UNIVERSITÄT DORTMUND

Algorithmen zur Konstruktion und Ausdünnung von Spanner-Graphen im *Cache-Oblivious*-Modell

Fabian Gieseke

Algorithm Engineering Report
TR07-3-005
Juni 2007
ISSN 1864-4503

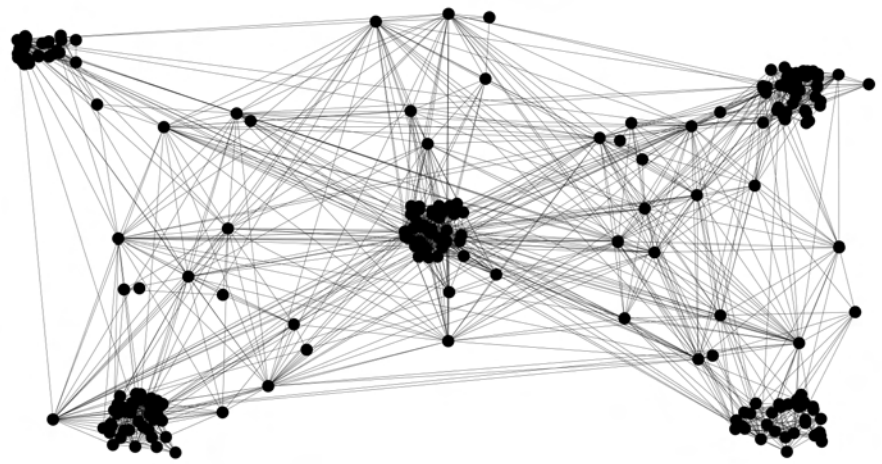
Universität Dortmund
Fachbereich Informatik
Algorithm Engineering (LS 11)
44221 Dortmund / Germany
http://ls11-www.cs.uni-dortmund.de/

Diplomarbeit

Algorithmen zur Konstruktion und Ausdünnung von Spanner-Graphen im *Cache-Oblivious*-Modell

Fabian Gieseke

7. Dezember 2006



Betreuer: Prof. Dr. Jan Vahrenhold

Institut für Informatik
Fachbereich Mathematik und Informatik
Westfälische Wilhelms-Universität Münster

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation und Hintergrund 1

1.2 Aufbau der Arbeit 4

1.3 Vorbemerkungen und Definitionen 5

1.3.1 Voraussetzungen 5

1.3.2 Graphen 5

2 Berechnungsmodelle **9**

2.1 Reale Computerarchitekturen 9

2.1.1 Speicherhierarchien 10

2.1.2 Lokalität von Programmen 12

2.1.3 Latenzzeiten 12

2.2 Das *Real*-RAM-Modell 13

2.3 Das I/O-Modell 14

2.4 Das *Cache-Oblivious*-Modell 16

2.4.1 Modellbeschreibung 16

2.4.2 Rechtfertigung des Modells 18

2.4.3 Annahme eines großen internen Speichers 18

2.4.4 Grundlegende Ergebnisse und Notationen 19

3 Algorithmen und Datenstrukturen **21**

3.1 Entwurfs- und Analysetechniken 21

3.1.1 *Divide and Conquer* 22

3.1.2 Rekursive Speicherstrukturen 22

3.1.3 Amortisierte Analyse 24

3.2 Algorithmen 25

3.2.1 Traversieren 25

3.2.2 Sortieren 26

3.3 Datenstrukturen 33

3.3.1 k-Verschmelzer 33

3.3.2 Prioritätswarteschlangen 40

4 Die *Well-Separated Pair Decomposition* 53

4.1 Bezeichnungen und Definitionen 54

4.2 Konstruktion einer WSPD im RAM- bzw. CREW PRAM-Modell 59

4.2.1 Optimaler sequentieller Algorithmus 59

4.2.2 Optimaler paralleler Algorithmus 68

4.3 Konstruktion einer WSPD im *Cache-Oblivious*-Modell 69

4.3.1 Vorbemerkungen 69

4.3.2 Konstruktion eines fairen Schnittbaums 72

4.3.3 Konstruktion einer WSPD 91

5 Konstruktion dünner Spanner-Graphen 95

5.1 Vorbemerkungen 95

5.2 Allgemeines Verfahren 97

5.3 Konstruktionsverfahren im RAM- und I/O-Modell 101

5.3.1 Algorithmen 102

5.3.2 Untere Schranken 103

5.4 Konstruktionsverfahren im *Cache-Oblivious*-Modell 104

6 Ausdünnung dichter Spanner-Graphen 107

6.1 Vorbemerkungen 107

6.2 Allgemeines Verfahren 108

6.3 Ausdünnungsverfahren im RAM- und I/O-Modell 112

6.4 Ausdünnungsverfahren im *Cache-Oblivious*-Modell 114

6.4.1 Orthogonale Bereichsanfragen 115

6.4.2 Markierungstechniken für Bäume 126

6.4.3 Der Ausdünnungsalgorithmus 128

7 Implementierung 133

7.1 Grundlegendes 133

7.1.1 Zielsetzung 133

7.1.2 Verwendete Bibliotheken 134

7.2 Implementierungsdetails 134

7.2.1 Datenstrukturen 134

7.2.2 Algorithmen 135

7.3 Programm 137

7.3.1 Funktionsumfang 137

7.3.2 Bedienung 139

7.4 Ergebnisse 139

7.4.1 Konstruktion von Spanner-Graphen 139
7.4.2 Ausdünnung von Spanner-Graphen 139

8 Zusammenfassung und Ausblick **145**
8.1 Zusammenfassung 145
8.2 Ausblick 147

A Weitere Beispiele **149**

Abbildungsverzeichnis **154**

Algorithmenverzeichnis **155**

Literaturverzeichnis **162**

Erklärung **162**

Kapitel 1

Einleitung

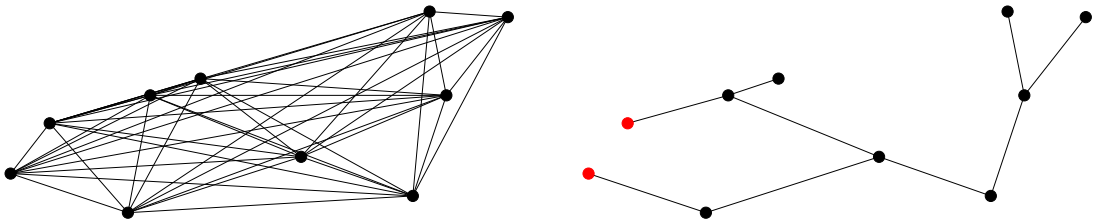
1.1 Motivation und Hintergrund

Beim Entwurf von Kommunikations-, Verkehrs- und Versorgungsnetzwerken besteht eine der wesentlichen Aufgaben darin, die beteiligten Objekte möglichst kostengünstig miteinander zu verbinden. Gleichzeitig müssen diese Netzwerke gewisse Rahmenbedingungen erfüllen. Zum Beispiel könnte gefordert sein, dass der „Weg“ zwischen zwei verschiedenen Objekten eine bestimmte Länge nicht überschreiten darf.

Zur Modellierung solcher Netzwerke, bzw. allgemein zur Modellierung einer Ansammlung von geometrischen Objekten mit Beziehungen untereinander, werden in der Informatik sogenannte *geometrische* bzw. *euklidische* Graphen verwendet. Ein geometrischer Graph ist ein ungerichteter Graph $G = (S, E)$, dessen Eckenmenge S eine endliche Punktmenge aus dem $\mathbb{R}^d$ ist. Wird zudem jeder Kante $e = \{p, q\} \in E$ eines solchen geometrischen Graphen $G = (S, E)$ der Euklidische Abstand der beiden Endecken p und q als Kantengewicht zugeordnet, so wird der dadurch induzierte gewichtete geometrische Graph als ein euklidischer Graph bezeichnet.

Bei der Modellierung eines der oben angesprochenen realen Netzwerke anhand eines geometrischen Graphen $G = (S, E)$ korrespondieren die geometrischen Objekte mit den Ecken aus S und die Verbindungen zwischen den geometrischen Objekten mit den Kanten aus E. Weiterhin können mit Hilfe von zusätzlichen Kantengewichten die Kosten für die einzelnen Verbindungen zwischen den geometrischen Objekten modelliert werden. Zur Vereinfachung wird dabei in der Regel angenommen, dass die Kosten einer Verbindung zweier Objekte der geometrischen Distanz zwischen diesen Objekten entsprechen. Die Hinzunahme von Kantengewichten führt demnach zu euklidischen Graphen.

Beim Entwurf eines Netzwerks für geometrische Objekte, also dem Entwurf eines euklidischen Graphen $G = (S, E)$ für eine endliche Punktmenge $S \subset \mathbb{R}^d$, müssen im Allgemeinen gewisse Kriterien berücksichtigt werden. In vielen Anwendungen ist es z. B. wichtig, eine möglichst „kurze“ Verbindung zwischen zwei Punkten $p, q \in S$ zu gewährleisten.



(a) Vollständiger Graph (b) Spannbaum

Abbildung 1.1: Der Entwurf eines euklidischen Graphen einer Punktmenge in der Ebene als vollständiger Graph (a) und als Spannbaum (b). Der vollständige Graph besitzt $\binom{10}{2} = 45$ Kanten. Die euklidische Länge des kürzesten Weges im Spannbaum zwischen den beiden rot markierten Punkten ist um ein Vielfaches größer als der Euklidische Abstand beider Punkte.

In diesen Fällen besteht ein offensichtlicher Ansatz darin, den vollständigen euklidischen Graphen $G = (S, E)$ als Graphen zur Modellierung des (gewünschten) Netzwerks zu wählen. Der Nachteil dieses Ansatzes besteht jedoch darin, dass der vollständige Graph zwar für zwei Ecken $p, q \in S$ die Kante $\{p, q\}$ enthält, die Kantenmenge E jedoch aus $\binom{|S|}{2} \in \mathcal{O}(|S|^2)$ Kanten besteht, vgl. Abbildung 1.1 (a). Für die meisten Anwendungen wäre das durch diesen Graphen modellierte Netzwerk aufgrund der quadratischen Anzahl an Verbindungen und der sich dadurch ergebenden (hohen) Kosten vermutlich nicht von Interesse.

Ein anderer Ansatz für den Entwurf eines solchen Netzwerks ergibt sich über die Konstruktion eines Spannbaums: Wird für die Punktmenge $S \subset \mathbb{R}^d$ ein Spannbaum als zugrundeliegender Graph gewählt, so besitzt dieser im Gegensatz zum vollständigen Graphen eine in der Anzahl der Punkte aus S lineare Anzahl von Kanten. Bei einem solchen Spannbaum kann jedoch die (euklidische) Länge des kürzesten Weges zwischen zwei Punkten $p, q \in S$ beliebig länger sein als der Euklidische Abstand beider Punkte, vgl. Abbildung 1.1 (b).

Eine Alternative zu diesen beiden Extremfällen bieten sogenannte Spanner-Graphen. Dabei ist ein (euklidischer) Spanner-Graph für eine endliche Punktmenge $S \subset \mathbb{R}^d$ ein euklidischer Graph $G = (S, E)$, dessen Kantenmenge E die Kanten eines Spannbaums für S enthält. Gilt zudem, dass die (euklidische) Länge des kürzesten Weges zwischen zwei Ecken p und q aus S höchstens t-mal so lang ist, wie der Euklidische Abstand der beiden Ecken, so heißt ein solcher Spanner-Graph ein t-Spanner für S. Der minimale Wert t', für den ein euklidischer Graph $G = (S, E)$ ein t'-Spanner für seine Eckenmenge S ist, wird als die *Dilatation* von G bezeichnet.

Der vollständige euklidische Graph einer endlichen Punktmenge $S \subset \mathbb{R}^d$ besitzt mit den obigen Definitionen eine Dilatation von 1. Da die mit Hilfe von vollständigen Graphen modellierten Netzwerke jedoch aufgrund der hohen Kantenanzahl in vielen Anwendungen zu kostspielig sein dürften, ist es von Interesse, sogenannte *dünn besetzte* Spanner-Graphen mit geringer Dilatation zu konstruieren. Ein dünn besetzter Spanner-Graph einer endli-

chen Punktmenge $S \subset \mathbb{R}^d$ ist ein Spanner-Graph $G = (S, E)$, dessen Kantenanzahl linear in der Anzahl der Ecken ist, d. h. für den $|E| \in \mathcal{O}(|S|)$ gilt. In der Literatur finden sich viele Arbeiten zur Konstruktion und Analyse von (dünn besetzten) Spanner-Graphen. Wesentlichen Einfluss auf dieses Gebiet hat eine von Callahan und Kosaraju [27] entwickelte Datenstruktur namens *well-separated pair decomposition* (WSPD). Wie Callahan und Kosaraju zeigen, ist es mit Hilfe dieser Datenstruktur möglich, zu einer beliebigen endlichen Punktmenge $S \subset \mathbb{R}^d$ und einer beliebigen reellen Konstante $\varepsilon > 0$ in einer Laufzeit von $\mathcal{O}(|S| \log |S|)$ einen dünn besetzten $(1 + \varepsilon)$-Spanner für S zu berechnen.

Ein mit der Konstruktion von dünn besetzten Spanner-Graphen verwandtes Problem ist die sogenannte „Ausdünnung“ von Spanner-Graphen. Mehrere Autoren, unter ihnen Gudmundsson *et al.* [43], haben sich mit der Frage beschäftigt, bis zu welchem Grad ein gegebener t-Spanner ($t \geq 1$) durch Weglassen von Kanten ausgedünnt werden kann, ohne dass sich die Dilatation zu sehr vergrößert. Gudmundsson *et al.* sind dabei zu dem Ergebnis gelangt, dass zu einem gegebenen t-Spanner $G = (S, E)$ einer endlichen Punktmenge $S \subset \mathbb{R}^d$ für jede reelle Konstante $\varepsilon > 0$ ein Teilgraph $G' = (S, E')$ mit $|E'| \in \mathcal{O}(|S|)$ konstruiert werden kann, welcher eine Dilatation von höchstens $(1 + \varepsilon)t$ besitzt.[1]

Dieser Ausdünnungsalgorithmus, welcher im Modell der *real random access machine* (*real*-RAM) formuliert und analysiert worden ist, wurde anschließend von Gudmundsson und Vahrenhold [45] in das von Aggarwal und Vitter [2] vorgestellte I/O-Modell übertragen. Im I/O-Modell wird die Untersuchung von Algorithmen formalisiert, die Daten auf zwei Ebenen einer zugrundeliegenden Speicherhierarchie verwalten. Ein (klassisches) Beispiel für ein solches Paar von Ebenen ist das Paar Hauptspeicher-Sekundärspeicher. Dieses Paar veranschaulicht besonders gut, dass zwischen den Zugriffszeiten auf Datenelemente unterschiedlicher Ebenen erhebliche Geschwindigkeitsunterschiede bestehen können, so dass bei Bearbeitung großer Datenmengen der Transfer von Datenelementen und nicht die Anzahl der von der CPU ausgeführten Operationen für die tatsächliche Laufzeit eines Algorithmus auf einer real existierenden Computerarchitektur entscheidend ist.

Im I/O-Modell von Aggarwal und Vitter treten neben dem Parameter N für die Anzahl der untersuchten Datenelemente auch die Parameter M und B auf. Der Parameter M gibt dabei die Anzahl der Datenelemente an, die gleichzeitig im Hauptspeicher gehalten werden können. Der Hauptspeicher und der Sekundärspeicher sind weiterhin in Blöcke fester Größe partitioniert. Die Größe eines Blocks von Datenelementen wird im I/O-Modell durch den Parameter B festgelegt. Die CPU kann im I/O-Modell weiterhin nur auf Datenelementen operieren, die sich im Hauptspeicher befinden. Die dafür benötigten Datenelemente werden dazu mittels sogenannter *I/O-Operationen* zwischen dem Sekundärspeicher und dem Hauptspeicher bewegt. Eine solche Operation besteht aus dem Transfer eines Blocks zusammenhängender Datenelemente vom Sekundärspeicher in den Hauptspeicher oder vom Hauptspeicher in den Sekundärspeicher. Als Kriterium für die Güte eines Algorithmus

[1] Zu beachten ist hierbei, dass der Wert ε in die „versteckte“ Konstante der $\mathcal{O}$-Notation eingeht.

werden im I/O-Modell die Anzahl der benötigten I/O-Operationen sowie die Anzahl der verwendeten Sekundärspeicherblöcke verwendet.

In der Literatur finden sich zahlreiche Arbeiten, die sich mit im Kontext des I/O-Modells effizienten Algorithmen befassen. Ein grundlegendes Ergebnis hinsichtlich der I/O-effizienten Konstruktion von dünn besetzten Spanner-Graphen stellt das von Govindarajan *et al.* [40] entwickelte I/O-effiziente Verfahren zur Konstruktion einer WSPD für eine endliche Punktmenge aus dem $\mathbb{R}^d$ dar. Mit Hilfe dieses Verfahrens zur Konstruktion einer WSPD zeigten Govindarajan *et al.*, dass analog zu der oben angesprochenen Übertragung des Ausdünnungsalgorithmus vom RAM-Modell in das I/O-Modell eine Übertragung des Verfahrens zur Konstruktion dünn besetzter Spanner-Graphen vom RAM-Modell in das I/O-Modell möglich ist.

Ein allgemeines Modell für mehrstufige Speicherhierarchien, welches das I/O-Modell als Spezialfall enthält, ist das von Frigo *et al.* [38] vorgestellte *cache-oblivious*-Modell. Die Grundidee dieses Modells besteht darin, einen Algorithmus wie im RAM-Modell (ohne Annahme einer Speicherhierarchie) zu entwerfen und ihn wie im I/O-Modell ohne Kenntnis der genauen Belegungen der Variablen M und B zu analysieren. Dies hat zur Konsequenz, dass ein in diesem Modell als optimal erkannter Algorithmus simultan auf *jeder* Ebene der zugrundeliegenden Speicherhierarchie optimal ist.

Ziel dieser Arbeit ist es, die im RAM- bzw. I/O-Modell formulierten Algorithmen zur Konstruktion dünn besetzter Spanner-Graphen bzw. zur Ausdünnung dichter Spanner-Graphen in das *cache-oblivious*-Modell zu übertragen, d. h. die entsprechenden algorithmischen Bausteine so zu realisieren, dass sie im Kontext dieses Modells eine möglichst gute asymptotische Komplexität besitzen. Zur Veranschaulichung der Arbeitsweise der Konstruktions- und Ausdünnungsalgorithmen soll zudem eine programmtechnische Realisierung durchgeführt werden, der die entsprechenden Algorithmen zugrunde liegen.

1.2 Aufbau der Arbeit

Im weiteren Verlauf dieses Kapitels werden einige Voraussetzungen festgelegt und ausgewählte mathematische Grundlagen aus dem Bereich der Graphentheorie dargelegt.

In Kapitel 2 werden die oben angesprochenen Berechnungsmodelle beschrieben. Dabei wird auch auf einige Aspekte real existierender Computerarchitekturen und deren Speicherhierarchien eingegangen.

In Kapitel 3 erfolgt die Beschreibung und Analyse einiger für das *cache-oblivious*-Modell konzipierter Algorithmen und Datenstrukturen, die in den darauffolgenden Kapiteln verwendet werden.

Wesentlich für das Verfahren zur Konstruktion dünn besetzter Spanner-Graphen bzw. zur Ausdünnung dichter Spanner-Graphen ist die WSPD-Datenstruktur von Callahan und Kosaraju [27]. Diese wird in Kapitel 4 ausführlich beschrieben. Zudem wird ein im Kon-

text des *cache-oblivious*-Modell optimales Verfahren zur Konstruktion einer WSPD gegeben, welches sich durch einige in dieser Arbeit entwickelte Änderungen des I/O-effizienten Verfahrens von Govindarajan *et al.* [40] ergibt.

Wie auch im RAM- bzw. I/O-Modell führt die effiziente Berechnung einer WSPD im *cache-oblivious*-Modell direkt zu einem effizienten Verfahren zur Konstruktion dünn besetzter Spanner-Graphen. Dieses im Kontext des *cache-oblivious*-Modells effiziente Verfahren zur Konstruktion von Spanner-Graphen ist Inhalt des Kapitels 5.

In Kapitel 6 wird analog zur Übertragung der Konstruktionsalgorithmen das I/O-effiziente Verfahren von Gudmundsson und Vahrenhold [45] zur Ausdünnung dichter Spanner-Graphen in das *cache-oblivious*-Modell übertragen.

Zur Veranschaulichung der Arbeitsweise der Konstruktions- und Ausdünnungsalgorithmen wurde eine programmtechnische Realisierung durchgeführt. Diese wird in Kapitel 7 ausführlich beschrieben.

Abschließend werden in Kapitel 8 die in dieser Arbeit entwickelten Ergebnisse zusammengefasst und ein Ausblick auf mögliche Anwendungen dieser Ergebnisse gegeben.

1.3 Vorbemerkungen und Definitionen

1.3.1 Voraussetzungen

Zum Verständnis der Arbeit werden Grundkenntnisse aus dem Bereich der Informatik als bekannt vorausgesetzt, die i. A. im Grundstudium eines Informatikstudiums oder eines ähnlichen Studiengangs behandelt werden. Als Nachschlagewerk für evtl. unbekannte Begriffe und Methoden eignet sich z. B. das Werk von Cormen *et al.* [32].

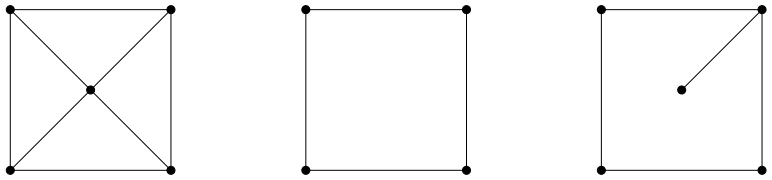
1.3.2 Graphen

Die folgenden Ausführungen basieren auf den Werken von Bollobás [18], Jungnickel [47] und Diestel [35] und geben ein kurzen Überblick über einige in dieser Arbeit verwendete Konzepte aus dem Bereich der Graphentheorie geben.

Grundlegende Definitionen

Ein *(ungerichteter) Graph* ist ein Paar $G = (V, E)$ disjunkter Mengen V und E mit $E \subseteq [V]^2$, wobei $[V]^2$ die Menge aller zweielementigen Teilmengen von V ist. Die Elemente aus V werden als *Ecken* und die Elemente aus E als *Kanten* bezeichnet. Entsprechend heißt die Menge V die *Eckenmenge* und die Menge E die *Kantenmenge* von G. Besitzt die Menge V unendlich viele Elemente, so heißt der Graph *unendlich*; ansonsten heißt er *endlich*.[2] Die Mächtigkeit $|V|$ der Eckenmenge wird als die *Ordnung* von G bezeichnet. Endliche Graphen sind also Graphen mit endlicher und unendliche Graphen mit unendlicher Ordnung.

[2] In dieser Arbeit werden ausschließlich endliche Graphen betrachtet.



(a) G (b) G' (c) G''

Abbildung 1.2: Ein Graph G und zwei Teilgraphen G' und G'' von G, vgl. Diestel [35]. Im Gegensatz zu G' ist G'' kein Untergraph von G.

Weiterhin heißt der Graph $G = (\emptyset, \emptyset)$ der *leere* Graph. Alle anderen Graphen werden als *nicht-leere* Graphen bezeichnet.

Ein Graph $G' = (V', E')$ heißt *Teilgraph* eines Graphen $G = (V, E)$, geschrieben $G' \subseteq G$, wenn $V' \subseteq V$ und $E' \subseteq E$ gilt. Der Graph G ist in diesem Fall ein *Obergraph* von G' und man sagt, dass der Graph G den Graphen G' *enthält*. Der Teilgraph G' heißt *aufgespannt von V' in G*, wenn er alle Kanten $\{x, y\} \in E$ mit $x, y \in V'$ enthält. Dieser spezielle Typ eines Teilgraphen von G wird als *Untergraph* von G bezeichnet. Anhand Abbildung 1.2 wird veranschaulicht, dass nicht jeder Teilgraph eines Graphen G ein Untergraph von G sein muss.

Sei $G = (V, E)$ ein nicht-leerer Graph. Eine Ecke $v \in V$ heißt mit einer Kante $e \in E$ *inzident*, wenn $v \in e$ gilt. Die beiden mit einer Kante inzidenten Ecken heißen die *Endecken* der Kante. Zwei Ecken v und w aus V sind *benachbart* und heißen *Nachbarn*, wenn $\{v, w\} \in E$ gilt. Die Menge der mit einer Ecke $v \in V$ benachbarten Ecken aus V wird als die *Nachbarschaft* $\Gamma_G(v)$ von v bezeichnet. Entsprechend heißen zwei Kanten $e, f \in E$ mit $e \neq f$ *benachbart*, falls sie eine gemeinsame Endecke besitzen. Sind je zwei (beliebige) Ecken $v, w \in V$ benachbart, so heißt der Graph G *vollständig*. Der *Grad* $d_G(v) = d(v)$ einer Ecke $v \in V$ wird als die Anzahl der mit v inzidenten Kanten definiert; es gilt somit $d(v) = |\Gamma_G(v)|$. Eine Ecke vom Grad 0 heißt *isoliert*.

Ein Graph $G = (V, E)$ heißt ein *gewichteter Graph*, wenn es eine Abbildung $w : E \to \mathbb{R}$ gibt, die jeder Kante $e \in E$ ein *Kantengewicht* $w(e) \in \mathbb{R}$ zuordnet.

Wege und Zusammenhang

Ein *Weg* ist ein nicht-leerer Graph $P = (V, E)$ der Gestalt

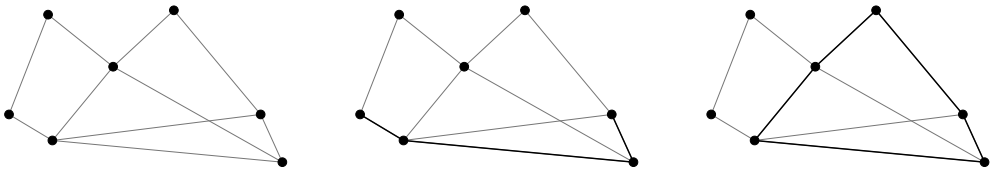
$$V = \{x_0, x_1, \ldots, x_k\}, \quad E = \{\{x_0, x_1\}, \{x_1, x_2\}, \ldots, \{x_{k-1}, x_k\}\},$$

wobei die Ecken x_i paarweise verschieden sind. Die Ecken $x_1, \ldots, x_{k-1}$ werden als die *inneren* Ecken von P bezeichnet. Die *Länge* des Weges P ist definiert als die Anzahl seiner Kanten. Ein Weg wird oft durch die natürliche Folge $P = x_0 x_1 \ldots x_k$ seiner Ecken beschrieben. Die Ecke x_0 heißt weiterhin die *Anfangsecke* und die Ecke x_k die *Endecke*

von P. Ein Weg $P = x_0x_1 \dots x_k$ wird als ein Weg *von* x_0 *nach* x_k oder auch als ein Weg *zwischen* x_0 *und* x_k bezeichnet. Ist $P = x_0x_1 \dots x_{k-1}$ ein Weg und $k \geq 3$, so heißt der Graph

$$C = (\{\{x_0, \dots, x_{k-1}\}, \{\{x_0, x_1\}, \{x_1, x_2\}, \dots, \{x_{k-2}, x_{k-1}\}, \{x_{k-1}, x_0\}\})$$

ein *Kreis*, vgl. Abbildung 1.3. Enthält ein Graph keine Kreise, so wird er *kreisfrei* oder *azyklisch* genannt.



(a) G (b) P (c) C

Abbildung 1.3: Ein Weg P und ein Kreis C in G, vgl. Diestel [35]

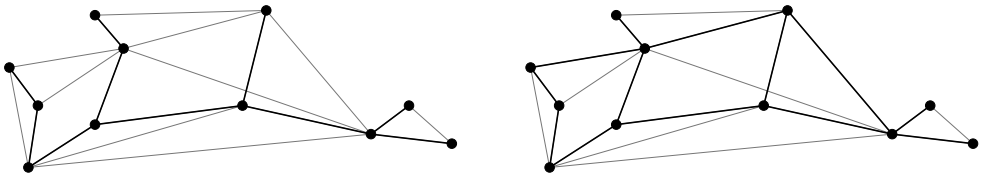
Ein nicht-leerer Graph $G = (V, E)$ heißt *zusammenhängend*, wenn er für je zwei verschiedene Ecken $x, y \in V$ einen Weg von x nach y enthält.

Bäume und Wälder

Enthält ein Graph keinen Kreis, so heißt er *Wald*. Ein zusammenhängender Wald wird als ein *Baum* bezeichnet. Die Ecken eines Baums werden zum besseren Verständnis im Folgenden vorwiegend als *Knoten* bezeichnet. Die Knoten eines Baums mit Grad 1 werden *Blätter* genannt. Ein *Teilbaum* eines Baums $T = (V, E)$ ist ein Teilgraph von T, welcher selbst wieder ein Baum ist. Ein Baum wird weiterhin als ein *Wurzelbaum* oder *gewurzelter Baum* bezeichnet, wenn es in ihm einen fest gewählten ausgezeichneten Knoten gibt. Der ausgezeichnete Knoten ist in diesem Fall die *Wurzel* des Baums. In einem gewurzelten Baum $T = (V, E)$ mit Wurzel $r \in V$ wird ein Knoten $v \in V$ als ein *Vorfahre* (oder *Vorgänger*) eines weiteren Knotens $w \in V$ und w als ein *Nachfahre* (oder *Nachfolger*) bezeichnet, wenn v in dem (einzigen) Weg von r nach w enthalten ist. Weiterhin heißt ein Knoten $v \in V$ der *Vater* eines weiteren Knotens $w \in V$ und der Knoten w ein *Kind* von v, wenn v ein Vorfahre von w ist und $\{v, w\} \in E$ gilt. Für zwei Knoten $v, w \in V$ wird ein Knoten $u \in V$ als der *kleinste gemeinsame Vorfahre* bezeichnet, wenn u sowohl von v als auch von w ein Vorfahre ist und es keinen Nachfahren von u mit dieser Eigenschaft gibt.

Spannbäume und Spanner-Graphen

Ein zusammenhängender Teilgraph $G' = (V, E')$ eines zusammenhängenden Graphen $G = (V, E)$ wird ein *Spanner-Graph* von G genannt. Ist ein Spanner-Graph $G' = (V, E')$ eines



(a) Spannbaum (b) Spanner-Graph

Abbildung 1.4: Ein Spannbaum bzw. Spanner-Graph eines Graphen

zusammenhängenden Graphen $G = (V, E)$ ein Baum, so wird G' auch als ein *Spannbaum* von G bezeichnet, vgl. Abbildung 1.4.

Gerichtete Graphen

In dieser Arbeit werden gelegentlich auch *gerichtete* Graphen betrachtet. Nach Jungnickel [47] ist ein gerichteter Graph ein Paar $G = (V, E)$ disjunkter Mengen V und E, wobei E eine Menge aus geordneten Paaren (v, w) mit $v, w \in V$ und $v \neq w$ ist. Die Elemente aus V werden analog zu einem (ungerichteten) Graphen als *Ecken* und die Elemente aus E als *Kanten* bezeichnet. Für eine Kante $e = (v, w) \in E$ werden v und w die *Endecken* und v bzw. w die *Anfangsecke* bzw. *Endecke* von e genannt. Eine Ecke $w \in V$ heißt ein *direkter Nachfolger* einer Ecke $v \in V$ mit $v \neq w$, wenn $(v, w) \in E$ gilt. In diesem Fall heißt die Ecke v auch ein *direkter Vorgänger* von w. Der *Ausgangsgrad* $d_G^-(v)$ einer Ecke $v \in V$ ist definiert als die Anzahl aller Kanten aus E, die v als Anfangsecke besitzen. Entsprechend gibt der *Eingangsgrad* $d_G^+(v)$ einer Ecke $v \in V$ die Anzahl aller Kanten aus E an, die v als Endecke besitzen. Die für ungerichtete Graphen eingeführten Begriffe und Notationen lassen sich i. A. direkt auf gerichtete Graphen übertragen, vgl. Jungnickel [47].

Aus der Definition eines gerichteten Graphen ergibt sich eine alternative Definition eines ungerichteten Graphen, die an einigen Stellen in dieser Arbeit verwendet wird: Ein gerichteter Graph $G = (V, E)$ wird auch als ein ungerichteter Graph bezeichnet, wenn für jede Kante $(v, w) \in E$ auch $(w, v) \in E$ gilt.

Kapitel 2

Berechnungsmodelle

Zur Entwicklung und Analyse von Algorithmen werden in der Informatik sogenannte *Berechnungsmodelle* herangezogen. Diese stellen Abstraktionen real existierender Computerarchitekturen dar und sollen im Allgemeinen zwei wesentliche Eigenschaften erfüllen: Zum Einen soll ein Berechnungsmodell die zu modellierenden Computerarchitekturen möglichst gut repräsentieren, d. h. detailliert genug sein, um die „Effizienz“ von Algorithmen möglichst gut vorhersagen zu können. Zum Anderen soll ein Berechnungsmodell hinreichend simpel gehalten sein, um einen einfachen Entwurf und eine einfache Analyse der Algorithmen zu gewährleisten. Berechnungsmodelle können also als idealisierte Modellrechner aufgefasst werden, um die Effizienz von Algorithmen zu analysieren, die auf (verschiedenen) realen Computerarchitekturen ablaufen.

In diesem Kapitel werden drei Berechnungsmodelle vorgestellt. Das erste ist das in der Informatik weit verbreitete *(real-)*RAM-Modell. Dieses wird in Abschnitt 2.2 kurz dargelegt. In den Abschnitten 2.3 und 2.4 werden anschließend das I/O- und das *cache-oblivious*-Modell ausführlich beschrieben. In Hinblick auf die Beschreibung bzw. auf ein besseres Verständnis dieser beiden Modelle wird dazu in Abschnitt 2.1 das Prinzip von Speicherhierarchien beschrieben, welche sich heutzutage in vielen Computerarchitekturen wiederfinden.

2.1 Reale Computerarchitekturen

In diesem Abschnitt werden einige grundsätzliche Merkmale heutiger Computerarchitekturen besprochen, welche zur Beschreibung des I/O- bzw. *cache-oblivious*-Modells wichtig sind. Dabei wird nicht auf die spezifischen Eigenschaften ausgewählter Computersysteme eingegangen, sondern vielmehr auf allgemeine Konzepte, die bei der Konstruktion dieser Systeme Anwendung finden.

2.1.1 Speicherhierarchien

Aus ökonomischen und technologischen Gründen besitzen heutige Computerarchitekturen im Allgemeinen Speicherhierarchien, die sich jeweils aus mehreren Speicherebenen zusammensetzen. Jede Ebene einer solchen Hierarchie greift auf Speichertypen unterschiedlicher Bauart zurück, welche sich in ihren Kosten und Leistungseigenschaften unterscheiden. Wesentliches Merkmal dieser Speicherhierarchien ist, dass der Speicher einer Speicherebene größer wird, je weiter die Ebene von der CPU „entfernt“ liegt, die Zugriffszeiten auf die Speicherzellen der jeweiligen Speicher jedoch mit wachsender Entfernung der zugehörigen Ebene zur CPU (stark) ansteigen; die Ebenen einer solchen Hierarchie sind also i. A. nach sinkender Zugriffszeit und steigender Speicherkapazität angeordnet. In Abbildung 2.1 wird eine typische Speicherhierarchie schematisch dargestellt. Diese setzt sich aus einem L1-Cache, L2-Cache, L3-Cache, Hauptspeicher und einer oder mehreren Festplatte zusammen. Die Register werden dabei oft als Bestandteil der CPU und nicht als eigenständige Speicherebene einer solchen Hierarchie angesehen.

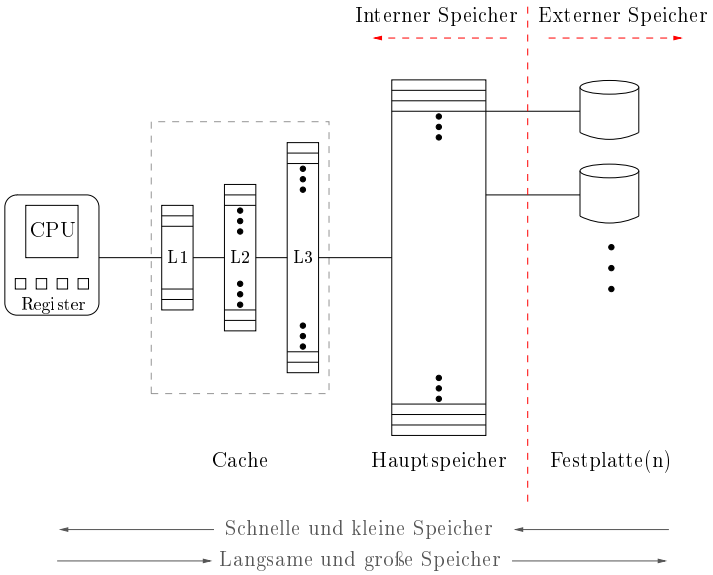


Abbildung 2.1: Eine (typische) Speicherhierarchie, die sich aus mehreren Ebenen zusammensetzt. Die Ebenen unterscheiden sich unter anderem in der Größe ihrer Speicher und in den Zugriffszeiten auf diese Speicher.

Aufgrund der großen Unterschiede in den Zugriffszeiten auf die Speicherzellen zwischen dem Hauptspeicher und den Festplatten werden die Speicherebenen von der CPU bis einschließlich zum Hauptspeicher oft als der *interne Speicher* und alle weiteren, also Festplatten und andere Sekundärspeichermedien, als der *externe Speicher* einer Computerarchitektur obigen Typs bezeichnet.

Reale Speicherhierarchien erfüllen weiterhin i. A. eine sogenannte *Inklusionseigenschaft*: Sind die Ebenen einer Speicherhierarchie der Größe ihrer Speicher nach aufsteigend numeriert, so bilden die Datenelemente im Speicher der Ebene i eine (echte) Teilmenge der Datenelemente im Speicher $i+1$. Der Speicher der von der CPU am weitesten entfernt liegenden Ebene enthält dabei alle (aktuellen) Datenelemente.

Die Inklusionseigenschaft von Speicherhierarchien wird in Abbildung 2.2 vereinfacht dargestellt. Die CPU hat dabei nur auf den Speicher der obersten Ebene direkten Zugriff. Befindet sich bei einem Zugriff der CPU auf den Speicher der obersten Ebene ein referenziertes Datenelement in diesem Speicher, so tritt ein *Treffer* (*cache hit*) auf und das referenzierte Datenelement wird der CPU übergeben. Ansonsten tritt ein *Fehlzugriff* (*cache miss*) auf, was zu einem Transfer von Datenelementen aus dem Speicher einer unteren Ebene in den Speicher der obersten Ebene führt.

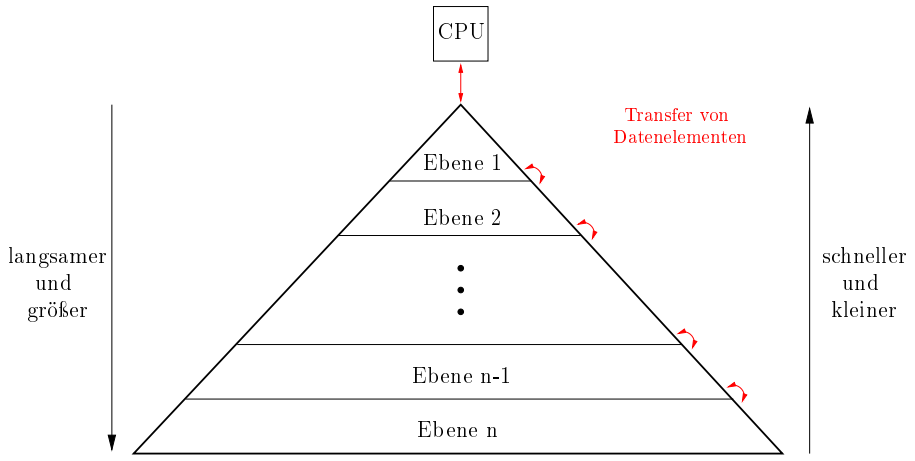


Abbildung 2.2: Inklusionseigenschaft von Speicherhierarchien

Für den effizienten Transfer von Datenelementen sind die Speicher aller Ebenen jeweils in *Blöcke* aufgeteilt. Ein solcher Block besteht aus einem zusammenhängenden Teil des zugehörigen Speichers und besitzt pro Ebene eine feste Größe. Die Blockgrößen unterschiedlicher Ebenen variieren dabei im Allgemeinen und nehmen mit wachsender Entfernung zur CPU zu. Die Bereiche eines Speichers, in denen ein Block abgelegt werden kann, werden die *Cache-Zeilen* des Speichers genannt.

Der bei einem Fehlzugriff stattfindende Transfer von Datenelementen findet jeweils nur zwischen zwei benachbarten Ebenen der Speicherhierarchie statt: Befindet sich ein referenziertes Datenelement nicht im Speicher der Ebene i, so wird der Zugriff an den Speicher der Ebene $i+1$ weitergeleitet. Da sich alle Datenelemente im Speicher der untersten Ebene befinden, bricht dieser Vorgang nach endlich vielen Schritten ab. Der Datentransfer zwischen zwei benachbarten Ebenen i und $i+1$ geschieht dabei mit Hilfe von *Blocktransfers*, also dem Transfer ganzer Blöcke von Datenelementen.

Ist der Speicher der Ebene i vor einem solchen Transfer voll belegt, so muss zuvor ein Block aus dem Speicher der Ebene i in den Speicher der Ebene $i+1$ ausgelagert werden. Die Auswahl eines solchen Blocks geschieht mit Hilfe der *Cache-Assoziativität* und der *Ersetzungsstrategie* des Speichers der Ebene i. Die *Cache*-Assoziativität eines Speichers begrenzt dabei die in Frage kommenden Cache-Zeilen, in denen ein Block abgelegt werden kann. Sind dabei bei einem Transfer eines Blocks in den Speicher der Ebene i alle für diesen Block möglichen Cache-Zeilen durch weitere Blöcke belegt, so muss einer dieser Blöcke in den Speicher der Ebene $i+1$ ausgelagert werden. Die Auswahl eines entsprechenden Blocks geschieht mit Hilfe der Ersetzungsstrategie des Speichers der Ebene i. Auf beide Begriffe soll hier jedoch nicht näher eingegangen werden.

