestellten Klassen *org.netbeans.lib.awtextra.AbsoluteLayout* und *org.netbeans.lib.awtextra.AbsoluteConstraints* zur absoluten Positionierung der Steuerelemente in *frmMain* verwendet.

6.2 Relevante Methoden

In *frmMain*:

private boolean *CreateContours*()
Erstellt für jeden unteren Randpunkt p_u die entsprechenden Farbkonturen genau so, wie es auch in Abschnitt 4.2 beschrieben worden ist.

private boolean CreateContours2()
Erstellt, wie im Abschnitt 5.2 beschrieben, die Farbkonturen für zwei untere Randpunkte.

private void *InitMaxElemInLinearTime*()
Initialisiert die Liste *MaxElem* wie in Abschnitt 4.6 besprochen.

private void *InitMaxElemInLinearTime2*()
Initialisiert die Liste *MaxElem* für zwei untere Randpunkte.

private int *GetStartIndex*()
Liefert den Index des ersten Punktes p_i des nach Y-Koordinate absteigend sortierten Arrays *PTS*, über dem alle k-1 Farben $\{c \mid c \neq p_i.col\}$ vorkommen und somit p_i der erste untere Randpunkt eines nicht verkleinerbaren Rechtecks sein kann.

private void *FindSmallestColorSpanningRectangle*()
Sucht die achsenparallelen Rechtecke, wie in Abschnitt 4.4 und 4.7 angegeben.

private void *Rotate*(double Alpha, MyPoint[] A)
Rotiert alle Punkte des Arrays `A` um den Winkel `Alpha` gegen den Uhrzeigersinn.

private MyOrient GetXOrient(MyPoint p1, MyPoint p2)
Berechnet die X-Orientierung zwischen den Punkten `p1` und `p2`.

private MyOrient GetYOrient(MyPoint p1, MyPoint p2)
Berechnet die Y-Orientierung zwischen den Punkten `p1` und `p2`.

private void FindSmallestColorSpanningFreeOrientedRectangle()
Sucht, wie in Abschnitt 5.2 vorgestellt, die Rechtecke beliebiger Orientierung.

public void SaveOrigPoints()
Wird benötigt, um die Original-Koordinaten der Punktmenge zu sichern, da nach der Berechnung der kleinsten Rechtecke beliebiger Orientierung jeder Punkt ca. n^2 mal rotiert wurde und sich somit erhebliche Rundungsfehler eingeschlichen haben. Eine erneute Suche mit der scheinbar gleichen Punktmenge kann dann zu unterschiedlichen Ergebnissen führen.

public void RestoreOrigPoints()
Wird nach der Berechnung der kleinsten Rechtecke beliebiger Orientierung aufgerufen.

`public void NewPointSet()`
Erstellt eine neue Punktmenge und sorgt gleichzeitig für die in Abschnitt 2.1 angegebenen Voraussetzungen.

In *MyMaxElem*:

`private boolean Dominates(MyPoint Dominator, MyPoint Dominated)`
Liefert `true`, wenn der Punkt *Dominator* den Punkt *Dominated* bzgl. dem Ankerpunkt (0, 1) dominiert.

`… void NoticeDominatedElementsForDeletionInStack(MyPoint Lf)`
Merkt sich für eine spätere Löschung alle Elemente in einem Stapel, die von einem neu eingefügten Element *Lf* dominiert werden.

`public boolean Insert(double x, double y, int col)`
Fügt das Element mit den Koordinaten (*x*, *y*) und der Farbe *col* in die nach X- bzw. Y-Koordinate verketteten Listen ein, und löscht neu dominierte Punkte.

`private MyPoint GetNextElem(AVLNode CurNd, double SpVal)`
Liefert den ersten Punkt, der rechts von *ExclL* oder oberhalb von *InclR* liegt.

`private boolean IsValidPair(MyPoint Left, MyPoint Right)`
Überprüft, ob der Punkt *Left* in dem vertikalen Bereich zwischen *ExclL* und *InclL* liegt, und ob der Punkt *Right* in dem horizontalen Bereich zwischen *InclR* und *ExclR* liegt.