2.1.2 Lokalität von Programmen

Der Nutzen von Speicherhierarchien basiert auf der *Lokalitätseigenschaft* von Computerprogrammen. Diese Eigenschaft setzt sich aus der *räumlichen Lokalität* und der *zeitlichen Lokalität* zusammen:

Das Prinzip der räumlichen Lokalität besagt, dass bei Abarbeitung eines Computerprogramms nach einem Zugriff auf ein Datenelement im Speicher in naher Zukunft mit hoher Wahrscheinlichkeit auch Zugriffe auf benachbarte Datenelemente stattfinden werden. Ein bekanntes, diesen Sachverhalt darstellendes Programmfragment, ist der sukzessive Zugriff auf die Elemente eines Arrays, wie er z. B. durch eine `for`-Schleife realisiert werden kann.

Der Begriff der zeitlichen Lokalität beschreibt den Sachverhalt, dass bei Abarbeitung eines Computerprogramms nach einem Zugriff auf ein Datenelement mit hoher Wahrscheinlichkeit in naher Zukunft erneut auf dieses Datenelement zugegriffen wird. Zwei Beispiele für die zeitliche Lokalität von Programmen sind z. B. das mehrmalige Traversieren eines Arrays bei der Multiplikation von (kleinen) Matrizen oder der Zugriff auf eine Zählvariable innerhalb einer `for`-Schleife.

2.1.3 Latenzzeiten

Mit dem Begriff *Latenzzeit* wird in der Informatik das Zeitintervall vom Ende eines Ereignisses bis zum Beginn einer Reaktion auf dieses Ereignis bezeichnet. Im Kontext von Speicherzugriffen der CPU auf den Speicher bzw. auf die Speicherhierarchie bezeichnet die Latenzzeit die Wartezeit der CPU, die nach einem Zugriff auf eine Speicherzelle auftritt.

Speicherhierarchien stellen den Versuch dar, die Wartezeit der CPU bei Abarbeitung von Programmen möglichst gering zu halten. Dabei nutzen sie die räumliche und zeitliche Lokalität dieser Programme, um für die CPU die Illusion eines Speichers zu erschaffen, dessen Größe der des Speichers der untersten Ebene und dessen Zugriffszeiten denen des Speichers der obersten Ebene der zugrundeliegenden Speicherhierarchie entsprechen.

Dieser Ansatz gelingt jedoch nur bedingt: Bei der Bearbeitung großer Datenmengen tritt das Problem auf, dass die Datenelemente nicht allesamt in die schnellen oberen Speicher der Speicherhierarchie passen, was zu einem ständigen Transfer von Daten aus den Speichern der unteren Ebenen in die Speicher der oberen Ebenen führt. Da die Zugriffszeiten der Speicher der unteren Ebenen sich i. A. erheblich von denen der Speicher der oberen Ebenen unterscheiden, kann bei Bearbeitung solcher massiven Datenmengen der Transfer von Daten eine wesentliche Rolle bei der tatsächlichen Laufzeit der Programme spielen.

2.2 Das *Real*-RAM-Modell

Eines der bedeutendsten mathematischen Modelle zur Analyse von Algorithmen und Datenstrukturen stellt die *random access machine* (RAM) dar. Im RAM-Modell wird ein Computer mit einem Speicher ausgestattet, der aus einer abzählbar unendlichen Menge von Speicherzellen besteht. Die CPU hat direkten Zugriff auf die Zellen, von denen jede eine ganze Zahl beliebiger Größe speichern kann, vgl. Abbildung 2.3. Sowohl für den Zugriff der CPU auf die Speicherzellen als auch für arithmetische Operationen der CPU wird ein *konstanter* Zeitbedarf veranschlagt.

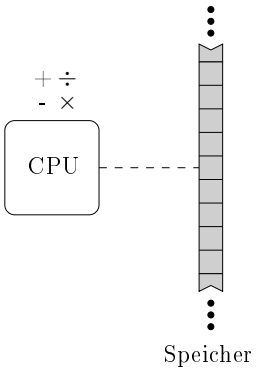


Abbildung 2.3: Das RAM-Modell

Als Erweiterung dieses Modells kann das Modell der *real random access machine* (*real*-RAM) [52] gesehen werden. Eine Speicherzelle der *real*-RAM kann im Gegensatz zu einer Speicherzelle der RAM eine beliebige reelle Zahl speichern. Zudem sind die folgenden Operationen auf der *real*-RAM möglich, für die ebenfalls unabhängig von der Größe der Operanden ein konstanter Zeitbedarf veranschlagt wird [52]:

1. Arithmetische Operationen $(+, -, \times, \div)$
2. Vergleiche zweier reeller Zahlen $(<, \leq, =, \neq, \geq, >)$
3. Indirekte Adressierung des Speichers (durch ganzzahlige Adressen)

Optional:

4. k-te Wurzeln, trigonometrische Funktionen, exp und log (allgemein: analytische Funktionen)

Sowohl im RAM- als auch im *real*-RAM-Modell wird also eine „flache“ Speicherhierarchie angenommen, d. h. für die Zugriffe der CPU auf die Speicherzellen ein konstanter Zeitbedarf veranschlagt. Diese Annahme ermöglicht zwar einen einfachen Entwurf und eine einfache Analyse von Algorithmen, jedoch bleibt die Problematik von unterschiedlichen Zugriffszeiten auf die Speicher realer Speicherhierarchien unberücksichtigt; eine Analyse im RAM- bzw. *real*-RAM-Modell erfasst also i. A. nicht die tatsächliche Effizienz eines Algorithmus beim Umgang mit großen Datenmengen.

2.3 Das I/O-Modell

Aufgrund dieses Nachteils der beiden oben beschriebenen Modelle wurden in der Vergangenheit zahlreiche *Mehrspeichermodelle* entworfen. Das bekannteste dieser Modelle stellt das von Aggarwal und Vitter [2] vorgestellte I/O-Modell dar. Im I/O-Modell wird ein Computer mit einem *internen Speicher* und einem *externen Speicher* ausgestattet, vgl. Abbildung 2.4. Der interne Speicher kann eine feste Anzahl von Datenelementen aufnehmen und ist in *Blöcke* partitioniert, die ihrerseits aus einer Folge von zusammenhängend im Speicher abgelegten Datenelementen bestehen. Der externe Speicher ist ebenso in Blöcke derselben Größe partitioniert, kann jedoch eine (nahezu) unbegrenzte Anzahl von Datenelementen fassen. Die durch das Modell festgelegten Parameter lauten

$$\begin{aligned} N &= \#\text{ der Datenelemente der Probleminstanz,} \\ M &= \#\text{ der Datenelemente, die in den internen Speicher passen,} \\ B &= \#\text{ der Datenelemente pro Block,} \end{aligned}$$

wobei $1 \leq B \leq M \ll N$ gefordert wird.[1] Durch die Ungleichung wird also gefordert, dass ein Block mindestens ein Element enthält und dass nicht alle N Elemente des zu untersuchenden Problems in den internen Speicher passen. Berechnungen der CPU können nur auf Datenelementen getätigt werden, die sich im internen Speicher befinden. Die für die Berechnungen benötigten Datenelemente werden dazu zwischen dem externen und internen Speicher durch *Ein-/Ausgabeoperationen* (I/O-Operationen) bewegt; eine solche Operation besteht aus dem Transfer eines Blocks zusammenhängender Datenelemente vom externen in den internen Speicher oder aus dem Transfer eines solchen vom internen in den externen Speicher. Die Kontrolle über den Transfer von Datenelementen zwischen dem internen und

[1] Im I/O-Modell von Aggarwal und Vitter [2] wird durch einen zusätzlichen Parameter P der Aspekt paralleler Speicherzugriffe berücksichtigt. Wie auch in anderen Arbeiten bleibt dies im Folgenden unberücksichtigt.

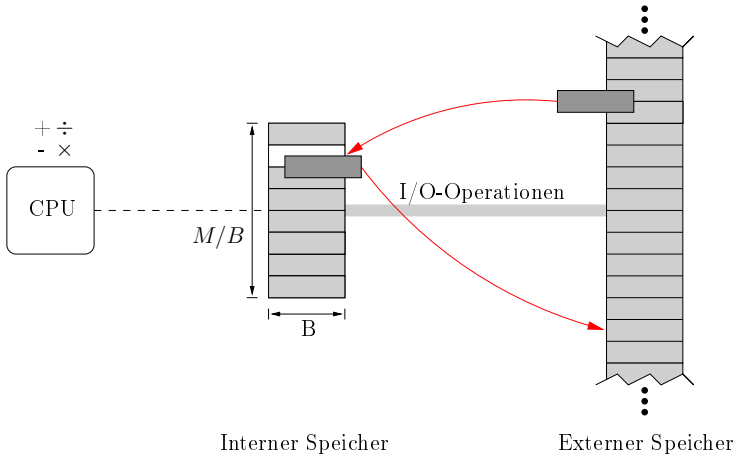


Abbildung 2.4: Das I/O-Modell von Aggarwal und Vitter [2, 34]

externen Speicher sowie die Verwaltung der Datenelemente in beiden Speichern liegt dabei vollständig in der Hand der Algorithmen.

Die Effizienz eines Algorithmus wird im I/O-Modell durch die Anzahl der benötigten I/O-Operationen, durch den benötigten Speicherplatz im externen Speicher und durch die Laufzeit im internen Speicher, d. h. die Laufzeit des Algorithmus im RAM-Modell, gemessen.

Die Stärken des I/O-Modells begründen sich darin, dass es die Problematik des Transfers von Daten zwischen zwei Speicherebenen beim Umgang mit massiven Datenmengen erfasst und zugleich hinreichend simpel gehalten ist, um einen einfachen Entwurf und eine einfache Analyse zu gewährleisten. Aufgrund dieser beiden Eigenschaften wurden in der Vergangenheit zahlreiche im Kontext dieses Modells effiziente Algorithmen für Probleme aus den unterschiedlichsten Bereichen entwickelt. Eine generelle Zusammenfassung relevanter Ergebnisse findet sich z. B. in den Arbeiten von Arge [5] oder von Vitter [57].

Das I/O-Modell weist jedoch zwei entscheidende Nachteile auf: Zum Einen modelliert es nur den Transfer von Daten zwischen *zwei* Ebenen der zugrundeliegenden Speicherhierarchie. Zum Anderen beruhen effiziente Implementierungen von im I/O-Modell konzipierten Algorithmen ausschlaggebend auf der genauen Kenntnis der Eigenschaften der modellierten Ebenen der (realen) Speicherhierarchie, d. h. auf der Kenntnis der Werte für die Parameter M und B.

Um auch den Transfer von Daten zwischen mehreren Ebenen zu modellieren, wurden weitere Modelle mit komplexeren Eigenschaften entworfen [57]. Diese modellieren zwar i. A. real existierende Computerarchitekturen besser als das I/O-Modell, erschweren jedoch durch ihre Komplexität den Entwurf und die Analyse von Algorithmen. Zudem beruhen analog zum I/O-Modell effiziente Implementierungen von in diesen Modellen konzipierten Algorithmen auf den Kenndaten der zugrundeliegenden (realen) Speicherhierarchie.

2.4 Das *Cache-Oblivious*-Modell

Eine elegante Lösung zur Modellierung von Computerarchitekturen mit komplexen Speicherhierarchien stellt das von Frigo *et al.* [38, 53] vorgestellte *cache-oblivious*-Modell (auch *ideal-cache*-Modell genannt) dar. Die Grundidee dieses Modells besteht darin, einen Algorithmus wie im RAM-Modell (ohne Zugrundelegung einer Speicherhierarchie) zu beschreiben und wie im I/O-Modell von Aggarwal und Vitter [2] für beliebige Werte M und B zu analysieren. Dabei wird bei der Analyse eines Algorithmus angenommen, dass die (benötigten) I/O-Operationen automatisch mittels einer optimalen Ersetzungsstrategie durchgeführt werden.

2.4.1 Modellbeschreibung

Im *cache-oblivious*-Modell wird ein Computer wie im I/O-Modell mit einer zweistufigen Speicherhierarchie, d. h. einem *internen* und einem *externen* Speicher, ausgestattet. Der interne Speicher kann analog zum I/O-Modell eine begrenzte und der externe Speicher eine (nahezu) unbegrenzte Anzahl von Datenelementen aufnehmen. Beide Speicher sind ebenfalls in Blöcke fester Größe partitioniert. Weiterhin stimmen die durch das Modell festgelegten Parameter mit denen des I/O-Modells überein, d. h. mit $B \geq 1$ wird die Blockgröße, mit $M \geq B$ die Größe des internen Speichers und mit $N \gg M$ die Anzahl der Datenelemente der untersuchten Probleminstanz bezeichnet. In Anlehnung an reale Computerarchitekturen wird der interne Speicher des *cache-oblivious*-Modells auch als *cache* und die M/B Blöcke im internen Speicher als *cache*-Zeilen bezeichnet.

Die CPU kann nur auf Datenelementen operieren, die sich im internen Speicher befinden. Greift die CPU auf ein Datenelement zu, welches sich im internen Speicher befindet, so tritt ein *cache hit* auf und das Datenelement wird dem Prozessor übergeben. Ansonsten tritt ein *cache miss* auf und der das Datenelement enthaltende Block wird (automatisch) aus dem externen in den internen Speicher transferiert. Der durch einen *cache miss* ausgelöste Transfer eines Blocks wird als *Speichertransfer* bezeichnet.[2]

Der interne Speicher des *cache-oblivious*-Modells ist *vollassoziativ* [38, 53]: Ein in den internen Speicher eingelesener Block kann in jeder der M/B möglichen Positionen des internen Speichers abgelegt werden. Ist der interne Speicher voll belegt, so muss ein Block in den externen Speicher ausgelagert werden. Im Gegensatz zum I/O-Modell wird dies *nicht* explizit durch die Algorithmen gesteuert. Stattdessen wird im *cache-oblivious*-Modell eine optimale (automatische) Ersetzungsstrategie (*optimal offline replacement strategy*) angenommen: Es wird automatisch derjenige Block ausgelagert, auf dessen Datenelemente am längsten nicht mehr zugegriffen wird. Aufgrund dieser Eigenschaft wird der interne Speicher in der Literatur auch *ideal-cache* genannt.

[2] In dieser Arbeit werden die Blocktransfers im I/O-Modell als I/O-Operationen und die (automatischen) Blocktransfers im *cache-oblivious*-Modell als Speichertransfers bezeichnet.

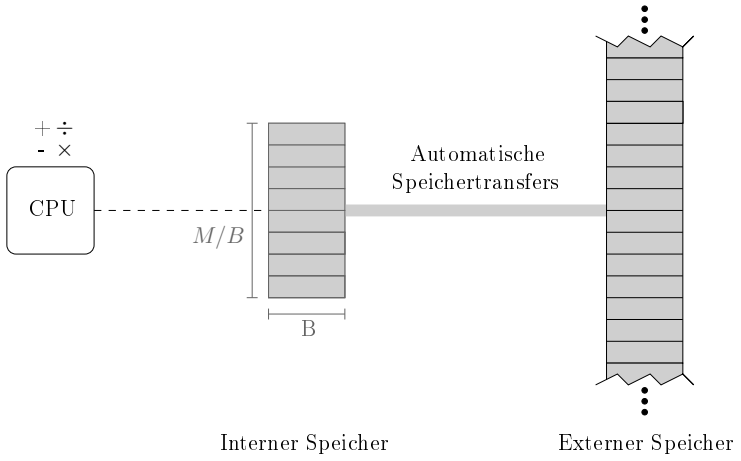


Abbildung 2.5: Das *cache-oblivious*-Modell von Frigo *et al.* [38, 53]

Die Effizienz eines Algorithmus wird im *cache-oblivious*-Modell durch die Anzahl der verursachten Speichertransfers, durch den im externen Speicher benötigten Speicherplatz und durch die Laufzeit im internen Speicher, d.h. die Laufzeit des Algorithmus im RAM-Modell, gemessen.[3]

Der entscheidende Unterschied zum I/O-Modell besteht darin, dass die genauen Werte für die Größen M und B im *cache-oblivious*-Modell nicht bekannt sind, vgl. Abbildung 2.5. Algorithmen werden also im *cache-oblivious*-Modell ohne Kenntnis der genauen Werte für die Parameter M und B entworfen und anschließend hinsichtlich jeder konkreten Belegung dieser beiden Variablen analysiert.

Diese (anscheinend kleine) Änderung hat zwei erstaunliche Konsequenzen: Da die Analyse eines Algorithmus für jede Belegung der Variablen M und B erfolgt, treffen die Aussagen der Analyse auf *jedes* Paar von benachbarten Ebenen der (modellierten) Speicherhierarchie zu, d.h. durch Optimierung eines Algorithmus bzgl. eines Paars benachbarter Ebenen mit unbekannten Kenndaten, wird dieser automatisch bzgl. jedes Paars benachbarter Ebenen der Speicherhierarchie optimiert. Die erste Konsequenz ist also, dass durch Optimierung bzgl. eines Paars benachbarter Ebenen eine simultane Optimierung bzgl. jedes Paars benachbarter Ebenen der Speicherhierarchie stattfindet. Die zweite Konsequenz ist, dass die Portabilität von im *cache-oblivious*-Modell konzipierten Algorithmen erhöht wird, da die (unbekannten) Werte M und B nicht fest in Implementierungen dieser Algorithmen verankert sein können.

[3] In dieser Arbeit liegt dabei das Hauptaugenmerk auf dem durch einen Algorithmus benötigten Speicherplatz und die durch ihn verursachten Speichertransfers. Der Laufzeitaspekt, also die Analyse des Algorithmus im Kontext des RAM-Modells, bleibt im Folgenden unberücksichtigt.

2.4.2 Rechtfertigung des Modells

Das *cache-oblivious*-Modell stellt als Berechnungsmodell eine Abstraktion realer Computerarchitekturen dar. Die vier im Modell getroffenen Grundannahmen lauten:

- Optimale Ersetzungsstrategie
- Zwei Speicherebenen
- Automatische Ersetzung
- Vollassoziativität

Frigo *et al.* [38] rechtfertigen diese (unrealistischen) Annahmen, indem sie eine Reihe von Reduktionen angeben, durch welche das *cache-oblivious*-Modell in ein realistischeres Berechnungsmodell abgeändert wird. In diesem realistischeren Modell wird von einer LRU-Ersetzungsstrategie (*least-recently used replacement strategy*), von mehreren Speicherebenen und von direkt abgebildeten Speichern ausgegangen. Weiterhin zeigen sie, dass die Anzahl der verursachten Speichertransfers eines Algorithmus im Kontext des *cache-oblivious*-Modells sich asymptotisch gesehen in diesem realistischeren Modell nicht ändert. Auf die Details dieser Reduktionen soll hier jedoch nicht näher eingegangen werden.

2.4.3 Annahme eines großen internen Speichers

Eine realistische und bei dem Entwurf und bei der Analyse von Algorithmen im Kontext des *cache-oblivious*-Modells häufig getroffene Annahme ist, dass der interne Speicher „höher“ als „breit“ ist, d. h., dass die Anzahl M/B der im internen Speicher Platz findenden Blöcke größer ist, als die Anzahl B der Elemente eines Blocks. In vielen Fällen wird dieser Sachverhalt durch eine Bedingung von der Form

$$M \in \Omega(B^2)$$

oder allgemeiner von der Form

$$M \in \Omega(B^{1+\varepsilon})$$

mit beliebigem $\varepsilon > 0$ vorausgesetzt. Eine Bedingung obigen Typs wird als eine *Annahme eines großen internen Speichers* oder als eine *tall-cache*-Annahme bezeichnet.

Die Bedeutung einer Annahme eines großen internen Speichers wurde von Brodal und Fagerberg [22] untersucht. Sie zeigen, dass das vergleichsbasierte Sortieren nicht ohne die Annahme eines großen internen Speichers, also durch eine Bedingung obiger Art, möglich ist.

2.4.4 Grundlegende Ergebnisse und Notationen

Ein Algorithmus wird im Kontext des I/O- bzw. *cache-oblivious*-Modells als *cache-oblivious* bezeichnet, wenn seine Effizienz hinsichtlich der benötigten I/O-Operationen bzw. hinsichtlich der verursachten Speichertransfers nicht von der genauen Kenntnis der Belegungen der Parameter M und B abhängt. Dementsprechend wird ein Algorithmus als *cache-aware* bezeichnet, wenn seine Effizienz hinsichtlich der I/O-Operationen bzw. Speichertransfers explizit von den Belegungen der Parameter M und B abhängt.

Da *cache-aware*-Algorithmen genauere Kenntnis über die zugrundeliegende Speicherhierarchie besitzen, können diese die Eigenschaften der Hierarchie besser nutzen. Demnach übertragen sich alle für das I/O-Modell entwickelten unteren Schranken direkt auf das *cache-oblivious*-Modell.

Im Kontext des I/O-Modells entwickelten Aggarwal und Vitter [2] für einige fundamentale Probleme obere und untere Schranken in Bezug auf die benötigten I/O-Operationen. Unter anderem zeigten sie, dass die Sortierung von N (im externen Speicher vorliegenden) Datenelementen insgesamt $\Theta(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/O-Operationen erfordert. Die Traversierung von N Datenelementen, d. h. der sukzessive Zugriff der CPU auf die Elemente (um z. B. ein „maximales“ Element zu bestimmen), benötigt offensichtlich $\Theta(\frac{N}{B})$ I/O-Operationen.

Die beiden unteren Schranken übertragen sich aufgrund des gerade genannten Arguments direkt auf das *cache-oblivious*-Modell. In Kapitel 3 werden für das Sortieren und Traversieren von Datenelementen effiziente *cache-oblivious*-Algorithmen gegeben, die diese Komplexität erreichen, d. h. die N Datenelemente mit $\mathcal{O}(\frac{N}{B}\log_{M/B}\frac{N}{B})$ bzw. mit $\mathcal{O}(\frac{N}{B})$ Speichertransfers sortieren bzw. traversieren.

In der Literatur werden die beiden obigen Schranken für die Traversierung und die Sortierung von N Datenelementen oft durch

$$\mathcal{O}\left(scan(N)\right) = \mathcal{O}\left(\frac{N}{B}\right) \quad \text{bzw.} \quad \mathcal{O}\left(sort(N)\right) = \mathcal{O}\left(\frac{N}{B}\log_{M/B}\frac{N}{B}\right)$$

abgekürzt. Diese Notationen werden im Folgenden ebenfalls verwendet.

Kapitel 3

Algorithmen und Datenstrukturen

Die in Kapitel 2 beschriebenen Vorteile des *cache-oblivious*-Modells gegenüber anderen Mehrspeichermodellen führten dazu, dass in diesem Modell zahlreiche Algorithmen und Datenstrukturen beispielsweise für Probleme aus der algorithmischen Geometrie [1, 9, 14, 20], für Probleme aus der Graphentheorie [7, 24, 30] und für die effiziente Bearbeitung von Suchanfragen [15, 16, 23] entwickelt wurden. Eine generelle Zusammenfassung relevanter Algorithmen und Datenstrukturen findet sich z. B. in den Arbeiten von Demaine [34] und Arge *et al.* [8].

In diesem Kapitel werden einige ausgewählte Algorithmen und Datenstrukturen vorgestellt, welche im weiteren Verlauf der Arbeit Anwendung finden werden. In Hinblick auf den Entwurf und die Analyse dieser Algorithmen und Datenstrukturen werden dazu zu Beginn dieses Kapitels in Abschnitt 3.1 einige im Kontext des *cache-oblivious*-Modells häufig angewendete Entwurfs- und Analysetechniken vorgestellt. In Abschnitt 3.2 wird dann das Traversieren und Sortieren von Datenelementen im Kontext des *cache-oblivious*-Modells untersucht. Bei der Analyse der Sortierung von Datenelementen wird dabei zunächst ein hinsichtlich der verursachten Speichertransfers suboptimales Sortierverfahren betrachtet. Anschließend wird ein von Brodal und Fagerberg [20] vorgestelltes im Kontext des *cache-oblivious*-Modells optimales Sortierverfahren detailliert beschrieben. Kernkomponente dieses optimalen Sortierverfahrens ist eine Datenstruktur namens „k-Verschmelzer“. Diese Datenstruktur und eine *cache-oblivious* Prioritätswarteschlange werden in Abschnitt 3.3 genauer erläutert.

3.1 Entwurfs- und Analysetechniken

In diesem Kapitel werden allgemeine Techniken zum Entwurf und zur Analyse von Algorithmen und Datenstrukturen im Kontext des *cache-oblivious*-Modells behandelt. Dabei wird zunächst auf das *divide and conquer*-Paradigma eingegangen, dessen Anwendung in vielen (wenn auch nicht in allen) Fällen zu optimalen *cache-oblivious*-Algorithmen führt.

Anschließend wird der Nutzen von rekursiven Speicherstrukturen bei der Konstruktion von *cache-oblivious*-Datenstrukturen anhand eines Beispiels veranschaulicht. Zuletzt wird ein bei der Analyse von Algorithmen und Datenstrukturen im *cache-oblivious*-Modell oft verwendetes Analyseverfahren beschrieben.

3.1.1 *Divide and Conquer*

Die Anwendung des *divide and conquer*-Paradigmas führt im *cache-oblivious*-Modell in vielen Fällen zu effizienten Algorithmen. Allgemein teilen *divide and conquer*-Algorithmen das zu lösende Problem in mehrere (unabhängig voneinander zu lösende) Teilprobleme mit geringerer Größe auf, lösen diese Teilprobleme rekursiv und kombinieren nach Beendigung aller rekursiven Aufrufe die ermittelten Teillösungen zu einer Lösung des Gesamtproblems. Da die Größe der rekursiven Teilprobleme mit zunehmender Rekursionstiefe sukzessiv abnimmt, passt ein solches Teilproblem ab einem gewissen Zeitpunkt vollständig in den internen Speicher und kann dort effizient gelöst, d. h. ohne weitere Speichertransfers zu verursachen, bearbeitet werden.

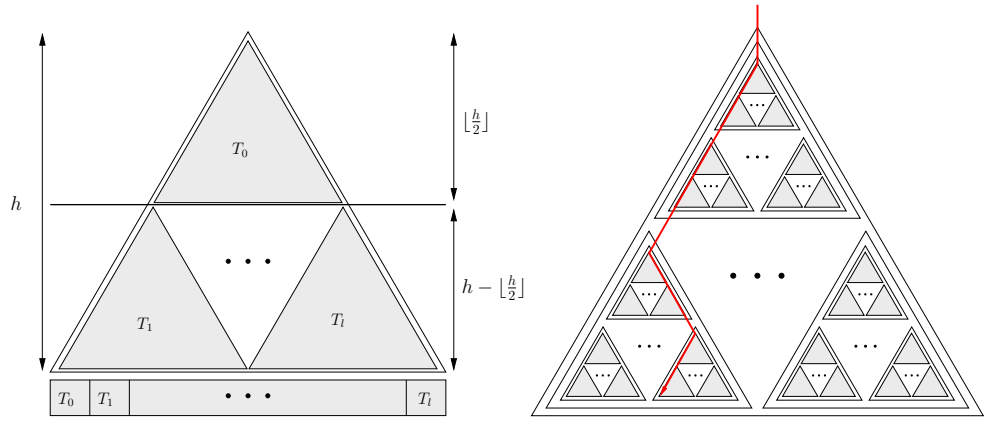
Wie Demaine [34] bemerkt, verursachen *divide and conquer*-Algorithmen im Kontext des *cache-oblivious*-Modells oft eine asymptotisch optimale Anzahl an Speichertransfers – speziell wenn durch das Aufteilen und Kombinieren der Teilprobleme nur wenige Speichertransfers verursacht oder die Kosten für das Aufteilen und Kombinieren durch die Blattkosten des zugehörigen Rekursionsbaums dominiert werden (d. h., dass die Anzahl der Blätter in dem Rekursionsbaum polynomial größer ist als die Anzahl der durch das Aufteilen und Kombinieren der Teilprobleme verursachten Speichertransfers).

Obwohl dieses Paradigma in vielen Fällen zu optimalen *cache-oblivious*-Algorithmen führt, muss dies nicht immer der Fall sein. Ein Beispiel für einen hinsichtlich der verursachten Speichertransfers suboptimalen *divide and conquer*-Algorithmus wird zu Beginn des Abschnitts 3.2.2 gegeben.

3.1.2 Rekursive Speicherstrukturen

Analog zu der bei *divide and conquer*-Algorithmen stattfindenden rekursiven Aufteilung des Gesamtproblems in Teilprobleme geringerer Größe führt im Kontext des *cache-oblivious*-Modells die Anwendung von Rekursion beim Entwurf von Datenstrukturen in vielen Fällen zu effizienten Strukturen.

Exemplarisch soll hier kurz die von Prokop [53] vorgestellte *cache-oblivious* Version eines B-Baums [13, 32] vorgestellt und analysiert werden. Im I/O-Modell können mit Hilfe eines B-Baums Suchanfragen effizient implementiert werden: Die Höhe eines B-Baums für N Elemente beträgt $\mathcal{O}(\log_B N)$. Da bei einer Suchanfrage für die Bearbeitung jedes Knotens im I/O-Modell höchstens $\mathcal{O}(1)$ I/O-Operationen benötigt werden, ergibt sich eine I/O-Komplexität von $\mathcal{O}(\log_B N)$ für eine einzelne Suchanfrage [57].



(a) Das *van Emde Boas layout* (b) Partitionierung in Basisbäume

Abbildung 3.1: Das *van Emde Boas layout* (Abbildung (a)) und die Partitionierung eines Baums in Basisbäume (Abbildung (b)), vgl. [8]. Der rot markierte Pfad durchläuft höchstens $\mathcal{O}(\log_B N)$ Basisbäume. Die Bearbeitung eines Basisbaums verursacht dabei $\mathcal{O}(1)$ Speichertransfers.

Wie Arge *et al.* [8] bemerken, scheint es schwierig eine *cache-oblivious* Variante eines B-Baums zu entwickeln, da der Wert B ein grundlegender Bestandteil in der Definition und Konstruktion der Datenstruktur ist. Um die obige I/O-Komplexität auch im Kontext des *cache-oblivious*-Modells zu erreichen, also Suchanfragen mit $\mathcal{O}(\log_B N)$ Speichertransfers zu ermöglichen, legt Prokop [53] einen (vollständigen) Binärbaum mit Hilfe eines rekursiv definierten Speicherlayouts im Speicher ab. Dieses rekursiv definierte Speicherlayout wird allgemein als das *van Emde Boas layout* bezeichnet und soll nun kurz erläutert werden.

Der Einfachheit halber werden hier nur vollständige Binärbäume betrachtet. Das *van Emde Boas layout* eines vollständigen Binärbaums T mit N Blättern wird wie folgt rekursiv definiert: Besitzt T nur einen Knoten, so besteht das *van Emde Boas layout* dieses Baums aus diesem einen im Speicher abgelegten Knoten. Ansonsten sei $h = \log N$ die Höhe des Baums.[1] Der *obere Baum* T_0 von T ist derjenige Teilbaum von T, dessen Knoten alle eine Tiefe von maximal $\lfloor h/2 \rfloor$ besitzen. Die *unteren Bäume* $T_1, \ldots, T_l$ sind die in den Knoten mit Tiefe $h - \lfloor h/2 \rfloor$ gewurzelten Teilbäume von T. Das *van Emde Boas layout* von T besteht dann aus dem *van Emde Boas layout* von T_0 gefolgt von den *van Emde Boas layouts* der unteren Bäume $T_1, \ldots, T_l$, vgl. Abbildung 3.1 (a). Aufgrund von $2^{(\log N)/2} = \sqrt{N}$ besitzt jeder der Teilbäume eine Größe von $\Theta(\sqrt{N})$ und es gilt $l \in \Theta(\sqrt{N})$.

Ein anhand des *van Emde Boas layouts* im Speicher abgelegter vollständiger Binärbaum unterstützt effiziente Suchanfragen:

3.1.1 Theorem ([8, 53]). *Sei T ein anhand des van Emde Boas layouts im Speicher abgelegter vollständiger Binärbaum mit N Blättern. Dann werden durch eine Suchanfrage*

[1] In dieser Arbeit wird mit log der Logarithmus $\log_2$ zur Basis 2 bezeichnet.

(also durch eine Traversierung eines Wurzel-Blatt-Pfades) höchstens $\mathcal{O}(\log_B N)$ *Speichertransfers verursacht.*

Beweis. Um die Anzahl der verursachten Speichertransfers zu analysieren, sei das Rekursionslevel der rekursiven Definition des *van Emde Boas layouts* betrachtet, ab der die Teilbäume eine Größe kleiner als B besitzen. Der gesamte Baum wird durch dieses Rekursionslevel in eine Menge von Teilbäumen partitioniert, die jeweils eine Größe zwischen $\Theta(\sqrt{B})$ und $\Theta(B)$ und somit eine Höhe von $\Omega(\log \sqrt{B}) = \Omega(\log B)$ besitzen, vgl. Abbildung 3.1 (b). Diese Teilbäume seien als *Basisbäume* bezeichnet. Nach Definition des *van Emde Boas layouts* wird jeder Basisbaum in $\Theta(B)$ zusammenhängend im Speicher liegenden Speicherplätzen abgelegt. Der Zugriff auf einen solchen Basisbaum verursacht demnach höchstens $\mathcal{O}(1)$ Speichertransfers. Betrachtet man nun einen Pfad von der Wurzel ausgehend zu einem Blatt von T, so durchläuft dieser Pfad insgesamt höchstens $\mathcal{O}((\log N)/\log B) = \mathcal{O}(\log_B N)$ Basisbäume. Eine Suchanfrage verursacht somit insgesamt höchstens $\mathcal{O}(\log_B N)$ Speichertransfers. □

Im Kontext des *cache-oblivious*-Modells führen rekursiv definierte Speicherstrukturen obiger Art oft zu effizienten Datenstrukturen. Ein Beispiel einer solchen rekursiv definierten Datenstruktur wird in Abschnitt 3.3 gegeben.

3.1.3 Amortisierte Analyse

Mit Hilfe der *amortisierten Analyse* [32] werden die Durchschnittskosten (z. B. die Laufzeit oder die verursachten Speichertransfers) einer einzelnen Datenstruktur-Operation im schlimmsten Fall innerhalb einer Sequenz von Operationen analysiert. Diese Analyse unterscheidet sich von der randomisierten Analyse [32], da in dieser die Kosten im erwarteten Fall, d. h. die Durchschnittskosten über alle möglichen Eingabefälle untersucht werden; bei der amortisierten Analyse werden hingegen die Durchschnittskosten jeder Operation im schlimmsten Fall betrachtet.

Wie Cormen *et al.* bemerken, sind die drei bekanntesten bei der amortisierten Analyse verwendeten Techniken die *Aggregat-*, *Bankkonto-* und *Potentialfunktion-Methode*. Die Aggregat- und Bankkonto-Methode werden nun kurz erläutert. Für weitere Details und eine Beschreibung der Potentialfunktion-Methode siehe Cormen *et al.* [32].

Aggregat-Methode

Bei Anwendung der Aggregat-Methode wird zunächst eine obere Schranke $T(N)$ für die Gesamtkosten einer beliebigen Sequenz von N Operationen ermittelt. Jeder Operation werden dann die *amortisierten Kosten* $T(N)/N$ zugeschrieben. Die für eine Operation veranschlagten amortisierten Kosten sind also unabhängig vom Typ der Operation immer gleich.

Bankkonto-Methode

Bei der Bankkonto-Methode können im Gegensatz zur Aggregat-Methode den Operationen abhängig vom Typ unterschiedliche amortisierte Kosten zugeschrieben werden. Einigen Operationen werden dadurch höhere und anderen geringere (amortisierte) Kosten zugeschrieben, als sie tatsächlich verursachen. Bei Betrachtung einer beliebigen Sequenz von Operationen wird dabei gefordert, dass zu jedem Zeitpunkt die Differenz zwischen den bis dahin insgesamt veranschlagten amortisierten und den bis dahin insgesamt verursachten Kosten größer oder gleich null ist: Werden die tatsächlich verursachten Kosten der i-ten Operation in einer beliebigen Sequenz von N Operationen mit c_i und die amortisierten Kosten dieser Operation mit $\hat{c_i}$ bezeichnet, so muss

$$\sum_{i=1}^{k} \hat{c_i} - \sum_{i=1}^{k} c_i \geq 0$$

für alle k mit $1 \leq j \leq N$ erfüllt sein.

Weitere Erläuterungen und einführende Beispiele dieser (beiden) Analysetechniken sind z.B. bei Cormen *et al.* [32] zu finden. In dieser Arbeit wird vor allem die Bankkonto-Methode bei der Analyse der vorgestellten Algorithmen und Datenstrukturen Anwendung finden.

3.2 Algorithmen

In diesem Abschnitt soll auf zwei fundamentale Bausteine bei der Konzeption von Algorithmen eingegangen und im Kontext des *cache-oblivious*-Modells analysiert werden: Das Traversieren und Sortieren von Datenelementen.

3.2.1 Traversieren

Das Traversieren von Datenelementen ist eine der am häufigsten verwendeten Techniken beim Entwurf von Algorithmen in Mehrspeichermodellen und speziell dem *cache-oblivious*-Modell. Im (*real*-)RAM-Modell können N Datenelemente in einer Laufzeit von $\mathcal{O}(N)$ traversiert, d.h. von dem Prozessor sukzessiv bearbeitet werden. Liegen die Datenelemente im *cache-oblivious*-Modell in beliebigen Blöcken des externen Speichers vor, so kann eine Traversierung dieser Elemente insgesamt N Speichertransfers benötigen, da ggf. bei jedem Zugriff des Prozessors auf ein Datenelement ein Speichertransfer verursacht wird. Liegen die Elemente jedoch zusammenhängend im externen Speicher vor, so kann eine Traversierung dieser Elemente im *cache-oblivious*-Modell analog zum I/O-Modell mit $\Theta(\frac{N}{B})$ Speichertransfers durchgeführt werden, vgl. Abbildung 3.2:

3.2.1 Theorem. *Das Traversieren von N Datenelementen, die in einem zusammenhängenden Bereich des externen Speichers abgelegt sind, verursacht im cache-oblivious-Modell mindestens $\lceil \frac{N}{B} \rceil$ und höchstens $\lceil \frac{N}{B} \rceil + 1$ Speichertransfers.*

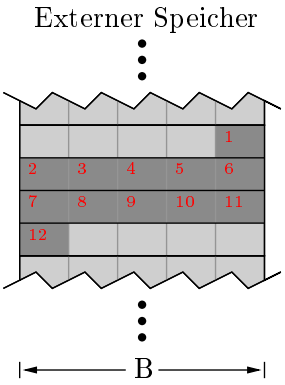


Abbildung 3.2: Traversierung von zusammenhängend im externen Speicher abgelegten Datenelementen im *cache-oblivious*-Modell. Die Traversierung der zwölf Elemente verursacht höchstens $\lceil 12/5 \rceil + 1 = 4$ Speichertransfers.

Beweis. Der zusammenhängende Bereich belegt eine endliche Anzahl von Blöcken im externen Speicher. Gilt $\frac{N}{B} \in \mathbb{N}$, so werden zur Speicherung der Elemente im optimalen Fall genau $\frac{N}{B} = \lceil \frac{N}{B} \rceil$ und im ungünstigsten Fall genau $\frac{N}{B} + 1 = \lceil \frac{N}{B} \rceil + 1$ Blöcke zur Speicherung benötigt. Gilt $\frac{N}{B} \notin \mathbb{N}$, so werden mindestens $\lceil \frac{N}{B} \rceil$ und höchstens $\lceil \frac{N}{B} \rceil + 1$ Blöcke belegt. □

Aufgrund der genauen Kenntnis der Belegungen der Variablen M und B kann im I/O-Modell ein Array mit N Elementen in $\lceil N/B \rceil$ Speicherblöcken der Größe B abgelegt und somit mit maximal $\lceil N/B \rceil$ I/O-Operationen traversiert werden. Da die genauen Belegungen der beiden Variablen im *cache-oblivious*-Modell nicht bekannt sind, kann das Array in diesem Fall nicht bzgl. der Blockgröße ausgerichtet werden, wodurch insgesamt ein zusätzlicher Speichertransfer verursacht werden kann.

3.2.2 Sortieren

Das Sortieren von Datenelementen ist eines der am meisten untersuchten Probleme in der Informatik und stellt sich im Kontext des *cache-oblivious*-Modells in vielen Fällen als Hauptkomponente effizienter Algorithmen heraus. Aggarwal und Vitter [2] zeigten, dass für das vergleichsbasierte Sortieren von N Elementen im I/O-Modell $\Theta(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/O-Operationen benötigt werden. Da Algorithmen im I/O-Modell die Blocktransfers explizit kontrollieren und somit die Blocktransfers optimal ausnutzen können, überträgt sich die untere Schranke für das vergleichsbasierte Sortieren auch auf Algorithmen im *cache-oblivious*-Modell. Die ersten optimalen *cache-oblivious*-Sortieralgorithmen wurden von Frigo *et al.* entwickelt: Einer dieser (beiden) Sortieralgorithmen, genannt *funnelsort*, kann als Variante eines *mergesort*-Verfahrens gesehen werden. Der andere Algorithmus ist eine Adaption des *distributionsort*-Verfahrens. Für beide Algorithmen wird die Annahme eines großen internen Speichers der Form $M \in \Omega(B^2)$ getroffen.

Eine vereinfachte Version des *funnelsort*-Verfahrens wurde von Brodal und Fagerberg [20] vorgestellt. Dieses Verfahren, genannt *lazy funnelsort*, trifft eine Annahme der Gestalt $M \in \Omega(B^{1+\varepsilon})$, wobei ε eine beliebig kleine positive Konstante ist. Brodal und Fagerberg zeigten weiterhin, dass das vergleichsbasierte Sortieren von Elementen ohne Annahme eines großen Speichers nicht möglich ist [22].

Es wird nun zunächst ein im *cache-oblivious*-Modell suboptimales Sortierverfahren vorgestellt und analysiert. Obwohl dieses Verfahren auf dem *divide and conquer*-Prinzip basiert, stellt sich heraus, dass durch Anwendung dieses Verfahrens eine suboptimale Anzahl von Speichertransfers verursacht wird. Das *divide and conquer*-Prinzip muss also nicht zwangsweise zu optimalen *cache-oblivious*-Algorithmen führen. Anschließend wird das von Brodal und Fagerberg [20] vorgestellte optimale *lazy funnelsort*-Verfahren vorgestellt und analysiert.

Zwei-Wege-*Mergesort*

Unter dem Begriff *mergesort* (Sortieren durch Verschmelzen) wird eine Gruppe von Sortieralgorithmen zusammengefasst, die alle nach dem gleichen Prinzip funktionieren: Um ein Array mit N Elementen zu sortieren, teilen *mergesort*-Algorithmen dieses zunächst in Teilarrays auf, welche anschließend rekursiv sortiert und nach Abarbeitung aller rekursiven Aufrufe zu einem sortierten Array zusammengefügt werden. Detaillierte Beschreibungen verschiedener *mergesort*-Algorithmen finden sich z.B. bei Ottmann und Widmayer [51]. Einer der bekanntesten *mergesort*-Algorithmen ist der *Zwei-Wege-mergesort*-Algorithmus, welcher nun beschrieben und im Kontext des *cache-oblivious*-Modells analysiert werden soll.

Algorithmus Durch den Zwei-Wege-*mergesort*-Algorithmus wird ein (unsortiertes) Array in zwei (etwa) gleich große Teilarrays aufgeteilt, welche anschließend rekursiv sortiert und nach Abarbeitung der rekursiven Aufrufe zu einem sortierten Array zusammengefügt werden. Eine mögliche Pseudocode-Darstellung dieses Verfahrens wird durch die Prozedur MERGESORT (Algorithmus 3.1) gegeben.

Die Prozedur MERGESORT verwendet die Hilfsprozedur MERGE(A, l, m, r), welche die beiden sortierten Teilarrays $A[l..m]$ und $A[m+1..r]$ zu einem sortierten Teilarray vereint und anschließend das ursprüngliche Teilarray $A[l..r]$ durch dieses neue ersetzt.

Analyse Die Laufzeit des Zwei-Wege-*Mergesort*-Verfahrens bei Anwendung auf ein Array der Länge N beträgt im RAM-Modell offensichtlich $\Theta(N \log N)$. Obwohl dieses Verfahren auf dem *divide and conquer*-Prinzip basiert, verursacht es bei Anwendung eine im Kontext des *cache-oblivious*-Modells suboptimale Anzahl an Speichertransfers:

Prozedur MERGESORT(A, l, r)

Eingabe: Ein Array A mit Gesamtlänge N und zwei natürliche Zahlen l, r mit $0 \leq l \leq r \leq N-1$.

1: **if** $l < r$ **then**
2: $m = \lfloor (l+r)/2 \rfloor$
3: MERGESORT(A, l, m)
4: MERGESORT$(A, m+1, r)$
5: MERGE(A, l, m, r)
6: **end if**

Algorithmus 3.1: Das Zwei-Wege-*Mergesort*-Sortierverfahren, vgl. [51, 53]

3.2.2 Lemma ([34, 53]). *Unter der Annahme von $M \geq 3B$ verursacht die Anwendung der Prozedur* MERGESORT *auf ein Array A der Länge N insgesamt $\Theta\left(\frac{N}{B} \log \frac{N}{B}\right)$ Speichertransfers.*

Beweis. Die Hilfsprozedur MERGE kann aufgrund der Mindestgröße des internen Speichers zwei Teilarrays effizient zu einem (sortierten) Array verschmelzen. Weiterhin lassen sich die durch Anwendung der Prozedur MERGESORT auf ein N-elementiges Array verursachten Speichertransfers durch die Rekurrenz

$$T(N) = \begin{cases} \Theta(1) & \text{für } N \leq B, \\ 2 \cdot T(N/2) + \Theta(N/B) & \text{sonst,} \end{cases}$$

beschreiben: Gilt $N \leq B$, so müssen nur konstant viele Blöcke in den internen Speicher geladen werden. Ansonsten teilt die Prozedur MERGESORT das Array in zwei Teilarrays der Größe $N/2$ auf und sortiert diese rekursiv. Nach Abarbeitung des zweiten rekursiven Aufrufs (Zeile 4) sind aufgrund der im *cache-oblivious*-Modell angenommenen optimalen Ersetzungsstrategie die meisten der Blöcke des ersten rekursiven Aufrufs (Zeile 3) aus dem internen Speicher verdrängt worden. Diese $\Theta(N)$ Elemente müssen während der Verschmelzung (Zeile 5) wieder in den internen Speicher geladen werden, wodurch $\Theta(N/B)$ Speichertransfers verursacht werden.

Der durch die Rekurrenz induzierte Rekursionsbaum besitzt $\Theta(N/B)$ Blätter, deren Bearbeitung insgesamt $\Theta(N/B)$ Speichertransfers verursacht. Da die Anzahl der verursachten Speichertransfers auf allen anderen Ebenen ebenso jeweils $\Theta(N/B)$ beträgt und der Rekursionsbaum eine Tiefe von $\Theta(\log \frac{N}{B})$ aufweist, werden insgesamt $\Theta(\frac{N}{B} \log \frac{N}{B})$ Speichertransfers verursacht. □

Durch das obige *mergesort*-Verfahren wird also *nicht* die optimale obere Schranke von $\mathcal{O}(sort(N)) = \mathcal{O}(\frac{N}{B} \log_{M/B} \frac{N}{B})$ für die Anzahl der verursachten Speichertransfers erreicht.

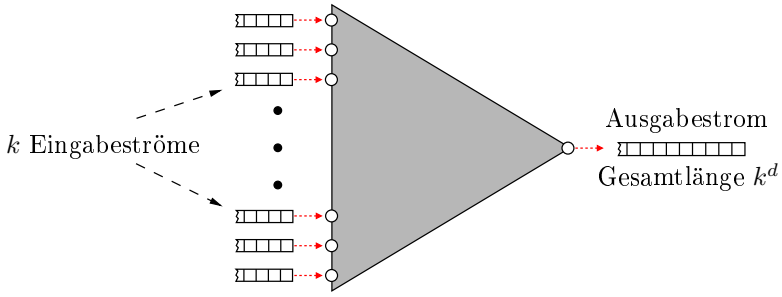


Abbildung 3.3: Verschmelzung von k (sortierten) Listen mit Gesamtgröße k^d ($d \geq 2$) zu einer (sortierten) Liste mit Hilfe eines k-Verschmelzers. Dieser besitzt einen Speicherplatzbedarf von $\Theta(k^{(d+1)/2})$ und verursacht pro Anwendung $\mathcal{O}(\frac{k^d}{B} \log_M(k^d) + k)$ Speichertransfers.

Lazy Funnelsort

Es wird nun das von Brodal und Fagerberg [20] entwickelte *lazy funnelsort*-Verfahren beschrieben. Die zentrale Idee dieses Verfahrens basiert auf einer statischen Datenstruktur namens k-Verschmelzer (*k-merger*). In Abschnitt 3.3 wird diese Datenstruktur genauer erläutert. Zunächst soll ein k-Verschmelzer jedoch als eine Black Box mit k Eingängen und einem Ausgang aufgefasst werden, welche bei Anwendung die Elemente aus k sortierten Eingabeströmen mit Gesamtgröße k^d ($d \geq 2$) zu einem sortierten Ausgabestrom vereint. Dazu werden die k Eingabeströme zu Beginn einer Anwendung mit den k Eingängen des k-Verschmelzers verknüpft. Der Ausgang liefert dann während der Anwendung einen sortierten Ausgabestrom, welcher aus den Elementen der vereinten Eingabeströme besteht, vgl. Abbildung 3.3. In Hinblick auf die Beschreibung des *lazy funnelsort*-Verfahrens werden noch die folgenden Eigenschaften eines k-Verschmelzers benötigt: Für den Speicherplatzbedarf $S(k)$ eines k-Verschmelzers gilt

$$k^{(d+1)/2} \leq S(k) \leq c \cdot k^{(d+1)/2},$$

wobei $c > 1$ eine reelle Konstante ist. Zu beachten ist hierbei, dass der Speicherplatzbedarf für die Eingabeströme und für den Ausgabestrom *nicht* in $S(k)$ enthalten ist. Weiterhin verursacht ein k-Verschmelzer unter der Annahme von

$$B^{(d+1)/(d-1)} \leq M/3c$$

pro Anwendung höchstens $\mathcal{O}(\frac{k^d}{B} \log_M(k^d) + k)$ Speichertransfers.

Algorithmus Um N in einem Array gespeicherte Elemente zu sortieren, kann ein N-Verschmelzer auf die N einelementigen Teilarrays angewendet werden. Wie Brodal und Fagerberg [20] bemerken, ist diese (direkte) Anwendung eines N-Verschmelzers nicht effizient: Da ein N-Verschmelzer mindestens $N^{(d+1)/2}$ Speicherplatz benötigt, würde diese Vorgehensweise zu einem Algorithmus mit nicht-linearem Speicherplatzbedarf führen. Weiterhin

sind k-Verschmelzer auch nur dann effizient, wenn die k Eingabeströme eine Gesamtgröße von k^d aufweisen. Die N einelementigen Teilarrays besitzen jedoch zusammen nur eine Gesamtgröße von N ($< N^d$).

Das von Brodal und Fagerberg [20] effiziente *lazy funnelsort*-Verfahren zur Sortierung eines Arrays mit N Elementen hat (deshalb) den folgenden Aufbau: Zunächst wird das Array in $N^{1/d}$ zusammenhängende Segmente der Größe $N^{1-1/d}$ aufgeteilt. Die Segmente werden anschließend rekursiv sortiert und nach Abarbeitung aller rekursiven Aufrufe mit Hilfe eines $N^{1/d}$-Verschmelzers wieder zu einem sortierten Array zusammengefügt. In der Prozedur LAZY-FUNNELSORT (Algorithmus 3.2) wird eine Pseudocode-Darstellung für diese Vorgehensweise gegeben. Da der kleinstmögliche k-Verschmelzer ein 2-Verschmelzer ist und dieser bei Anwendung 2^d Elemente ausgibt, bricht die Rekursion ab, sobald eine Liste weniger als 2^d Elemente besitzt, vgl. Ronn [54].

Prozedur LAZY-FUNNELSORT(A)

Eingabe: Ein Array A mit N Elementen.

1: **if** $N < 2^d$ **then**
2: Sortiere die N Elemente mit Hilfe eines beliebigen Sortierverfahrens.
3: **else**
4: Teile das Array in $N^{1/d}$ zusammenhängende Segmente der Größe $N^{1-1/d}$ auf. Seien $L_1, \ldots, L_{N^{1/d}}$ die entstehenden Segmente.
5: **for** $i = 1$ to $N^{1/d}$ **do**
6: LAZY-FUNNELSORT(L_i)
7: **end for**
8: Verschmelze die (sortierten) Segmente $L_1, \ldots, L_{N^{1/d}}$ mit Hilfe eines $N^{1/d}$-Verschmelzers.
9: **end if**

Algorithmus 3.2: Das *lazy funnelsort*-Verfahren, vgl. [20, 54]

Analyse Besteht das Eingabearray der Prozedur aus weniger als 2^d Elementen, so wird dieses in Zeile 2 mit Hilfe eines beliebigen Sortierverfahrens (z. B. mit dem obigen Zwei-Wege-*mergesort*-Verfahren) sortiert. Ansonsten findet eine Aufteilung der Liste in $N^{1/d}$ Segmente gleicher Größe statt (Zeile 4). Die Segmente werden anschließend rekursiv bearbeitet (Zeile 6) und nach Abarbeitung aller rekursiven Aufrufe mit Hilfe eines $N^{1/d}$-Verschmelzers wieder zu einem sortierten Array verschmolzen (Zeile 8). Die Korrektheit dieses Verfahrens ist somit offensichtlich.

Für die Anwendung des $N^{1/d}$-Verschmelzers in Zeile 8 wird zusätzlicher Speicherplatz benötigt. Da nach Voraussetzung $d \geq 2$ gilt, folgt für den Speicherplatzbedarf $S(N^{1/d})$ des $N^{1/d}$-Verschmelzers

$$S(N^{1/d}) \leq c \cdot N^{(d+1)/2d} < c \cdot N.$$

Der $N^{1/d}$-Verschmelzer benötigt also sublinear viel zusätzlichen Speicherplatz. Da weiterhin zu jedem Zeitpunkt nur ein k-Verschmelzer aktiv ist und der $N^{1/d}$-Verschmelzer den größten der involvierten Verschmelzer darstellt, reicht dieser zusätzliche Speicherplatz für die Anwendungen aller Verschmelzer aus. Um die sortierten Teilsegmente (Zeile 6) zwischenzuspeichern, wird zudem ein globales temporäres Array linearer Größe benötigt. Insgesamt besitzt das Verfahren also einen linearen Speicherplatzbedarf.

Da jeder Sortieralgorithmus im *cache-oblivious*-Modell mindestens $\Omega(\frac{N}{B}\log_{M/B}\frac{N}{B})$ Speichertransfers verursacht, wird die hinsichtlich der verursachten Speichertransfers optimale Arbeitsweise des obigen Verfahrens durch das folgende Theorem belegt:

3.2.3 Theorem ([20]). *Seien $d \geq 2$ und $c > 1$ wie oben gewählt. Unter der Annahme von $B^{(d+1)/(d-1)} \leq M/3c$ verursacht die Anwendung der Prozedur* Lazy-Funnelsort *auf ein Array A mit N Elementen höchstens $\mathcal{O}\left(d \cdot \frac{N}{B}\log_{M/B}\frac{N}{B}\right) = \mathcal{O}(sort(N))$ Speichertransfers.*

Beweis. Die N Eingabeelemente werden durch das Verfahren in $N^{1/d}$ Segmente der Größe $N^{1-1/d}$ eingeteilt, welche anschließend rekursiv sortiert und nach Beendigung aller rekursiven Aufrufe mit Hilfe eines $N^{1/d}$-Verschmelzers zu einem sortierten Array verschmolzen werden. Die sortierten Teilsegmente werden dabei in einem globalen Array der Größe N zwischengespeichert. Für den Ausgabestrom des $N^{1/d}$-Verschmelzers kann dann der ursprüngliche Speicherplatz der N Elemente verwendet werden.

Da für den Speicherplatzbedarf $S(k)$ eines k-Verschmelzers $k^{(d+1)/2} \leq S(k) \leq c{\cdot}k^{(d+1)/2}$ gilt und dieser bei Anwendung k^d Elemente ausgibt, ist $S(k)$ linear in der Anzahl der verschmolzenen Elemente. Der (gesamte) Speicherplatzbedarf eines rekursiven Aufrufs der Prozedur Lazy-Funnelsort mit Rekursionstiefe i setzt sich aus dem Speicherplatzbedarf für die Eingabeelemente, für die Zwischenergebnisse und für den $N^{(1-1/d)^i\cdot 1/d}$-Verschmelzer zusammen und lässt sich weiterhin aufgrund obiger Bemerkungen wie folgt abschätzen:

$$\begin{aligned} N^{(1-1/d)^i} + N^{(1-1/d)^i} + S(N^{(1-1/d)^i\cdot 1/d}) &\leq 2\cdot N^{(1-1/d)^i} + c\cdot N^{(1-1/d)^i} \\ &= (2+c)\cdot N^{(1-1/d)^i} \end{aligned}$$

Da die Größe der Segmente mit zunehmender Rekursionstiefe sukzessiv abnimmt, gilt ab einer gewissen Rekursionstiefe l

$$(2+c)\cdot N^{(1-1/d)^l} < M.$$

Demnach passen die Elemente des (rekursiv zu sortierenden) Segments, das Array für die Zwischenergebnisse und der zugehörige k-Verschmelzer ab dieser Rekursionstiefe gemeinsam in den internen Speicher. Da bei der rekursiven Sortierung der $N^{(1-1/d)^l}$ Elemente immer nur ein k-Verschmelzer aktiv ist und der $N^{(1-1/d)^l\cdot 1/d}$-Verschmelzer den größten dieser k-Verschmelzer dargestellt, müssen alle involvierten Blöcke im Zuge der restlichen Rekursion jeweils nur einmal in den internen Speicher geladen werden. Werden also *pro*

Element amortisierte $\mathcal{O}\left(\frac{1}{B}\right)$ Speichertransfers veranschlagt, so sind die Kosten für alle rekursiven Aufrufe mit Rekursionstiefe $\geq l$ gedeckt.

Für die auf den höheren Rekursionsebenen verwendeten k-Verschmelzer gilt $k^d \geq \frac{1}{2+c}M$ und somit

$$k^d \geq \frac{3c}{2+c} \cdot B^{(d+1)/(d-1)}.$$

Die letzte Ungleichung impliziert

$$k^{d-1} \geq y \cdot B^{(d+1)/d} > y \cdot B$$

mit $y = \left(\frac{3c}{2+c}\right)^{\frac{d-1}{d}}$, woraus schließlich $k^d/B > y \cdot k$ folgt. Die auf den höheren Rekursionsebenen verwendeten k-Verschmelzer verursachen demnach bei Anwendung höchstens

$$\mathcal{O}\left(\frac{k^d}{B}\log_M k^d + k\right) = \mathcal{O}\left(\frac{k^d}{B}\log_M k^d\right)$$

Speichertransfers, was $\mathcal{O}\left(\frac{1}{B} \cdot \log_M k^d\right)$ amortisierten Speichertransfers *pro* Element entspricht. Weiterhin verursacht sowohl das Lesen der Eingabelemente eines rekursiven Aufrufs als auch das Schreiben des sortierten Segments in das globale Array $\mathcal{O}\left(\frac{1}{B}\right)$ amortisierte Speichertransfers *pro* Element. Insgesamt werden deshalb durch alle rekursiven Aufrufe *pro* Element höchstens

$$\mathcal{O}\left(\frac{1}{B}\left(1 + \sum_{i=0}^{\infty} \log_M N^{(1-1/d)^i}\right)\right) = \mathcal{O}\left(d \cdot \frac{\log_M N}{B}\right)$$

amortisierte Speichertransfers verursacht. Für die Bearbeitung aller Elemente werden somit höchstens $\mathcal{O}\left(d \cdot \frac{N}{B}\log_M N\right)$ Speichertransfers benötigt.

Es bleibt noch $\mathcal{O}\left(d \cdot \frac{N}{B}\log_M N\right) = \mathcal{O}\left(d \cdot \frac{N}{B}\log_{\frac{M}{B}} \frac{N}{B}\right)$ zu zeigen. Da nach Voraussetzung $M \geq 3c \cdot B^{(d+1)/(d-1)}$ gilt, folgt

$$\log M \geq \log\left(3c \cdot B^{(d+1)/(d-1)}\right) = \underbrace{\frac{d+1}{d-1}}_{>1}\log B + \log 3c$$

und somit

$$\log_{\frac{M}{B}} N = \frac{\log N}{\log M - \log B} \in \frac{\log N}{\Theta(\log M)} = \Theta\left(\log_M N\right).$$

Der Logarithmus zur Basis M entspricht also in diesem Fall dem Logarithmus zur Basis M/B, vgl. Demaine [34]. Mit

$$\log N \geq \log M \geq \underbrace{\frac{d+1}{d-1}}_{>1}\log B + \log 3c$$

folgt dann analog zu obiger Argumentation

$$\begin{aligned}
\mathcal{O}\left(\frac{N}{B}\log_{\frac{M}{B}} \frac{N}{B}\right) &= \mathcal{O}\left(\frac{N}{B}\log_{\frac{M}{B}} N - \frac{N}{B}\log_{\frac{M}{B}} B\right) \\
&= \mathcal{O}\left(\frac{N}{B}\log_{\frac{M}{B}} N\right) \\
&= \mathcal{O}\left(\frac{N}{B}\log_M N\right).
\end{aligned}$$

□

Der Kerngedanke der obigen Analyse besteht also darin, dass ab einer gewissen Rekursionstiefe der Speicherplatzbedarf für die Eingabeelemente, den Hilfsspeicher und den zugehörigen k-Verschmelzer hinreichend klein ist, so dass alle involvierten Blöcke im Zuge der restlichen rekursiven Aufrufe nur einmal in den internen Speicher geladen werden müssen. Für die Bearbeitung der Segmente auf den höheren Rekursionsebenen kann $k^d/B > y \cdot k$ ($y > 0$) gezeigt und somit die Anzahl der durch Anwendung eines k-Verschmelzers verursachten Speichertransfers mittels der oberen Schranke $\mathcal{O}\left(\frac{k^d}{B} \log_M k^d\right)$ begrenzt werden, was letztendlich zum obigen Ergebnis führt.

Parameter d Im obigen Verfahren kann durch Wahl der Konstanten d die Anforderung an die Größe des internen Speichers variiert werden. In der Literatur [8, 34, 38] wird oft $d = 3$ gewählt und somit eine Annahme der Form $M \in \Omega\left(B^2\right)$ getroffen. Für $d > 3$ ergibt sich zwar eine schwächere Bedingung der Gestalt $M \in \Omega(B^{1+\varepsilon})$ mit $\varepsilon \in (0, 1)$, jedoch verursacht das obige Verfahren dann um einen Faktor aus $\Theta\left(\frac{1}{\varepsilon}\right)$ mehr Speichertransfers als in dem Fall $M \in \Omega(B^2)$. Die Wahl der Konstanten d stellt also einen Kompromiss zwischen der Anforderung an den internen Speicher und dem konstanten Vorfaktor in der (in allen Fällen asymptotisch optimalen) Grenze für die verursachten Speichertransfers dar. Wie Brodal und Fagerberg [22] bemerken, ist dieser Kompromiss der bestmögliche für das vergleichsbasierte Sortieren.

3.3 Datenstrukturen

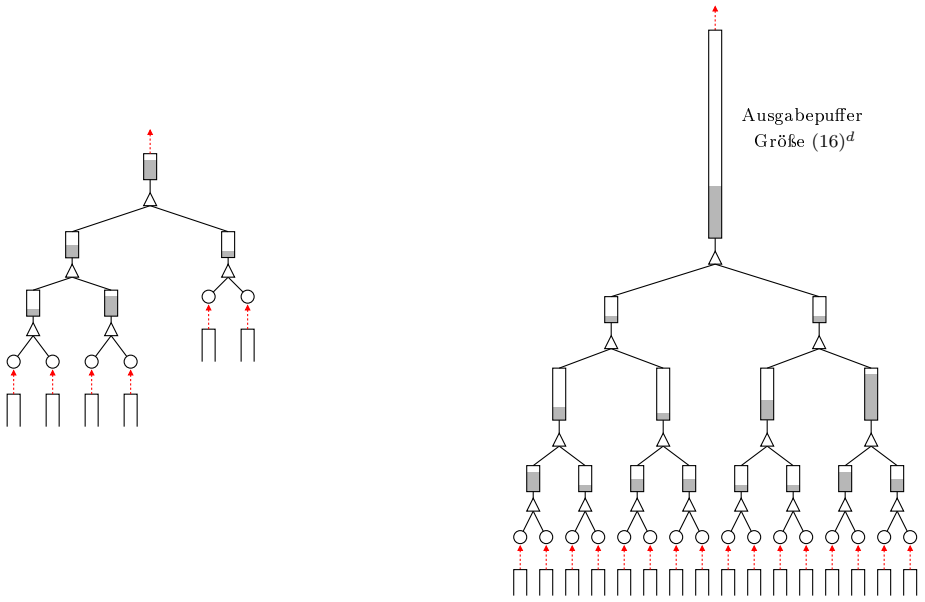
In diesem Abschnitt werden zwei *cache-oblivious*-Datenstrukturen genauer erläutert. Die erste ist die weiter oben schon angesprochene Struktur eines k-Verschmelzers. Die zweite Datenstruktur ist eine von Brodal und Fagerberg [21] entwickelte Prioritätswarteschlange namens *funnel heap*.

3.3.1 k-Verschmelzer

In den Arbeiten von Frigo *et al.* [38, 53] wird ein k-Verschmelzer rekursiv anhand von $\sqrt{k}$-Verschmelzern und Puffern definiert. Hier soll jedoch eine (vereinfachte) Version eines solchen k-Verschmelzers behandelt werden, die auf Brodal und Fagerberg [20] zurückgeht.

Struktur

Zur Definition eines k-Verschmelzers wird die Definition eines *binären Verschmelzers* (*binary merger*) benötigt. Ein binärer Verschmelzer vereint zwei sortierte Eingabeströme zu einem sortierten Ausgabestrom: Pro Verschmelzungsschritt wird ein Element vom Kopf eines der beiden Eingabeströme an das Ende des Ausgabestroms bewegt. Der Kopf beider Eingabeströme sowie das Ende des Ausgabestroms befinden sich jeweils in *Puffern*, die



(a) Binärer Verschmelzungsbaum (b) 16-Verschmelzer

Abbildung 3.4: In Abbildung (a) wird ein aus fünf binären Verschmelzern zusammengesetzter binärer Verschmelzungsbaum dargestellt. Abbildung (b) zeigt einen 16-Verschmelzer ($d \geq 2$), vgl. [8]. Die Größen der Puffer unterscheiden sich dabei je nach Rekursionstiefe.

eine *begrenzte* Anzahl von Elementen aufnehmen können. Ein Puffer (*buffer*) ist ein Array mit zusätzlichen Feldern, die die Größe des Arrays und Zeiger auf das erste und das letzte Element des Arrays speichern.

Binäre Verschmelzer können zu *binären Verschmelzungsbäumen* (*binary merge trees*) [8] kombiniert werden, um mehrere Eingabeströme gleichzeitig zu vereinen. Ein binärer Verschmelzungsbaum ist ein Binärbaum, dessen innere Knoten mit binären Verschmelzern korrespondieren. Mit Ausnahme der Wurzel bildet der Ausgabestrom eines inneren Knotens einen der beiden Eingabeströme seines Vaterknotens. Der Ausgabestrom des Wurzelknotens stellt den Ausgabestrom des gesamten Verschmelzungsbaums dar. Jede Kante zwischen zwei inneren Knoten korrespondiert demnach mit einem Ausgabepuffer. Zusätzlich besitzt ein Verschmelzungsbaum einen mit seinem Wurzelknoten korrespondierenden Ausgabepuffer. Die zu verschmelzenden Elemente sind in (externen) Eingabeströmen enthalten, welche mit den Blättern des Verschmelzungsbaums verknüpft werden, vgl. Abbildung 3.4 (a).

Ein *k-Verschmelzer* (*k-merger*) [20] ist ein binärer Verschmelzungsbaum, dessen Puffer spezielle Größen besitzen. Wie auch in anderen Arbeiten [8, 20, 21] wird der Einfachheit halber angenommen, dass k von der Form $k = 2^j$ für eine positive ganze Zahl j ist. Ein k-Verschmelzer ist somit ein vollständiger Binärbaum mit $k-1$ binären Verschmelzern, vgl. Abbildung 3.4 (b). Der Ausgabepuffer des Wurzelknotens eines k-Verschmelzers T besitzt

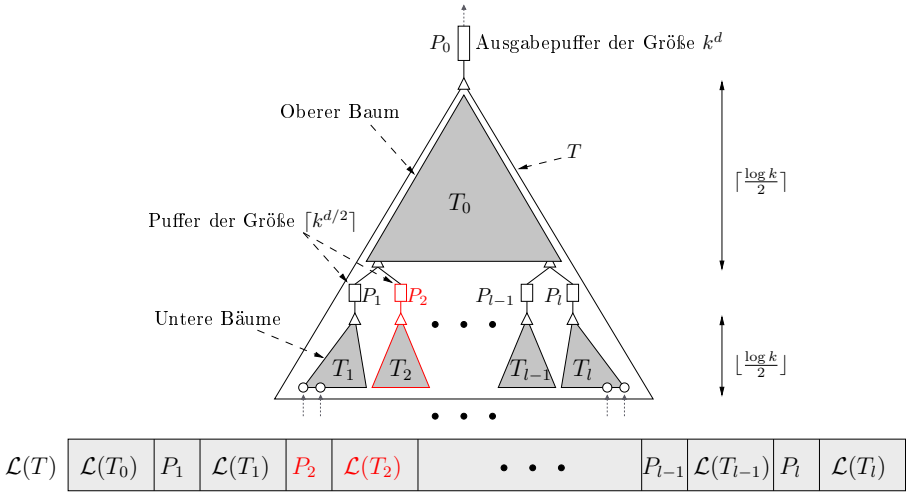


Abbildung 3.5: Rekursive Definition der Puffergrößen und Speicherplatzbelegung eines k-Verschmelzers T, vgl. [8, 34, 54]. Die Puffer zwischen dem oberen Teilbaum und den $l = 2^{\lceil \log k^{1/2} \rceil}$ unteren Teilbäumen besitzen eine Größe von $\lceil k^{d/2} \rceil$. Ein unterer Teilbaum wird gemeinsam mit seinem Ausgabepuffer (falls ein solcher vorhanden ist – ein Blatt besitzt keinen Ausgabepuffer) im Speicher abgelegt. Der Ausgabepuffer P_0 wird separat gespeichert.

eine Größe von k^d. Die Größen der (restlichen) internen Puffer von T werden rekursiv definiert: Sei $i = \log k$ die Tiefe von T. Der *obere Baum* von T ist derjenige Teilbaum, dessen Knoten alle eine Tiefe von maximal $\lceil i/2 \rceil$ besitzen. Die *unteren Bäume* von T sind die in den Knoten mit Tiefe $\lceil i/2 \rceil + 1$ gewurzelten Teilbäume von T. Die Größe der Puffer zwischen dem oberen und den unteren Bäumen beträgt $\lceil k^{d/2} \rceil$. Innerhalb der Teilbäume werden die Größen der Puffer durch Rekursion definiert, vgl. Abbildung 3.5.

Die Speicherplatzbelegung $\mathcal{L}(T)$ eines k-Verschmelzers T wird ebenso rekursiv anhand der obigen Teilbäume definiert: Besteht T aus nur einem Knoten, so wird dieser ohne seine Eingabepuffer und ohne seinen Ausgabepuffer im Speicher abgelegt. Ansonsten seien T_0 der obere und $T_1, \ldots, T_l$ die unteren Bäume von T ($l = 2^{\lceil \log k^{1/2} \rceil}$). Die Speicherplatzbelegung $\mathcal{L}(T)$ von T besteht dann aus der Speicherplatzbelegung $\mathcal{L}(T_0)$ von T_0 gefolgt von den Speicherplatzbelegungen $\mathcal{L}(T_1), \ldots, \mathcal{L}(T_l)$ der unteren Bäume. Der Ausgabepuffer eines unteren Teilbaums T_i (falls ein solcher vorhanden ist) wird dabei vor der Speicherplatzbelegung $\mathcal{L}(T_i)$ zusammenhängend mit dieser im Speicher abgelegt, vgl. Abbildung 3.5. Der Ausgabepuffer des Wurzelknotens gehört *nicht* zur Speicherplatzbelegung des k-Verschmelzers. Dieser wird separat gespeichert und kann als Ausgabemechanismus für den Ausgabestrom des k-Verschmelzers gesehen werden.

Ein k-Verschmelzer entspricht somit einem anhand des *van Emde Boas layouts* im Speicher abgelegten Binärbaums, bis auf den Unterschied, dass für jeden inneren Knoten mit Ausnahme des Wurzelknotens zusätzlich ein Puffer gespeichert wird und dass (bei

Prozedur FILL(v)

Eingabe: Ein innerer Knoten v eines Verschmelzungsbaums T.

1: **while** Der Ausgabepuffer von v nicht voll und mindestens einer der beiden Eingabeströme von v nicht aufgebraucht ist **do**
2: **if** Linker Eingabepuffer von v leer und linker Eingabestrom nicht aufgebraucht **then**
3: FILL(linkes Kind von v)
4: **end if**
5: **if** Rechter Eingabepuffer von v leer und rechter Eingabestrom nicht aufgebraucht **then**
6: FILL(rechtes Kind von v)
7: **end if**
8: **if** Mindestens einer der beiden Eingabepuffer von v ist nicht leer **then**
9: Führe einen Verschmelzungsschritt aus. Dadurch wird ein Element aus einem der beiden Eingabepuffer von v in den Ausgabepuffer von v bewegt.
10: **end if**
11: **end while**

Algorithmus 3.3: Die Verschmelzungsprozedur, vgl. [20, 21]

Betrachtung ungerundeter Werte) der obere Teilbaum eine größere Höhe als ein unterer Teilbaum besitzt.

Anwendung eines Verschmelzungsbaums

Mit Hilfe der Prozedur FILL (Algorithmus 3.3) können die Eingabeströme eines Verschmelzungsbaums T vereint werden. Allgemein führt die Anwendung der Prozedur auf einen inneren Knoten v von T solange zu rekursiven Aufrufen, bis entweder der Ausgabepuffer von v voll ist, oder beide Eingabeströme von v aufgebraucht sind. Wird die Prozedur auf den Wurzelknoten des Verschmelzungsbaums T (dessen Pufferspeicher zu Beginn leer sind) angewendet, so wird dadurch ein Ausgabestrom produziert, der aus den Elementen der verschmolzenen mit dem k-Verschmelzer verknüpften Eingabeströme besteht. Bei einem k-Verschmelzer besitzt dieser Ausgabestrom (pro Anwendung der Prozedur FILL auf seinen Wurzelknoten) eine Länge von k^d.

Wie Brodal und Fagerberg [20] bemerken, besteht der Unterschied zu dem von Frigo *et al.* [38] vorgestellten k-Verschmelzer darin, dass die Puffer nicht auf ihren Inhalt überprüft und anschließend ggf. gefüllt werden müssen. Bei der hier vorgestellten Variante eines k-Verschmelzers wird durch die (rekursive) Ausführung der Prozedur FILL ein Puffer einfach wieder aufgefüllt, sobald das letzte Element aus diesem entnommen wurde und der entsprechende Eingabestrom noch nicht aufgebraucht ist.

Analyse

Es wird nun zunächst die Speicherplatzkomplexität eines k-Verschmelzers bestimmt, aus der anschließend die Anzahl der pro Anwendung eines solchen verursachten Speichertransfers hergeleitet wird.

Speicherplatzbedarf Der von einem k-Verschmelzer belegte Speicherplatz besteht aus dem (rekursiv definierten) Speicherplatz für den oberen und die unteren Bäume des k-Verschmelzers und dem Speicherplatz für die Pufferspeicher zwischen dem oberen und den unteren Bäumen. Analog zu Brodal und Fagerberg [20] werden im Folgenden die Werte für die Größen dieser Teilbäume gerundet, d. h. sowohl der obere als auch die unteren Bäume eines k-Verschmelzers als $k^{1/2}$-Verschmelzer angesehen. Der insgesamt benötigte Speicherplatz eines k-Verschmelzers wird durch das folgende Lemma analysiert:

3.3.1 Lemma ([20]). *Sei $d \geq 2$ und bezeichne $S(k)$ den durch einen k-Verschmelzer belegten Speicherplatz. Dann existiert eine Konstante $c > 1$, so dass*

$$k^{(d+1)/2} \leq S(k) \leq c \cdot k^{(d+1)/2}$$

für alle $k \in \mathbb{N}$ erfüllt ist.

Beweis. Da ein k-Verschmelzer aus $k^{1/2}$ Puffern der Größe $k^{d/2}$ und $(k^{1/2}+1)$ rekursiv definierten Teilbäumen der Größe $S(k^{1/2})$ besteht, kann der Speicherplatzbedarf $S(k)$ eines solchen durch die Rekurrenz

$$S(k) = k^{1/2} \cdot k^{d/2} + (k^{1/2}+1) \cdot S(k^{1/2})$$

beschrieben werden.[2] Diese lässt sich mit Hilfe der Substitutionsmethode [32] wie folgt auflösen: Sei $c > 1$ eine beliebige fest gewählte Konstante aus $\mathbb{R}$. Unter der Annahme von $S(k) \leq c \cdot k^{\frac{d+1}{2}}$ gilt

$$\begin{aligned} S(k) &= k^{1/2} \cdot k^{d/2} + (k^{1/2}+1) \cdot S(k^{1/2}) \\ &\leq k^{\frac{d+1}{2}} + (k^{1/2}+1) \cdot c \cdot k^{\frac{d+1}{4}} \\ &= c \cdot k^{\frac{d+1}{2}} \underbrace{-(c-1) \cdot k^{\frac{d+1}{2}} + c \cdot (k^{1/2}+1) \cdot k^{\frac{d+1}{4}}}_{\leq 0 \text{ für alle } k \text{ hinreichend groß}}. \end{aligned}$$

Der hintere Term ist für alle hinreichend großen k kleiner oder gleich 0: Da $\frac{d+1}{2} > \frac{d+3}{4} \Leftrightarrow d > 1$ nach Wahl von d erfüllt ist, wächst die Funktion $k \mapsto k^{\frac{d+1}{2}}$ asymptotisch echt schneller als die Funktion $k \mapsto k^{1/2} \cdot k^{\frac{d+1}{4}} = k^{\frac{d+3}{4}}$. Aufgrund von $(c-1) > 0$ existiert somit eine Konstante $k_0 \in \mathbb{N}$, so dass $(c-1) \cdot k^{\frac{d+1}{2}} \geq c \cdot (k^{1/2}+1) \cdot k^{\frac{d+1}{4}}$ und folglich

$$S(k) \leq c \cdot k^{\frac{d+1}{2}}$$

[2] Der Ausgabepuffer der Größe k^d gehört nach Definition nicht zur Speicherplatzbelegung des k-Verschmelzers dazu.

für alle $k \geq k_0$ erfüllt ist. Für die endlich vielen $k < k_0$ kann die Konstante c entsprechend vergrößert werden. Dies ergibt die Behauptung. □

Speichertransfers Das folgende Theorem analysiert die durch Anwendung der Prozedur FILL auf den Wurzelknoten eines k-Verschmelzers verursachten Speichertransfers. Für $d \geq 2$ und $c > 1$ wird dabei die Annahme

$$B^{(d+1)/(d-1)} \leq M/3c$$

getroffen, d. h. eine Mindestgröße des internen Speichers vorausgesetzt.

3.3.2 Theorem ([20]). *Sei $d \geq 2$ fest gewählt und c die Konstante aus Lemma 3.3.1. Unter der Annahme von $B^{(d+1)/(d-1)} \leq M/3c$ werden durch Anwendung der Prozedur* FILL *auf den Wurzelknoten eines k-Verschmelzers höchstens $\mathcal{O}\left(\frac{k^d}{B}\log_M(k^d) + k\right)$ Speichertransfers verursacht.*

Beweis. Um die verursachten Speichertransfers zu analysieren, wird die rekursive Definition der Puffergrößen eines k-Verschmelzers betrachtet. Je größer die Rekursionstiefe, desto geringer ist die Anzahl der in den entsprechenden Puffern Platz findenden Elemente und desto geringer ist der von den oberen und unteren Teilbäumen verwendete Speicherplatz. Sei nun diejenige Rekursionstiefe betrachtet, ab der diese Teilbäume (ohne die Ausgabepuffer der zugehörigen Wurzelknoten) jeweils weniger als $M/3$ Speicherplatz belegen, d. h. ab der

$$\bar{k}^{(d+1)/2} \leq M/3c$$

gilt, wobei $\bar{k}$ die Anzahl der Blätter eines solchen Teilbaums sei. Da $\bar{k}$ der erste Wert ist, für den

$$\bar{k}^{(d+1)/2} \leq M/3c$$

gilt, muss

$$(\bar{k}^2)^{(d+1)/2} > M/3c$$

und somit

$$\bar{k} > (M/3c)^{1/(d+1)}$$

erfüllt sein. Die Puffer, deren Größe durch Rekursion bis zur obigen Rekursionstiefe definiert ist, werden *große Puffer* (*large buffers*) genannt. Entfernt man die mit den großen Puffern korrespondierenden Kanten aus dem k-Verschmelzer, so wird dieser in Teilbäume zerlegt, welche als *Basisbäume* (*base trees*) bezeichnet werden. Nach Voraussetzung gilt $B^{(d+1)/(d-1)} \leq M/3c$ und somit

$$\begin{aligned} M/3c + B + \bar{k} \cdot B &\leq M/3c + M/3c + (M/3c)^{2/(d+1)} \cdot (M/3c)^{(d-1)/(d+1)} \\ &= M/c. \end{aligned}$$

Da $c > 1$ gilt, kann der interne Speicher folglich jeweils einen Basisbaum und einen Speicherblock für jeden der $\bar{k} + 1$ Puffer, die den Basisbaum umgeben, aufnehmen.

Ist der k-Verschmelzer selbst ein Basisbaum (gilt also $k^{(d+1)/2} \leq M/3c$), so passen dieser, ein Block des Ausgabestroms und jeweils ein Block pro Eingabestrom in den internen Speicher. Eine Anwendung der Prozedur FILL auf den Wurzelknoten füllt den Ausgabepuffer dieses Knotens mit k^d Elementen und verursacht bei diesem Vorgang höchstens $2(k^d/B) + k$ Speichertransfers: Da die Elemente nicht gleichmäßig auf die Eingabeströme verteilt sein müssen, verursacht das Einlesen der Elemente aus den Eingabeströmen insgesamt $k^d/B+k$ Speichertransfers. Der Transfer der Ausgabeelemente des k-Verschmelzers in den Ausgabepuffer seines Wurzelknotens verursacht weiterhin k^d/B Speichertransfers. Bei dem gesamten Vorgang werden also insgesamt höchstens $\mathcal{O}(k^d/B + k)$ Speichertransfers verursacht. Die Behauptung ist somit für diesen Fall erfüllt.

Ansonsten betrachte man die Zerlegung des k-Verschmelzers in die entsprechenden Basisbäume. Eine Anwendung der Prozedur FILL auf den Wurzelknoten v eines solchen Basisbaums füllt den Ausgabepuffer von v mit $\Omega(\bar{k}^d)$ Elementen (die Ausgabepuffer der Basisbäume können zwischen $\bar{k}^d$ und k^d Elemente aufnehmen). Unter der Annahme, dass kein direkt unter dem Basisbaum liegender Eingabepuffer geleert wird, verursacht die Abarbeitung dieses Prozeduraufrufs höchstens $\mathcal{O}\left(\bar{k}^d/B\right)$ Speichertransfers: Das Laden des Basisbaums und eines Speicherblocks für jeden der $\bar{k}$ Puffer direkt unter dem Basisbaum verursacht $\mathcal{O}\left(\bar{k}^{(d+1)/2}/B + \bar{k}\right)$ Speichertransfers. Dies entspricht $\mathcal{O}\left(1/B\right)$ amortisierten Speichertransfers pro ausgegebenen Element, da zum Einen aufgrund von $d \geq 2$

$$\bar{k}^{(d+1)/2}/B \leq \bar{k}^d/B$$

erfüllt ist und zum Anderen aus

$$\bar{k}^{d+1} > M/3c$$

unter Verwendung der Mindestgröße des internen Speichers

$$\bar{k}^{d-1} > (M/3c)^{(d-1)/(d+1)} \geq B$$

und somit

$$\bar{k} < \bar{k}^d/B$$

folgt. Die Kosten für das Einlesen der Elemente aus den Eingabeströmen des Basisbaums verursachen aufgrund obiger Abschätzung ebenso $\mathcal{O}\left(\bar{k}^d/B + \bar{k}\right) \subseteq \mathcal{O}\left(\bar{k^d}/B\right)$ Speichertransfers. Da die Kosten für das Schreiben der Elemente in den Ausgabepuffer trivialerweise $\mathcal{O}\left(\bar{k}^d/B\right)$ Speichertransfers betragen, werden also wie behauptet $\mathcal{O}(\frac{1}{B})$ amortisierte Speichertransfers pro Element verursacht.