`public boolean Recalc(boolean bolConsoleOutput)`
Sucht neue nicht verkleinerbare Rechtecke und liefert `true`, falls solch eins gefunden wurde.

`public void InitWithSortedArray(MyPoint[] W, int EndIndex)`
Erstellt die balancierten Binärbäume aus den nach X- und Y-Koordinate sortierten Punkten des Arrays *W*, wie in Abschnitt 4.6 und 5.2 beschrieben.

`public MyPoint getFirstLeaf()`
Liefert für die Visualisierung der Elemente das erste Element der verketteten Liste aller Elemente.

6.3 Bedienhinweise

In diesem Abschnitt wird die Bedienung und insbesondere die Funktion der Steuerelemente des Applets beschrieben. Abbildung 6.3.1 zeigt die Anwendung während einer visualisierten Berechnung von Rechtecken beliebiger Orientierung mit 300 Punkten und 10 Farben. Man sieht hier rechts die Punktmenge, die z.Zt. untere und obere Grenze, die beiden bisher kleinsten gefundenen Rechtecke (rot = kleinster Umfang, schwarz = kleinste Fläche), die Farbkonturen und das z.Zt. betrachtete nicht verkleinerbare Rechteck (grün ausgefüllt). Unten links werden die in *MaxElem* enthaltenen Elemente und die gültigen schraffierten Bereiche zwischen *ExclL* und *InclL*, sowie zwischen *InclR* und *ExclR* angezeigt.