Entleert sich ein Puffer direkt unter dem Basisbaum, so führt dies zu rekursiven Aufrufen der Prozedur FILL. Dadurch kann der Basisbaum aus dem Speicher verdrängt werden, infolgedessen dieser nach Abarbeitung der rekursiven Aufrufe evtl. erneut in den internen

Speicher geladen werden muss. Im Zuge der rekursiven Aufrufe wird jedoch der zuvor leere Puffer mit $\Omega(\bar{k}^d)$ Elementen gefüllt (mit Ausnahme des weiter unten beschriebenen Sonderfalls). Die Kosten für das erneute Laden des Basisbaums können somit diesen $\Omega(\bar{k}^d)$ Elementen zugeschrieben werden; jedes Element, welches in einen großen Puffer eingefügt wird, „zahlt“ also mit $\mathcal{O}(1/B)$ amortisierten Speichertransfers für das evtl. erneute Laden des darüberliegenden Basisbaums.

Der Ausnahmefall, dass ein großer Puffer nicht vollständig gefüllt wird, da der entsprechende Eingabestrom aufgebraucht ist, tritt pro großem Puffer höchstens einmal auf. Indem jeweils jede Position eines großen Puffers mit $\mathcal{O}(1/B)$ Speichertransfers belastet wird, sind die Kosten für das evtl. erneute Laden der entsprechenden Basisbäume in diesen Fällen ebenso gedeckt. Da die großen Puffer einen Teil des von einem k-Verschmelzer benötigten Speicherplatz ausmachen und dieser einen in der Anzahl der durch ihn ausgegebenen Elemente sublinearen Speicherplatz belegt, werden dadurch höchstens $\mathcal{O}(1/B)$ amortisierte Speichertransfers pro Element veranschlagt.

Indem also jedes Element mit $\mathcal{O}(1/B)$ amortisierten Speichertransfers belastet wird, sobald dieses in einen großen Puffer eingefügt wird, sind die Kosten für alle verursachten Speichertransfers gedeckt. Ein Basisbaum besitzt aufgrund von $\bar{k} > (M/3c)^{1/(d+1)}$ mindestens $F = (M/3c)^{1/(d+1)}$ Blätter. Jedes Element kann somit in höchstens

$$\begin{aligned}
\log_F k &= \frac{\log k}{\log \left(\frac{M}{3c}\right)^{1/(d+1)}} \\
&= (d+1) \cdot \frac{\log k}{\log(M/3c)} \\
&= (d+1) \cdot \log_M k \cdot \frac{\log M}{\log M - \log 3c}
\end{aligned}$$

große Puffer eingefügt werden. Da $\log M > \log 3c$ gilt, folgt $\log_F k \in \mathcal{O}(d \cdot \log_M k) = \mathcal{O}(\log_M k^d)$. Die (rekursive) Bearbeitung aller Elemente verursacht deshalb wie behauptet höchstens $\mathcal{O}(\frac{k^d}{B} \log_M(k^d) + k)$ Speichertransfers. □

3.3.2 Prioritätswarteschlangen

Eine *Prioritätswarteschlange* (*priority queue*) ist eine abstrakte Datenstruktur zur Verwaltung einer Menge von Elementen, denen jeweils eine Priorität (Schlüssel) zugeordnet ist. Prioritätswarteschlangen müssen dabei die Operationen INSERT und DELETEMIN ausführen können, wobei die INSERT-Operation ein neues Element in die Prioritätswarteschlange einfügt und die DELETEMIN-Operation das kleinste Element (also das Element mit dem kleinsten Schlüssel und somit größter Priorität) auffindet und aus der Prioritätswarteschlange entfernt. Gelegentlich wird noch eine DELETE-Operation von einer Prioritätswarteschlange unterstützt, welche ein beliebiges Element mit gegebenem Schlüssel aus der Prioritätswarteschlange entfernt.

Eine Prioritätswarteschlange kann im RAM-Modell effizient durch einen *heap* realisiert werden [49, 51]. Die Laufzeit einer INSERT- bzw. einer DELETEMIN-Operation beträgt in diesem Fall jeweils höchstens $\mathcal{O}(\log N)$, wobei N die Gesamtanzahl der Elemente des *heaps* angibt. Im I/O-Modell führt die sogenannte *buffer tree*-Technik von Arge [6] zu einer effizienten Implementierung einer Prioritätswarteschlange, bei der die beiden Operationen INSERT und DELETEMIN jeweils mit $\mathcal{O}\left(\frac{1}{B}\log_{M/B}\frac{N}{B}\right)$ amortisierten I/O-Operationen unterstützt werden. Wie Arge *et al.* [7, 8] bemerken, scheint es schwierig, diese Struktur an die Bedürfnisse des *cache-oblivious*-Modells anzupassen, da in ihr periodisch die $\Theta(M)$ kleinsten Elemente gesucht und im internen Speicher ablegt werden.

Im *cache-oblivious*-Modell impliziert die Existenz eines dynamischen *cache-oblivious* B-Baums [15] die Existenz einer *cache-oblivious* Prioritätswarteschlange, bei der die Operationen INSERT und DELETEMIN jeweils höchstens $\mathcal{O}(\log_B N)$ Speichertransfers verursachen, vgl. Arge *et al.* [7, 8]. Diese Grenze ist aber bei einer amortisierten Analyse der verursachten Speichertransfers suboptimal. Die optimale Grenze von $\mathcal{O}\left(\frac{1}{B}\log_{M/B}\frac{N}{B}\right)$ Speichertransfers für die Ausführung einer INSERT- bzw. DELETEMIN-Operation wurde zuerst durch eine von Arge *et al.* [7] entwickelte *cache-oblivious* Prioritätswarteschlange erreicht. Etwas später entwickelten Brodal und Fagerberg [21] eine *cache-oblivious* Prioritätswarteschlange, deren Hauptkomponente aus einem binären Verschmelzungsbaum besteht, welcher sich wiederum aus Puffern und k-Verschmelzern verschiedener Größen zusammensetzt.

Im Rest dieses Kapitels wird die von Brodal und Fagerberg [21] vorgestellte Prioritätswarteschlange, genannt *funnel heap*, vorgestellt. Dabei basieren die folgenden Ausführungen auf den Arbeiten von Brodal und Fagerberg [21] und Arge *et al.* [8].

Struktur

Die Hauptkomponente der *funnel heap*-Datenstruktur besteht aus einer Sequenz von k-Verschmelzern ($d = 3$), die mit Hilfe von binären Verschmelzern und Puffern zu einer verketteten Liste zusammengefügt werden, vgl. Abbildung 3.6. Dieser Teil der Datenstruktur stellt insgesamt einen einzigen binären Verschmelzungsbaum dar. Der zweite Teil besteht aus einem separaten Eingabepuffer I.

Die Details der Struktur werden induktiv mittels der wie folgt definierten Werte k_i und s_i ($i \geq 1$) gegeben:

$$\begin{aligned}(k_1, s_1) &= (2, 8)\\ s_{i+1} &= s_i \cdot (k_i + 1)\\ k_{i+1} &= \lceil\lceil {s_{i+1}}^{1/3} \rceil\rceil\end{aligned}$$

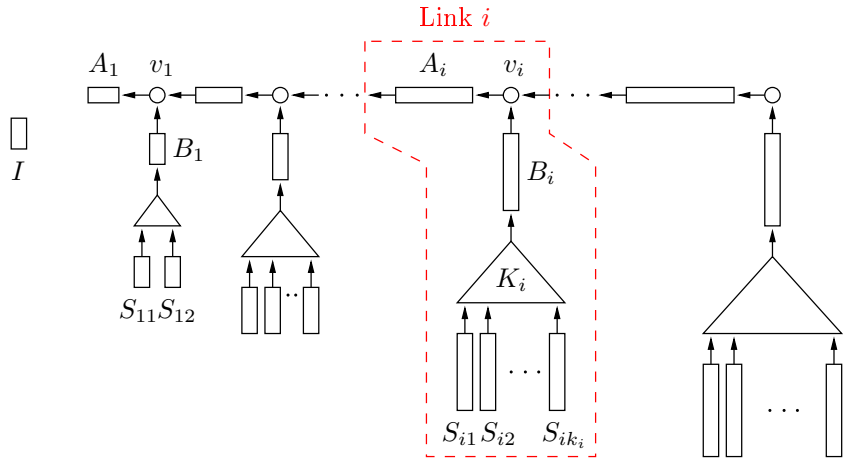


Abbildung 3.6: Die *funnel heap*-Prioritätswarteschlange, vgl. [8, 21]. Diese besteht aus einer Sequenz von k-Verschmelzern (Dreiecke), die mit Hilfe von Puffern (Rechtecke) und binären Verschmelzern (Kreise) zu einem binären Verschmelzungsbaum verknüpft werden, und einem separaten Eingabepuffer I. Die Komponenten des i-ten Links sind rot umrandet.

Dabei bezeichnet die Notation $\lceil\lceil x \rceil\rceil$ die kleinste Potenz von 2, die größer als x ist, d. h. $\lceil\lceil x \rceil\rceil = 2^{\lceil \log x \rceil}$. Für die Werte k_i und s_i gilt dabei $s_{i+1} \in \Theta\left(s_i^{4/3}\right)$ und $k_{i+1} \in \Theta\left(k_i^{4/3}\right)$: Da $k_{i+1} = \lceil\lceil {s_{i+1}}^{1/3} \rceil\rceil$ gilt, folgt direkt

$$s_i^{1/3} \leq k_i < 2s_i^{1/3}$$

und somit $s_{i+1} = s_i \cdot (k_i + 1) > s_i k_i \geq s_i^{4/3}$ bzw. $s_{i+1} = s_i \cdot (k_i + 1) < s_i \cdot (2s_i^{1/3} + 1) \leq s_i \cdot 3 \cdot s_i^{1/3} = 3 \cdot s_i^{4/3}$, also insgesamt

$$s_i^{4/3} < s_{i+1} < 3 \cdot s_i^{4/3}.$$

Induktiv kann weiterhin leicht $s_i = s_1 + \sum_{j=1}^{i-1} k_j s_j$ für $i \geq 1$ gezeigt werden: Für $i = 1$ ist die Gleichung trivialerweise erfüllt. Ansonsten folgt mit Hilfe der Induktionsvoraussetzung direkt

$$s_{i+1} = s_i \cdot (k_i + 1) = s_i k_i + \left(s_1 + \sum_{j=1}^{i-1} k_j s_j \right) = s_1 + \sum_{j=1}^{i} k_j s_j$$

und somit die Behauptung.

Wie bereits oben erwähnt, setzt sich die *funnel heap*-Datenstruktur aus zwei Komponenten zusammen. Die erste Komponente besteht aus einer Sequenz von *Links*, wobei der i-te Link seinerseits zwei Puffer A_i und B_i, einen binären Verschmelzer v_i und einen k_i-Verschmelzer mit Eingabepuffern $S_{i1}, \ldots, S_{ik_i}$ umfasst, vgl. Abbildung 3.6. Der Puffer B_i, der k_i-Verschmelzer K_i und die Eingabepuffer $S_{i1}, \ldots, S_{ik_i}$ stellen den *unteren Teil* des i-ten Links dar. Die Puffer A_i und B_i können jeweils beide k_i^3 und die Puffer $S_{i1}, \ldots, S_{ik_i}$ jeweils s_i Elemente aufnehmen. Zu jedem Link i gehört weiterhin eine Zählvariable c_i, für

die $1 \leq c_i \leq k_i + 1$ gilt. Der Startwert dieser Variablen beträgt 1 für jeden der Links. Die zweite Komponente der *funnel heap*-Datenstruktur ist ein Eingabepuffer I der Größe s_1.

Ein *funnel heap* wird in der Reihenfolge $I, \text{Link } 1, \text{Link } 2, \ldots$ zusammenhängend im Speicher abgelegt, wobei die Speicherplatzbelegung eines Links i den Aufbau $c_i, A_i, v_i, B_i, K_i, S_{i1}, \ldots, S_{ik_i}$ hat.

Operationen

Es werden nun vier Operationen auf der *funnel heap*-Datenstruktur beschrieben. Die ersten beiden Operationen sind die für eine Prioritätswarteschlange benötigten Operationen INSERT und DELETEMIN. Die dritte Operation ist die optionale DELETE-Operation, welche ein beliebiges Element mit Hilfe seines Schlüssels in der Prioritätswarteschlange löschen kann. Zuletzt wird eine SWEEP-Operation beschrieben, welche der INSERT-Operation als Hilfsfunktion dient. Bei der Ausführung aller Operationen werden die folgenden Invarianten aufrecht erhalten:

Invariante 1: Für jeden Link i enthalten die Eingabepuffer $S_{ic_i}, \ldots, S_{ik_i}$ keine Elemente.

Invariante 2: Auf jedem Pfad des gesamten Verschmelzungsbaums von einem beliebigen Puffer zum Puffer A_1 treten die Elemente in absteigender Ordnung (bzgl. ihrer Schlüssel) auf.

Invariante 3: Die Elemente im Puffer I sind sortiert.

Wie Arge *et al.* [8] bemerken, entspricht die zweite Invariante der Forderung, dass der gesamte Verschmelzungsbaum die für einen *heap* geforderte Ordnungsbedingung [32] erfüllt. Aus dieser folgt direkt, dass die Elemente in den Puffern jeweils in sortierter Form vorliegen und dass sich die kleinsten Elemente der Prioritätswarteschlange entweder in A_1 oder I befinden. Weiterhin bleibt diese Ordnungsbedingung auch bei Anwendung der Prozedur FILL (Algorithmus 3.3) auf einen beliebigen binären Verschmelzer des Verschmelzungsbaums erhalten.

INSERT Mit Hilfe der INSERT-Operation kann ein neues Element in die Prioritätswarteschlange eingefügt werden. Dazu wird dieses unter Aufrechterhaltung der dritten Invariante in den Puffer I eingefügt. Ist I nach Einfügen dieses Elements voll belegt, so wird derjenige Link i gesucht, für den $c_i \leq k_i$ gilt. Anschließend wird die SWEEP-Operation mit Parameter i ausgeführt.

DELETEMIN Die DELETEMIN-Operation stellt die einfachste der Operationen dar. Um das Element mit dem kleinsten Schlüssel (und somit der höchsten Priorität) aus der Prioritätswarteschlange zu entfernen, wird zunächst geprüft, ob der Puffer A_1 leer ist. Ist dies

der Fall, so wird die Prozedur FILL auf den Knoten v_1 angewendet. Falls die Prioritätswarteschlange überhaupt Elemente enthält, so sind die kleinsten dieser Elemente anschließend entweder in A_1 oder I (falls A_1 Elemente enthält) oder nur in I enthalten. Durch einen Vergleich des Inhalts beider Puffer kann dann das gesuchte Element aus der Prioritätswarteschlange entfernt und zurückgegeben werden.

DELETE Die DELETE-Operation kann mit Hilfe der INSERT- und DELETEMIN-Operation von der Prioritätswarteschlange unterstützt werden: Um ein beliebiges Element mit gegebenen Schlüssel aus der Prioritätswarteschlange zu entfernen, wird ein spezielles *delete*-Element mit gleichem Schlüssel in die Prioritätswarteschlange eingefügt. Dadurch kann bei Ausführung einer später folgenden DELETEMIN-Operation mittels eines weiteren Aufrufs von DELETEMIN festgestellt werden, ob das Element zuvor gelöscht wurde oder nicht.

SWEEP Durch einen Aufruf der Operation SWEEP mit Parameter i wird zunächst der Pfad p von A_1 nach S_{ic_i} traversiert und jeder angetroffene Puffer mit der Anzahl von Elementen markiert, die er zu diesem Zeitpunkt enthält. Anschließend wird durch eine Traversierung des Teilpfads von A_i nach S_{ic_i} ein sortierter Strom σ_1 gebildet, der aus den Elementen besteht, welche sich in den Puffern auf diesem Teilpfad befinden. Hiernach wird ein zweiter Strom σ_2 erzeugt, der sich aus den Elementen des Puffers I und aus den Elementen der Links $1, \ldots, i-1$ zusammensetzt. Dazu wird der Puffer A_i vorübergehend als „aufgebraucht“ markiert und die Operation DELETEMIN solange ausgeführt, bis diese keine Elemente mehr liefert. Die beiden Ströme σ_1 und σ_2 werden dann zu einem sortierten Strom σ verschmolzen. Im Zuge dieser Verschmelzung werden die vordersten (kleinsten) Elemente von σ in die Puffer des Pfads p eingefügt. Hierbei werden mit Hilfe der zu Beginn erstellten Markierungen jeweils nur so viele Elemente in einen Puffer eingefügt, wie dieser vor Aufruf der SWEEP-Operation enthielt. Die restlichen Elemente von σ werden in den Puffer S_{ic_i} eingefügt. Zuletzt wird die Zählvariable c_i um eins erhöht und die Zählvariablen $c_1, \ldots, c_{i-1}$ auf eins zurückgesetzt.

Analyse

Es wird zunächst die Korrektheit der oben angegebenen Operationen überprüft und anschließend der für die Prioritätswarteschlange insgesamt benötigte Speicherplatz angegeben. Zuletzt werden die durch Anwendung der Operationen verursachten Speichertransfers analysiert.

Korrektheit Die Korrektheit der DELETEMIN-Operation folgt direkt aus der (gültigen) zweiten Invariante. Für die beiden Operationen INSERT und SWEEP muss gezeigt werden, dass diese die Invarianten aufrecht erhalten und dass der Eingabepuffer S_{ic_i} bei Anwendung

einer SWEEP(i)-Operation nicht überläuft. Die Korrektheit der DELETE-Operation ergibt sich direkt aus der Korrektheit der INSERT- und DELETEMIN-Operation.

Die INSERT-Operation fügt unter Aufrechterhaltung der dritten Invariante ein neues Element in den Puffer I ein. Enthält dieser Puffer anschließend weniger als acht Elemente, so findet keine weitere Aktion statt. Die Korrektheit der ersten und zweiten Invariante ist in diesem Fall offensichtlich. Ansonsten ist I nach Einfügen des Elements voll belegt und es wird derjenige Link i mit $c_i \leq k_i$ ausgewählt. Der c_i-te Eingabepuffer des i-ten Links enthält aufgrund der (gültigen) ersten Invariante keine Elemente. Durch den darauf folgenden Aufruf der SWEEP-Operation bleiben weiterhin alle Invarianten erhalten: Im Laufe des SWEEP(i)-Aufrufs wird ein sortierter Strom σ gebildet, der aus den Elementen des Pfades p und den unteren Teilen der Links $1, \ldots, i-1$ besteht. Die vordersten (kleinsten) Elemente dieses Stroms werden sodann in die Puffer des Pfades p eingefügt. Dabei werden jeweils nur so viele Elemente in einen der involvierten Puffer eingefügt, wie dieser vor Aufruf der SWEEP-Operation enthielt. Ein neues Element des Pfades p kann deshalb höchstens einen kleineren Schlüssel besitzen, als das Element, welches zuvor diese Position des entsprechenden Puffers belegt hat. Daraus ergibt sich die Gültigkeit der zweiten Invariante. Die Gültigkeit der ersten und dritten Invariante ist offensichtlich, da ein SWEEP(i)-Aufruf den Puffer I und die unteren Teile der Links $1, \ldots, i-1$ vollständig entleert. Wird die SWEEP-Operation unabhängig von einer INSERT-Operation aufgerufen, so ergibt sich die Gültigkeit aller Invarianten analog.

Es muss noch gezeigt werden, dass der Puffer S_{ic_i} bei Abarbeitung eines SWEEP(i)-Aufrufs nicht überläuft. Dies ergibt sich aus den folgenden Überlegungen: Bei Ausführung einer SWEEP(i)-Operation werden die Elemente der unteren Teile der Links $1, \ldots, i-1$ aus den entsprechenden Puffern entfernt und in den Puffer S_{ic_i} eingefügt. Zudem werden die Zählvariablen $c_1, \ldots, c_{i-1}$ auf eins zurückgesetzt. Allgemein wird eine Zählvariable c_j des j-ten Links also nur dann auf eins zurückgesetzt, wenn zuvor der untere Teil dieses Links vollständig entleert wurde. Insofern kann die Anzahl der in dem unteren Teil des j-ten Links enthaltenen Elemente nie größer sein, als die Anzahl der Elemente, welche durch die maximal k_j letzten SWEEP(j)-Operationen in die Prioritätswarteschlange eingefügt wurden. Aufgrund von $s_j = s_1 + \sum_{l=1}^{j-1} k_l s_l$ folgt dann induktiv, dass der Puffer S_{ic_i} bei Ausführung einer SWEEP(i)-Operation nicht überläuft.

Speicherplatzbedarf Um zu gewährleisten, dass die gesamte Datenstruktur nur einen linearen Speicherplatzbedarf besitzt, wird ein Link i erst dann erstellt, wenn dieser das erste Mal benötigt wird, d. h. wenn zum ersten Mal ein SWEEP(i)-Aufruf stattfindet. Bei Erstellung des i-ten Links werden zunächst nur der Zähler c_i, der binäre Verschmelzer v_i, der k_i-Verschmelzer K_i und die Puffer A_i, B_i und S_{i1} angelegt. Dies benötigt insgesamt $\Theta(s_i)$ neuen Speicherplatz. Bei jeder der nächsten $k_i - 1$ SWEEP(i)-Operationen wird der Puffer S_{ic_i} angelegt. Da vor jedem SWEEP(i)-Aufruf $s_i = s_1 + \sum_{j=1}^{i-1} k_j s_j$ Elemente in die

Prioritätswarteschlange eingefügt wurden, besitzt die *funnel heap*-Datenstruktur insgesamt einen linearen Speicherplatzbedarf.

Um sicherzustellen, dass die Prioritätswarteschlange zusammenhängend im Speicher abgelegt werden kann, wird die gesamte Datenstruktur in einen größeren Speicherbereich transferiert, sobald diese um einen konstanten Faktor gewachsen ist. Der zusammenhängende Speicherbereich muss dabei so gewählt werden, dass genügend freier Speicherplatz bis zum nächsten Transfer dieser Art zur Verfügung steht. Die amortisierten Kosten eines solchen Transfers betragen offensichtlich $\mathcal{O}(1/B)$ amortisierte Speichertransfers pro insgesamt eingefügtem Element.

Speichertransfers Die folgende Analyse der durch die Operationen verursachten Speichertransfers stellt den umfangreichsten Teil der gesamten Analyse der Datenstruktur dar und ist in vier Teile gegliedert: Im ersten Teil werden die durch den Aufstieg von Elementen in den Puffer A_1 verursachten amortisierten Speichertransfers analysiert. Im zweiten Teil werden die durch den Aufstieg der Elemente verursachten Kosten den jeweiligen SWEEP-Operationen zugeschrieben und die durch die SWEEP-Operationen selbst verursachten Kosten analysiert. In diesen beiden ersten Teilen wird angenommen, dass keine involvierten Eingabeströme aufgebraucht werden. Diese Sonderfälle werden im dritten Teil der Analyse behandelt. Im vierten Teil werden letztendlich die durch die einzelnen Operationen verursachten amortisierten Speichertransfers angegeben. Für die gesamte Analyse wird die Annahme eines großen Speichers von der Gestalt $M \in \Omega\left(B^2\right)$ getroffen.

Aufstieg eines Elements in den Puffer A_1 Um die amortisierten Kosten für eine INSERT- oder DELETE-Operation zu analysieren, wird zunächst geprüft, wie viele Speichertransfers durch den Aufstieg von Elementen in den Puffer A_1, d. h. durch Anwendung von entsprechenden binären Verschmelzern des Verschmelzungsbaums, verursacht werden. Dabei werden die Kosten für das Füllen eines Ausgabepuffers gleichmäßig auf die in den Ausgabepuffer eingefügten Elemente verteilt. Unter der (vorläufigen) Annahme, dass keiner der involvierten Eingabeströme aufgebraucht wird, werden dadurch jedem Element aus einem Eingabepuffer eines k_i-Verschmelzers K_i insgesamt $\mathcal{O}(\frac{1}{B}\log_{M/B} s_i)$ Speichertransfers als Kosten für den Aufstieg in den Puffer A_1 zugeschrieben:

Wie Arge *et al.* [8] bemerken, beruht der Beweis der obigen Aussage auf der im *cache-oblivious*-Modell angenommenen optimalen Ersetzungsstrategie, durch welche die ersten (kleineren) Links der Datenstruktur im internen Speicher gehalten werden. Um die Anzahl der Links zu ermitteln, welche im internen Speicher Platz finden, gebe $\triangle_{L_i}$ den Speicherplatzbedarf des i-ten Links und $\triangle_i$ den Speicherplatzbedarf der ersten i Links an, d. h.

$$\triangle_i = \sum_{k=1}^{i} \triangle_{L_k}.$$

Aufgrund des sublinearen Speicherplatzbedarfs eines k-Verschmelzers wird der Speicherplatzbedarf $\triangle_{L_i}$ des i-ten Links durch den Speicherplatzbedarf seiner k_i Eingabepuffer $S_{i1}, \ldots, S_{ik_i}$ dominiert, d. h. es gilt

$$\triangle_{L_i} \in \Theta(s_i k_i) = \Theta(k_i^4).$$

Unter Verwendung von $k_{i+1} \in \Theta(k_i^{4/3})$ folgt dann für den Speicherplatzbedarf $\triangle_i$ der ersten i Links

$$\begin{aligned} \triangle_i &\in \Theta(k_1^4) + \Theta(k_2^4) + \quad \ldots \quad + \Theta(k_i^4) \\ &= \Theta\Big((k_1^4)^{(4/3)^0}\Big) + \Theta\Big((k_1^4)^{(4/3)^1}\Big) + \quad \ldots \quad + \Theta\Big((k_1^4)^{(4/3)^{i-1}}\Big). \end{aligned}$$

Da der Speicherplatzbedarf $\triangle_{L_i}$ des i-ten Links den Speicherplatzbedarf der ersten $i-1$ Links dominiert[3], also

$$\sum_{j=0}^{i-2} \Theta\left((k_1^4)^{(4/3)^j}\right) \in \Theta\left((k_1^4)^{(4/3)^{i-1}}\right) = \Theta\left(k_i^4\right)$$

gilt, folgt für den Speicherplatzbedarf $\triangle_i$ insgesamt

$$\triangle_i \in \Theta(k_i^4) = \Theta(s_i^{4/3}).$$

Mit i_M sei nun derjenige Index bezeichnet, für den

$$\triangle_{i_M} \leq M < \triangle_{i_M+1}$$

erfüllt ist. Aufgrund der obigen Analyse folgt dann für drei reelle Konstanten $c_1, c_2, c_3 > 0$

$$c_1 \cdot s_{i_M}^{4/3} \leq M < c_2 \cdot s_{i_M+1}^{4/3}$$

und somit unter Verwendung von $s_{i+1} \in \Theta\left(s_i^{4/3}\right)$ weiter

$$\log_M(c_1) + \left(\frac{4}{3}\right)\log_M(s_{i_M}) \leq 1 < \log_M(c_2) + \left(\frac{4}{3}\right)\log_M(c_3) + \left(\frac{4}{3}\right)^2 \log_M(s_{i_M}).$$

Es gilt demnach $\log_M(s_{i_M}) \in \Theta(1)$.

Sei nun ein Element in einem Eingabepuffer eines k_i-Verschmelzers K_i gegeben. Gilt $i \leq i_M$, so verursacht der Aufstieg des Elements in den Puffer A_1 keine Speichertransfers, da sich alle involvierten Komponenten im internen Speicher befinden. Für $i > i_M$ wird das Element mit den Kosten für den Transfer durch den k_i-Verschmelzer K_i in den Puffer B_i und für den Transfer durch die binären Verschmelzer v_j in die Puffer A_j für $j = i, i-1, \ldots, i_M$ belastet. Seien zunächst die Kosten für den Aufstieg in B_i betrachtet:

[3] Zur Verifikation dieser Aussage betrachte man $\sum_{j=0}^{i-2} \log\left(c \cdot (k_1^4)^{(4/3)^j}\right)$ für eine passende reelle Konstante $c > 0$.

Nach Theorem 3.3.2 verursacht der k_i-Verschmelzer K_i bei Anwendung $\mathcal{O}(\frac{k_i^3}{B}\log_M k_i^3 + k_i)$ Speichertransfers und füllt den Ausgabepuffer B_i mit k_i^3 Elementen. Mit Hilfe von

$$M < \triangle_{i_M+1} \in \Theta(k_{i_M+1}^4) \subseteq \mathcal{O}(k_i^4)$$

folgt dann unter Verwendung von $M \in \Omega(B^2)$ zunächst $B \in \mathcal{O}(k_i^2)$ und somit weiter $k_i \in \mathcal{O}(k_i^3/B)$. Unter der Annahme, dass keine Eingabeströme aufgebraucht werden, wird ein Element demnach beim Aufstieg in den Puffer B_i mit

$$\mathcal{O}\left(\frac{1}{B}\log_{M/B} k_i^3\right) = \mathcal{O}\left(\frac{1}{B}\log_{M/B} s_i\right)$$

amortisierten Speichertransfers belastet. Für den weiteren Transfer des Elements durch die binären Verschmelzer v_j nach A_j für $j = i, i-1, \ldots, i_M$ wird jedes Element mit weiteren $\mathcal{O}(\frac{1}{B}\log_{M/B} s_i)$ Speichertransfers belastet: Das Füllen eines Puffers A_j verursacht $\mathcal{O}(1 + |A_j|/B)$ Speichertransfers. Analog zu obiger Begründung folgt

$$B \in \mathcal{O}(k_{i_M+1}^2) \subset \mathcal{O}(k_{i_M}^{8/3})$$

und somit unter Beachtung von $|A_j| = k_j^3$ und $k_j^3 \geq k_{i_M}^3 > k_{i_M}^{8/3}$ insgesamt

$$\mathcal{O}(1 + |A_j|/B) = \mathcal{O}(|A_j|/B).$$

Ein Element wird also für jedes A_j mit $\mathcal{O}(1/B)$ Speichertransfers belastet. Es muss noch die Anzahl $i - i_M$ der traversierten Puffer begrenzt werden: Aus $s_{i-1}^{4/3} < s_i$ folgt induktiv $s_{i_M}^{(4/3)^{i-i_M}} < s_i$ und somit

$$\begin{aligned} \log\log_M s_i &> \log\log_M s_{i_M}^{(4/3)^{i-i_M}} \\ &= (i - i_M) \cdot \log\frac{4}{3} + \log\log_M s_{i_M}. \end{aligned}$$

Da $\log_M(s_{i_M}) \in \Theta(1)$ und $M \in \Omega(B^2)$ gilt, folgt schließlich

$$i - i_M \in \mathcal{O}(\log\log_M s_i) \subset \mathcal{O}(\log_M s_i) = \mathcal{O}(\log_{M/B} s_i).$$

Unter der Annahme, dass keiner der involvierten Eingabeströme aufgebraucht wird, werden demnach jedem Element aus einem der Eingabepuffer von K_i insgesamt $\mathcal{O}(\frac{1}{B}\log_{M/B} s_i)$ Speichertransfers als Kosten für seinen Aufstieg in den Puffer A_1 zugeschrieben.

Kredit-Invariante und Kosten einer SWEEP(i)-Operation Ein Element wird nach obigen Ausführungen für den Aufstieg aus einem Eingabepuffer von K_i in den Puffer A_1 mit $\mathcal{O}(\frac{1}{B}\log_{M/B} s_i)$ Speichertransfers belastet. Arge *et al.* [8] stellen eine *Kredit-Invariante* auf, durch die gefordert wird, dass jedes Element genug „Krediteinheiten“ besitzt, um den Aufstieg in den Puffer A_1 zu „bezahlen“. Die Krediteinheiten entsprechen

dabei den amortisierten Speichertransfers. Ein Element muss also $\mathcal{O}(\frac{1}{B}\log_{M/B} s_i)$ Krediteinheiten besitzen, wenn es in einen der Eingabepuffer eines k_i-Verschmelzers K_i eingefügt wird. Die Kosten für diese Krediteinheiten werden den SWEEP-Operationen zugeschrieben, d. h. eine SWEEP(i)-Operation wird mit den Kosten der Elemente belastet, die durch die Ausführung der Operation in einen der Eingabepuffer von K_i eingefügt werden. Insgesamt werden die durch die binären Verschmelzer verursachten Kosten somit durch die SWEEP-Operation im Voraus bezahlt.

Eine SWEEP(i)-Operation verursacht bei Ausführung selbst auch Speichertransfers. Diese werden nun analysiert. Im ersten Schritt einer SWEEP(i)-Operation wird der Strom σ_1 erstellt, welcher aus den Elementen des Pfades p von A_1 nach S_{ic_i} besteht. Da alle Komponenten zusammenhängend im Speicher abgelegt sind, werden dadurch höchstens

$$\mathcal{O}\left(\left(\triangle_{i-1} + |A_i| + |B_i| + |K_i|\right)/B\right) \subset \mathcal{O}\left(k_i^3/B\right) = \mathcal{O}\left(s_i/B\right)$$

Speichertransfers verursacht. Im zweiten Schritt einer SWEEP(i)-Operation wird der Strom σ_2 mit Hilfe von DELETEMIN-Operationen erstellt. Die Kosten für diesen Vorgang sind schon gedeckt, da jedes involvierte Element aufgrund der Kredit-Invariante genug Krediteinheiten für seinen Aufstieg in den Puffer A_1 besitzt. Im letzten Schritt der SWEEP(i)-Operation werden die beiden Ströme σ_1 und σ_2 zu einem sortierten Strom σ vereint und die Elemente auf die Puffer des Pfades p und den Eingabepuffer S_{ic_i} verteilt. Die durch das Einfügen der Elemente in die Puffer des Pfades p verursachte Anzahl an Speichertransfers kann aufgrund obiger Abschätzung durch $\mathcal{O}(s_i/B)$ begrenzt werden. Da der Puffer S_{ic_i} weiterhin höchstens s_i Elemente aufnehmen kann, werden durch diesen gesamten Vorgang höchstens $\mathcal{O}(s_i/B)$ Speichertransfers verursacht.

Nach Ausführung der SWEEP(i)-Operation muss die Kredit-Invariante wiederhergestellt werden. Jedes der $\mathcal{O}(s_i)$ in den Puffer S_{ic_i} eingefügten Elemente muss dabei insgesamt $\mathcal{O}(\frac{1}{B}\log_{M/B} s_i)$ Krediteinheiten für den (bevorstehenden) Aufstieg in den Puffer A_1 erhalten. Des Weiteren müssen die in die Puffer des Pfades p eingefügten Elemente Krediteinheiten für den Aufstieg in den Puffer A_1 erhalten, da die Krediteinheiten der Elemente, welche die entsprechenden Positionen der Puffer zuvor belegt haben, durch die DELETEMIN-Operationen aufgebraucht wurden. Diese $\mathcal{O}(\triangle_{i-1}) = \mathcal{O}(s_i)$ Elemente müssen aufgrund der obigen Analyse jeweils maximal $\mathcal{O}(\frac{1}{B}\log_{M/B} s_i)$ Krediteinheiten erhalten. Insgesamt werden also jeder SWEEP(i)-Operation

$$\mathcal{O}\left(\frac{s_i}{B}\right) + \mathcal{O}\left(\frac{s_i}{B}\log_{M/B} s_i\right) = \mathcal{O}\left(\frac{s_i}{B}\log_{M/B} s_i\right)$$

Speichertransfers als Kosten für die Ausführung der Operation und das Aufrechterhalten der Kredit-Invariante zugeschrieben.

Aufgebrauchte Eingabeströme In der bisherigen Analyse wurde angenommen, dass die involvierten Eingabeströme nicht aufgebraucht werden. Die Kosten für das partielle Füllen eines Puffers sind dabei jedoch höchstens so hoch, wie die Kosten für das

komplette Füllen eines solchen. Die letzteren wurden schon weiter oben analysiert: Gilt $i < i_M$, so werden keine Kosten verursacht. Für $i > i_M$ werden durch das Füllen von A_i höchstens $\mathcal{O}(|A_i|/B) = \mathcal{O}(k_i/B)$ und für das Füllen von B_i höchstens $\mathcal{O}(\frac{|B_i|}{B} \log_{M/B} s_i) = \mathcal{O}(\frac{k_i}{B} \log_{M/B} s_i)$ Speichertransfers verursacht. Wird der Puffer B_i aufgebraucht, so können nur durch eine SWEEP(i)-Operation wieder Elemente in den Puffer B_i eingefügt werden. Wird der Puffer A_i aufgebraucht, so kann sich dieser Zustand nur durch eine SWEEP(j)-Operation für $j \geq i$ wieder verändern. Der Puffer B_i kann also pro SWEEP(i)-Operation höchstens einmal aufgebraucht werden. Da höchstens eine SWEEP(j)-Operation mit $j > i$ zwischen einer SWEEP(i)-Operation und der nächsten ausgeführt wird, kann der Puffer A_i pro SWEEP(i)-Operation höchstens zweimal aufgebraucht werden. Werden also pro SWEEP(i)-Operation zusätzliche

$$\mathcal{O}\left(\frac{k_i^3}{B} \log_{M/B} s_i\right) = \mathcal{O}\left(\frac{s_i}{B} \log_{M/B} s_i\right)$$

Speichertransfers veranschlagt, so sind die Kosten für das partielle Füllen von Puffern gedeckt.

Amortisierte Kosten Insgesamt sind die durch die INSERT- und DELETEMIN-Operationen verursachten Kosten also gedeckt, wenn pro SWEEP(i)-Operation $\mathcal{O}(\frac{s_i}{B} \log_{M/B} s_i)$ Speichertransfers veranschlagt werden. Aufgrund von $s_i = s_1 + \sum_{j=1}^{i-1} k_j s_j$ folgt, dass zwischen zwei SWEEP(i)-Operationen mindestens s_i Elemente in die Datenstruktur eingefügt werden. Werden die Kosten für eine SWEEP(i)-Operation also den letzten s_i in die Datenstruktur eingefügten Elementen zugeschrieben, so wird dadurch jedes in die Prioritätswarteschlange eingefügte Element mit $\mathcal{O}(\frac{1}{B} \log_{M/B} s_i)$ Speichertransfers belastet.

Sei nun eine Folge von Operationen auf einer zu Beginn leeren *funnel heap*-Prioritätswarteschlange gegeben und sei $i_{\max}$ der größte Index i, für den eine SWEEP(i)-Operation eintritt. Für diesen gilt

$$s_{i_{\max}} = s_1 + \sum_{j=1}^{i_{\max}-1} k_j s_j \leq N,$$

wobei N die Anzahl der INSERT-Operationen in der Folge der Operationen angebe. Da jede INSERT-Operation durch höchstens eine SWEEP(i) für $i = 1, \ldots, l_{\max}$ belastet wird, belaufen sich die für eine INSERT-Operation veranschlagten Kosten somit auf

$$\mathcal{O}\left(\sum_{k=0}^{\infty} \frac{1}{B} \log_{M/B} N^{(3/4)^k}\right) = \mathcal{O}\left(\frac{1}{B} \log_{M/B} N\right) = \mathcal{O}\left(\frac{1}{B} \log_{M/B} \frac{N}{B}\right).$$

Die durch die INSERT-, DELETEMIN- und DELETE-Operationen verursachten Kosten können also den Kosten für die INSERT-Operationen zugeschrieben werden. Insgesamt ergibt sich somit das folgende Theorem:

3.3.3 Theorem ([8, 21]). *Unter der Annahme von $M \in \Omega(B^2)$ verursacht die Abarbeitung von N* INSERT-, DELETEMIN- *und* DELETE-*Operationen auf einer anfangs leeren*

funnel heap-Prioritätswarteschlange pro Operation höchstens $\mathcal{O}\left(\frac{1}{B}\log_{M/B}\frac{N}{B}\right)$ *amortisierte Speichertransfers. Die Struktur belegt dabei* $\mathcal{O}\left(\frac{N}{B}\right)$ *zusammenhängend im Speicher liegende Speicherblöcke.*

Kapitel 4

Die *Well-Separated Pair Decomposition*

In diesem Kapitel wird die sogenannte *well-separated pair decomposition* (WSPD) einer Punktmenge aus dem $\mathbb{R}^d$ vorgestellt. Diese Datenstruktur wurde von Callahan und Kosaraju [27] eingeführt und anschließend von mehreren Autoren zur Lösung geometrischer Probleme im d-dimensionalen euklidischen Raum herangezogen.

Eine WSPD einer endlichen Punktmenge $P \subset \mathbb{R}^d$ besteht aus zwei Komponenten. Die erste Komponente ist ein Binärbaum, dessen Blätter in eindeutiger Weise mit den Punkten aus P korrespondieren. Ein innerer Knoten v dieses Baums stellt dabei diejenige Teilmenge von P dar, die durch die Blätter in dem in v gewurzelten Teilbaum von T repräsentiert wird. Die zweite Komponente ist eine Liste, welche Paare von sogenannten „scharf getrennten" Teilmengen von P enthält. Für jedes Paar $\{A, B\}$ dieser Liste existieren in T Knoten, die die Mengen A und B repräsentieren.

Callahan und Kosaraju [27] geben einen sequentiellen und einen parallelen Algorithmus zur Berechnung einer WSPD einer endlichen nicht-leeren Punktmenge $P \subset \mathbb{R}^d$ an. Beide Algorithmen ermöglichen die Konstruktion einer WSPD, deren Liste aus $\mathcal{O}(|P|)$ Paaren von Teilmengen besteht. Der Nutzen einer linearen Größe dieser Datenstruktur begründet sich in folgendem Sachverhalt: Die Lösung vieler geometrischer Probleme bezieht jeweils die Menge aller verschiedenen Punktpaare der zu untersuchenden Punktmenge $P \subset \mathbb{R}^d$ mit ein. Bei einigen dieser Probleme stellt sich heraus, dass jeweils eine konstante Anzahl von Berechnungen auf einem Paar $\{A, B\}$ einer WSPD von P ausreicht, anstatt $|A| \cdot |B|$ Berechnungen auf der mit dem Paar $\{A, B\}$ korrespondierenden Menge von (ungeordneten) Punktpaaren $\{\{a, b\} \mid a \in A \wedge b \in B \wedge a \neq b\}$ durchzuführen. Die effiziente Berechnung einer WSPD linearer Größe führt somit zu einer effizienten Lösung dieser Probleme. In den Kapiteln 5 und 6 werden zwei Anwendungen der WSPD dieser Art detailliert beschrieben.

Govindarajan *et al.* [40] geben einen I/O-effizienten Algorithmus zur Berechnung einer WSPD an, welcher auf dem parallelen Konstruktionsalgorithmus von Callahan und Kosara-

ju [27] basiert. In diesem Kapitel wird gezeigt, wie dieser I/O-effiziente Algorithmus mittels einiger Änderungen zu einem im Kontext des *cache-oblivious*-Modells effizienten Algorithmus angepasst werden kann. Die Beschreibung des sich dadurch ergebenden effizienten *cache-oblivious*-Algorithmus zur Konstruktion einer WSPD für eine endliche Punktmenge aus dem $\mathbb{R}^d$ wird den größten Teil dieses Kapitels in Anspruch nehmen.

Dieses Kapitel ist wie folgt aufgebaut: In Abschnitt 4.1 werden benötigte Bezeichnungen und Definitionen eingeführt. In Abschnitt 4.2 wird der im RAM-Modell optimale sequentielle und der im CREW PRAM-Modell[1] effiziente parallele Algorithmus zur Konstruktion einer WSPD beschrieben. Beide gerade genannten Abschnitte basieren auf den Arbeiten von Callahan und Kosaraju [25, 27]. In Abschnitt 4.3 wird der im *cache-oblivious*-Modell effiziente auf den Arbeiten von Govindarajan *et al.* [40, 59] basierende Algorithmus zur Berechnung einer WSPD detailliert beschrieben.

4.1 Bezeichnungen und Definitionen

Sei P eine endliche nicht-leere Punktmenge aus dem $\mathbb{R}^d$. Ein *(abgeschlossenes) Rechteck* ist ein kartesisches Produkt der Form $R = [x_1, x_1'] \times \cdots \times [x_d, x_d'] \subset \mathbb{R}^d$ von abgeschlossenen Intervallen. Ein kartesisches Produkt aus offenen Intervallen der Gestalt $R = (x_1, x_1') \times \cdots \times (x_d, x_d') \subset \mathbb{R}^d$ wird als *offenes Rechteck* und ein kartesisches Produkt aus halboffenen Intervallen der Gestalt $R = [x_1, x_1') \times \cdots \times [x_d, x_d') \subset \mathbb{R}^d$ als *halboffenes Rechteck* bezeichnet. Der Punkt $(\frac{x_1+x_1'}{2}, \ldots, \frac{x_d+x_d'}{2}) \in \mathbb{R}^d$ ist das *Zentrum* eines Rechtecks R obigen Typs. Weiterhin werden die durch die beiden Werte x_i und x_i' induzierten $(d-1)$-dimensionalen Seitenflächen eines solchen Rechtecks R als die *Seiten* von R in der Dimension i bezeichnet.

Die *Länge* eines Rechtecks R in der Dimension $i \in \{1, \ldots, d\}$ wird durch $l_i(R) = x_i' - x_i$ definiert. Die *Maximal-* bzw. *Minimallänge* von R sei durch

$$l_{max}(R) = \max\{l_i(R) \mid 1 \leq i \leq d\} \quad \text{bzw.} \quad l_{min}(R) = \min\{l_i(R) \mid 1 \leq i \leq d\}$$

gegeben. Mit $i_{max}(R) \in \{1, \ldots, d\}$ bzw. $i_{min}(R) \in \{1, \ldots, d\}$ werde die kleinste Dimension bezeichnet, welche der Bedingung $l_{i_{max}(R)}(R) = l_{max}(R)$ bzw. $l_{i_{min}(R)}(R) = l_{min}(R)$ genügt.

Gilt $l_{max}(R) = l_{min}(R)$, so heißt R ein *d-Würfel* mit $l(R)$ als abkürzende Schreibweise für dessen Länge. Ein Rechteck R wird als *Box* bezeichnet, wenn $l_{max}(R) \leq 3l_{min}(R)$ gilt. Das *Begrenzungsrechteck* $R(P)$ von P sei definiert als das kleinste Rechteck, welches alle

[1] Die *parallel random access machine* (PRAM) ist ein Berechnungsmodell zur Analyse paralleler Algorithmen, vgl. Callahan [25]. Für dieses Modell existieren mehrere Varianten, die sich darin unterscheiden, ob gleichzeitige Lese- bzw. Schreibzugriffe auf eine identische Speicherzelle zugelassen sind oder nicht. Das CREW PRAM-Modell modelliert z. B. den *gleichzeitigen* Lese- und den *exklusiven* Schreibzugriff auf eine identische Speicherzelle. Da dieses Modell und dessen Varianten im Folgenden nur eine untergeordnete Rolle spielen, wird hier auf eine genauere Beschreibung verzichtet.

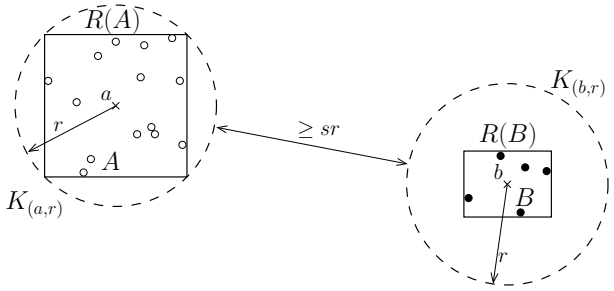


Abbildung 4.1: Zwei bzgl. eines reellen Wertes $s > 0$ scharf getrennte Punktmengen A und B in der Ebene, vgl. [25]

Punkte aus P enthält. Weiterhin seien $l_i(P)$, $l_{max}(P)$ bzw. $l_{min}(P)$ abkürzende Schreibweisen für $l_i(R(P))$, $l_{max}(R(P))$ bzw. $l_{min}(R(P))$.

Mit $d(x, y)$ wird der euklidische Abstand $d(x, y) = \sqrt{\sum_{i=1}^{d}(x_i - y_i)^2}$ zweier Punkte $x, y \in \mathbb{R}^d$ bezeichnet. Für zwei nicht-leere Teilmengen X und Y des $\mathbb{R}^d$ wird der Abstand dieser Mengen zueinander durch $d(X, Y) = \inf\{d(x, y) \mid x \in X \wedge y \in Y\}$ definiert. Die Menge $K_{(a,r)} = \{x \in \mathbb{R}^d \mid d(x, a) \leq r\}, r \in \mathbb{R}_0^+, a \in \mathbb{R}^d$, wird als *d-Kugel* mit Radius r und Mittelpunkt a bezeichnet.

Seien A und B zwei endliche nicht-leere Punktmengen aus dem $\mathbb{R}^d$ und $s \in \mathbb{R}^+$. Dann heißen A und B *scharf getrennt bzgl.* s, wenn es zwei d-Kugeln $K_{(a,r)}$ und $K_{(b,r)}$ mit Radius $r \in \mathbb{R}_0^+$ und Mittelpunkten $a, b \in \mathbb{R}^d$ gibt, so dass $R(A) \subset K_{(a,r)}$, $R(B) \subset K_{(b,r)}$ und $d(K_{(a,r)}, K_{(b,r)}) \geq sr$ gilt, vgl. Abbildung 4.1.

Das *Interaktionsprodukt* $A \otimes B$ zweier Punktmengen $A, B \subset \mathbb{R}^d$ wird definiert durch

$$A \otimes B = \{\{a, b\} \mid a \in A \wedge b \in B \wedge a \neq b\}.$$

Eine *Realisation* $\mathcal{R}$ eines solchen Interaktionsprodukts $A \otimes B$ ist eine Menge $\{\{A_1, B_1\}, \ldots, \{A_k, B_k\}\}$ $(k \geq 1)$, die den folgenden Bedingungen genügt, vgl. Callahan [27]:

(R1) $\emptyset \neq A_i \subseteq A$ und $\emptyset \neq B_i \subseteq B$ für alle $i = 1, \ldots, k$

(R2) $A_i \cap B_i = \emptyset$ für alle $i = 1, \ldots, k$

(R3) $(A_i \otimes B_i) \cap (A_j \otimes B_j) = \emptyset$ für alle i, j mit $1 \leq i < j \leq k$

(R4) $A \otimes B = \bigcup_{i=1}^{k} A_i \otimes B_i$

Eine Realisation $\mathcal{R}$ heißt *scharf getrennt bzgl.* $s \in \mathbb{R}^+$, wenn zusätzlich die folgende Bedingung erfüllt ist:

(R5) A_i und B_i sind scharf getrennt bzgl. s für alle $i = 1, \ldots, k$.

Die *Größe* der Realisation wird durch die Anzahl k der Paare $\{A_i, B_i\}$ definiert.

Mit der Punktmenge P kann ein Binärbaum T *verknüpft* werden. Ein Blatt dieses Baums stellt dabei eine einelementige Teilmenge von P dar, wobei die Menge aller durch die Blätter dargestellten Mengen eine disjunkte Zerlegung von P bildet. Ein innerer Knoten v von T stellt die Vereinigung aller Mengen dar, welche mittels der Blätter in dem in v gewurzelten Teilbaum von T repräsentiert werden. Die durch einen Knoten v aus T dargestellte Teilmenge von P wird mit $\sigma(v)$ bezeichnet. Die *Größe* eines Knotens v wird durch $|\sigma(v)|$ definiert. Da Blätter einelementige Teilmengen von P darstellen, besitzen diese demnach allesamt die Größe 1.

Seien $A, B \subseteq P$ und T ein mit P verknüpfter Binärbaum. Eine Realisation $\mathcal{R} = \{\{A_1, B_1\}, \ldots, \{A_k, B_k\}\}$ von $A \otimes B$ *benutzt* den Baum T, wenn es für alle A_i und B_i Knoten v_i und w_i in T mit $\sigma(v_i) = A_i$ und $\sigma(w_i) = B_i$ gibt. Eine *well-separated pair decomposition* (WSPD) $\mathcal{D} = (T, \mathcal{R})$ von P bzgl. eines Wertes $s \in \mathbb{R}^+$ besteht aus einem mit P verknüpften Binärbaum T und einer bzgl. s scharf getrennten Realisation $\mathcal{R}$ von $P \otimes P$, die T benutzt. Die *Größe* der WSPD wird durch die Größe der Realisation $\mathcal{R}$ definiert.

Das im nächsten Abschnitt angegebene Verfahren zur Konstruktion einer WSPD einer endlichen nicht-leeren Punktmenge aus dem $\mathbb{R}^d$ basiert auf der Berechnung einer speziellen Datenstruktur, die durch sukzessives „Aufteilen“ der Punktmenge entsteht. Eine *Aufteilung* von P ist eine Zerlegung von P in zwei nicht-leere Teilmengen, die durch eine zu einer der Koordinatenachsen senkrecht stehenden Hyperebene, die sogenannte *Aufteilungshyperebene*, getrennt werden. Hierbei darf kein Punkt in der Aufteilungshyperebene liegen.

Ein *Aufteilungsbaum* T der Menge P ist ein mit P verknüpfter Binärbaum und wird wie folgt rekursiv definiert:

- Gilt $|P| = 1$, so besteht der Aufteilungsbaum T aus einem Knoten v mit $\sigma(v) = P$.
- Für $|P| > 1$ besteht T aus einem Wurzelknoten v mit $\sigma(v) = P$ und zwei in v gewurzelten Aufteilungsbäumen der Teilmengen von P, die durch eine beliebige Aufteilung von P hervorgegangen sind.

Die Definition eines *unvollständigen Aufteilungsbaums* T' der Menge P stimmt mit der Definition eines Aufteilungsbaums von P überein, bis auf den Unterschied, dass die Blätter des Baums auch mehrelementige Teilmengen von P darstellen dürfen, d. h. es wird $|\sigma(v)| \geq 1$ für ein Blatt v aus T' zugelassen.

Sei v ein Knoten eines (unvollständigen) Aufteilungsbaums. Falls v nicht die Wurzel des Baums ist, so sei der Vaterknoten von v mit $p(v)$ bezeichnet. Das *äußere Rechteck* $\hat{R}(v)$ von v wird ebenso rekursiv definiert: Ist v die Wurzel des Baums, so ist $\hat{R}(v)$ ein d-Würfel mit Länge $l_{max}(\sigma(v))$ und gleichem Zentrum wie $R(\sigma(v))$. Ansonsten teilt die Aufteilungshyperebene, durch die $\sigma(p(v))$ in zwei Teilmengen zerlegt wurde, das Rechteck $\hat{R}(p(v))$ in zwei Rechtecke. $\hat{R}(v)$ ist dann dasjenige Rechteck dieser beiden Rechtecke, welches $\sigma(v)$ enthält. Eine *faire Aufteilung* von $\sigma(v)$ ist eine Aufteilung von $\sigma(v)$, bei dem

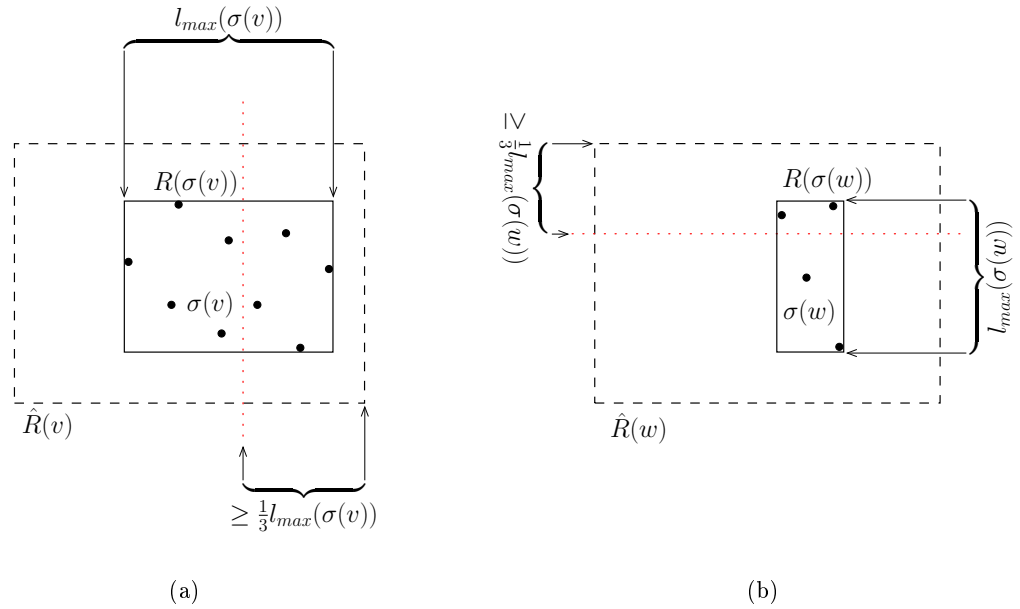


Abbildung 4.2: Zwei Beispiele für faire Aufteilungen von Punktmengen in der Ebene. Die gepunkteten Linien skizzieren die Aufteilungshyperebenen, vgl. [25].

die Aufteilungshyperebene H mindestens einen Abstand von $l_{max}(\sigma(v))/3$ von jeder der beiden zu H parallelen Seiten von $\hat{R}(v)$ hat, vgl. Abbildung 4.2. Ein Aufteilungsbaum, der ausschließlich durch faire Aufteilungen hervorgegangen ist, wird als *fairer Aufteilungsbaum* bezeichnet, vgl. Abbildung 4.3. Analog zu einem fairen Aufteilungsbaum ist ein *unvollständiger fairer Aufteilungsbaum* ein nur durch faire Aufteilungen entstandener unvollständiger Aufteilungsbaum.

Im nächsten Abschnitt wird gezeigt, wie man mittels eines fairen Aufteilungsbaums T von P effizient eine WSPD von P mit linearer Größe konstruieren kann. Die effiziente Konstruktion dieser WSPD wird durch zwei wesentliche Eigenschaften des fairen Aufteilungsbaums ermöglicht. Zum Einen gilt für jeden inneren Knoten $v \in T$ mit Kindern v_1 und v_2

$$|\sigma(v_1)| < |\sigma(v)| \quad \text{bzw.} \quad |\sigma(v_2)| < |\sigma(v)|.$$

Die Höhe des Aufteilungsbaums ist also linear in der Anzahl der Punkte aus P. Zum Anderen verkleinert sich die geometrische Größe der den Knoten zugeordneten äußeren Rechtecke stark bei zunehmender Tiefe der Knoten.

Callahan [25] vergleicht die Struktur eines fairen Aufteilungsbaum mit der eines *quadtrees* bzw. *kd*-Baums.[2] Ebenso wie mit einem fairen Aufteilungsbaum kann mit Hilfe eines *quadtrees* bzw. eines *kd*-Baums die Punktmenge P „zerlegt“ werden. Das Problem bei der Verwendung eines *quadtrees* besteht jedoch darin, dass durch geometrisch nah beieinander

[2] Eine Beschreibung dieser beiden Datenstrukturen findet sich z. B. bei de Berg *et al.* [17]

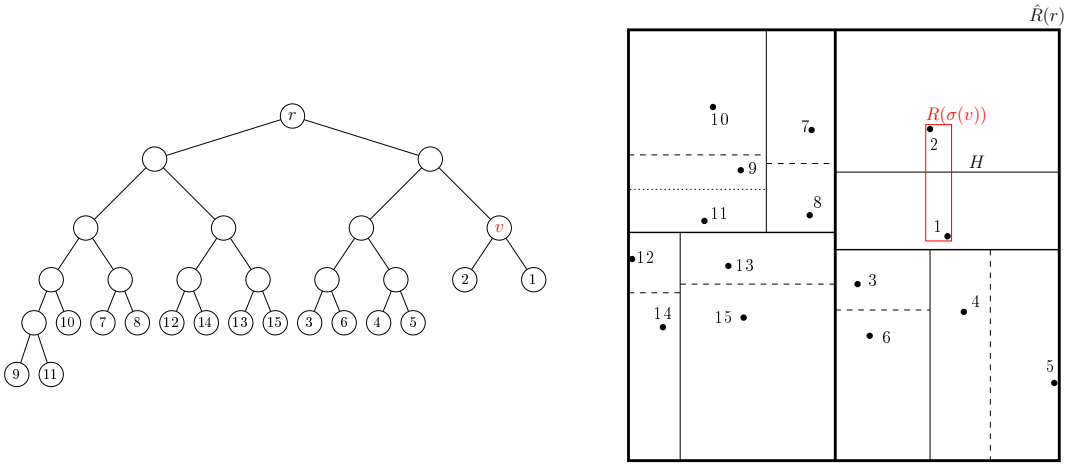
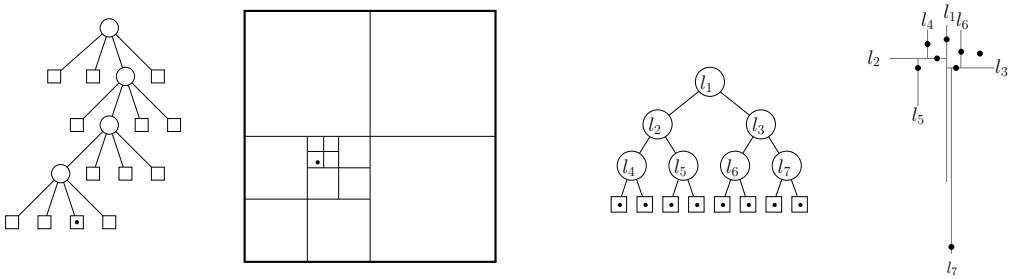


Abbildung 4.3: Darstellung eines fairen Aufteilungsbaums einer Punktmenge in der Ebene. Exemplarisch ist die faire Aufteilung der Menge $\sigma(v) = \{1, 2\}$ skizziert. Die Aufteilungshyperebene H hat bei dieser Aufteilung einen Abstand von mindestens $\frac{l_{max}(\sigma(v)}{3}$ zu den beiden zu H parallelen Seiten von $\hat{R}(v)$, vgl. auch Abbildung 4.2.

liegende Punkte eine beliebig lange Kette von Rechtecken entstehen kann, bei der jedes Rechteck nur ein nicht-leeres Kind besitzt, vgl. Abbildung 4.4 (a). Die Höhe eines *quadtrees* muss also nicht wie bei einem fairen Aufteilungsbaum linear in der Anzahl der Punkte von P sein. Ein *kd*-Baum für P besitzt hingegen eine Höhe von $\mathcal{O}(\log N)$. Der entscheidende Nachteil bei Verwendung eines *kd*-Baums besteht jedoch darin, dass keine Aussage über die geometrische Verteilung der Punkte getroffen werden kann, d. h., dass die den Knoten zugeordneten Punktmengen nicht wie bei einem *quadtree* oder bei einem fairen Aufteilungsbaum in Rechtecken enthalten sind, deren Größe sich bei zunehmender Tiefe der Knoten stark verkleinert, vgl. Abbildung 4.4 (b).



(a) Unbalancierter *quadtree* (b) *kd*-Baum

Abbildung 4.4: Zerlegungen zweier Punktmengen in der Ebene mit Hilfe eines *quadtrees* bzw. *kd*-Baums, vgl. [25]

4.2 Konstruktion einer WSPD im RAM- bzw. CREW PRAM-Modell

Callahan und Kosaraju [27] geben einen sequentiellen und einen parallelen Algorithmus zur Berechnung einer WSPD linearer Größe für eine endliche nicht-leere Punktmenge P aus dem $\mathbb{R}^d$ an. Der sequentielle Algorithmus benötigt zur Konstruktion einer solchen WSPD eine optimale Laufzeit von $\mathcal{O}(|P| \log |P|)$. Der ursprüngliche parallele Algorithmus [27] erfordert $\mathcal{O}(\log^2 |P|)$ Zeit auf einer CREW PRAM mit $\mathcal{O}(|P|)$ Prozessoren. Dieser Algorithmus wurde anschließend von Callahan [25] verbessert. Diese neue Variante, welche den ursprünglichen parallelen Algorithmus als Hilfsfunktion nutzt, benötigt nur eine Laufzeit von $\mathcal{O}(\log |P|)$ auf einer CREW PRAM mit $\mathcal{O}(|P|)$ Prozessoren. Sie bietet zudem die Grundlage für das I/O-effiziente Verfahren von Govindarajan *et al.* [40].

Sowohl die beiden in diesem Abschnitt vorgestellten Algorithmen, als auch die I/O-effiziente Variante von Govindarajan *et al.* und deren, in Abschnitt 4.3 beschriebene, abgewandelte *cache-oblivious* Version berechnen zunächst einen fairen Aufteilungsbaum der zu untersuchenden Punktmenge P um anschließend aus diesem die gesuchte Realisation von $P \otimes P$ zu entwickeln.

Die folgenden Ausführungen basieren auf den Arbeiten von Callahan und Kosaraju [25, 27] und Govindarajan *et al.* [40, 59].

4.2.1 Optimaler sequentieller Algorithmus

Der sequentielle Algorithmus ähnelt in seinem Aufbau den parallelen Algorithmen bzw. deren I/O-effizienten Varianten und vermittelt so einen guten Eindruck über das allgemeine Vorgehen bei der Erstellung eines fairen Aufteilungsbaums und der daraus entstehenden WSPD. Des Weiteren ermöglicht seine einfache Struktur eine im Vergleich zu den anderen Algorithmen relativ einfache Analyse und Implementierung.

Konstruktion eines fairen Aufteilungsbaums

Ein fairer Aufteilungsbaum T für eine endliche nicht-leere Punktmenge P lässt sich wie folgt rekursiv konstruieren: Gilt $|P| = 1$, so besteht T aus genau einem Knoten v mit $\sigma(v) = P$. Der Punkt $p \in P$ mit $\sigma(v) = \{p\}$ wird explizit in dem Knoten gespeichert. Gilt $|P| > 1$, so wird $R(P)$ durch diejenige Hyperebene in zwei geometrisch gleich große Hälften geteilt, welche senkrecht zur $i_{max}(P)$-ten Koordinatenachse steht.[3] Da die Hyperebene einen Abstand von mindestens $l_{max}(P)/3$ von beiden zu der Hyperebene parallelen Seiten

[3] O. B. d. A. sei angenommen, dass keine Punkte in der Aufteilungshyperebene liegen. Ansonsten wird eine zur ursprünglichen Aufteilungshyperebene H parallele Hyperebene $\hat{H}$ für die Aufteilung verwendet, die so gewählt wird, dass die Bedingung an eine faire Aufteilung erfüllt bleibt. Dazu wird ein Punkt $p_0 \in P$ mit $d(\{p_0\}, H) = \min_{p \in P}\{d(\{p, H\}) \mid d(\{p, H\}) > 0\}$ gesucht. Wählt man $\hat{H}$ so, dass $d(H, \hat{H}) < \frac{d(\{p_0\}, H)}{3}$, so wird die Bedingung an eine faire Aufteilung von $\hat{H}$ erfüllt. Zudem kann kein Punkt aus P in $\hat{H}$ liegen.

von $R(P)$ hat, handelt es sich um eine faire Aufteilung der Menge P. Für die dadurch entstehenden Teilmengen P_1 und P_2 werden rekursiv faire Aufteilungsbäume T_1 und T_2 berechnet. Der faire Aufteilungsbaum T besteht dann aus einer Wurzel v mit $\sigma(v) = P$ und den beiden in v gewurzelten Bäumen T_1 und T_2.

Eine naive Implementierung der obigen rekursiven Konstruktion kann jedoch zu einer quadratischen Komplexität führen, da bei einem Aufteilungsschritt evtl. eine der beiden Seiten nur einen Punkt enthält. Der von Callahan und Kosaraju [27] vorgestellte effiziente RAM-Algorithmus berechnet deshalb zuerst einen unvollständigen fairen Aufteilungsbaum von P. Anschließend werden die Blätter des unvollständigen fairen Aufteilungsbaums durch rekursiv berechnete faire Aufteilungsbäume für die korrespondierenden Mengen ersetzt. Im Folgenden soll dieses Vorgehen genauer erläutert werden. Analog zu der Arbeit von Callahan [25] wird zunächst eine Aufteilungsphase beschrieben, die der anschließende Hauptalgorithmus als Hilfsfunktion nutzt.

Die Aufteilungsphase Als Eingabe benötigt diese Phase eine endliche nicht-leere Menge $P_0 \subset \mathbb{R}^d$ und d Listen $L_1^0, \ldots, L_d^0$, wobei eine Liste L_i^0 (Referenzen auf) die $|P_0|$ Punkte in aufsteigend sortierter Reihenfolge bzgl. der i-ten Koordinate speichert.[4]

Die Ausgabe der Aufteilungsphase besteht aus einem unvollständigen fairen Aufteilungsbaum T' von P_0, dessen Blätter $v_1, \ldots, v_k$ Teilmengen $\sigma(v_1), \ldots, \sigma(v_k)$ von P_0 mit höchstens $|P_0|/2$ Punkten repräsentieren. Des Weiteren werden in jedem Blatt v_j des Baums d Listen gespeichert, die die Punkte aus $\sigma(v_j)$ enthalten. Diese sind dabei pro Dimension $i \in \{1, \ldots, d\}$ in aufsteigend sortierter Reihenfolge bzgl. der i-ten Dimension in genau einer der Listen enthalten.

Zu Beginn der Phase werden zunächst die Eingabelisten $L_1^0, \ldots, L_d^0$ erweitert: Durch eine Traversierung aller d Listen lassen sich aus diesen doppelt-verkettete Listen mit Querverweisen zwischen den einzelnen Elementen erstellen. Die erweiterten Listen seien wieder mit $L_1^0, \ldots, L_d^0$ bezeichnet. Zudem werden Kopien dieser Listen erstellt, welche am Ende der Aufteilungsphase benötigt werden.

Der Algorithmus berechnet anschließend eine Sequenz fairer Aufteilungen der Menge P_0. Nach jeder Aufteilung liegen zwei nicht-leere Mengen vor. Die Menge mit der größeren Anzahl von Punkten wird jeweils sukzessiv weiter aufgeteilt. Der Aufteilungsprozess bricht ab, sobald nach einer Aufteilung beide Mengen weniger als $|P_0|/2$ Punkte enthalten. Da P_0 eine endliche Punktmenge ist, geschieht dies nach endlich vielen Schritten. Für die folgenden Betrachtungen wird die nach der j-ten Aufteilung ($j \geq 1$) zahlenmäßig größere der beiden Mengen mit P_j und die kleinere Menge mit $\hat{P}_j$ bezeichnet. Während des Aufteilungsprozesses werden weiterhin für die Mengen P_j bzw. $\hat{P}_j$ Listen $L_1^j, \ldots, L_d^j$ bzw. $\hat{L}_1^j, \ldots, \hat{L}_d^j$ erstellt, wobei eine Liste L_i^j bzw. $\hat{L}_i^j$ die Punkte aus P_j bzw. $\hat{P}_j$ in aufsteigend sortierter Reihenfolge bzgl. der i-ten Koordinate enthält.

[4] Alle in diesem Algorithmus verwendeten Listen speichern Referenzen auf die jeweiligen Punkte.

Zur Beschreibung der j-ten Aufteilung sei nun die Menge P_{j-1} gegeben. Für $j = 1$ ist dies die Startmenge P_0. Im Fall $j > 1$ ist P_{j-1} die zahlenmäßig größere der beiden Mengen, welche durch die $(j-1)$-te Aufteilung entstanden sind. Die Menge P_{j-1} wird nun analog zu der eingangs beschriebenen fairen Aufteilung in zwei Teilmengen P_j und $\hat{P}_j$ zerlegt. Die für diese Aufteilung benötigte Dimension $i_{max} = i_{max}(P_{j-1})$ kann dabei mit Hilfe der für die Punktmenge P_{j-1} vorliegenden Listen $L_1^{j-1}, \ldots, L_d^{j-1}$ in einer Laufzeit von $\mathcal{O}(1)$ bestimmt werden. Durch die faire Aufteilung wird die Liste $L_{i_{max}}^{j-1}$ in zwei jeweils zusammenhängende Segmente aufgeteilt. Die Trennposition für diese Aufteilung kann deshalb mittels einer linearen Suche, welche von den beiden Endpunkten der Liste aus alternierend zur Mitte hin verläuft, in einer zu der Länge des kleineren Segments proportionalen Laufzeit gefunden werden.

Die Punkte aus $\hat{P}_j$ werden anschließend aus den Listen $L_1^{j-1}, \ldots, L_d^{j-1}$ gelöscht. Die dadurch entstehenden Listen seien mit $L_1^j, \ldots, L_d^j$ bezeichnet. Diese werden für den nächsten Aufteilungsschritt benötigt und enthalten die Punkte aus P_j in sortierter Reihenfolge bzgl. der jeweiligen Koordinate. Der Löschvorgang kann durch eine Traversierung der Liste $L_{i_{max}}^{j-1}$ beginnend von dem der Trennposition näher liegenden Ende mit Hilfe der Querverweise bewerkstelligt werden. Ebenso wie die Suchzeit ist die für diesen Vorgang benötigte Zeit proportional zu der Länge des kürzeren Segments.

Die aus den Listen $L_1^{j-1}, \ldots, L_d^{j-1}$ gelöschten Punkte sind die Punkte aus der Menge $\hat{P}_j$. Für diese Menge müssen ebenfalls sortierte Listen $\hat{L}_1^j, \ldots, \hat{L}_d^j$ erstellt werden. Da die gelöschten Punkte für $i \neq i_{max}$ in einer beliebigen Reihenfolge in der Liste L_i^{j-1} enthalten sind, ist die Erstellung der neuen Listen an dieser Stelle nicht effizient. Anstatt diese Listen zu diesem Zeitpunkt zu erstellen, werden deshalb zunächst leere Listen $\hat{L}_1^j, \ldots, \hat{L}_d^j$ erzeugt und die aus den Listen $L_1^{j-1}, \ldots, L_d^{j-1}$ gelöschten Punkte jeweils um d Zeiger auf diese (leeren) Listen ergänzt. *Alle* in den einzelnen Schritten gelöschten Punkte werden dann am Ende der Phase mittels *einer* Traversierung der zu Beginn der Phase erstellten Kopien der Listen $L_1^0, \ldots, L_d^0$ in die für sie vorgesehenen Listen eingefügt.

Der obige Aufteilungsprozess wird, beginnend mit der Startmenge P_0, so lange wiederholt, bis $|P_j| \leq \frac{|P_0|}{2}$ gilt. Die Anzahl der dafür nötigen Aufteilungsschritte sei k. Da alle Aufteilungen faire Aufteilungen sind, entsteht durch diesen Prozess ein unvollständiger fairer Aufteilungsbaum T' von P_0, vgl. Abbildung 4.5: Die Mengen $P_0, P_1, \ldots, P_k$ werden durch eine Kette von Knoten $v_0, v_1, \ldots, v_k$ mit $\sigma(v_j) = P_j$ repräsentiert, die während des Prozesses sukzessiv erstellt werden. Für jeden Knoten v_j wird weiterhin ein Geschwisterknoten $\hat{v}_j$ mit $\sigma(\hat{v}_j) = \hat{P}_j$ zu T' hinzugefügt. Der Gesamtaufwand für diesen Aufteilungsprozess ist linear in der Anzahl der gelöschten Punkte und somit insgesamt linear. In jedem der Blätter $\hat{v}_1, \ldots, \hat{v}_k, v_k$ werden während dieses Prozesses die zugehörigen Listen gespeichert.

Nach der Berechnung der Aufteilungen müssen die in den einzelnen Schritten gelöschten Punkte in die für sie vorgesehenen Listen $\hat{L}_1^j, \ldots, \hat{L}_d^j$ $(1 \leq j \leq k)$ eingefügt werden. Dazu iteriert man über alle d Dimensionen. Pro Dimension i wird die zu Beginn der Phase

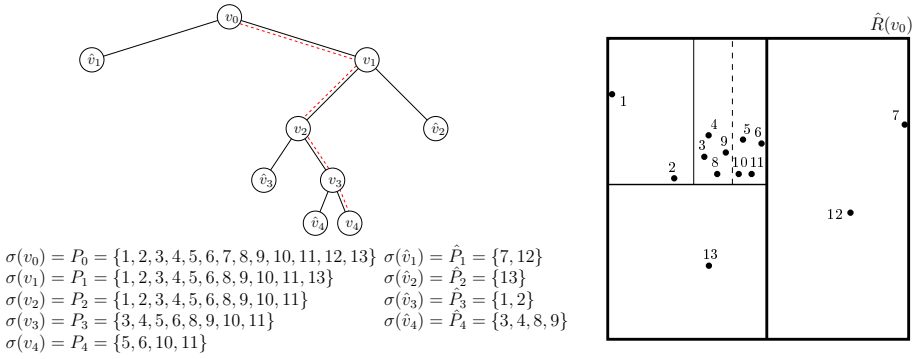


Abbildung 4.5: Schematische Darstellung eines durch den Aufteilungsprozess entstehenden unvollständigen fairen Aufteilungsbaums einer Punktmenge $P_0 \subset \mathbb{R}^2$, die aus 13 Punkten besteht. Der Weg von v_0 nach v_4 wird durch die gestrichelten Linien hervorgehoben. Die Blätter des Baums repräsentieren die Mengen $\hat{P}_1, \hat{P}_2, \hat{P}_3, \hat{P}_4$ und P_4, welche jeweils höchstens $\frac{13}{2}$ Punkte enthalten.

angefertigte Kopie der Liste L_i^0 in aufsteigend sortierter Reihenfolge durchlaufen. Ein traversierter Punkt enthält entweder keinen Zeiger auf eine der obigen Listen (dann ist der Punkt in der Menge P_k) oder enthält d Zeiger auf Listen $\hat{L}_1^{j'}, \ldots, \hat{L}_d^{j'}$ mit $j' \in \{1, \ldots, k\}$. Enthält ein Punkt Zeiger auf Listen $\hat{L}_1^{j'}, \ldots, \hat{L}_d^{j'}$, so wird dieser bei dem aktuellen Durchlauf an das Ende der Liste $\hat{L}_i^{j'}$ angehängt. Nach diesem Vorgang enthält für $j = 1, \ldots, k$ die Liste $\hat{L}_i^j$ alle Punkte aus $P_{j-1} - P_j$ in sortierter Reihenfolge bzgl. der i-ten Dimension. Da d konstant ist, benötigt die Verteilung aller Punkte auf die entsprechenden Listen insgesamt eine in der Anzahl der Punkte aus $|P_0|$ lineare Laufzeit.

Die Aufteilungsphase erstellt also in linearer Zeit einen unvollständigen fairen Aufteilungsbaum für die vorsortierte Eingabemenge P_0. Jedes Blatt des Baums repräsentiert eine Menge, welche aus höchstens $|P_0|/2$ Elementen besteht. Die Ausgabe der Phase beinhaltet zudem in den Blättern gespeicherte sortierte Listen für diese Mengen, welche den folgenden *divide and conquer*-Algorithmus ermöglichen.

Der Hauptalgorithmus Um die Funktion SequentialFairSplitTree (Algorithmus 4.1) auf eine Punktmenge aus dem $\mathbb{R}^d$ anwenden zu können, müssen entsprechende sortierte Listen $L_1, \ldots, L_d$ vorhanden sein. Die für den initialen Aufruf der Funktion SequentialFairSplitTree benötigten Listen werden dazu in einem Vorsortierungsschritt erstellt. Die für die rekursiven Aufrufe der Funktion benötigten sortierten Listen werden durch die Ausgabe der oben beschriebenen Aufteilungsphase bereitgestellt.

Der Hauptalgorithmus zur Konstruktion eines fairen Aufteilungsbaums besteht dann aus diesem Vorsortierungsschritt und der darauf folgenden Anwendung der Funktion SequentialFairSplitTree auf die Startmenge mit den zugehörigen sortierten Listen. Die Analyse dieses Algorithmus im RAM-Modell wird durch das folgende Lemma gegeben.

Funktion SequentialFairSplitTree$(P, L_1, \ldots, L_d)$

Eingabe: Eine nicht-leere endliche Punktmenge $P \subset \mathbb{R}^d$ und d Listen $L_1, \ldots, L_d$, wobei eine Liste L_i die $|P|$ Punkte in aufsteigend sortierter Reihenfolge bzgl. der i-ten Koordinate enthält.

Ausgabe: Ein fairer Aufteilungsbaum T von P.

1: **if** $|P| = 1$ **then**
2: Der Aufteilungsbaum T von P besteht aus einem Knoten v mit $\sigma(v) = P$. Der Punkt p mit $P = \{p\}$ wird explizit in diesem Knoten gespeichert.
3: **else**
4: Berechne mittels der Aufteilungsphase einen unvollständigen fairen Aufteilungsbaum T' von P. Seien $v_1, \ldots, v_k$ die Blätter von T'. Für jede Menge $\sigma(v_i)$ liegen wiederum sortierte Listen $L_1^i, \ldots, L_d^i$ vor und es gilt $|\sigma(v_i)| \leq |P|/2$.
5: **for** $i = 1$ to k **do**
6: Wende die Funktion SequentialFairSplitTree rekursiv auf die Punktmenge $\sigma(v_i)$ und deren zugehörigen Listen an um einen fairen Aufteilungsbaum T_i von $\sigma(v_i)$ zu berechnen. Das Blatt v_i von T' wird in Zeile 8 durch den Baum T_i ersetzt.
7: **end for**
8: $T = T' \cup T_1 \cup \cdots \cup T_k$
9: **end if**
10: **return** T

Algorithmus 4.1: Berechnung eines fairen Aufteilungsbaums im RAM-Modell, vgl. [25]

4.2.1 Lemma ([25, 27]). *Sei eine endliche nicht-leere Punktmenge $P \subset \mathbb{R}^d$ gegeben. Dann kann in einer Laufzeit von $\mathcal{O}(|P| \log |P|)$ ein fairer Aufteilungsbaum für P konstruiert werden. Der Speicherplatzbedarf des fairen Aufteilungsbaums liegt dabei in $\mathcal{O}(|P|)$.*

Beweis. In einer Laufzeit von $\mathcal{O}(|P| \log |P|)$ lassen sich die Punkte bzgl. einer Dimension sortieren und in der zugehörigen Liste speichern. Da d konstant ist, benötigt diese Vorsortierung für alle d Listen insgesamt $\mathcal{O}(d \cdot |P| \log |P|) = \mathcal{O}(|P| \log |P|)$ Zeit. Eine Anwendung der Funktion SequentialFairSplitTree auf die Punktmenge P und die erstellten Listen liefert dann den gewünschten fairen Aufteilungsbaum T von P: Besteht die Punktmenge P aus nur einem Punkt, so wird in Zeile 2 der Funktion der aus diesem einen Knoten bestehende faire Aufteilungsbaum T erstellt und anschließend zurückgegeben. Ansonsten wird mittels der Aufteilungsphase ein unvollständiger fairer Aufteilungsbaum T' von P berechnet, dessen Blätter Mengen mit höchstens $|P|/2$ Elementen repräsentieren. Für diese Mengen werden in Zeile 6 jeweils rekursiv faire Aufteilungsbäume berechnet. In Zeile 8 werden dann die Blätter des unvollständigen fairen Aufteilungsbaums durch diese rekursiv berechneten Bäume ersetzt. Der so konstruierte Baum T ist offensichtlich ein fairer Aufteilungsbaum von P.

Um den Gesamtaufwand dieser rekursiven Konstruktion zu bestimmen, sei der Aufwand betrachtet, der jeweils pro Rekursionsebene anfällt: Da die durch einen Funktionsaufruf ent-

stehenden Mengen $\sigma(v_1), \ldots, \sigma(v_k)$ eine disjunkte Zerlegung der jeweiligen Eingabemenge bilden, ist die Vereinigung der durch alle Funktionsaufrufe einer Rekursionsebene induzierten Mengen eine disjunkte Zerlegung der ursprünglichen Startmenge P. Aufgrund des linearen Laufzeitbedarfs der Aufteilungsphasen in Zeile 4 beträgt der Gesamtaufwand für alle Aufteilungsphasen einer Rekursionsebene demnach insgesamt höchstens $\mathcal{O}(|P|)$. Die Ersetzung der Blätter des unvollständigen fairen Aufteilungsbaums T' in Zeile 8 durch die rekursiv berechneten fairen Aufteilungsbäume kann ebenso in einer Laufzeit von $\mathcal{O}(|P|)$ realisiert werden. Der Gesamtaufwand *aller* Funktionsaufrufe einer Rekursionsebene ist demnach linear in der Anzahl der Punkte aus P.