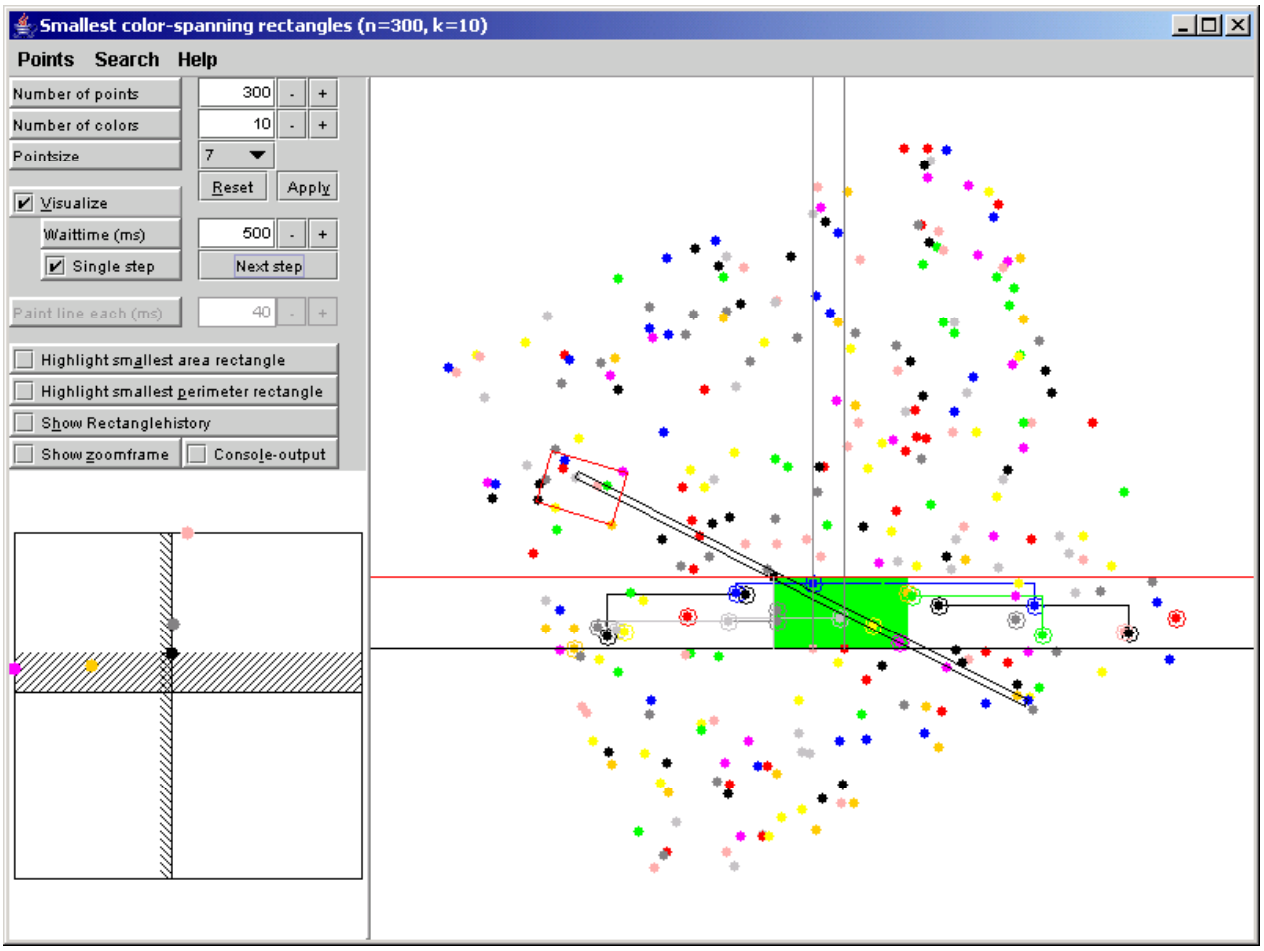


Abb. 6.3.1) Das Applet bei der Berechung der kleinsten Rechtecke beliebiger Orientierung für 300 Punkte und 10 Farben.

Oben links können neben der Anzahl der Farben und der Anzahl der Punkte noch weitere Optionen angeben werden, die jetzt näher erläutert werden.

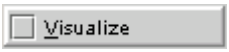
Die Steuerelemente:

Anzahl der Punkte. Mit den Buttons „+“ und „-“ kann die Zahl – wie auch bei den anderen drei Zahleingabefeldern – in logarithmischer Größenordnung verändert werden. Die maximale Punktanzahl wird auf 100 000 beschränkt.

Anzahl der Farben. Diese kann nicht größer werden, als die Anzahl der Punkte und nicht kleiner als 2.

Die Punkte können einen Durchmesser von einem bis neun Pixel haben. Bei mehr als 10 000 Punkten wird die Punktgröße automatisch auf eins gesetzt.

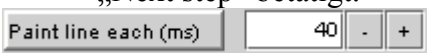
Die oberen drei angegebenen Werte werden erst durch Betätigung von „Apply“ übernommen. Während einer Berechnung können diese Werte nicht verändert werden. „Reset“ setzt die noch nicht übernommenen Werte wieder zurück.



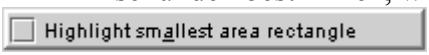
Schaltet die Visualisierung ein. Während solch einer Visualisierung hat man noch folgende Arten der Visualisierung zur Auswahl:

Zwischen jedem Schritt wird eine entsprechende Anzahl an Millisekunden gewartet, bis automatisch weitergerechnet wird.

Man kann sich auch jeden Schritt so lange anzeigen lassen, bis man den Button „Next step“ betätigt.



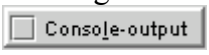
Wenn man sich die Berechnung nicht visualisieren lässt, dann kann man hier in Millisekunden bestimmen, wie oft das Ausgabepanel neu gezeichnet wird.



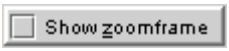
Markiert das Rechteck mit der kleinsten Fläche durch zwei dieses Rechteck kreuzende Balken. Diese und auch die folgende Option ist hilfreich, wenn man sehr viele Punkte und wenig Farben hat, weil dann die gefundenen Rechtecke u.U. so klein sind, dass sie auf den ersten Blick nicht zu sehen sind.

Markiert das Rechteck mit dem kleinsten Umfang.

Zeigt alle temporär kleinsten Rechtecke an.



Gibt detailliertere Informationen der Berechnung über die Konsole aus. Diese Ausgabe ist aber sehr zeitintensiv und sollte daher nicht permanent eingeschaltet sein. Nützlich ist aber eine Kombination von dieser Option mit der Einzellschrittvisualisierung.



Wem die ☑Highlight – Optionen nicht reichen, kann sich ein Zoomfenster anzeigen lassen. Die Bedienung wird vom dem Zoomfenster durch eine entsprechende Textausgabe selbst erklärt.