Da die Anzahl der Eingabeelemente der rekursiven Funktionsaufrufe pro Rekursionsebene mindestens halbiert wird, besitzt der durch alle Funktionsaufrufe induzierte Rekursionsbaum eine maximale Tiefe von $\log |P|$. Der Gesamtaufwand für die Konstruktion des fairen Aufteilungsbaums beträgt demnach insgesamt $\mathcal{O}(|P| \log |P|)$. Die Aussage über den verwendeten Speicherplatz ergibt sich aus der linearen Größe des Baums und der Tatsache, dass in jedem Knoten nur konstant viele Informationen gespeichert sind. □

Während der Erstellung eines Knotens v des fairen Aufteilungsbaums T durch den obigen Algorithmus kann mit Hilfe der sortierten Listen in $\mathcal{O}(d) = \mathcal{O}(1)$ Zeit das Begrenzungsrechteck $R(\sigma(v))$ von $\sigma(v)$ bestimmt werden. Für die folgenden Betrachtungen kann deshalb angenommen werden, dass in jedem Knoten v aus T das zugehörige Rechteck $R(\sigma(v))$ gespeichert ist. Die Berechnung der kleinsten d-Kugel, welche das Rechteck $R(\sigma(v))$ enthält, ist demnach für jeden Knoten v aus T in einer konstanten Laufzeit möglich.

Konstruktion einer WSPD

Sei $P = \{p_1, \ldots, p_n\}$ eine endliche nicht-leere Punktmenge aus dem $\mathbb{R}^d$. Zu dieser lässt sich mittels des obigen Algorithmus ein fairer Aufteilungsbaum T von P berechnen. Es wird nun gezeigt, wie sich mit Hilfe eines solchen Baums eine scharf getrennte Realisation $\mathcal{R} = \{\{A_1, B_1\}, \ldots, \{A_k, B_k\}\}$ von $P \otimes P$ linearer Größe konstruieren lässt. Die Existenz einer scharf getrennten Realisation von $P \otimes P$ ist gesichert, da die Menge $\{\{p_i, p_j\} \mid 1 \leq i < j \leq n\}$ eine bzgl. jeden Wertes $s \in \mathbb{R}^+$ scharf getrennte Realisation von $P \otimes P$ darstellt. Diese Realisation besteht aber aus $\binom{n}{2}$ vielen Paaren und besitzt demnach eine quadratische Größe.

Aufgrund der obigen Bemerkung kann angenommen werden, dass in jedem Knoten $v \in T$ das zugehörige Begrenzungsrechteck $R(\sigma(v))$ gespeichert ist. Für zwei beliebige Knoten v und w aus T kann deshalb in konstanter Laufzeit geprüft werden, ob die Mengen $\sigma(v)$ und $\sigma(w)$ bzgl. eines Wertes $s \in \mathbb{R}^+$ scharf getrennt sind oder nicht. Die Paare $\{A_i, B_i\}$ der im Folgenden berechneten Realisation werden weiterhin durch Knotenpaare $\{v_i, w_i\}$ mit Knoten aus T und *nicht* explizit durch die Punkte der Mengen A_i und B_i dargestellt. Der Grund dafür ist, dass eine explizite Darstellung der Realisation anhand

der Punkte aus den Mengen $A_1, B_1, \ldots, A_k, B_k$ zu einer quadratischen Komplexität führen kann, d.h. $\sum_{i=1}^{k} (|A_i| + |B_i|)$ kann quadratisch in $|P|$ sein, vgl. Callahan [25].

Die Funktion ComputeWSR (Algorithmus 4.2) stellt den Algorithmus zur Berechnung einer Realisation linearer Größe dar. Diese Funktion greift auf eine Hilfsfunktion namens FindPairs zurück, für deren Beschreibung die folgende Relation zwischen zwei Knoten v und w aus T benötigt wird: Für zwei Knoten v und w aus T sei die Relation $v \prec w$ definiert durch

$$v \prec w \quad :\Leftrightarrow \quad l_{max}(\sigma(v)) \leq l_{max}(\sigma(w)) \quad \vee \quad \Big(l_{max}(\sigma(v)) = l_{max}(\sigma(w)) \wedge \nu(v) < \nu(w)\Big),$$

wobei ν eine beliebige, fest gewählte Postorder-Nummerierung sei. Für zwei verschiedene Knoten v und w gilt somit entweder $v \prec w$ oder $w \prec v$. Zudem gilt $v \prec p(v)$ für jeden von der Wurzel verschiedenen Knoten $v \in T$ mit Vaterknoten $p(v)$. Die Relation $v \preceq w$ wird weiterhin durch $v \preceq w :\Leftrightarrow v \prec w \vee v = w$ definiert.

Durch das folgende Lemma wird nun zunächst die Hilfsfunktion FindPairs analysiert.

4.2.2 Lemma ([25, 27]). *Sei ein fairer Aufteilungsbaum T einer endlichen nicht-leeren Punktmenge $P \subset \mathbb{R}^d$ und eine Konstante $s \in \mathbb{R}^+$ gegeben. Die durch die Ausgabe der Funktion* FindPairs*, angewendet auf T, s und ein Paar $\{v, w\}$ von Knoten aus T, dargestellte Menge $\{\{\sigma(v_1), \sigma(w_1)\}, \ldots, \{\sigma(v_k), \sigma(w_k)\}\}$ ist eine bzgl. s scharf getrennte Realisation von $\sigma(v) \otimes \sigma(w)$, die T benutzt.*

Beweis. Sind die Mengen $\sigma(v)$ und $\sigma(w)$ bzgl. s scharf getrennt, so ist die Behauptung trivial. Ansonsten kann induktiv angenommen werden, dass die Funktion Darstellungen

$$\mathcal{R}'_1 = \{\{c_1, d_1\}, \ldots, \{c_{k_1}, d_{k_1}\}\} \quad \text{bzw.} \quad \mathcal{R}'_2 = \{\{e_1, f_1\}, \ldots, \{e_{k_2}, f_{k_2}\}\}$$

von bzgl. s scharf getrennten Realisationen der Interaktionsprodukte $\sigma(\bar{v}_1) \otimes \sigma(w)$ bzw. $\sigma(\bar{v}_2) \otimes \sigma(w)$ berechnet. Die Vereinigung $\mathcal{R}' = \mathcal{R}'_1 \cup \mathcal{R}'_2 = \{\{v_1, w_1\}, \ldots, \{v_k, w_k\}\}$ dieser Paare ist dann eine Darstellung einer bzgl. s scharf getrennten Realisation von $\sigma(v) \otimes \sigma(w)$: Die zu zeigenden Eigenschaften R1, R2 und R5 folgen leicht aus $\sigma(\bar{v}_1), \sigma(\bar{v}_2) \subset \sigma(v)$ und den Eigenschaften der durch $\mathcal{R}'_1$ und $\mathcal{R}'_2$ dargestellten Realisationen. Die Eigenschaften R3 bzw. R4 folgen aus $\sigma(\bar{v}_1) \cap \sigma(\bar{v}_2) = \emptyset$ bzw. aus $\sigma(v) = \sigma(\bar{v}_1) \cup \sigma(\bar{v}_2)$, ebenso unter Verwendung der Eigenschaften der durch $\mathcal{R}'_1$ und $\mathcal{R}'_2$ dargestellten Realisationen. Die Terminierung eines Funktionsaufrufs von FindPairs ergibt sich aus $|\sigma(\bar{v}_1)| \cdot |\sigma(w)| < |\sigma(v)| \cdot |\sigma(w)|$ bzw. $|\sigma(\bar{v}_2)| \cdot |\sigma(w)| < |\sigma(v)| \cdot |\sigma(w)|$. □

Es wird nun die Funktion ComputeWSR analysiert. Um eine Darstellung der gewünschten Realisation zu berechnen, wird in Zeile 2 dieser Funktion zunächst für jeden inneren Knoten aus T mit Kindern v und w das Paar $\{v, w\}$ zu der Menge R hinzugefügt. Wie aus dem Beweis des folgenden Lemmas ersichtlich ist, stellt die auf diese Weise konstruierte Menge R eine Realisation von $P \otimes P$ dar. Da diese im Allgemeinen jedoch nicht bzgl. $s \in \mathbb{R}^+$

Funktion ComputeWSR(P, T, s)

Eingabe: Eine endliche nicht-leere Punktmenge $P \subset \mathbb{R}^d$, ein fairer Aufteilungsbaums T von P und eine Konstante $s \in \mathbb{R}^+$.

Ausgabe: Eine Menge $\mathcal{R} = \{\{v_1, w_1\}, \ldots, \{v_k, w_k\}\}$ von Knotenpaaren aus T. Die durch die Knotenpaare dargestellte Menge $\{\{\sigma(v_1), \sigma(w_1)\}, \ldots, \{\sigma(v_k), \sigma(w_k)\}\}$ ist eine bzgl. des Wertes s scharf getrennte Realisation von $P \otimes P$, die T benutzt.

1: $R \leftarrow \emptyset$
2: Füge für jeden inneren Knoten aus T mit Kindern v und w das Paar $\{v, w\}$ zu R hinzu.
3: $\mathcal{R} \leftarrow \emptyset$
4: **for** jedes Paar $\{v, w\} \in R$ **do**
5: Entferne $\{v, w\}$ aus R.
6: $\mathcal{R} \leftarrow \mathcal{R} \cup$ FindPairs$(T, \{v, w\}, s)$
7: **end for**
8: **return** $\mathcal{R}$

Funktion FindPairs$(T, \{v, w\}, s)$

Eingabe: Ein fairer Aufteilungsbaum T einer endlichen nicht-leeren Punktmenge $P \subset \mathbb{R}^d$, ein Paar $\{v, w\}$ von Knoten aus T und eine Konstante $s \in \mathbb{R}^+$.

Ausgabe: Eine Menge $\mathcal{R}' = \{\{v_1, w_1\}, \ldots, \{v_k, w_k\}\}$ von Knotenpaaren aus T. Die durch die Knotenpaare dargestellte Menge $\{\{\sigma(v_1), \sigma(w_1)\}, \ldots, \{\sigma(v_k), \sigma(w_k)\}\}$ ist eine bzgl. s scharf getrennte Realisation von $\sigma(v) \otimes \sigma(w)$, die T benutzt.

1: O. B. d. A. gelte $w \prec v$.
2: **if** $\sigma(v)$ und $\sigma(w)$ sind scharf getrennt bzgl. s **then**
3: $\mathcal{R}' = \{\{v, w\}\}$
4: **else**
5: Es gilt $|\sigma(v)| > 1$.
6: Seien $\bar{v}_1$ und $\bar{v}_2$ die beiden Kinder von v in T.
7: $\mathcal{R}' =$ FindPairs$(T, \{\bar{v}_1, w\}, s) \cup$ FindPairs$(T, \{\bar{v}_2, w\}, s)$
8: **end if**
9: **return** $\mathcal{R}'$

Algorithmus 4.2: Berechnung einer scharf getrennten Realisation im RAM-Modell, vgl. [40, 59]

scharf getrennt ist, wird in Zeile 6 von ComputeWSR für jedes Paar $\{v, w\} \in R$ mit Hilfe der Funktion FindPairs eine Darstellung einer bzgl. s scharf getrennten Realisation von $\sigma(v) \otimes \sigma(w)$ berechnet. Dass die Vereinigung $\mathcal{R}$ aller dadurch entstehenden Knotenpaare eine Darstellung einer bzgl. s scharf getrennten Realisation von $P \otimes P$ ist, wird nun durch das folgende Lemma gezeigt:

4.2.3 Lemma ([25, 27]). *Sei eine endliche nicht-leere Punktmenge $P \subset \mathbb{R}^d$, ein fairer Aufteilungsbaum T von P und eine Konstante $s \in \mathbb{R}^+$ gegeben. Die durch die Ausgabe der Funktion* ComputeWSR *(Algorithmus 4.2), angewendet auf T, P und s, dargestellte*

Menge $\{\{\sigma(v_1), \sigma(w_1)\}, \ldots, \{\sigma(v_k), \sigma(w_k)\}\}$ *ist eine bzgl.* s *scharf getrennte Realisation von* $P \otimes P$*, die* T *benutzt.*

Beweis. Nach Lemma 4.2.2 berechnet die Funktion FindPairs zu jedem Paar $\{v, w\}$ eine Darstellung einer bzgl. s scharf getrennten Realisation von $\sigma(v) \otimes \sigma(w)$. Somit erfüllt die Menge $\{\{\sigma(v_1), \sigma(w_1)\}, \ldots, \{\sigma(v_k), \sigma(w_k)\}\}$ die Eigenschaften R1, R2 und R5. Um die Eigenschaften R3 und R4 zu verifizieren, muss bewiesen werden, dass jedes Paar verschiedener Punkte aus P in genau einem der Interaktionsprodukte $\sigma(v_i) \otimes \sigma(w_i)$ enthalten ist. Seien dazu ein Paar $\{p_1, p_2\}$ aus $P \otimes P$ und die beiden Blätter a, b aus T mit $\sigma(a) = \{p_1\}$ und $\sigma(b) = \{p_2\}$ gegeben. Weiterhin sei c der kleinste gemeinsame Vorfahre von a und b sowie c_1 und c_2 die Kinder von c. Eine Anwendung der Funktion FindPairs auf das Knotenpaar $\{c_1, c_2\}$ liefert dann eine Darstellung einer bzgl. s scharf getrennten Realisation von $\sigma(c_1) \otimes \sigma(c_2)$. Demnach existiert ein $i' \in \{1, \ldots k\}$ mit $\{p_1, p_2\} \in \sigma(v_{i'}) \otimes \sigma(w_{i'})$.

Es muss noch gezeigt werden, dass es keinen Index $i'' \in \{1, \ldots, k\}$ mit $i'' \neq i'$ und $\{p_1, p_2\} \in \sigma(v_{i''}) \otimes \sigma(w_{i''})$ gibt. Dazu sei ein von c verschiedener innerer Knoten $\hat{c}$ aus T gegeben. Für die Kinder $\hat{c}_1$ und $\hat{c}_2$ dieses Knotens gilt dann genau einer der drei folgenden Fälle:

(i) $p_1 \notin \sigma(\hat{c}_1) \cup \sigma(\hat{c}_2)$

(ii) $p_2 \notin \sigma(\hat{c}_1) \cup \sigma(\hat{c}_2)$

(iii) $(p_1, p_2 \in \sigma(\hat{c}_1) \wedge p_1, p_2 \notin \sigma(\hat{c}_1)) \quad \vee \quad (p_1, p_2 \in \sigma(\hat{c}_2) \wedge p_1, p_2 \notin \sigma(\hat{c}_2)) \quad \vee \quad (p_1, p_2 \notin \sigma(\hat{c}_1) \cup \sigma(\hat{c}_2)$

Durch eine Anwendung der Funktion FindPairs auf die Kinder eines von c verschiedenen inneren Knoten kann somit kein weiteres Paar $\{v_{i''}, w_{i''}\}$ mit $i'' \in \{1, \ldots, k\}$, $i'' \neq i'$ und $\{p_1, p_2\} \in \sigma(v_{i''}) \otimes \sigma(w_{i''})$ entstehen. □

Um die Laufzeit dieses Konstruktionsverfahrens zu bestimmen, führen Callahan und Kosaraju [27] den Begriff eines *Berechnungsbaums* ein. Der Berechnungsbaum $T(\{v, w\})$ eines Knotenpaares $\{v, w\}$ mit Knoten v und w aus T und $w \prec v$ ist ein Binärbaum mit Wurzelknoten (v, w). Sind die beiden Mengen $\sigma(v)$ und $\sigma(w)$ bzgl. s scharf getrennt, so besteht $T(\{v, w\})$ nur aus diesem einen Knoten. Ansonsten seien v_1 und v_2 die beiden Kinder von v. Der Berechnungsbaum $T(\{v, w\})$ von $\{v, w\}$ besteht dann aus der Wurzel (v, w) und den beiden in (v, w) gewurzelten Berechnungsbäumen $T(\{v_1, w\})$ und $T(\{v_2, w\})$.

Die Funktion ComputeWSR wendet die Hilfsfunktion FindPairs auf die Kinder v und w eines jeden inneren Knotens des fairen Aufteilungsbaums T an. Der Berechnungsbaum $T(\{v, w\})$ von $\{v, w\}$ stellt dann die rekursive Berechnung der Knotenpaare durch die Funktion FindPairs dar. Ein Blatt des Berechnungsbaums korrespondiert dabei mit einem Knotenpaar der berechneten Menge. Aufgrund dieser Korrespondenz zwischen den

Blättern eines Berechnungsbaums und den jeweiligen Knotenpaaren folgt, dass die Gesamtgröße *aller* Berechnungsbäume $T(\{v, w\})$ mit $\{v, w\} \in R$ linear in der Anzahl der insgesamt berechneten Knotenpaare ist. Eine Anwendung der Funktion ComputeWSR benötigt demnach eine in der Anzahl der berechneten Knotenpaare lineare Laufzeit.

Um die Anzahl der insgesamt berechneten Knotenpaare zu bestimmen, verwenden Callahan und Kosaraju [27] ein Kompaktheitsargument. Callahan [25] präzisiert dieses Verfahren, indem er den Begriff der sogenannten *kanonischen Realisation* einführt. Auf diese Vorgehensweisen soll hier jedoch nicht näher eingegangen werden. Stattdessen wird nur das Gesamtergebnis dieser Überlegungen gegeben:

4.2.4 Theorem ([25, 27]). *Sei eine endliche nicht-leere Punktmenge $P \subset \mathbb{R}^d$, ein fairer Aufteilungsbaum T von P und eine Konstante $s \in \mathbb{R}^+$ gegeben. Dann kann in einer Laufzeit von $\mathcal{O}(s^d|P|) = \mathcal{O}(|P|)$ eine WSPD $\mathcal{D} = (T, \mathcal{R})$ von P bzgl. s berechnet werden, deren Größe $\mathcal{O}(s^d|P|) = \mathcal{O}(|P|)$ beträgt.*

Insgesamt ergibt sich also ein Konstruktionsverfahren, welches für eine endliche nicht-leere Punktmenge $P \subset \mathbb{R}^d$ eine bzgl. eines Wertes $s \in \mathbb{R}^+$ scharf getrennte WSPD $\mathcal{D} = (T, \mathcal{R})$ von P linearer Größe konstruiert. Die Laufzeit dieses Verfahrens setzt sich aus der Laufzeit für die Berechnung des fairen Aufteilungsbaums T und der Laufzeit für die Berechnung der Realisation $\mathcal{R}$ zusammen und beträgt somit $\mathcal{O}(|P| \log |P| + s^d|P|) = \mathcal{O}(|P| \log |P|)$. Die durch dieses Verfahren konstruierte Realisation $\mathcal{R}$ wird dabei durch eine Menge $\mathcal{R}_T = \{\{v_1, w_1\}, \ldots, \{v_k, w_k\}\}$ von Knotenpaaren mit Knoten $v_i, w_i \in T$ dargestellt.

4.2.2 Optimaler paralleler Algorithmus

Der optimale parallele Algorithmus von Callahan [25] bietet die Grundlage für die I/O-effiziente Variante von Govindarajan *et al.* [40], welche wiederum in weiten Teilen mit dem im Kontext des *cache-oblivious*-Modells effizienten Algorithmus aus Abschnitt 4.3 übereinstimmt. Da dieser in Abschnitt 4.3 ausführlich beschrieben wird, soll hier nur kurz auf die parallelen Algorithmen [25, 27] zur Konstruktion einer WSPD eingegangen werden.

Die ursprüngliche Version des parallelen Algorithmus [27] zur Konstruktion einer WSPD berechnet zuerst einen fairen Aufteilungsbaum T der zu untersuchenden endlichen nicht-leeren Punktmenge $P \subset \mathbb{R}^d$. Die dafür benötigte (suboptimale) Laufzeit beträgt $\mathcal{O}(\log^2 |P|)$ auf einer CREW PRAM mit $\mathcal{O}(|P|)$ Prozessoren. Mit Hilfe dieses Aufteilungsbaums wird dann unter Verwendung von $\mathcal{O}(|P|)$ Prozessoren in einer Laufzeit von $\mathcal{O}(\log |P|)$ die benötigte Realisation berechnet. Durch das von Callahan [27] entwickelte verbesserte Konstruktionsverfahrens eines fairen Aufteilungsbaums entsteht dann ein effizientes Verfahren zur Konstruktion einer WSPD auf einer CREW PRAM:

4.2.5 Theorem ([25]). *Sei eine endliche nicht-leere Punktmenge P aus dem $\mathbb{R}^d$ und eine Konstante $s \in \mathbb{R}^+$ gegeben. Dann kann in einer Laufzeit von $\mathcal{O}(\log |P|)$ auf einer CREW*

PRAM mit $\mathcal{O}(|P|)$ Prozessoren eine WSPD von P bzgl. s konstruiert werden, deren Größe linear in der Anzahl der Punkte aus P ist.

Die verbesserte Konstruktion des fairen Aufteilungsbaums basiert auf einer Technik namens *multi-way divide and conquer*. Zur Lösung eines Problems der Größe N zerlegt diese Technik das Problem in Teilprobleme der Größe $\mathcal{O}(N^\alpha)$ mit $0 \leq \alpha < 1$ und löst diese Teilprobleme rekursiv. Bei Anwendung dieser Technik auf das Problem der Konstruktion eines fairen Aufteilungsbaums muss jedoch darauf geachtet werden, dass die Menge nur durch faire Aufteilungen zerlegt wird – es dürfen also nicht beliebige Hyperebenen zur Aufteilung der jeweiligen Mengen verwendet werden, vgl. Callahan [25]

Callahans optimaler paralleler Algorithmus zur Konstruktion eines fairen Aufteilungsbaums T einer endlichen nicht-leeren Punktmenge $P \subset \mathbb{R}^d$ ist wie folgt aufgebaut: Für P wird zunächst ein unvollständiger fairer Aufteilungsbaum T' konstruiert, dessen Blätter Mengen mit höchstens $|P|^\alpha$ $(0 \leq \alpha < 1)$ Elementen repräsentieren. Anschließend werden die Blätter von T' durch rekursiv berechnete faire Aufteilungsbäume für die entsprechenden Mengen ersetzt. Der Aufbau des Algorithmus ähnelt also in seiner Struktur der des sequentiellen Algorithmus.

Kernidee bei der Konstruktion des unvollständigen fairen Aufteilungsbaums T' ist die Konstruktion eines gewissen Obergraphen G, dessen Ecken mit sogenannten „konstruierbaren Rechtecken“ korrespondieren. Anschließend wird aus diesem Obergraphen ein Teilgraph T_c extrahiert, welcher die Grundlage für den Baum T' bildet. Die Struktur des parallelen Algorithmus wird von dem im folgenden Abschnitt beschriebenen Algorithmus weitgehend übernommen.

4.3 Konstruktion einer WSPD im *Cache-Oblivious*-Modell

Das in diesem Abschnitt beschriebene Konstruktionsverfahren stimmt in weiten Teilen mit der I/O-effizienten Variante von Govindarajan *et al.* [40] überein. Der größte Teil des adaptierten Verfahrens besteht aus Sortierungen und Traversierungen von gewissen Datenelementen. Da Sortierungen und Traversierungen von Datenelementen im *cache-oblivious*-Modell asymptotisch gesehen mit genauso wenig Speichertransfers wie im I/O-Modell möglich sind, können diese Stellen fast identisch übernommen werden.

4.3.1 Vorbemerkungen

Es wird zunächst eine alternative Definition eines fairen Aufteilungsbaums T einer endlichen nicht-leeren Punktmenge $P \subset \mathbb{R}^d$ gegeben. Für die Definition eines solchen Baums wird in einem ersten Schritt der Begriff eines „fairen Schnitts“ eingeführt. Anschließend wird eine Technik namens *time-forward processing* genauer erläutert.

Faire Schnittbäume

Ein *Schnitt* eines Rechtecks R ist eine Zerlegung von R in zwei Rechtecke R_1 und R_2, die durch eine achsenparallele Hyperebene H, die sogenannte *Schnitthyperebene*, getrennt werden. Die Hyperebene H muss dabei einen echt positiven Abstand zu den beiden zu H parallelen Seiten von R haben. Ein *fairer Schnitt* [25] eines Rechtecks R ist ein Schnitt dieses Rechtecks, bei dem die entsprechende Hyperebene H einen Abstand von mindestens $\frac{1}{3}l_{max}(R)$ zu den beiden zu H parallelen Seiten von R hat. Analog zu Callahan [25] wird für zwei Rechtecke R und R' die Notation $R \rightsquigarrow R'$ eingeführt, um zu kennzeichnen, dass R' durch eine Sequenz von fairen Schnitten aus R entstehen kann, d. h., dass es eine Sequenz von Rechtecken $R = R_1, \ldots, R_k = R'$ gibt, so dass R_{i+1} eines der Rechtecke ist, welches durch einen fairen Schnitt von R_i entstanden ist $(i = 1, \ldots, k-1)$.

Ein *fairer Schnittbaum* T einer endlichen nicht-leeren Punktmenge $P \subset \mathbb{R}^d$ ist ein mit P verknüpfter Binärbaum, der die folgenden Bedingungen erfüllt, vgl. Callahan [25] und Govindarajan *et al.* [40]:

- Jedem Knoten v aus T ist ein Rechteck $\tilde{R}(v)$ mit $\sigma(v) \subset \tilde{R}(v)$ zugeordnet. Für die Wurzel r ist $\tilde{R}(r)$ ein d-Würfel mit Länge $l_{max}(P)$ und gleichem Zentrum wie $R(P)$.
- Für jeden inneren Knoten v mit Kindern v_1 und v_2 gilt: $\tilde{R}(v)$ kann durch einen fairen Schnitt in zwei Rechtecke R_1 und R_2 zerlegt werden, so dass $\sigma(v_1) = \sigma(v) \cap R_1$, $\sigma(v_2) = \sigma(v) \cap R_2$, $R_1 \rightsquigarrow \tilde{R}(v_1)$ und $R_2 \rightsquigarrow \tilde{R}(v_2)$ gilt.

Da ein fairer Schnittbaum ein mit der Punktmenge P verknüpfter Baum ist, gilt $|\sigma(v)| \geq 1$ für jeden Knoten v aus T. Für einen inneren Knoten v aus T mit Kindern v_1 und v_2 gilt weiterhin $\sigma(v_1) \cap \sigma(v_2) = \emptyset$, $\sigma(v_1) \cup \sigma(v_2) = \sigma(v)$ und $|\sigma(v_1)| < |\sigma(v)|$ bzw. $|\sigma(v_2)| < |\sigma(v)|$.

Wie das folgende Lemma zeigen wird, besitzt ein fairer Schnittbaum einer endlichen nicht-leeren Punktmenge P aus dem $\mathbb{R}^d$ im Wesentlichen dieselben Eigenschaften wie ein fairer Aufteilungsbaum von P. Im Unterschied zu einem fairen Aufteilungsbaum können bei einem fairen Schnittbaum jedoch Punkte aus P in den Schnitthyperebenen liegen, durch welche der Baum definiert ist. Nach Definition einer (fairen) Aufteilung wird dies bei einem fairen Aufteilungsbaum nicht zugelassen. Mit Ausnahme dieser Eigenschaft ist jeder faire Schnittbaum ein fairer Aufteilungsbaum:

4.3.1 Lemma. *Mit Ausnahme der oben angesprochenen Eigenschaft ist jeder faire Schnittbaum T einer endlichen nicht-leeren Punktmenge $P \subset \mathbb{R}^d$ ein fairer Aufteilungsbaum von P.*

Beweis. Sei v ein innerer Knoten aus T mit Kindern v_1 und v_2. Nach Definition eines fairen Schnittbaums kann das Rechteck $\tilde{R}(v)$ mittels eines fairen Schnitts in zwei Rechtecke R_1 und R_2 mit $R_1 \rightsquigarrow \tilde{R}(v_1)$ und $R_2 \rightsquigarrow \tilde{R}(v_2)$ zerlegt werden. Da $l_{max}(\tilde{R}(v)) \geq l_{max}(\sigma(v))$ und $\tilde{R}(v) \subseteq \hat{R}(v)$ gilt, hat die Schnitthyperebene H, durch welche der faire Schnitt definiert

ist, einen Abstand von mindestens $l_{max}(\sigma(v))/3$ zu den beiden zu H parallelen Seiten von $\hat{R}(v)$. □

In Anlehnung an einen unvollständigen fairen Aufteilungsbaum wird ein *unvollständiger fairer Schnittbaum* definiert: Die Definition eines unvollständigen fairen Schnittbaums T' einer endlichen nicht-leeren Punktmenge $P \subset \mathbb{R}^d$ stimmt mit der Definition eines fairen Schnittbaums von P überein, bis auf den Unterschied, dass die Blätter von T' auch mehrelementige Teilmengen von P darstellen dürfen, d. h. es wird $|\sigma(v)| \geq 1$ für ein Blatt v aus T' zugelassen.

Time-Forward Processing-Technik

Im Rest dieses Kapitels wird intensiv von einer Technik namens *time-forward processing* Gebrauch gemacht. Diese soll nun kurz beschrieben werden. Die folgende Beschreibung basiert dabei auf der Arbeit von Govindarajan *et al.* [40].

Die *time-forward processing*-Technik stellt ein elegantes Verfahren zur Bearbeitung bzw. Traversierung von gerichteten azyklischen Graphen dar. Dieses Verfahren wurde ursprünglich von Chiang *et al.* [29] vorgestellt und anschließend von Arge [6] verbessert. Wie Govindarajan *et al.* bemerken, kann mit Hilfe dieser Technik das folgende Problem gelöst werden:

Sei ein gerichteter azyklischer Graph $G = (V, E)$ gegeben, dessen Eckenmenge V *topologisch sortiert* ist, d. h., dass für jede Kante $(v, w) \in E$ die Anfangsecke v der Endecke w vorausgeht. Jeder Ecke $v \in V$ wird dadurch eine Priorität zugeordnet, welche der Position der Ecke in dieser sortierten Liste der Eckenmenge entspricht. Weiterhin sei in jeder Ecke $v \in V$ eine Markierung $\phi(v)$ gespeichert und f eine Funktion, welche auf die Eckenmenge V von G angewendet werden soll, um für jede Ecke $v \in V$ mit direkten Vorgängern $u_1, \ldots, u_k$ eine neue Markierung $\psi(v) = f(\phi(v), \psi(u_1), \ldots, \psi(u_k))$ zu berechnen.

Um die Berechnung der neuen Markierungen durchzuführen, werden die Ecken aus V mittels der *time-forward processing*-Technik in topologisch sortierter Reihenfolge traversiert. Nach Bearbeitung einer Ecke $v \in V$ wird deren neue Markierung $\psi(v)$ an ihre direkten Nachfolger „geschickt“, um sicherzustellen, dass die Markierung $\psi(v)$ bei Bearbeitung dieser direkten Nachfolger vorliegt. Govindarajan *et al.* beschreiben die von Arge [6] entwickelte Umsetzung dieser Technik wie folgt: Das „Senden“ von Informationen wird durch eine Prioritätswarteschlange Q realisiert. Soll nach Bearbeitung einer Ecke $v \in V$ die (neue) Markierung $\psi(v)$ an einen direkten Nachfolger w von v „geschickt“ werden, so wird die Information $\psi(v)$ in die Prioritätswarteschlange Q mit der Priorität von w eingefügt. Bei Bearbeitung einer Ecke $w \in V$ werden alle Einträge aus Q mit der Priorität von w entfernt. Da für jeden direkten Vorgänger v von w die für die Berechnung des neuen Werts $\psi(w)$ benötigte Information $\psi(v)$ aufgrund der topologischen Sortierung der Eckenmenge zuvor mit der Priorität von w in die Prioritätswarteschlange Q eingefügt wurde, liegen bei

der Bearbeitung von w alle Informationen für die Berechnung von $\psi(w)$ vor. Die für die Bearbeitung einer Ecke benötigten Informationen können demnach durch eine Folge von DeleteMin-Operationen aus der Prioritätswarteschlange entfernt werden.

Sowohl im I/O-Modell als auch im *cache-oblivious*-Modell kann die obige Anwendung der *time-forward processing*-Technik mit Hilfe einer Prioritätswarteschlange mit $\mathcal{O}(sort(|V| + |E| + |I|))$ I/O-Operationen bzw. Speichertransfers umgesetzt werden, wobei I die insgesamt entlang der Kanten von G „verschickte“ Menge von Informationen ist.[5] Im Folgenden wird diese Technik zudem auf Bäume anwendet, und gesagt, dass die Knoten dieser Bäume von den Blättern ausgehend zur Wurzel hin bzw. von der Wurzel ausgehend zu den Blättern hin mittels der *time-forward processing*-Technik traversiert werden. Damit ist gemeint, dass eine Traversierung der entsprechenden gerichteten Graphen stattfindet.

4.3.2 Konstruktion eines fairen Schnittbaums

Analog zu Govindarajan *et al.* [40] kann das folgende Ergebnis gezeigt werden:

4.3.2 Theorem. *Sei eine endliche nicht-leere Punktmenge P aus dem $\mathbb{R}^d$ gegeben. Dann kann mit Hilfe der Funktion* FairCutTree *(Algorithmus 4.3) unter Verwendung von $\mathcal{O}(\frac{|P|}{B})$ Speicherblöcken mit $\mathcal{O}(sort(|P|))$ Speichertransfers ein fairer Schnittbaum T von P berechnet werden.*

Beweis. Mit $\mathcal{O}(sort(|P|))$ Speichertransfers können d Listen erstellt werden, die die Punkte jeweils sortiert bzgl. einer der Dimensionen enthalten. Aus diesen Listen kann mit $\mathcal{O}(1)$ Speichertransfers der d-Würfel R_0 mit Länge $l_{max}(R_0) = l_{max}(P)$ und gleichem Zentrum wie $R(P)$ berechnet werden.

Sei α eine reelle Konstante aus dem halboffenen Intervall $[1 - \frac{1}{4d}, 1)$. Eine Anwendung der Funktion FairCutTree auf P, R_0 und α liefert dann den gewünschten fairen Schnittbaum T von P: Gilt $|P| = 1$, so wird in Zeile 2 der aus einem Knoten v bestehende faire Schnittbaum T von P konstruiert und anschließend zurückgegeben. Andernfalls wird in Zeile 4 ein unvollständiger fairer Schnittbaum T' von P berechnet, dessen Blätter Mengen repräsentieren, die aus höchstens $|P|^\alpha$ Punkten bestehen. Für diese Mengen werden in Zeile 6 rekursiv faire Schnittbäume berechnet, die dann in Zeile 8 die korrespondierenden Blätter ersetzen. Der dadurch entstehende Baum T ist offensichtlich ein fairer Schnittbaum von P.

Wie das folgende Lemma zeigen wird, verursacht die in Zeile 4 stattfindende Berechnung des unvollständigen fairen Schnittbaums T' mittels der Funktion PartialFairCutTree (Algorithmus 4.4) insgesamt höchstens $\mathcal{O}(sort(|P|))$ Speichertransfers. Weiterhin können die Blätter von T' sowie die für die rekursiven Aufrufe benötigten Informationen durch eine

[5] Für diese Menge kann $|I| \in \omega(|V| + |E|)$ gelten, falls die Markierungen $\phi(v)$ und $\psi(v)$ einer Ecke $v \in V$ nicht-konstante Informationen speichern.

Funktion FairCutTree(P, R_0, α)

Eingabe: Eine endliche nicht-leere Punktmenge $P \subset \mathbb{R}^d$, eine Box R_0 mit $P \subset R_0$ und eine reelle Konstante $\alpha \in [0, 1)$.

Ausgabe: Ein fairer Schnittbaum T von P.

1: **if** $|P| = 1$ **then**
2: Erzeuge einen fairen Schnittbaum T von P, der aus einem Knoten v mit $\sigma(v) = P$ und $\tilde{R}(v) = R_0$ besteht, und speichere den Punkt $P = \{p\}$ explizit in diesem Knoten.
3: **else**
4: Wende die Funktion PartialFairCutTree auf P, R_0 und α an, um einen unvollständigen fairen Schnittbaum T' von P zu berechnen. Seien $v_1, \ldots, v_k$ die Blätter von T'. In jedem Blatt v_i sind die Menge $\sigma(v_i)$ und das Rechteck $\tilde{R}(v_i)$ gespeichert und es gilt $|\sigma(v_i)| \leq |P|^\alpha$.
5: **for** $i = 1$ to k **do**
6: Wende die Funktion FairCutTree rekursiv auf die Punktmenge $\sigma(v_i)$, das Rechteck $\tilde{R}(v_i)$ und die Konstante α an, um einen fairen Schnittbaum T_i von $\sigma(v_i)$ zu konstruieren. Das Blatt v_i von T' wird in Zeile 8 durch den Baum T_i ersetzt.
7: **end for**
8: $T = T' \cup T_1 \cup \cdots \cup T_k$
9: **end if**
10: **return** T

Algorithmus 4.3: Berechnung eines fairen Schnittbaums im *cache-oblivious*-Modell, vgl. [40]

Traversierung der $\mathcal{O}(|P|)$ großen Knotenmenge von T' mit $\mathcal{O}(scan(|P|))$ Speichertransfers extrahiert werden (dies ist aufgrund der Art und Weise möglich, auf die der unvollständige faire Schnittbaum durch den im Rest dieses Abschnitts beschriebenen Konstruktionsalgorithmus im Speicher abgelegt wird). Durch Zeile 4 werden demnach höchstens $\mathcal{O}(sort(|P|))$ Speichertransfers verursacht.

Die für die rekursive Berechnung insgesamt benötigten Speichertransfers können somit durch folgende Rekurrenz beschrieben werden:

$$\mathcal{I}(|P|) \leq c \cdot sort(|P|) + \sum_{i=1}^{k} \mathcal{I}(N_i)$$

Die Konstante $c \in \mathbb{R}^+$ ist dabei so gewählt, dass ein Aufruf der Funktion PartialFairCutTree höchstens $c \cdot sort(|P|)$ Speichertransfers verursacht (aufgrund des folgenden Lemmas existiert eine solche Konstante). Weiterhin geben die Werte N_i die Größen der

Mengen $\sigma(v_i)$ an, d.h. $N_i = |\sigma(v_i)|$ für $i = 1, \ldots, k$. Da für diese Werte $N_i \leq |P|^\alpha$ und $|P| = \sum_{i=1}^{k} N_i$ gilt, lässt sich die Rekurrenz wie folgt auflösen:

$$\begin{aligned}
\mathcal{I}(|P|) &\leq c \cdot sort(|P|) + \sum_{i=1}^{k} \mathcal{I}(N_i) \\
&\leq c \cdot sort(|P|) + \sum_{i=1}^{k} \left(c \cdot sort(N_i) + \sum_{j=1}^{k_i} \mathcal{I}(N_{ij}) \right) \\
&\leq c \cdot sort(|P|) + \sum_{i=1}^{k} c \cdot \frac{N_i}{B} \log_{M/B} \frac{|P|^\alpha}{B} + \sum_{i=1}^{k} \sum_{j=1}^{k_i} \mathcal{I}(N_{ij}) \\
&= c \cdot sort(|P|) \cdot (1 + \alpha) + \sum_{i=1}^{k} \sum_{j=1}^{k_i} \mathcal{I}(N_{ij}) \\
&\leq \ldots \\
&\leq c \cdot sort(|P|) \cdot \sum_{i=0}^{\infty} \alpha^i
\end{aligned}$$

Für die Rekurrenz gilt demnach $\mathcal{I}(|P|) \leq \frac{c}{1-\alpha} \cdot sort(|P|) \in \mathcal{O}(sort(|P|))$. Ein Aufruf der Funktion FairCutTree verursacht also höchstens $\mathcal{O}(sort(|P|))$ Speichertransfers. Die Aussage über den Speicherplatz ergibt sich aus dem folgenden Lemma. □

Es ist noch das folgende Lemma zu zeigen.

4.3.3 Lemma. *Sei eine endliche nicht-leere Punktmenge $P \subset \mathbb{R}^d$, eine Box R_0 mit $P \subset R_0$ und eine reelle Konstante $\alpha \in [1 - \frac{1}{4d}, 1)$ gegeben. Dann benötigt eine Anwendung der Funktion* PartialFairCutTree *(Algorithmus 4.4) auf P, R_0 und α zur Berechnung eines unvollständigen fairen Schnittbaums T' von P insgesamt $\mathcal{O}(\frac{|P|}{B})$ Speicherblöcke und $\mathcal{O}(sort(|P|))$ Speichertransfers. Für jedes Blatt v von T' gilt zudem $|\sigma(v)| \leq |P|^\alpha$.*

Analog zu Govindarajan *et al.* [40] wird die Topologie des Baums T' durch die Funktion PartialFairCutTree in drei Schritten erstellt. Im ersten Schritt wird ein Binärbaum T_c erstellt, dessen Knoten mit Boxen korrespondieren. Jede durch ein Blatt von T_c dargestellte Box enthält höchstens $|P|^\alpha$ Punkte aus P. Im zweiten Schritt wird der Baum T_c zu einem Baum T'' erweitert. Dabei werden ggf. einige Blätter an T_c angehängt und einige Kanten von T_c durch Bäume ersetzt, welche aus Sequenzen von fairen Schnitten entstehen. Nach Schritt 2 ist sichergestellt, dass jeder Punkt aus P in genau einer durch ein Blatt von T'' dargestellten Box liegt. Dabei kann es sein, dass eine durch ein Blatt von T'' dargestellte Box keine Punkte aus P enthält. Diese Blätter werden im dritten Schritt entfernt. Anschließend werden alle Wege in dem entstehenden Baum, die aus Knoten vom Grad 2 bestehen, durch eine einzelne Kante ersetzt. Der resultierende Baum ist dann der gesuchte faire Schnittbaum T'. Diese drei Schritte werden im Folgenden detailliert beschrieben und analysiert.

Funktion PartialFairCutTree(P, R_0, α)

Eingabe: Eine endliche nicht-leere Punktmenge $P \subset \mathbb{R}^d$, eine Box R_0 mit $P \subset R_0$ und eine reelle Konstante $\alpha \in [0, 1)$.

Ausgabe: Ein unvollständiger fairer Schnittbaum T' von P. Für ein Blatt v des Baums gilt $|\sigma(v)| \leq |P|^\alpha$.

1: Berechne den Baum T_c. Jeder Knoten stellt eine Box dar, welche explizit in diesem Knoten gespeichert ist.

2: Expandiere T_c zu T''. Jeder Knoten von T'' stellt eine Box dar, welche explizit in diesem Knoten gespeichert ist. In einem Blatt von T'' ist zusätzlich die (möglicherweise leere) Teilmenge von P gespeichert, welche in der durch das Blatt dargestellten Box liegt.

3: Entferne jedes Blatt aus T'', welches keine Punkte speichert. Fasse alle Wege zu einer einzelnen Kante zusammen, die aus Knoten vom Grad 2 bestehen. Der resultierende Baum ist der Baum T'.

4: **return** T'

Algorithmus 4.4: Berechnung eines unvollständigen fairen Schnittbaums im *cache-oblivious*-Modell, vgl. [40]

Um die Lesbarkeit zu verbessern, wird bei der Bezeichnung von Knoten bzw. von den durch die Knoten dargestellten Rechtecke nicht zwischen Knoten und Rechtecken unterschieden. Mit einer Bezeichnung R kann also kontextabhängig der Knoten R oder das durch den Knoten dargestellte Rechteck R gemeint sein.

Es wird nun gezeigt, wie die drei Schritte der Funktion PartialFairCutTree implementiert werden können, um die in Lemma 4.3.3 angegebene asymptotische Schranke für die Anzahl der verursachten Speichertransfers bzw. des benötigten Speicherplatzes einzuhalten. Seien dazu die durch die Eingabe der Funktion PartialFairSplitTree mitgelieferte endliche nicht-leere Punktmenge $P \subset \mathbb{R}^d$, die Box R_0 mit $P \subset R_0$ und die reelle Konstante $0 \leq \alpha < 1$ gegeben.

Konstruktion von T_c

Um T_c zu konstruieren, wird das Rechteck R_0 zunächst in jeder Dimension in „Scheiben“ eingeteilt, welche jeweils höchstens $|P|^\alpha$ Punkte aus P enthalten. Diese Einteilung wird pro Dimension mit Hilfe von $\left\lceil \frac{|P|}{\lfloor |P|^\alpha \rfloor} \right\rceil + 1$ achsenparallelen Hyperebenen vollzogen, wobei die beiden äußeren Hyperebenen die beiden Seiten von R_0 in dieser Dimension enthalten. Die Hyperebenen werden im Folgenden als *Begrenzungsebenen* bezeichnet. Für die Konstruktion von T_c werden ausschließlich die durch die Begrenzungsebenen gegebenen Informationen über die Punkte verwendet.

Sind die Begrenzungsebenen für das Rechteck R_0 berechnet worden, so kann der Baum T_c durch den folgenden Algorithmus konstruiert werden, welcher das gegebene Rechteck

R_0 sukzessiv anhand der drei weiter unten beschriebenen Fälle aufteilt: Da alle Seiten des Startrechtecks R_0 in Begrenzungsebenen liegen, wird dieses durch den Fall 1 (s. u.) in zwei Rechtecke R_1 und R_2 aufgeteilt. Für die beiden Rechtecke R_1 und R_2 werden jeweils mittels der drei weiter unten beschriebenen Fälle rekursiv entsprechende Bäume T_{c_1} und T_{c_2} berechnet. Die rekursive Aufteilung bricht ab, sobald ein Rechteck in mindestens einer Dimension von keiner der zugehörigen Begrenzungsebenen geschnitten wird. Der Baum T_c besteht dann aus der Wurzel R_0 und aus den beiden in R_0 gewurzelten Bäumen T_{c_1} und T_{c_2}. Am Ende der Konstruktion liegt jedes Blatt von T_c in mindestens einer Dimension vollständig in einer der zugehörigen Scheiben. Dadurch wird sichergestellt, dass jedes dieser Rechtecke höchstens $|P|^\alpha$ Punkte aus P enthält.

Zur Beschreibung der drei Fälle sei nun ein Rechteck R gegeben, welches den unten beschriebenen Bedingungen (i)–(iii) genügt und weiter aufgeteilt werden soll. Dabei bezeichne R' das größte in R enthaltene Rechteck, dessen Seiten in Begrenzungsebenen liegen. Bei jeder Aufteilung mittels einer der drei unten beschriebenen Fälle werden die folgenden Invarianten aufrecht erhalten:

(i) Pro Dimension liegt mindestens eine der beiden Seiten von R in einer Begrenzungsebene

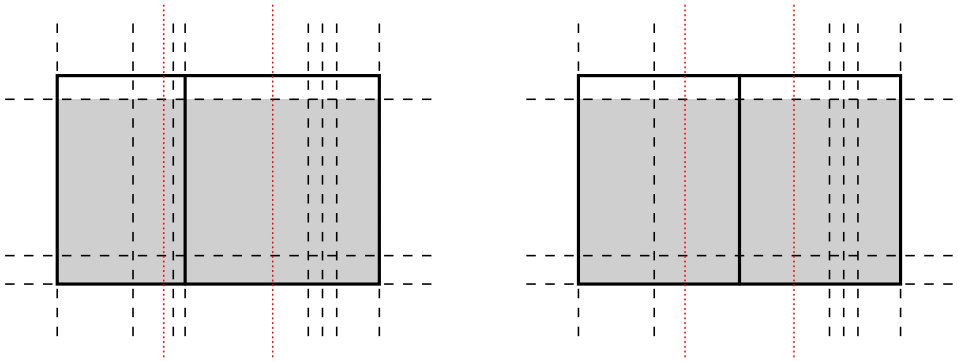
(ii) Für alle $i = 1, \ldots, d$ gilt entweder $l_i(R') = l_i(R)$ oder $l_i(R') \leq \frac{2}{3} l_i(R)$

(iii) $l_{min}(R) \geq \frac{1}{3} l_{max}(R)$

Das Startrechteck R_0 erfüllt alle drei Invarianten: R_0 ist entweder ein d-Würfel oder ein Rechteck, welches durch einen vorherigen Aufruf der Funktion PartialFairCutTree entstanden ist. Somit erfüllt R_0 die Invariante (iii). Die beiden Invarianten (i) und (ii) folgen aus der speziellen Platzierung der Begrenzungsebenen zu Beginn der Konstruktion von T_c.

Durch Anwendung eines der nun beschriebenen Fälle entstehen jeweils ein oder zwei neue Rechtecke. Es wird gesagt, dass die jeweiligen Fälle diese Rechtecke *produzieren*. Für die folgenden Betrachtungen sei weiterhin i_{max} eine abkürzende Schreibweise für $i_{max}(R)$.

Fall 1: $l_{max}(R) = l_{i_{max}}(R')$. Beide Seiten von R in der Dimension i_{max} liegen in Begrenzungsebenen. Es wird nun diejenige Begrenzungsebene in dieser Dimension gesucht, die das Rechteck möglichst „gut“ in zwei geometrisch gleich große Hälften aufteilt. Liegt diese Begrenzungsebene in einem Abstand von mindestens $\frac{1}{3} l_{max}(R)$ von den beiden zur Begrenzungsebene parallelen Seiten von R entfernt, so wird diese Ebene für die Aufteilung verwendet (Fall 1a). Andernfalls wird R in der Dimension i_{max} in zwei gleich große Hälften aufgeteilt (Fall 1b). Dieser Fall produziert genau zwei Rechtecke. Eine vereinfachte Darstellung einer Aufteilung anhand dieses Falls wird durch Abbildung 4.6 gegeben.



(a) Fall 1a (b) Fall 1b

Abbildung 4.6: Das äußere fett gedruckte Rechteck ist das aufzuteilende Rechteck R. Die mittlere fett gedruckte vertikale Linie ist die Aufteilungslinie, durch die R zerlegt wird. Die Begrenzungsebenen werden durch gestrichelte Linien skizziert. Der Bereich zwischen den beiden längeren vertikalen gepunkteten Linien stellt das mittlere Drittel von R in der Dimension $i_{max}(R)$ dar. Durch den grau schattierten Bereich wird das Rechteck R' schematisch dargestellt.

Gilt Fall 1 nicht, so folgt aufgrund der zweiten Invariante

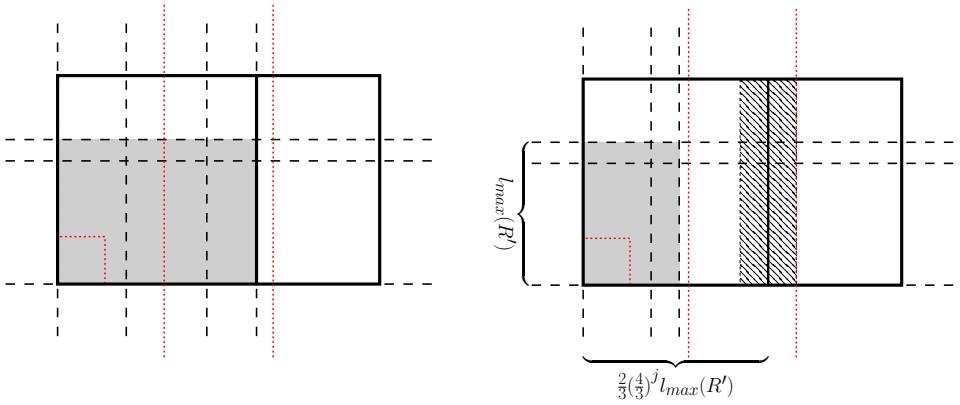
$$l_{i_{max}}(R') \leq \frac{2}{3} l_{max}(R).$$

Es liegt somit nur eine der beiden Seiten von R in der Dimension i_{max} in Begrenzungsebenen. Diese Begrenzungsebene sei mit H bezeichnet. Des Weiteren ist c eine reelle Konstante mit Wert $c = \frac{4}{27}$. Der Grund für diese spezielle Wahl von c wird im Beweis zu Lemma 4.3.8 ersichtlich werden.

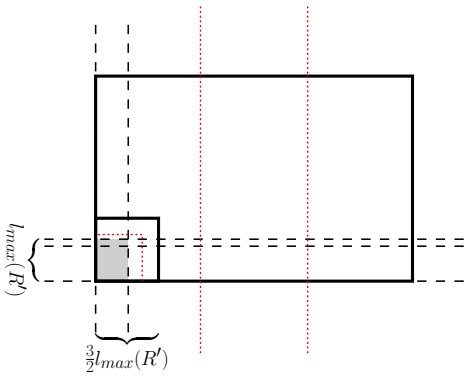
Fall 2: $l_{max}(R') \geq c \cdot l_{max}(R)$. Gilt $l_{i_{max}}(R') \geq \frac{1}{3} l_{max}(R)$ (Fall 2a), so wird R durch diejenige Begrenzungsebene aufgeteilt, welche eine Seite von R' in der Dimension i_{max} enthält und nicht mit H übereinstimmt, vgl. Abbildung 4.7 (a). Ansonsten (Fall 2b) wird R durch eine zu H parallele Hyperebene aufgeteilt. Um die Invarianten aufrecht zu erhalten, wird diese Hyperebene in einem Abstand von $y = \frac{2}{3}(\frac{4}{3})^j l_{max}(R')$ zu H platziert, wobei j eine ganze Zahl ist und so gewählt wird, dass $y \in (\frac{1}{2} l_{max}(R), \frac{2}{3} l_{max}(R)]$ gilt, vgl. Abbildung 4.7 (b).

Der Fall 2b produziert nur das Rechteck, welches R' enthält. Das andere Rechteck wird in diesem Aufteilungsprozess nicht weiter betrachtet und ist somit kein Knoten von T_c. Der Grund dafür ist, dass das zweite Rechteck die Invariante (i) nicht erfüllt. Da dieses jedoch vollständig in einer Scheibe von R_0 liegt, sind höchstens $|P|^\alpha$ Punkte aus P in ihm enthalten. Dadurch kann es in der zweiten Phase des Algorithmus 4.4 als Blatt an T'' angehängt werden.

Fall 3: $l_{max}(R') < c l_{max}(R)$. Aufgrund von $c = \frac{4}{27}$ und der für das Rechteck R gültigen



(a) Fall 2a (b) Fall 2b



(c) Fall 3

Abbildung 4.7: Beschriftungen wie in Abbildung 4.6. Hinzu kommt das Rechteck (punktierter Rand) in der linken unteren Ecke von R mit Länge $\frac{4}{27} \cdot l_{max}(R)$. In Fall 2*a* liegt eine Begrenzungsebene im gekennzeichneten mittleren Bereich. In Fall 2*b* liegt keine Begrenzungsebene im mittleren Bereich, jedoch existiert mindestens eine Begrenzungsebene, welche zwar R jedoch nicht das kleine Rechteck (mit punktiertem Rand) schneidet. Ansonsten (Fall3) liegen alle Begrenzungsebenen im kleinen Rechteck.

Invariante (iii) folgt, dass R' einen einzigen gemeinsamen Eckpunkt E mit R teilt. Sei C ein d-Würfel mit Länge $l(C) = \frac{3}{2}l_{max}(R')$, der R' enthält und E als einen Eckpunkt besitzt. Dieser Fall produziert das Rechteck C, vgl. Abbildung 4.7 (c). Die Kante zwischen R und C ist eine *komprimierte Kante* von T_c, die während der Konstruktion von T'' durch einen Baum ersetzt wird, welcher aus einer Folge von fairen Schnitten hervorgeht.

Callahan [25] zeigt, dass die Invarianten (i)–(iii) bei einer Aufteilung anhand der drei obigen Fälle erhalten bleibt:

4.3.4 Lemma ([25]). *Sei R ein Rechteck, welches den drei obigen Invarianten (i)–(iii) genügt. Ein durch Anwendung einer der drei Fälle produziertes Rechteck erfüllt wieder die drei Invarianten (i)–(iii).*

Auf den Beweis dieses Lemmas soll hier verzichtet werden. Wie Govindarajan *et al.* [40] bemerken, führt eine direkte Konstruktion von T_c durch den obigen Aufteilungsalgorithmus, der das Rechteck R_0 rekursiv anhand der drei Fälle aufteilt, nicht zu einem I/O-effizienten Verfahren. Stattdessen orientieren sie sich an dem parallelen Konstruktionsverfahren von Callahan [25] und erstellen zunächst einen Obergraphen G, aus welchem T_c anschließend extrahiert wird. Die Größe dieses Obergraphen kann durch eine entsprechende Wahl der Konstanten α variiert werden, wodurch die effiziente Extraktion von T_c gewährleistet wird. Dieses Vorgehen wird hier übernommen.

Zur Definition des Obergraphen G sind zunächst folgende Überlegungen nötig: Nach Callahan [25] und Govindarajan *et al.* [40] lassen sich die in den Fällen 1–3 verwendeten Aufteilungshyperebenen jeweils durch zwei Begrenzungsebenen und konstant viele zusätzliche Informationen eindeutig beschreiben. Da alle Seiten des Rechtecks R_0 in Begrenzungsebenen liegen, gilt dies auch für alle Seiten von R_0. Somit lässt sich jedes Rechteck aus T_c pro Dimension durch zwei Begrenzungsebenen und konstant viele zusätzliche Informationen beschreiben.

Das Gegenteil muss nicht zutreffen; es kann Rechtecke geben, die durch diese beschränkten Informationen eindeutig beschrieben werden können aber keine Knoten von T_c sind. Callahan [25] betrachtet *alle* Rechtecke, die durch die obigen beschränkten Informationen eindeutig beschrieben werden können und nennt sie *konstruierbare Rechtecke*. Der Schlüssel zu einem effizienten Verfahren wird durch das folgende Lemma gegeben, auf dessen Beweis wiederum verzichtet wird.

4.3.5 Lemma ([25, 40]). *Es gibt $\mathcal{O}(|P|^{2d(1-\alpha)})$ konstruierbare Rechtecke.*

Callahan [25] definiert den Obergraphen G wie folgt: Die Eckenmenge von G besteht aus allen konstruierbaren Rechtecken. Zwischen zwei Rechtecken R_1 und R_2, welche beide die Invarianten (i)–(iii) erfüllen, existiert eine gerichtete Kante (R_1, R_2), wenn R_2 durch Anwendung einer der drei obigen Fälle aus R_1 produziert werden kann.

In dieser Arbeit wird die Definition dieses Obergraphen, dem sogenannten *Produktionsgraphen*, weitgehend übernommen. Um zu gewährleisten, dass kein Rechteck R mit $l_i(R) = 0$ für ein $i \in \{1, \ldots, d\}$ Teil der Eckenmenge des Produktionsgraphen ist, wird in Ergänzung zur obigen Definition gefordert, dass von einem Rechteck R_1 nur dann Kanten ausgehen, wenn es in keiner Dimension vollständig in einer der zu dieser Dimension gehörenden Scheiben liegt.

Der gesuchte Baum T_c ist der kleinste Teilgraph von G, welcher alle von R_0 aus erreichbaren Ecken enthält. Da jede Ecke aus G höchstens zwei ausgehende Kanten besitzt, ist die Gesamtgröße des Graphen G linear in der Anzahl der konstruierbaren Rechtecke. Wird nun für die Konstante α ein Wert aus $[1 - \frac{1}{4d}, 1)$ gewählt, so ergibt sich für den Graphen G nach Lemma 4.3.5 eine Gesamtgröße von $\mathcal{O}(\sqrt{|P|})$.

Die effiziente Extraktion des Baums T_c aus G kann mit Hilfe der *time-forward processing*-Technik realisiert werden. Um diese Technik anwenden zu können, wird ein effizientes Verfahren benötigt, welches den Graphen G topologisch sortiert. Dies wird durch die folgende Beobachtung ermöglicht:

4.3.6 Beobachtung. Seien R_1 und R_2 zwei konstruierbare Rechtecke. Existiert die Kante (R_1, R_2) in G, so gilt $\sum_{j=1}^{d} l_j(R_2) < \sum_{j=1}^{d} l_j(R_1)$.

Diese Eigenschaft von G ermöglicht eine topologische Sortierung desselben, indem die Ecken in absteigender Reihenfolge bzgl. der Summen über ihre Seitenlängen sortiert werden. Der Graph G lässt sich zudem durch die obige Beobachtung als ein gerichteter azyklischer Graph erkennen. Weiterhin ist leicht ersichtlich, dass der Teilgraph T_c ein Baum ist.

Es wird nun der im Kontext des *cache-oblivious*-Modells effiziente Algorithmus zur Konstruktion von T_c mit Hilfe des Obergraphen G beschrieben. Dieser Algorithmus besteht aus den folgenden drei Schritten:

Schritt 1a: Berechnung der Begrenzungsebenen, durch die das Eingaberechteck R_0 in Scheiben unterteilt wird

Schritt 1b: Konstruktion des Produktionsgraphen G anhand dieser Einteilung

Schritt 1c: Extraktion von T_c aus G

Schritt 1a: Berechnung der Begrenzungsebenen Das Rechteck R_0 soll pro Dimension in Scheiben eingeteilt werden, die jeweils höchstens $|P|^\alpha$ Punkte aus P enthalten. Dazu wird über alle Dimensionen des Rechtecks R_0 iteriert und pro Dimension $i \in \{1, \ldots, d\}$ werden die Punkte aus P anhand ihrer i-ten Koordinate sortiert. Anschließend werden die sortierten Punkte durchlaufen und die Begrenzungsebenenliste der Dimension i erzeugt. Ein Eintrag dieser Liste besteht aus einem reellen Wert, der eine Begrenzungsebene in dieser Dimension eindeutig beschreibt. Bei der Traversierung der Punkte wird für jedes $j = 1, \ldots, \left\lceil \frac{|P|}{\lfloor |P|^\alpha \rfloor} \right\rceil - 1$ zwischen dem $(j \cdot |P|^\alpha)$-ten und $(j \cdot |P|^\alpha + 1)$-ten Punkt eine Begrenzungsebene platziert und der reelle Wert in die Begrenzungsebenenliste eingetragen, welcher diese Ebene eindeutig beschreibt. Zudem werden zwei Begrenzungsebenen hinzugefügt, die die Seiten von R_0 in der i-ten Dimension enthalten.[6]

[6] Sonderfälle der Punkteverteilung können durch eine ε-Scherung des Koordinatensystems behandelt werden. Beim Erstellen der Begrenzungsebenenliste wird parallel auf zwei Listen zugegriffen (die Liste der Punkte und die Begrenzungsebenenliste). Dadurch könnte der aktuelle Block der Begrenzungsebenenliste (in den geschrieben wird) aus dem internen Speicher verdrängt werden. Die im *cache-oblivious*-Modell angenommene optimale Ersetzungsstrategie gewährleistet jedoch, dass dieser Block im Speicher bleibt. Es werden für diesen Vorgang also höchstens $\mathcal{O}(\frac{N^{1-\alpha}}{B}) + \mathcal{O}(scan(N))$ (und nicht $\mathcal{O}(N^{1-\alpha}) + \mathcal{O}(scan(N))$) Speichertransfers benötigt.

Durch diesen Vorgang entsteht also pro Dimension eine Liste von Begrenzungsebenen und somit entstehen insgesamt d dieser Listen. Da d konstant ist, benötigt dieser gesamte Schritt $\mathcal{O}(d \cdot sort(|P|)) = \mathcal{O}(sort(|P|))$ Speichertransfers.

Schritt 1b: Konstruktion des Produktionsgraphen G Nach Lemma 4.3.5 liegt die Anzahl der konstruierbaren Rechtecke in $\mathcal{O}(|P|^{2d(1-\alpha)}) \subseteq \mathcal{O}(\sqrt{|P|})$ und die Eckenmenge des zu erstellenden Graphen G besteht aus genau diesen Rechtecken. Jedes dieser Rechtecke kann, wie oben beschrieben, pro Dimension eindeutig durch zwei Begrenzungsebenen und konstant viele zusätzliche Informationen dargestellt werden.

Die Knotenmenge von G lässt sich durch verschachtelte Traversierungen der Begrenzungsebenenlisten erstellen. Dazu wird zunächst eine Knotenliste mit leeren Einträgen und passender Größe erstellt. Ein Eintrag dieser Liste ist ein $3d$-Tupel von der Form $(s_1, s'_1, \alpha_1, \ldots, s_d, s'_d, \alpha_d)$ und stellt ein konstruierbares Rechteck eindeutig dar. Dabei werden die beiden zur eindeutigen Identifizierung des Rechtecks nötigen Begrenzungsebenen in der Dimension i durch die Parameter s_i und s'_i festgehalten. Die zusätzlich pro Dimension i nötigen Informationen werden in dem Parameter α_i gespeichert.

Das Füllen der Tupelliste mit Werten geschieht wie folgt: Die entsprechenden s_1-Werte werden in die Liste eingefügt, indem die Tupelliste und die Begrenzungsebenenliste der ersten Dimension traversiert werden. Für jeden s_1-Wert wird nun erneut die Begrenzungsebenenliste der ersten Dimension durchlaufen und es werden entsprechende s'_1-Werte in die Tupelliste eingetragen. Für ein dadurch entstehendes Paar (s_1, s'_1) trägt man alle in Frage kommenden α_1-Werte ein. Pro Tripel (s_1, s'_1, α_1) werden nun die Begrenzungsebenenlisten der zweiten Dimension traversiert und die s_2-Werte in die Tupelliste eingetragen. Auf diese Weise fährt man fort, bis die Werte aller $3d$ Dimensionen eingetragen sind. Der gesamte Vorgang lässt sich durch einen verschachtelten Durchlauf realisieren, durch welchen die Tupelliste einmal und die verschiedenen Begrenzungsebenenlisten mehrere Male traversiert werden. Da die Gesamtzahl der durch die Traversierung der Begrenzungsebenenlisten gelesenen Elemente gleich der Anzahl der Elemente in der Tupelliste ist, entspricht der Gesamtaufwand für das Durchlaufen der Tupelliste und aller Begrenzungsebenenlisten einer zweimaligen Traversierung der Tupelliste. Nach Wahl von $\alpha \geq 1 - \frac{1}{4d}$ besteht diese aus insgesamt $\mathcal{O}(|P|^{2d(1-\alpha)}) \subseteq \mathcal{O}(\sqrt{|P|})$ Elementen. Somit benötigt dieser verschachtelte Durchlauf höchstens $\mathcal{O}(scan(\sqrt{|P|}))$ Speichertransfers.

Jeder Eintrag der entstandenen Liste enthält eine vollständige Darstellung eines konstruierbaren Rechtecks $R = [x_1, x'_1] \times \cdots \times [x_d, x'_d]$. In Hinblick auf die Erstellung der Eckenmenge von G wird nun ein solcher Tupeleintrag um eine vollständige Darstellung des größten in R liegenden Rechtecks R' ergänzt, dessen Seiten in Begrenzungsebenen liegen. In dem I/O-effizienten Algorithmus von Govindarajan *et al.* wird zur Umsetzung dieses Schritts pro Dimension i ein *buffer tree* [6] über den Elementen der Begrenzungsebenenliste der Dimension i konstruiert. Da der *buffer tree* nicht *cache-oblivious* ist, wird hier

jedoch eine andere Vorgehensweise gewählt. Um jeden Tupeleintrag mit den entsprechenden Informationen zu erweitern, wird über alle d Dimensionen iteriert und pro Dimension i wie folgt verfahren: Durch eine Traversierung der Begrenzungsebenenliste der i-ten Dimension lässt sich derjenige Eintrag z_1 finden, welcher einen minimalen Abstand zu x_i hat und die Bedingung $z_1 - x_i \geq 0$ erfüllt. Während dieser Traversierung kann für x'_i analog der Wert z_2 mit $x'_i - z_2 \geq 0$ und minimalem Abstand zu x'_i bestimmt werden.

Pro Dimension wird also zur Untersuchung eines Rechtecks R ein Durchlauf der Begrenzungsebenenliste dieser Dimension benötigt. Insgesamt ergibt sich pro untersuchtem Rechteck für alle d Dimensionen ein Aufwand von höchstens $\mathcal{O}(2d \cdot scan(\left\lceil \frac{|P|}{\lfloor |P|^{\alpha} \rfloor} \right\rceil + 1)) \subseteq \mathcal{O}(scan(\frac{|P|}{|P|^{\alpha}-1} + 2)) \subseteq \mathcal{O}(scan(|P|^{1-\alpha})) \subseteq \mathcal{O}(scan(\sqrt{|P|}))$ Speichertransfers. Da G insgesamt nur aus $\mathcal{O}(\sqrt{|P|})$ konstruierbaren Rechtecken besteht, sind zur Ergänzung aller Rechtecke R aus G um das entsprechende Rechteck R' insgesamt höchstens $\mathcal{O}(scan(|P|))$ Speichertransfers erforderlich.

Um die Kantenmenge des Produktionsgraphen G zu konstruieren, müssen für jedes Rechteck R der schon erstellten Knotenmenge von G höchstens zwei ausgehende Kanten gefunden werden. Ein Eintrag in der Knotenliste von G enthält nach obigem Verfahren eine vollständige Darstellung eines Rechtecks R und des dazugehörenden Rechtecks R'. Durch diese Informationen kann ermittelt werden, ob von R Kanten ausgehen oder nicht: Gilt $l_k(R') = 0$ für ein $k \in \{1, \ldots, d\}$, so wird das Rechteck R in der Dimension k von keiner Begrenzungsebene geschnitten. Das Rechteck R hat somit keine ausgehenden Kanten. Ansonsten werden durch Anwendung eines der drei eingangs beschriebenen Fälle auf R ein oder zwei Rechtecke produziert, für welche entsprechende Kanten erzeugt werden müssen. Die durch einen Eintrag der Eckenliste von G gegebenen Informationen reichen dabei aus, um zwischen den Fällen $1, 2$ und 3 zu unterscheiden. Ebenso ist anhand der Informationen eine Fallunterscheidung der Fälle $2a$ und $2b$ möglich. Um die Fälle $1a$ und $1b$ unterscheiden zu können, benötigt man zusätzlich diejenige Begrenzungsebene, die das Rechteck R in der Dimension $i_{max}(R)$ möglichst „gut“ in zwei geometrisch gleich große Hälften teilt. Diese Begrenzungsebene kann analog zu dem obigen Verfahren bestimmt werden: Dazu werden pro Rechteck R die Begrenzungsebenenlisten der Dimension $i_{max}(R)$ durchlaufen und derjenige Eintrag z_3 mit minimalem Abstand zu $\frac{x_i + x'_i}{2}$ ausgewählt. Die Erweiterung aller Rechtecke um diese Information benötigt demnach wieder $\mathcal{O}(scan(|P|))$ Speichertransfers.[7] Anschließend kann die Kantenmenge mittels einer Traversierung der Knotenmenge von G erstellt werden. Anstatt die Kantenmenge von G separat zu speichern, wird bei der Erstellung der Kantenmenge eine Ecke R von G zu einem Tripel (R, R_1, R_2) bzw. (R, R_1, null) bzw. $(R, \mathit{null}, \mathit{null})$ erweitert, je nachdem, ob das Rechteck R durch einen fairen Schnitt in die

[7] Der I/O-effiziente Algorithmus konstruiert zur Erweiterung aller Rechtecke um diese Informationen erneut pro Dimension i einen *buffer tree* über den Elementen der Begrenzungsebenenliste der i-ten Dimension.

Rechtecke R_1 und R_2 bzw. in das Rechteck R_1 bzw. nicht mehr zerlegt wird. Insgesamt benötigt die Erstellung der Kantenmenge demnach $\mathcal{O}(scan(|P|))$ Speichertransfers.

Schritt 1c: Extraktion von T_c Um den gesuchten Teilgraphen T_c aus G extrahieren zu können, wird G zunächst mit Hilfe von Beobachtung 4.3.6 topologisch sortiert. Die Ecken werden also in absteigender Reihenfolge bzgl. der Summen über ihre Seitenlängen sortiert. Anschließend wird die Eckenmenge von G durchlaufen und jeder Knoten mit Ausnahme von R_0 als inaktiv markiert. Der Knoten R_0 wird als aktiv markiert. Mittels der *time-forward processing*-Technik werden dann alle von R_0 aus erreichbaren Ecken als aktiv ummarkiert. Eine aktive Ecke bleibt bei diesem Vorgang aktiv und sendet durch ihre ausgehenden Kanten eine Nachricht „aktiviere“ an ihre direkten Nachfolger. Eine inaktive Ecke, welche durch eine ihrer eingehenden Kanten die Nachricht „aktiviere“ erhält, wird als aktiv ummarkiert und leitet die Nachricht durch ihre ausgehenden Kanten an ihre direkten Nachfolger weiter.

Da der Graph G höchstens $\mathcal{O}(\sqrt{|P|})$ Ecken besitzt, benötigt die topologische Sortierung und die Anwendung der *time-forward processing*-Technik insgesamt $\mathcal{O}(sort(\sqrt{|P|}))$ Speichertransfers, vgl. Abschnitt 4.3.1.

Nach diesem Vorgang sind alle von R_0 aus erreichbaren Ecken von G als aktiv markiert. Durch eine weitere Traversierung der Eckenmenge von G können diese extrahiert werden. Die dadurch entstehende Eckenmenge ist die Eckenmenge des gesuchten Baums T_c. Die Kanten von T_c sind dabei implizit in den Ecken gespeichert.

Da alle Schritte zur Konstruktion von T_c jeweils höchstens $\mathcal{O}(sort(|P|))$ Speichertransfers und $\mathcal{O}(\frac{|P|}{B})$ Speicherblöcke benötigen, ergibt sich das folgende Lemma:

4.3.7 Lemma. *Sei eine endliche nicht-leere Punktmenge $P \subset \mathbb{R}^d$, eine Box R_0 mit $P \subset R_0$ und eine reelle Konstante $\alpha \in [1-\frac{1}{4d}, 1)$ gegeben. Dann kann der Baum T_c aus Algorithmus 4.4 unter Verwendung von $\mathcal{O}(\frac{|P|}{B})$ Speicherblöcken mit $\mathcal{O}(sort(|P|))$ Speichertransfers konstruiert werden.*

Konstruktion von T''

Mit Hilfe des in der vorherigen Phase berechneten Baums T_c soll in dieser Phase der Baum T'' aus Algorithmus 4.4 berechnet werden. Der Baum T'' kann durch die folgenden Schritte konstruiert werden:

Schritt 2a: Während der Konstruktion von T_c wurde bei einer Aufteilung durch den Fall 2 eines der beiden Rechtecke nicht weiter betrachtet, da es die Invariante (i) nicht erfüllt hat. Jedes dieser Rechtecke enthält höchstens $|P|^\alpha$ Punkte aus P, da es in mindestens einer Dimension vollständig in einer zu der Dimension gehörenden Scheibe liegt. Diese Rechtecke werden in diesem Schritt an den Baum T_c als Blätter angehängt. Der resultierende Baum wird mit T_c^+ bezeichnet.

Schritt 2b: Jeder Punkt aus P liegt entweder in einem Blatt von T_c^+ oder in einer Region $R \backslash C$ zwischen zwei Rechtecken R und C wobei (R, C) eine komprimierte Kante von T_c^+ ist.[8] Die Punkte aus P werden in diesem Schritt auf die Rechtecke bzw. Regionen verteilt.

Schritt 2c: In diesem Schritt werden alle komprimierten Kanten von T_c^+ expandiert. Jede komprimierte Kante (R, C) von T_c^+ wird jeweils durch einen Baum $T(R, C)$ ersetzt, der aus einer Sequenz $R = R_1, \ldots, R_k = C$ von fairen Schnitten hervorgeht. Der entstehende Baum ist T''.

Schritt 2a: Anhängen fehlender Blätter Um die durch den Fall 2 verworfenen Rechtecke als Blätter an den Baum T_c anzuhängen, wird die Knotenliste von T_c durchlaufen. Jeder Knoten der Form (R, R_1, null) wird dabei durch den Knoten (R, R_1, R_2) ersetzt, wobei sich $R_2 = R \setminus R_1$ aus den in dem Knoten (R, R_1, null) gespeicherten Informationen berechnen lässt. Des Weiteren wird ein Knoten der Form $(R_2, \mathit{null}, \mathit{null})$ zur Knotenmenge von T_c hinzugefügt. Da an jeden Knoten höchstens ein neues Kind angehängt wird, hat der entstehende Baum T_c^+ ebenso wie T_c eine Größe von $\mathcal{O}(\sqrt{|P|})$. Der gesamte Schritt benötigt somit höchstens $\mathcal{O}(\mathit{scan}(\sqrt{|P|}))$ Speichertransfers.

Schritt 2b: Verteilung der Punkte aus P Um die Punkte auf die Blätter von T_c^+ bzw. auf die durch die komprimierten Kanten von T_c^+ definierten Regionen zu verteilen, wird das Rechteck R_0 zuerst in Zellen partitioniert.[9] Anschließend wird jeder Punkt aus P mit derjenigen Zelle markiert, die ihn enthält. Durch diese Markierung eines jeden Punktes kann die Verteilung auf die Blätter bzw. Regionen effizient berechnet werden.

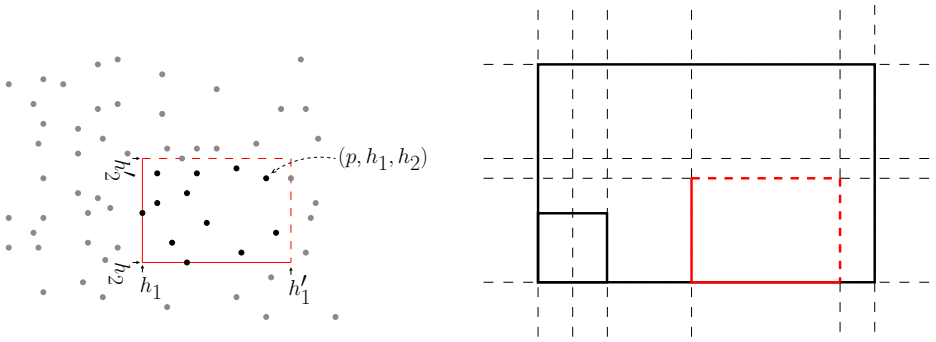
Die Partitionierung des Rechtecks R_0 in Zellen wird durch diejenigen Hyperebenen vollzogen, welche als Grenzen für die konstruierbaren Rechtecke auftreten. Eine Zelle Z ist dabei ein halboffenes Rechteck der Form $Z = [h_1, h_1') \times \cdots \times [h_d, h_d') \subset \mathbb{R}^d$ und wird durch Angabe des Tupels $(h_1, \ldots, h_d)$ eindeutig beschrieben, wobei der Wert h_i ein Eintrag aus der Begrenzungsebenenliste der Dimension i ist. Besitzt eine Zelle $Z = [h_1, h_1') \times \cdots \times [h_d, h_d') \subset \mathbb{R}^d$ einen maximalen Wert h_j' $(1 \leq j \leq d)$, so wird der „offene Rand“ dieser Zelle hinzugenommen, d.h. die Zelle zu $Z = [h_1, h_1') \times \cdots \times [h_j, h_j'] \times \cdots \times [h_d, h_d') \subset \mathbb{R}^d$ erweitert (es können natürlich mehrere Ränder hinzugenommen werden). Die Anzahl der entstehenden Zellen lässt sich analog zur Anzahl der konstruierbaren Rechtecken begrenzen: Pro Dimension kann eine Zelle eindeutig durch Angabe einer der obigen Hyperebenen

[8] Die Punkte können auch auf den Rändern der Rechtecke bzw. der Regionen liegen. Bei der Verteilung der Punkte wird darauf geachtet, dass jeder Punkt nur einem Rechteck bzw. einer Region zugeordnet wird.

[9] Der I/O-effiziente Algorithmus bearbeitet diese Verteilung der Punkte mit Hilfe von speziellen Anfragen an eine Datenstruktur namens *topology buffer tree* [59]. Hier wird die Verteilung der Punkte mit Hilfe einer Partitionierung der Rechtecke bzw. Regionen in Zellen umgesetzt, die auf der Arbeit von Callahan [25] basiert.

beschrieben werden. Da die Anzahl der Hyperebenen pro Dimension durch $\mathcal{O}(|P|^{1-\alpha})$ beschränkt ist, existieren folglich insgesamt höchstens $\mathcal{O}(|P|^{d(1-\alpha)}) \subset \mathcal{O}(\sqrt{|P|})$ Zellen. Eine Liste aller Zellen mit Einträgen der Form $(h_1, \ldots, h_d) \in \mathbb{R}^d$ kann demnach analog zur Berechnung der Eckenmenge von G mit $\mathcal{O}(scan(\sqrt{|P|}))$ Speichertransfers erstellt werden.

Um die Punkte aus P auf die Zellen zu verteilen, werden pro Dimension i die Punkte aus P und die Einträge der gerade erstellten Liste bzgl. der i-ten Koordinate sortiert. Ein Punkt $p \in P$ liegt in genau einer Zelle Z. Eine Traversierung beider Listen ermöglicht es somit, jeden Punkt aus P mit einem entsprechenden Wert h_i zu markieren. Nach Bearbeitung aller Dimensionen besitzt jeder Punkt eine Markierung der Gestalt $(h_1, \ldots, h_d)$, durch welche die Zelle, in der er liegt, eindeutig beschrieben wird, vgl. Abbildung 4.8 (a). Da d konstant ist, benötigt die Markierung aller Punkte insgesamt $\mathcal{O}(sort(|P|))$ Speichertransfers.



(a) Verteilung von Punkten auf eine Zelle

(b) Verteilung von Zellen auf eine Region

Abbildung 4.8: Verteilung von Punkten auf Zellen bzw. Verteilung von Zellen auf Rechtecke und Regionen.

Um jede Zelle mit dem Blatt bzw. der Region zu markieren, in dem bzw. in der sie liegt, wird die Knotenliste von T_c^+ traversiert. Für jedes traversierte Blatt bzw. jede komprimierte Kante wird die Liste der Zellen durchlaufen. Liegt eine Zelle in dem Rechteck bzw. in der durch die komprimierte Kante beschriebenen Region, so wird die Zelldarstellung um diese Information ergänzt, vgl. Abbildung 4.8 (b). Ein Blatt bzw. eine komprimierte Kante kann jeweils während der Traversierung mittels der impliziten Speicherung der Kantenmenge von T_c^+ erkannt werden. Da sowohl die Größe von T_c^+ als auch die Anzahl der Zellen durch $\mathcal{O}(\sqrt{|P|})$ begrenzt ist, benötigt die verschachtelte Traversierung insgesamt höchstens $\mathcal{O}(scan(|P|))$ Speichertransfers.

Die Punkte können nun auf die Blätter bzw. Regionen verteilt werden: Sowohl die um die jeweilige Zelldarstellung ergänzten Punkte aus P als auch die Einträge der Liste der Zellen werden dazu lexikographisch sortiert. Durch eine Traversierung beider Listen kann dann ein Punkt um die Darstellung des Blatts bzw. der Region ergänzt werden, in dem bzw. in der er liegt. Ein Punkt p wird also zu einem Tupel $(p, R, null)$ bzw. (p, R, C)

erweitert, wenn er in einem Rechteck R bzw. in einer durch eine komprimierte Kante (R, C) beschriebenen Region liegt.

Ein letzter Sortierschritt ermöglicht es, jedes Blatt R bzw. jede komprimierte Kante (R, C) von T_c^+ mit den Punkten zu markieren, die dem Blatt bzw. der komprimierten Kante zugeordnet wurden: Beide bzgl. der Rechtecke sortierten Listen werden dazu traversiert. Bei dieser Traversierung wird für den Baum eine neue Liste von Tupeln erstellt, wobei ein Tupel $(R, null, null)$, welches ein Blatt mit den Punkten $p_1, \ldots, p_j$ darstellt, durch die Folge von Tupeln $(R, null, null, p_1), \ldots, (R, null, null, p_j)$ ersetzt wird. Entsprechend wird ein eine komprimierte Kante darstellendes Tupel $(R, C, null)$ durch eine Folge von Tupeln der Form $(R, C, null, p_1), \ldots, (R, C, null, p_j)$ ersetzt, wenn der komprimierten Kante Punkte $p_1, \ldots, p_j$ zugeordnet wurden, vgl. Abbildung 4.9.