Das Menü:

Points

New pointset N

Erstellt eine neue Punktmenge, färbt die Punkte ein und sortiert und verkettet diese, wie in Abschnitt 2.1 angegeben.

Edit new pointset E

Startet den Punkteditor zur manuellen Eingabe der Punkte. Falls die aktuelle Punktmenge automatisch generiert wurde, kann man diese nicht mit dem Editor bearbeiten, sondern nur eine neue Punktmenge eingeben. Wenn aber die aktuelle Punktmenge bereits manuell eingegeben wurde, dann kann diese auch wieder durch den Editor verändert werden und der Menü-Text ändert sich zu **Edit current pointset** E.

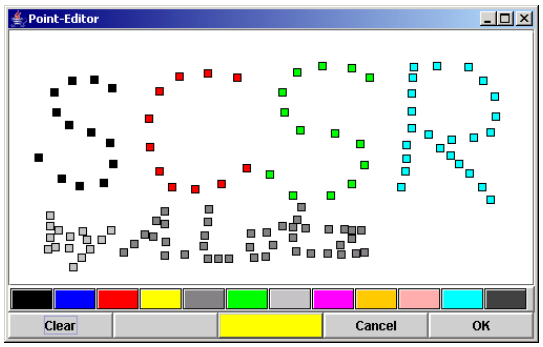


Abb. 6.3.2) Der Punkteditor.

Search

Smallest rectangle of fixed orientation F

Startet die Suche nach dem kleinsten achsenparallelen Rechteck.

Smallest free-oriented rectangle O

Startet die Suche nach dem kleinsten Rechteck beliebiger Orientierung.

Stop searching X

Beendet die Suche vorzeitig.

Help

About F1

Kurze Info über das Applet.

7 Zusammenfassung und Ausblick

7.1 Zusammenfassung

Diese Diplomarbeit beschäftigt sich zum Einen mit der Suche nach kleinsten achsenparallelen farbumspannenden Rechtecken. Aus einer früheren Arbeit [1] ist bekannt, das es bei n Punkten und k Farben insgesamt nicht mehr als $\Theta((n\text{-}k)^2)$ Kandidaten für die gesuchten Rechtecke gibt. Ein optimaler Algorithmus benötigt demnach auch mindestens $\Theta((n\text{-}k)^2)$ viel Zeit. Die Frage ist nun, wie effizient man diese Kandidaten auflisten kann.
In der oben erwähnten Arbeit wurde auch der – hier der Vollständigkeit halber nochmals vorgestellte – Algorithmus beschrieben, der $O((n\text{-}k)\ n \log^2 k)$ Zeit benötigt und somit zeitlich höchstens um einen Faktor

$$c_{alt} = \frac{n \log^2 k}{n-k}$$

von einer optimalen Lösung abweicht. Der Faktor $\log^2 k$ entstand hierbei durch die Notwendigkeit, sämtliche k Elemente zu verwalten. Insbesondere mussten solche Elemente verwaltet werden, die nicht maximal sind, aber – wegen der Festlegungsreihenfolge des oberen Grenzpunktes p_o von unten nach oben – trotzdem maximal werden können, obwohl sie selber nicht verschoben wurden.
In dieser Arbeit wird gezeigt, dass dieser Algorithmus noch nicht optimal war. Hierfür wird ein neuer Algorithmus angegeben, der die Festlegungsreihenfolge des oberen Grenzpunktes umkehrt. Weiterhin werden vor der Betrachtung sämtlicher oberer Grenzpunkte p_o für einen festen unteren Grenzpunkt p_u noch Farbkonturen erstellt, und mit deren Hilfe die ersten $O(k)$ Elemente in $O(n)$ – also linearer Zeit – in die Liste *MaxElem* eingefügt.
Hierdurch erhält man – in Abhängigkeit von n und k – entweder die Kosten $O((n\text{-}k)^2 \log k)$ und somit einen maximalen Faktor von

$$c_{neu\_1} = \log k\ ,$$

bzw. die Kosten $O((n\text{-}k)n)$ und den Faktor

$$c_{neu\_2} = \frac{n}{n-k}\ .$$

Zum anderen beschäftigt sich diese Arbeit mit der Suche nach kleinsten farbumspannenden Rechtecken beliebiger Orientierung. Hierfür wurde ein leicht geänderter Teil des obigen Algorithmus verwendet. Ausschlaggebend ist aber auch hier, dass wieder Farbkonturen erstellt werden. Die somit erzielten Kosten liegen in

$$O\left(\left(n^2 - k^2\right)\left(n + (n-k)\log k\right)\right)$$

und somit für den eher üblichen Fall mit $n > 2k$ in

$$O\left((n^2 - k^2)(n-k)\log k\right).$$

7.2 Ausblick

Wie in Abschnitt 3.3 bereits angesprochen, liegt die Anzahl nicht verkleinerbarer farbumspannender und achsenparalleler Rechtecke in $\Theta((n\text{-}k)^2)$. Es bleibt also die Frage, ob der in Kapitel 4 beschriebene Algorithmus optimal ist, oder ob der Faktor log k auch noch getilgt werden kann.

Auch der in Kapitel 5 beschriebene Algorithmus für nicht verkleinerbare farbumspannende Rechtecke beliebiger Orientierung lässt noch Fragen offen.
Zum einen ist noch nicht klar, wie viele nicht verkleinerbare Rechtecke beliebiger Orientierung es für gegebenes n und k mindestens gibt. Existieren also tatsächlich $\Theta((n^2\text{-}k^2)(n\text{-}k))$ viele, oder ist die Anzahl in Abhängigkeit von n eher quadratisch und nicht kubisch? Hierzu würde es genügen, ein Beispiel zu finden, das $\Omega((n^2\text{-}k^2)(n\text{-}k))$ viele erweiterte Rechtecke eines der in Abschnitt 5.3 vorgestellten Typen beinhaltet.
Zum anderen würde sich bei $\Theta((n^2\text{-}k^2)(n\text{-}k))$ vielen Rechtecken auch hier die Frage stellen, ob der zusätzliche Faktor log k immer notwendig und demnach der Algorithmus optimal ist.

Symbolverzeichnis

$[a, b]$	Geschlossenes Intervall	2-6
$]a, b[$	Offenes Intervall	2-3
(x, y)	Koordinaten eines Punktes/Elements	2-4
$a_{i,j}$	Anzahl der nicht verkleinerbaren Rechtecke mit dem unteren Randpunkt $p_u = p_i$ und dem oberen Randpunkt $p_o = p_j$	3-12
$\alpha_{[i],[i+1]}$	Y-Orientierung zwischen den Punkten $p_{[i]}$ und $p_{[i+1]}$ des Arrays $p_{[1,n]}$	5-28
$\alpha_{i,j}$	Y-Orientierung zwischen den Punkten p_i und p_j	5-27
β	Eine Orientierung, die nicht durch zwei Punkte p_i und p_j gebildet wird	5-37
c	Eine Farbe	2-5
C	Konvexe Hülle einer Punktmenge	5-38
CCP	Siehe *CurContourPoints*[1, 2][1, k]	4-17
c_i	Eine der k Farben mit $1 \leq i \leq k$	2-3
col.index	Eindeutige Nummer der Farbe *col*	4-17
CurContourPoints[1, 2][1, k]:	Enthält die obersten Farbkonturpunkte aller Farben auf der linken und rechten Seite von p_u	4-17
CurUpperBound	Das Gleiche wie p_o	4-17
e(c)	Element der Farbe c und den Koordinaten $e(c).x = L(c, p_u, p_o)$ und $e(c).y = R(c, p_u, p_o)$)	2-6
E	Die Menge aller Elemente $e(c_i)$	2-6
e.Next	Das nächste von e aus rechts und oberhalb liegende maximale Element	3-9
e.Previous	Das nächste von e aus links und unterhalb liegende maximale Element	3-9
$e_d(c_i)$	Ein nicht maximales Element der Farbe c_i	4-20
$e_D(c_j)$	Ein maximales Element der Farbe c_j	4-20
e_l	Element der Farbe $p_u.col$, dass in E einen linken unteren Randpunkt repräsentiert ($p_u = p_l$)	2-8
$e_L(c_L)$	Ein Element der Farbe c_L, dass einen linken Randpunkt eines nicht verkleinerbaren Rechtecks repräsentiert	3-9
e_r	Element der Farbe $p_u.col$, dass in E einen rechten unteren Randpunkt repräsentiert ($p_u = p_r$)	2-8
$e_R(c_R)$	Ein Element der Farbe c_R, dass einen rechten Randpunkt eines nicht verkleinerbaren Rechtecks repräsentiert	3-9
ExclL	Maximum von $L(p_o.col, p_u, p_o)$ und $L(p_u.col, p_u, p_o)$	2-5
ExclR	Minimum von $R(p_o.col, p_u, p_o)$ und $R(p_u.col, p_u, p_o)$	2-5
f	Eine Farbe	2-7
farbname	Ein Punkt oder Element der Farbe „*farbname*“	2-6
f_B	Verlauf der Breite eines rotierenden Rechtecks	5-38
f_H	Verlauf der Höhe eines rotierenden Rechtecks	5-38
f_{lci}	Linker Farbkonturpunkt bei fester Orientierung	5-36
f_P	Bijektive Abbildung zwischen erweiterten und nicht verkleinerbaren Rechtecken.	3-13
f_{rci}	Rechter Farbkonturpunkt bei fester Orientierung	5-36
f_U	Verlauf des Umfangs eines rotierenden Rechtecks	5-40
g_i	Waagerechte und durch $p_{[i]}$ verlaufende Gerade	5-33
h_i	Höhe eines Rechtecks	5-38

H_i	Geschlossene Halbebene, mit waagerechter Begrenzungsgerade g_i, die durch $p_{[i]}$ verläuft.	5-33
InclL	Minimum von $p_u.x$ und $p_o.x$	2-5
InclR	Maximum von $p_u.x$ und $p_o.x$	2-5
k	Anzahl der Farben	1-2
k_i	Eine der vier Kanten eines Rechtecks	5-38
$K_{i,j}$	Bipartiter Graph mit $i + j = n$	2-3
$L(c, p_u, p_o)$	Maximale X-Koordinate aller Punkte, die rechts und oberhalb von p_u und unterhalb von p_o liegen und die Farbe c haben.	2-5
$l_{i,j}$	Potenzieller linker Randpunkt eines erweiterten Rechtecks	3-13
$L(p_1, p_2)$	Liniensegment zwischen den Punkten p_1 und p_2	5-38
MaxElem	Struktur bzw. Liste zur Verwaltung aller maximalen Elemente	2-3
n	Anzahl der Punkte	1-2
null	Leerer Zeiger	4-17
$p_{[1,n]}$	Nach X-Koordinate verkettetes und nach Y-Koordinate sortiertes Array von n Punkten	2-3
$p_{[index]}$	*Index*'ter Punkt im Punkt-Array $p_{[1,n]}$. Es gilt: $p_{[p.Index]} = p$	2-3
$p_{[i],m}$	Punkt, der mit $p_{[i]}$ eine nicht zu überprüfende Orientierung bildet	5-33
P	Die Menge aller n Punkte	2-3
p.col	Farbe des Punktes p	2-3
p.index	Index des Punktes p im Punkt-Array $p_{[1,n]}$	2-3
p.Next	Der nächste von p aus rechts liegende Punkt	4-23
p.NextContourPointBelow: Zeigt auf den nächsten Farbkonturpunkt unterhalb von p		4-17
p.NextContourPointOfSameColorBelow: Zeigt auf den nächsten Farbkonturpunkt, der unterhalb von p und bzgl. p_u auf der gleichen Seite wie p liegt und die gleiche Farbe wie p hat		4-17
p.Previous	Der nächste von p aus links liegende Punkt	4-23
p.x	X-Koordinate des Punktes p	2-3
p.y	Y-Koordinate des Punktes p	2-3
p_A	Ankerpunkt bzgl. einer Dominanz	2-4
p_d	Dominierter Punkt	2-4
p_D	Dominierender Punkt	2-4
p_l	Linker Randpunkt eines nicht verkleinerbaren Rechtecks	2-3
P_L	Alle Punkte, die links oberhalb von p_u liegen	4-17
$P_L(c)$	Alle Punkte, die links oberhalb von p_u liegen und die Farbe c haben	4-17
p_m	Farbkonturpunkt, der zwischen p_{u1} und p_{u2} liegt	5-29
p_o	Potenzieller oberer Randpunkt eines nicht verkleinerbaren Rechtecks	2-3
p_r	Rechter Randpunkt eines nicht verkleinerbaren Rechtecks	2-3
p_u	Potenzieller unterer Randpunkt eines nicht verkleinerbaren Rechtecks	2-3
p_{u1}	Potenzieller linker unterer Randpunkt eines nicht verkleinerbaren Rechtecks beliebiger Orientierung	5-27
p_{u2}	Potenzieller rechter unterer Randpunkt eines nicht verkleinerbaren Rechtecks beliebiger Orientierung	5-27
$Q[1, k]$	Ein nach Farbindex sortiertes Punkt-Array mit der jeweils zusätzlichen Punkt-Eigenschaft *.bolfound*	4-23
q_i	Oberer Randpunkt eines erweiterten Rechtecks	3-13
$\{q_i\}$	Rechte Farbkontur der Farbe $p_u.col$	3-13

$R(c, p_u, p_o)$	Minimale X-Koordinate aller Punkte, die links und oberhalb von p_u und unterhalb von p_o liegen und die Farbe c haben.	2-5
$R_{ect}(x1, y1, x2, y2)$	Achsenparalleles Rechteck mit den entsprechenden Koordinaten	2-3
$R_{ex}(x1, y1, x2, y2)$	Erweitertes Rechteck	3-12
$r_{i,j}$	Potenzieller rechter Randpunkt eines erweiterten Rechtecks	3-13
$R_{i,j}$	Erweitertes Rechteck $R_{ex}(l_{i,j}.x, q_i.y, r_{i,j}.x, p_u.y)$	3-13
s_{il}	Waagerechter von $p_{[i]}$ ausgehender und nach links laufender Strahl	5-33
s_{ir}	Waagerechter von $p_{[i]}$ ausgehender und nach rechts laufender Strahl	5-33
S_u	Eine durch den unteren Randpunkt p_u gezeichnete senkrechte Gerade	2-5
$W[1, k + 1]$	Ein nach X- und Y- Koordinate aufsteigend sortiertes Punkt-Array mit k+1 Punkten	4-23
$\mathfrak{X}$	AVL-Baum zur Verwaltung aller n-1 relevanten X-Orientierungen	5-28
$\xi_{i,j}$	X-Orientierung zwischen den Punkten p_i und p_j	5-27
$\mathfrak{Y}$	AVL-Baum zur Verwaltung aller n-1 relevanten Y-Orientierungen	5-28

Abbildungsverzeichnis

1.1.1	Wohnungssuche	1-1
1.1.2	Haus mit erwünschten Lagen des Fahrstuhlschachtes	1-2
2.2.1	Dominanz	2-4
2.3.1	Einschränkung der Lage linker und rechter Randpunkte	2-5
2.4.1	Die Schranken *ExclL*, *InclL*, *InclR* und *ExclR*	2-6
2.4.2	Lage maximaler Elemente	2-6
3.3.1	m-contour von OVERMARS und LEEUWEN aus [3]	3-11
3.3.2	Erweiterte Rechtecke (Typ 1 bis Typ 3)	3-12
3.3.3	Lage von oberen und rechten Randpunkten eines erweiterten Rechtecks	3-13
3.3.4	Erweiterte Rechtecke mit $p_u = p_l$	3-14
3.3.5	Beispiel für $\Omega((n-k)^2)$ nicht verkleinerbare achsenparallele Rechtecke	3-14
4.1.1	Beispiel für Punktkonstellation und Lage maximaler Elemente	4-16
4.1.2	Verhalten maximaler Elemente bei Wanderung von unten nach oben	4-16
4.2.1	Farbkontur	4-18
4.2.2	Farbkontur unter Berücksichtigung der Farbe von p_u	4-18
4.2.3	Neue Definition von L(*Farbe*, p_u, p_o)	4-19
4.6.1	Initialisierung von *MaxElem* in linearer Zeit - *SchleifeA*	4-25
4.6.2	Initialisierung von *MaxElem* in linearer Zeit - *SchleifeC*	4-25
4.6.3	Resultat nach Initialisierung von *MaxElem*	4-25
4.7.1	Das Verhältnis zwischen n und k	4-26
5.1.1	Die Y-Orientierungen $\alpha_{i,j}$ und $\alpha_{j,i}$	5-27
5.1.2	Die kleinste Y-Orientierung liegt zwischen zwei aufeinanderfolgenden Punkten	5-28
5.2.1	Einschränkung der Konturen durch zwei Farben	5-29
5.2.2	Vorzeitige Beendigung der Farbkonturerstellung	5-29
5.2.3	Verknüpfung einiger Farbkonturen	5-29
5.2.4	Die Elemente $e(p_m.col)$ werden immer dominiert	5-30
5.2.5	Die beiden unteren Randpunkte können gleichzeitig Eckpunkte sein	5-30
5.3.1	Maximale Anzahl zu überprüfender Orientierungen für $k = 3$	5-32
5.3.2	Prozentualer Anteil der Orientierungen, die tatsächlich untersucht werden müssen	5-34
5.3.3	Erweiterte Rechtecke beliebiger Orientierung	5-35
5.3.4	Lage von oberen und rechten Randpunkten eines erweiterten Typ 1-Rechtecks	5-36
5.3.5	Orientierungslosigkeit	5-36
5.3.6	Lage von erweiterten Typ 2-Rechtecken bei fixierter Orientierung und Farbe	5-37
5.4.1	Rechteck mit unbekannter Orientierung	5-37
5.4.2	Entstehung eines lokalen Minimums	5-38
5.4.3	Kein lokales Minimum	5-38
5.4.4	Abstandsmessung zwischen p_l und g	5-39
5.4.5	Betrag des Vektors $(0, 1)^t + (\cos 2\beta, \sin 2\beta)^t$	5-39
5.4.6	Maße eines rotierten Rechtecks	5-40
6.3.1	Das Applet in Aktion	6-45
6.3.2	Der Punkteditor	6-47

Literatur- und Quellenverzeichnis

[1] Manuel ABELLANAS, Ferran HURTADO, Christian ICKING, Rolf KLEIN, Elmar LANGETEPE, LIHONG Ma, Belén PALOP, Vera SACRISTÁN
„*Smallest Color-Spanning Objects*"
Technical Report 283, Department of Computer Science, FernUniversität Hagen, Germany, 2001.
http://www.informatik.fernuni-hagen.de/forschung/informatikberichte/pdf-versionen/283.pdf

[2] „*Boomregion Shanghai – Der Bauboom*"
Sendung „hitec" vom 21.10.2004 auf 3Sat
http://www.3sat.de/hitec/magazin/71937/index.html

[3] M. H. OVERMARS und J. VAN LEEUWEN.
"*Maintenance of configurations in the plane (revised edition)*"
Technical Report RUU-CS-81-3, Department of Computer Science, University of Utrecht, Netherlands, 1981, pp. 33-41
http://archive.cs.uu.nl/pub/RUU/CS/techreps/CS-1981/1981-03.pdf

Für die Programmierung wurden zusätzlich folgende Bücher verwendet:

Matthew ROBINSON, Pavel A. VOROBIEV
"*Swing (2nd Edition)*"
Manning, 2003

Marc LOY, Robert ECKSTEIN, Dave WOOD
"*Java Swing (2nd Edition)*"
O'Reilly & Associates, Nov. 2002

Bruce ECKEL
"*Thinking in Java (3rd Edition)*"
Bruce Eckel's Free Electronic Books, Nov. 2002
http://64.78.49.204/TIJ-3rd-edition4.0.zip

Gerd FISCHER
"*Lineare Algebra (9. Auflage)*"
Vieweg, 1986

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit und das dazugehörige Java-Applet selbstständig angefertigt, keine anderen, als die angegebenen Hilfsmittel benutzt, sowie Zitate kenntlich gemacht habe.

Bonn, den 29. März 2005

Andreas Lotz