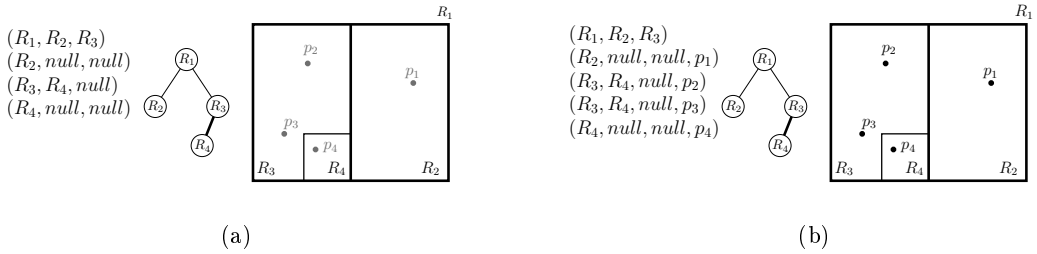


Abbildung 4.9: Erweiterung der Tupelliste

Der bis zu diesem Zeitpunkt erstellte Baum wird also durch eine Liste von Tupeln dargestellt, welche insgesamt $\mathcal{O}(\frac{|P|}{B})$ Speicherblöcke belegt. Der gesamte Schritt benötigt $\mathcal{O}(sort(|P|))$ Speichertransfers.

Schritt 2c: Komprimierte Kanten expandieren Um die komprimierten Kanten von T_c^+ zu ersetzen, simulieren Govindarajan *et al.* [40] eine Phase des sequentiellen Algorithmus aus Abschnitt 4.2. Sie ersetzen dabei eine komprimierte Kante (R, C) durch einen Baum $T(R, C)$. Der Baum $T(R, C)$ besteht aus einem Pfad von R nach C, wobei jeder innere Knoten als zusätzliches Kind ein Blatt besitzt. Durch die Ersetzung jeder komprimierten Kante (R, C) aus T_c^+ entsteht der Baum T''. Dieses Vorgehen zur Konstruktion von T'' wird hier übernommen.

Die Ersetzungsprozedur Zu einer komprimierten Kante (R, C) wird der Baum $T(R, C)$ mit Hilfe der folgenden Prozedur DecompressEdge konstruiert, welche beginnend mit $\bar{R} = R$ solange ausgeführt wird, bis $\bar{R}$ mit C übereinstimmt: Gilt $l_{max}(\bar{R}) > 3l(C)$, so wird das Rechteck $\bar{R}$ durch diejenige Hyperebene in zwei Rechtecke R_1 und R_2 gleicher Größe zerlegt, die senkrecht zur $i_{max}(\bar{R})$-ten Koordinatenachse steht. Ansonsten teilen $\bar{R}$ und C in der Dimension $i_{max}(\bar{R})$ eine gemeinsame Seite. Sei dann H diejenige Hyperebene, welche die andere Seite von C in der Dimension $i_{max}(\bar{R})$ enthält. Das Rechteck

$\bar{R}$ wird dann mittels H in zwei Rechtecke R_1 und R_2 aufgeteilt. In beiden Fällen sei R_1 das Rechteck der beiden, welches C enthält. Gilt $R_1 \neq C$, so wird das Rechteck $\bar{R} = R_1$ iterativ weiter aufteilt.

4.3.8 Lemma. *Die obige Prozedur* DecompressEdge *hält die folgenden Invarianten aufrecht:*

(I) Für jedes $i \in \{1, \ldots, d\}$ gilt entweder $l_i(\bar{R}) \geq \frac{3}{2} l(C)$ oder $l_i(\bar{R}) = l(C)$.

(II) Das Rechteck $\bar{R}$ ist eine Box.

Beweis. Das Rechteck $\bar{R} = R$ ist eine Box, da es die Invariante (iii) im Algorithmus zur Konstruktion des Baums T_c erfüllt. Aufgrund der speziellen Wahl der Konstanten c (siehe Fall 3) und $l_{max}(\bar{R}) \leq 3 l_{min}(\bar{R})$ folgt weiterhin $l(C) = \frac{3}{2} l_{max}(R') < \frac{3}{2} \cdot \frac{4}{27} l_{max}(\bar{R}) \leq \frac{2}{3} l_{min}(\bar{R})$ und somit $l_i(\bar{R}) \geq \frac{3}{2} l(C)$ für $i = 1, \ldots, d$. Die Invarianten (I) und (II) sind demnach für das Startrechteck R erfüllt.

Sei nun ein Rechteck $\bar{R} \neq R$ gegeben, welches durch Zerlegung eines Rechtecks $\hat{R}$ hervorgegangen ist. Dass $\bar{R}$ die Invariante (I) erfüllt, folgt direkt aus der Fallunterscheidung, anhand derer das Rechteck $\hat{R}$ aufgeteilt wurde. Um die Invariante (II) für das Rechteck $\bar{R}$ zu verifizieren, kann induktiv angenommen werden, dass das Rechteck $\hat{R}$ die beiden Invarianten erfüllt. Da das Rechteck $\hat{R}$ in der Dimension $i_{max}(\hat{R})$ aufgeteilt wurde, gilt $3 l_i(\bar{R}) = 3 l_i(\hat{R}) \geq l_{max}(\hat{R}) \geq l_{max}(\bar{R})$ für $i \neq i_{max}(\hat{R})$. Um zu zeigen, dass $\bar{R}$ eine Box ist, muss noch $3 l_i(\bar{R}) \geq l_{max}(\bar{R})$ für $i = i_{max}(\hat{R})$ bewiesen werden. Wurde $\hat{R}$ in zwei gleich große Hälften aufgeteilt, so ist die Behauptung offensichtlich, da $3 l_{i_{max}(\hat{R})}(\bar{R}) \geq l_{max}(\hat{R}) \geq l_{max}(\bar{R})$ gilt. Ansonsten kann aus $l_{max}(\hat{R}) \leq 3 l(C)$, der für das Rechteck $\hat{R}$ gültigen Invariante (I) und $\hat{R} \neq C$ ($\Rightarrow l_{max}(\hat{R}) > l(C)$) die Ungleichung

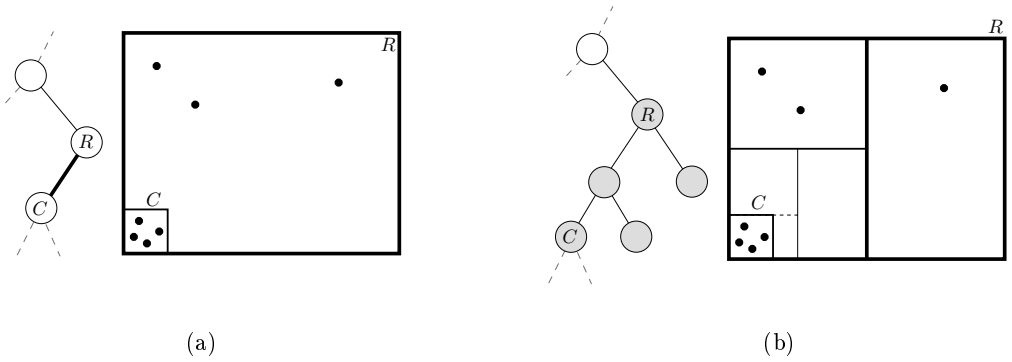
$$\frac{1}{3} l_{max}(\hat{R}) \leq l(C) \leq \frac{2}{3} l_{max}(\hat{R}).$$

gefolgert werden. Somit gilt auch in diesem Fall $3 l_{i_{max}(\hat{R})}(\bar{R}) \geq l_{max}(\hat{R}) \geq l_{max}(\bar{R})$. Das Rechteck $\bar{R}$ erfüllt also auch die zweite Invariante. □

Die Terminierung der Prozedur DecompressEdge ist offensichtlich. Aus den obigen Betrachtungen folgt weiterhin, dass es sich bei den Zerlegungen um faire Schnitte handelt und dass alle entstehenden Rechtecke Boxen sind. Somit ergibt sich unter anderem $R \rightsquigarrow C$, d.h. die Existenz einer durch faire Schnitte entstehenden Sequenz von Rechtecken $R = \bar{R}_1, \ldots, \bar{R}_k = C$ ist gesichert.

Der Baum $T(R, C)$ wird während des Aufteilungsprozesses wie folgt erstellt: Die Wurzel von $T(R, C)$ ist das Rechteck R. Enthält das Rechteck R_2 bei einer der obigen Zerlegungen eines Rechtecks $\bar{R}$ mindestens einen Punkt, so werden die Punkte aus $\bar{R} \setminus C$ auf die beiden Rechtecke R_1 und R_2 verteilt, R_1 und R_2 als Kinder an den Knoten $\bar{R}$ angehängt, und es wird der Prozess mit R_1 fortgesetzt. Andernfalls wird der Knoten $\bar{R}$ durch R_1 ersetzt

und der Prozess mit R_1 fortgeführt. Eine vereinfachte Darstellung der Ersetzung einer komprimierten Kante (R, C) durch den Baum $T(R, C)$ wird in Abbildung 4.10 gegeben.



(a) (b)

Abbildung 4.10: Ersetzung einer komprimierten Kante (R, C) durch den Baum $T(R, C)$.

Effiziente Umsetzung der Ersetzungsprozedur Es wird nun die hinsichtlich des *cache-oblivious*-Modells effiziente Umsetzung der Prozedur DecompressEdge, d. h. die im Kontext des *cache-oblivious*-Modells effiziente Berechnung der Sequenz $R = \bar{R}_1, \ldots, \bar{R}_k = C$, beschrieben. Die Kante (R, C) wurde durch den (vorherigen) Schritt $2a$ um diejenigen Punkte ergänzt, die in der Region $R \setminus C$ liegen. Diese Punkte bzw. Kopien dieser Punkte werden zunächst bzgl. jeder Dimension i sortiert und in entsprechenden Listen $L_1, \ldots, L_d$ gespeichert. Um ein Rechteck $\bar{R}$ in der Dimension $i_{max} = i_{max}(\bar{R})$ in die beiden Rechtecke R_1 und R_2 aufzuteilen, wird anschließend die Liste $L_{i_{max}}$ von dem zum Rechteck R_2 gehörenden Ende der Liste aus bis zum ersten Auftreten eines Punktes aus R_1 durchlaufen.[10] Wird dabei kein Punkt aus R_2 traversiert (dann enthält R_2 keinen Punkt aus $\bar{R} \setminus C$), so wird das Rechteck $\bar{R}$ in $T(R, C)$ durch R_1 ersetzt. Ansonsten werden die beiden Knoten R_1 und R_2 an $\bar{R}$ angehängt, die traversierten Punkte aus der Liste $L_{i_{max}}$ entfernt und die gelöschten Punkte zu dem Blatt R_2 hinzugefügt.

In beiden Fällen wird, wenn $R_1 \neq C$ gilt, der Aufteilungsprozess mit R_1 fortgeführt. Zuvor müssen jedoch die aus der Liste $L_{i_{max}}$ entfernten Punkte auch aus den Listen L_i mit $i \neq i_{max}$ entfernt werden. Da die Punkte in diesen Listen in beliebiger Reihenfolge gespeichert sein können, wäre das Entfernen der Punkte an dieser Stelle jedoch ineffizient. Stattdessen wird eine Traversierung einer Liste L_i bei einer der folgenden Zerlegungen wie folgt erweitert: Bei jedem traversierten Punkt wird getestet, ob dieser in dem aktuellen Rechteck R_2 liegt. Liegt der Punkt in R_2, so wird er wie oben beschrieben aus der aktuellen Liste gelöscht und zu dem Blatt R_2 hinzugefügt. Ansonsten liegt der Punkt in $R \setminus \bar{R}$ und wurde somit schon in einem vorherigen Schritt zu einem Blatt hinzugefügt. Ein solcher Punkt wird ohne weitere Aktionen aus L_i entfernt.

[10] Das entsprechende Ende kann mit Hilfe des Rechtecks C ohne weitere Speichertransfers identifiziert werden.

Insgesamt wird eine Liste durch die obigen Traversierungen nur einmal durchlaufen. Für das Durchlaufen aller Listen $L_1, \ldots, L_d$ werden also $\mathcal{O}(scan(|P \cap (R \setminus C)|))$ Speichertransfers benötigt. Die Ersetzung aller komprimierten Kanten von T_c^+ benötigt dann insgesamt $\mathcal{O}(sort(|P|))$ Speichertransfers: Bei jeder Ersetzung einer komprimierten Kante (R, C) durch einen Baum $T(R, C)$ müssen die Punkte aus $R \setminus C$ zuerst sortiert werden. Die durch diese Sortierung entstehenden Listen werden anschließend bei der Erstellung des Baums $T(R, C)$ insgesamt einmal traversiert. Jeder der Punkte aus P nimmt also bei der Ersetzung aller komprimierten Kanten insgesamt an konstant vielen Sortierschritten teil. Die Ersetzung aller komprimierten Kanten von T_c^+ verursacht somit höchstens $\mathcal{O}(sort(|P|))$ Speichertransfers.

Die Größe des durch das obige Verfahren konstruierten Baums T'' beträgt $\mathcal{O}(|P|)$: Die Größe des Baums T_c^+ beträgt, wie oben beschrieben, $\mathcal{O}(\sqrt{|P|})$. Bei der Ersetzung einer komprimierten Kante (R, C) durch den Baum $T(R, C)$ werden pro Aufteilung eines Rechtecks $\bar{R}$ in zwei Rechtecke R_1 und R_2 nur dann zwei neue Knoten hinzugefügt, wenn das Rechteck R_2 mindestens einen Punkt enthält. Durch die Ersetzung der komprimierten Kanten in T_c^+ können somit höchstens $\mathcal{O}(|P|)$ neue Knoten hinzukommen. Insgesamt belegt die Knotenliste des Baums demnach höchstens $\mathcal{O}(\frac{|P|}{B})$ Speicherblöcke.

4.3.9 Lemma. *Sei eine endliche nicht-leere Punktmenge $P \subset \mathbb{R}^d$, eine Box R_0 mit $P \subset R_0$ und eine reelle Konstante $\alpha \in [1-\frac{1}{4d}, 1)$ gegeben. Dann kann der Baum T'' aus Algorithmus 4.4 unter Verwendung von $\mathcal{O}(\frac{|P|}{B})$ Speicherblöcken mit $\mathcal{O}(sort(|P|))$ Speichertransfers konstruiert werden.*

Konstruktion von T'

Um den unvollständigen fairen Schnittbaum T' von P aus T'' zu erstellen, müssen in T'' alle Knoten R mit $R \cap P = \emptyset$ entfernt und alle Wege, die aus Knoten vom Grad 2 bestehen, komprimiert werden. Dazu wird die Knotenmenge von T'' traversiert. Während der Traversierung kann jedes Blatt, welches keine Punkt enthält, identifiziert und als „entfernen“ markiert werden. Enthält das Blatt Punkte aus P, so erhält es „behalten“ als Markierung.

Mittels der *time-forward processing*-Technik kann T'' anschließend von den Blättern aus zur Wurzel hin traversiert werden. Existiert zwischen zwei Knoten R_1 und R_2 aus T'' eine Kante (R_1, R_2), so gilt $\sum_{j=1}^{d} l_j(R_2) < \sum_{j=1}^{d} l_j(R_1)$. Die für die Anwendung der Technik benötigte topologische Sortierung der Knoten kann also wie zuvor durch einen Sortierschritt erstellt werden, der die Knoten in absteigender Reihenfolge bzgl. der Summen über ihre Seitenlängen sortiert. Bei der von den Blättern ausgehenden Traversierung kann dann jeder Knoten mit „behalten“, „zusammenziehen“ oder „entfernen“ markiert werden, je nachdem ob keins, eins oder beide seiner Kinder mit „entfernen“ markiert sind. Zusätzlich wird jeder Knoten, welcher nicht als „entfernen“ markiert ist, mit einem Knoten $K(R)$ markiert. Dabei gilt $K(R) = R$, wenn R selbst als „behalten“ markiert ist. Ansonsten

gilt $K(R) = K(R')$, wobei R' dasjenige der beiden Kinder von R ist, welches nicht die Markierung „entfernen“ trägt. Während dieses Vorgangs können die Kinder R_1 und R_2 eines jeden als „behalten“ markierten Knotens durch die entsprechenden Nachfolger $K(R_1)$ und $K(R_2)$ ersetzt werden.

In einem letzten Schritt werden alle als „behalten“ markierten Knoten aus der Knotenmenge von T'' extrahiert. Da die Kanten implizit in den Knoten gespeichert sind, ist das Ergebnis dieser Extraktion der gesuchte Baum T'. Die Konstruktion von T' mit Hilfe des Baums T'' benötigt somit insgesamt $\mathcal{O}(sort(|P|))$ Speichertransfers. Weiterhin belegt die Knotenliste von T' höchstens $\mathcal{O}(\frac{N}{|P|})$ Speicherblöcke.

Der konstruierte Baum T' erfüllt alle an einen unvollständigen fairen Schnittbaum gestellten Bedingungen: Für einen Knoten v sei $\tilde{R}(v)$ das in dem Knoten v gespeicherte Rechteck. Die durch einen Knoten v dargestellte Menge $\sigma(v)$ liegt vollständig in $\tilde{R}(v)$. Anhand des Konstruktionsverfahren lässt sich erkennen, dass die Knoten von T' die an einen unvollständigen fairen Schnittbaum gestellten Bedingungen erfüllen; insbesondere sind alle Rechtecke mit Ausnahme des Startrechtecks durch faire Schnitte entstanden. Sequenzen der Form $R \rightsquigarrow R'$ können durch das Expandieren von komprimierten Kanten und durch das Entfernen „leerer“ Blätter im letzten Schritt des gesamten Verfahrens entstehen. Da für jedes Blatt v das Rechteck $\tilde{R}(v)$ in einer Dimension vollständig in einer Scheibe liegt, gilt $|\sigma(v)| \leq |P|^{\alpha}$. Damit ist das Lemma 4.3.3 bewiesen.

Da jeder faire Schnittbaum nach Lemma 4.3.1 auch ein fairer Aufteilungsbaum ist, ergibt sich das folgende Korollar:

4.3.10 Korollar. *Sei eine endliche nicht-leere Punktmenge $P \subset \mathbb{R}^d$ gegeben. Dann kann unter Verwendung von $\mathcal{O}(\frac{|P|}{B})$ Speicherblöcken mit $\mathcal{O}(sort(|P|))$ Speichertransfers ein fairer Aufteilungsbaum T von P berechnet werden.*

Für eine endliche nicht-leere Punktmenge $P \subset \mathbb{R}^d$ lässt sich also mittels der Funktion FairCutTree unter Verwendung von $\mathcal{O}(\frac{|P|}{B})$ Speicherblöcken und $\mathcal{O}(sort(|P|))$ Speichertransfers ein fairer Schnitt- bzw. fairer Aufteilungsbaum T von P berechnen. Für das im Folgenden beschriebene im Kontext des *cache-oblivious*-Modells effiziente Konstruktionsverfahren einer WSPD von P muss jeder Knoten v eines solchen Baums noch um das zugehörige Begrenzungsrechteck $R(\sigma(v))$ ergänzt werden. Diese Erweiterung kann erneut mit Hilfe der *time-forward processing*-Technik realisiert werden: In jedem Knoten v von T ist das Rechteck $\tilde{R}(v)$ gespeichert. Für einen (von der Wurzel verschiedenen) Knoten v aus T gilt weiterhin $\sum_{j=1}^{d} l_j(\tilde{R}(v)) < \sum_{j=1}^{d} l_j(\tilde{R}(p(v)))$. Die Knoten aus T können also wieder topologisch sortiert und anschließend von den Blättern aus zur Wurzel hin durchlaufen werden. Das Begrenzungsrechteck eines Blatts v aus T mit $\sigma(v) = \{p\}, p \in P$, besteht aus dem Punkt $R(\sigma(v)) = p$. Für einen inneren Knoten v mit Kindern v_1 und v_2 ist das Begrenzungsrechteck $R(\sigma(v))$ das kleinste Rechteck, welches $R(\sigma(v_1))$ und $R(\sigma(v_2))$ enthält. Die

Begrenzungsrechtecke können somit während dieser Traversierung effizient erstellt werden. Der gesamte Vorgang benötigt $\mathcal{O}(sort(|P|))$ Speichertransfers.

4.3.3 Konstruktion einer WSPD

In diesem Abschnitt wird ein hinsichtlich des *cache-oblivious*-Modells effizienter Algorithmus beschrieben, der mit Hilfe eines vorliegenden fairen Aufteilungsbaums T einer endlichen nicht-leeren Punktmenge $P \subset \mathbb{R}^d$ eine WSPD von P bzgl. eines Wertes $s \in \mathbb{R}^+$ berechnet. Der im Folgenden beschriebene Algorithmus stimmt zwar mit dem Algorithmus von Govindarajan *et al.* [40] zur Konstruktion einer WSPD im I/O-Modell überein, soll jedoch an dieser Stelle der Vollständigkeit halber kurz beschrieben werden:

Sei dazu eine endliche nicht-leere Punktmenge $P \subset \mathbb{R}^d$ und ein fairer Aufteilungsbaum T von P gegeben. Aufgrund der obigen Bemerkung kann angenommen werden, dass in jedem Blatt v von T die Koordinaten des Punktes $p \in P$ mit $\sigma(v) = \{p\}$ und in jedem inneren Knoten v das Begrenzungsrechteck $R(\sigma(v))$ gespeichert sind. Analog zu Abschnitt 4.2.1 wird jedes Paar $\{A, B\}$ der zu berechnenden Realisation durch Knoten $v, w \in T$ mit $\sigma(v) = A$ und $\sigma(w) = B$ und *nicht* explizit durch die Punkte aus den Mengen $A, B \subseteq P$ dargestellt. Die berechnete Realisation $\mathcal{R}$ wird also insgesamt durch eine Menge $\mathcal{R}_T = \{\{v_1, w_1\}, \ldots, \{v_k, w_k\}\}$ von Knotenpaaren $\{v_i, w_i\}$ mit Knoten $v_i, w_i \in T$ repräsentiert.

Der Algorithmus zur Konstruktion einer WSPD simuliert die Berechnung einer bzgl. s scharf getrennten Realisation von $P \otimes P$ mittels der Funktion ComputeWSR (Algorithmus 4.2) aus Abschnitt 4.2.1. Die Simulation besteht aus der Traversierung eines Graphen $\bar{G}$ mit Hilfe der *time-forward processing*-Technik. Nach Govindarajan *et al.* [40] liegt die Schwierigkeit bei Anwendung dieser Technik darin, dass die Kantenmenge von $\bar{G}$ nicht im Voraus erstellt werden kann.

Der Graph $\bar{G}$, dessen Traversierung zur Simulation der Berechnungen von ComputeWSR (Algorithmus 4.2) genutzt wird, ist wie folgt definiert: Die Eckenmenge von $\bar{G}$ stimmt mit der Knotenmenge des fairen Aufteilungsbaums T überein. Für zwei Knoten v_1 und v_2 existiert in $\bar{G}$ eine gerichtete Kante von v_1 nach v_2, wenn es in einem Berechnungsbaum $T(\{v, w\})$ mit $\{v, w\} \in R$ (siehe Abschnitt 4.2.1) zwei Knoten (v_1, w_1) und (v_2, w_2) gibt, so dass (v_1, w_1) der Vaterknoten von (v_2, w_2) ist. Govindarajan *et al.* zeigen für den so definierten Graphen die Gültigkeit des folgenden Lemmas, auf dessen Beweis verzichtet wird.

4.3.11 Lemma ([40]). *Der Graph $\bar{G}$ ist ein gerichteter azyklischer Graph.*

Die Berechnung einer WSPD von P mittels der Funktion ComputeWSR simulieren Govindarajan *et al.* wie folgt: Eine Traversierung der Eckenmenge von G ermöglicht es, jedes Paar der Form $\{v, w\} \in R, w \prec v$ (siehe Algorithmus 4.2), in der Ecke $v \in G$ zu speichern. Anschließend wird die auf diese Weise veränderte Eckenmenge von G in topologisch sortierter Reihenfolge durchlaufen. Für jeden Knoten $v \in G$ werden dabei die in dem

Knoten gespeicherten Paare traversiert. Bei jedem traversierten Paar $\{v, w\}, w \prec v$, wird geprüft, ob $\sigma(v)$ und $\sigma(w)$ scharf getrennt sind. Ist dies der Fall, so wird das Paar $\{v, w\}$ zu der Menge $\mathcal{R}_T$ hinzugefügt. Andernfalls seien (v', w') und (v'', w'') die beiden Kinder des Knotens (v, w) in dem Berechnungsbaum, der den Knoten (v, w) enthält (zu dem Knoten (v, w) gibt es genau einen entsprechenden Berechnungsbaum). Nach Definition von $\bar{G}$ existieren in $\bar{G}$ Kanten der Form (v, v') und (v, v''), da es in dem obigen Berechnungsbaum Kanten von der Form $((v, w), (v', w'))$ und $((v, w), (v'', w''))$ geben muss. Somit kann das Paar $\{v', w'\}$ entlang der Kante (v, v') und das Paar $\{v'', w''\}$ entlang der Kante (v, v'') „gesendet“ werden, um diese Paare zu den Mengen hinzuzufügen, die jeweils in den Knoten v' bzw. v'' gespeichert sind.

Anstatt die Kantenmenge von $\bar{G}$ im Voraus zu berechnen, wird diese während des Ablaufs des im Folgenden beschriebenen Algorithmus erstellt. Die Kantenmenge von $\bar{G}$ muss also für die obige Traversierung nicht von Anfang an vorliegen. In einem Vorverarbeitungsschritt des Algorithmus wird die Knotenmenge von T wie folgt erweitert: Zuerst wird eine Postorder-Nummerierung der Knoten von T berechnet. Anschließend werden die Knoten bzgl. der durch die Postorder-Nummerierung definierten „$\prec$“-Relation sortiert. Jedem Knoten $v \in T$ wird dadurch eine Nummer $\eta(v)$ zugeordnet, die dessen Position in der sortierten Folge der Knoten beschreibt. Zusätzlich werden in jedem inneren Knoten $v \in T$ die Nummern $\eta(v_1)$ und $\eta(v_2)$ seiner beiden Kinder v_1 und v_2 gespeichert. Zuletzt erfolgt eine Sortierung der Knoten, bei welcher diese in absteigender Reihenfolge bzgl. ihrer zugehörigen Nummern sortiert werden. Der Wurzelknoten r von T befindet sich nach diesem Sortierschritt am Anfang der Knotenliste. Die Vorverarbeitungsphase verursacht insgesamt höchstens $\mathcal{O}(sort(|P|))$ Speichertransfers, da die Berechnung einer Postorder-Nummerierung und das Kopieren der Nummern $\eta(v)$ eines Knotens v aus T in den Vaterknoten von v mit Hilfe der *time-forward processing*-Technik realisiert werden können, vgl. Govindarajan *et al.* [40]. Des Weiteren wird die Knotenmenge von T zweimal sortiert, was ebenfalls höchstens $\mathcal{O}(sort(|P|))$ Speichertransfers verursachen kann.

Nach diesem Vorverarbeitungsschritt wird der Graph $\bar{G}$ traversiert. Dazu wird die Liste der Knoten von T durchlaufen. Für jeden inneren Knoten aus T mit Kindern v und w, $w \prec v$, wird bei dieser Traversierung ein Paar (v, w) mit Priorität $\eta(v)$ in eine Prioritätswarteschlange Q eingefügt. Nachdem alle Knoten in die Prioritätswarteschlange Q eingefügt wurden, werden die Knoten ihrem Auftreten nach bearbeitet und für jeden Knoten v alle Paare der Form (v, w) aus Q entfernt. Für jedes Paar (v, w) wird dabei getestet, ob $\sigma(v)$ und $\sigma(w)$ scharf getrennt bzgl. s sind. Ist dies der Fall, so wird das Paar (v, w) der Menge $\mathcal{R}_T$ hinzugefügt. Sind die Mengen nicht scharf getrennt, so besitzt v zwei Kinder v_1 und v_2. Gilt $\eta(v_1) > \eta(w)$, so wird ein Paar (v_1, w) mit Priorität $\eta(v_1)$ in die Prioritätswarteschlange Q eingefügt. Andernfalls wird das Paar (w, v_1) mit Priorität $\eta(w)$ in Q eingefügt. Auf die gleiche Weise wird mit dem zweiten Kind v_2 von v verfahren. Dieses Vor-

gehen entspricht dem „Senden“ der Paare $\{v_1, w\}$ und $\{v_2, w\}$ entlang der entsprechenden Kanten von G.

Wie Govindarajan *et al.* [40] bemerken, ist der obige Algorithmus eine standardmäßige Anwendung der *time-forward processing*-Technik bis auf den Unterschied, dass jede Ecke ihre direkten Nachfolger mittels der Informationen erzeugt, die sie durch ihre eingehenden Kanten erhält. Für eine mittels einer Ecke v erzeugte Ecke w gilt $\eta(v) > \eta(w)$. Somit wird jedes Paar der Form (v, w) zur Ecke w geschickt, bevor diese bearbeitet wird.

Alle Paare, die in die Prioritätswarteschlange Q eingefügt bzw. entfernt werden, korrespondieren mit Knoten in Berechnungsbäumen. Jedes dieser Paare wird höchstens einmal eingefügt und einmal gelöscht. Da die Größe *aller* Berechnungsbäume höchstens $\mathcal{O}(|P|)$ beträgt (siehe Abschnitt 4.2.1), ist die Anzahl der insgesamt benötigten Operationen auf der Prioritätswarteschlange durch $\mathcal{O}(|P|)$ begrenzt. Somit kann die Simulation der Funktion ComputeWSR durch die Traversierung des Graphen $\bar{G}$ mit Hilfe von höchstens $\mathcal{O}(sort(|P|))$ Speichertransfers realisiert werden. Nach Korollar 4.3.10 kann ein fairer Aufteilungsbaum von P mit $\mathcal{O}(sort(|P|))$ Speichertransfers berechnet werden. Es ergibt sich also das folgende Theorem:

4.3.12 Theorem. *Sei eine endliche nicht-leere Punktmenge $P \subset \mathbb{R}^d$ und eine Konstante $s \in \mathbb{R}^+$ gegeben. Dann kann unter Verwendung von $\mathcal{O}(\frac{|P|}{B})$ Speicherblöcken mit $\mathcal{O}(sort(|P|))$ Speichertransfers eine WSPD von P linearer Größe berechnet werden.*

Kapitel 5

Konstruktion dünner Spanner-Graphen

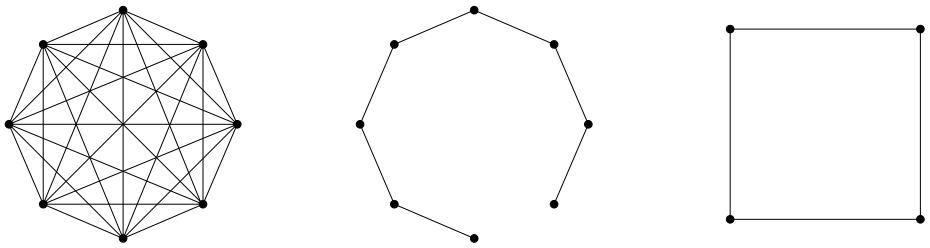
Dieses Kapitel handelt von im Kontext des RAM-, I/O- und *cache-oblivious*-Modell effizienten Verfahren zur Konstruktion sogenannter „dünn besetzter t-Spanner“ für endliche Punktmengen aus dem $\mathbb{R}^d$. Wesentlicher Bestandteil aller Konstruktionsverfahren ist dabei die im vorherigen Kapitel beschriebene WSPD-Datenstruktur.

In Hinblick auf diese Konstruktionsverfahren werden in Abschnitt 5.1 zunächst einige Begriffe eingeführt. In Abschnitt 5.2 wird anschließend ein allgemeines Verfahren zur Konstruktion von speziellen Spanner-Graphen beschrieben, welches unmittelbar zu effizienten Algorithmen im RAM-, I/O- und *cache-oblivious*-Modell führt. Der effiziente RAM- und der effiziente I/O-Algorithmus werden in Abschnitt 5.3 kurz erläutert. Der im Kontext des I/O-Modells effiziente Algorithmus lässt sich direkt in das *cache-oblivious*-Modell übertragen. Dieser *cache-oblivious*-Algorithmus wird in Abschnitt 5.4 ausführlich beschrieben.

5.1 Vorbemerkungen

Ein ungerichteter Graph $G = (S, E)$ wird als *geometrischer Graph* bezeichnet, wenn die Eckenmenge S eine endliche Punktmenge aus dem $\mathbb{R}^d$ ist. Weiterhin wird ein gewichteter geometrischer Graph $G = (S, E)$, bei dem das Kantengewicht $w(e)$ einer Kante $e = \{p, q\} \in E$ dem euklidischen Abstand $d(p, q)$ der beiden Endpunkte entspricht, als ein *euklidischer Graph* bezeichnet. In einem euklidischen Graphen wird die *Länge* eines Weges $p_0 p_1 \dots p_k$ nicht durch die Anzahl k der Kanten sondern durch die Summe

$$\sum_{i=0}^{k-1} d(p_i, p_{i+1})$$



(a) $\triangle(G) = 1$ (b) $\triangle(G) = 7$ (c) $\triangle(G) = \sqrt{2}$

Abbildung 5.1: Drei euklidische Graphen mit Eckenmengen aus dem $\mathbb{R}^2$ und jeweils zugehöriger Dilatation $\triangle(G)$

der Kantengewichte seiner Kanten definiert. Weiterhin ist für einen euklidischen Graphen $G = (S, E)$ die *Graphdistanz* $\delta_G(p, q)$ zweier Ecken $p, q \in S$ als die Länge des kürzesten Weges in G von p nach q definiert, d. h.

$$\delta_G(p, q) = \sum_{i=0}^{k-1} d(p_i, p_{i+1}),$$

falls in G ein Weg von p nach q existiert und $p_0 p_1 \dots p_k$ ein solcher Weg mit minimaler Länge ist bzw.

$$\delta_G(p, q) = \infty,$$

falls in G kein Weg von p nach q existiert.

Sei $t \geq 1$ eine reelle Konstante. Ein euklidischer Graph $G = (S, E)$ wird als *t-Spanner-Graph* (*t-Spanner*) für S bezeichnet, wenn es für jedes Paar voneinander verschiedener Ecken $p, q \in S$ einen Weg von p nach q gibt, dessen Länge höchstens t-mal so groß wie der Euklidische Abstand der beiden Ecken ist, d. h. wenn

$$\delta_G(p, q) \leq t \cdot d(p, q)$$

gilt. Der minimale Wert t', so dass ein euklidischer Graph $G = (S, E)$ ein t'-Spanner für seine Eckenmenge S ist, wird als die *Dilatation* $\triangle(G)$ von G bezeichnet; für diese gilt demnach

$$\triangle(G) = \max \left\{ \frac{\delta_G(p, q)}{d(p, q)} \mid p, q \in S, p \neq q \right\}.$$

Da für zwei voneinander verschiedene Ecken $p, q \in S$ eines vollständigen euklidischen Graphen $G = (S, E)$ stets $\delta_G(p, q) = d(p, q)$ gilt, besitzen solche eine Dilatation von $\triangle(G) = 1$, vgl. Abbildung 5.1. Weiterhin wird ein t-Spanner $G = (S, E)$ als *dünn (besetzt)* bezeichnet, wenn die Anzahl der Kanten linear in der Anzahl der Ecken ist, d. h. wenn $|E| \in \mathcal{O}(|S|)$ gilt.

In der Literatur finden sich zahlreiche Algorithmen zur Konstruktion dünn besetzter t-Spanner, welche jeweils für eine beliebige reelle Konstante $t > 1$ und eine endliche Punktmenge $S \subset \mathbb{R}^d$ einen dünn besetzten t-Spanner für S berechnen. Chen *et al.* [28] zeigten,

dass die untere Schranke für die Berechnung eines beliebigen t-Spanners für eine gegebene endliche Punktmenge $S \subset \mathbb{R}^d$ und eine reelle Konstante $t > 1$ im algebraischen Entscheidungsbaummodell $\Omega(|S| \log |S|)$ ist. Diese Schranke wird von den von Salowe [55], Vaidya [56] bzw. Callahan und Kosaraju [27] entwickelten Algorithmen angenommen; jeder dieser Algorithmen konstruiert einen entsprechenden dünn besetzten t-Spanner in einer Laufzeit von $\mathcal{O}(|S| \log |S|)$.

Zusätzlich zu der Forderung, dass die Anzahl der Kanten linear in der Anzahl der Ecken sein muss, können weitere Aspekte bei der Konstruktion eines t-Spanners $G = (S, E)$ berücksichtigt werden. Zu diesen zählen z. B. die Eigenschaft eines begrenzten Grades (d. h., dass der Grad einer jeden Ecke aus S durch eine globale Konstante beschränkt ist), die Eigenschaft eines geringen Durchmessers (d. h., dass jeweils zwei verschiedene Ecken aus S durch einen Weg verbunden werden können, welcher aus einer geringen Anzahl von Kanten besteht) oder die Eigenschaft eines geringen Gewichts (d. h., dass die die Summe aller Kantengewichte proportional zu dem Gewicht eines minimalen Spannbaums von S ist). Für eine Beschreibung dieser (im Folgenden unberücksichtigten) Eigenschaften und der entsprechenden Konstruktionsalgorithmen sei auf die Zusammenfassung von Eppstein [36] verwiesen.

5.2 Allgemeines Verfahren

Im vorherigen Abschnitt wurde das Konzept von dünn besetzten t-Spannern eingeführt. In diesem Abschnitt wird nun ein allgemeines Verfahren beschrieben, mit dessen Hilfe für beliebiges reelles $\varepsilon > 0$ ein dünn besetzter $(1 + \varepsilon)$-Spanner einer gegebenen endlichen Punktmenge $S \subset \mathbb{R}^d$ konstruiert werden kann. Dieses allgemeine Konstruktionsverfahren greift auf die im vorherigen Kapitel eingeführte WSPD-Datenstruktur zurück und basiert auf den Arbeiten von Callahan und Kosaraju [25, 26].

Für das weitere Vorgehen sei S eine endliche Punktmenge aus dem $\mathbb{R}^d$ und $\mathcal{R} = \{\{A_1, B_1\}, \ldots, \{A_k, B_k\}\}$ eine bzgl. eines Wertes $s > 0$ scharf getrennte Realisation von $S \otimes S$. Mit $\bar{d}(A_i, B_i)$ werde die minimale Distanz zweier Punkte $a \in A_i$ und $b \in B_i$ bezeichnet, d. h.

$$\bar{d}(A_i, B_i) = \min_{a \in A_i, b \in B_i} d(a, b).$$

Für eine reelle Konstante $\varepsilon > 0$ bezeichne $\mathbf{G}(S, \mathcal{R}, s, \varepsilon)$ weiterhin die Menge aller euklidischen Graphen $G = (S, E)$, deren Kantenmenge E jeweils für jedes Paar $\{A_i, B_i\}$ der bzgl. des Wertes s scharf getrennten Realisation $\mathcal{R}$ von $S \otimes S$ genau eine ungerichtete Kante $\{\bar{a}, \bar{b}\}$ mit $\bar{a} \in A_i$, $\bar{b} \in B_i$ und

$$d(\bar{a}, \bar{b}) \leq (1 + \varepsilon)\bar{d}(A_i, B_i)$$

enthält. Eine Kante $\{\bar{a}, \bar{b}\}$ der obigen Form wird als eine *ε-approximative Kante* zwischen A_i und B_i bezeichnet, vgl. Abbildung 5.2.

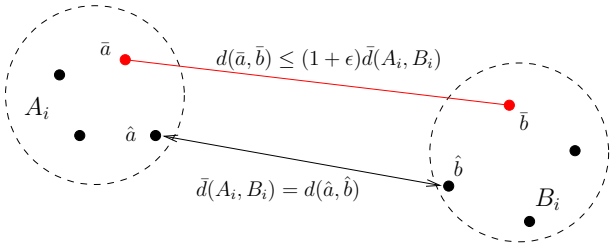


Abbildung 5.2: Eine ε-approximative Kante $\{\bar{a},\bar{b}\}$ zwischen zwei Punktmengen A_i und B_i ($\varepsilon > 0$)

Für ein Paar $\{A_i, B_i\}$ der Realisation $\mathcal{R}$ und zwei beliebige Punkte a, b mit $a \in A_i$ und $b \in B_i$ ergibt sich die folgende Abschätzung:

5.2.1 Lemma ([25, 26]). *Sei S eine endliche Punktmenge aus dem $\mathbb{R}^d$ und $\mathcal{R} = \{\{A_1, B_1\}, \ldots, \{A_k, B_k\}\}$ eine bzgl. eines Wertes $s > 0$ scharf getrennte Realisation von $S \otimes S$. Dann gilt*

$$d(a,b) \leq (1 + \frac{4}{s}) \cdot \bar{d}(A_i, B_i),$$

wobei $\{A_i, B_i\} \in \mathcal{R}$ ein Paar der Realisation und a bzw. b ein Punkt aus A_i bzw. B_i ist.

Beweis. Seien $\bar{a} \in A_i$ und $\bar{b} \in B_i$ zwei Punkte mit $d(\bar{a},\bar{b}) = \bar{d}(A_i, B_i)$. Da A_i und B_i bzgl. s scharf getrennt sind, gibt es zwei d-Kugeln K_1 und K_2 mit Radius $r \geq 0$, so dass $A_i \subset K_1$, $B_i \subset K_2$ und $d(K_1, K_2) \geq sr$ gilt. Für die beiden Punkte $\bar{a}$ und $\bar{b}$ folgt somit $4r = 4\frac{sr}{s} \leq \frac{4}{s} d(\bar{a},\bar{b})$ und demnach insgesamt

$$d(a,b) \leq d(a,\bar{a}) + d(\bar{a},\bar{b}) + d(\bar{b},b) \leq 2r + d(\bar{a},\bar{b}) + 2r \leq (1 + \frac{4}{s}) \cdot d(\bar{a},\bar{b}). \qquad \square$$

Das allgemeine Verfahren zur Konstruktion eines dünn besetzten $(1+\varepsilon)$-Spanners von S wird durch das folgende Lemma vorbereitet. Der Beweis dieses Lemmas macht von den Eigenschaften der Realisation $\mathcal{R}$ Gebrauch, aufgrund derer zu jedem ungeordneten Paar $\{a, b\}$ mit $a, b \in S$ und $a \neq b$ genau ein Paar $\{A_i, B_i\}$ der Realisation existiert, so dass $\{a, b\}$ in dem Interaktionsprodukt $A_i \otimes B_i$ von A_i und B_i enthalten ist, vgl. Kapitel 4. Diese „Überdeckungseigenschaft“ der Realisation im Verbund mit der Forderung, dass die Paare $\{A_i, B_i\}$ der Realisation jeweils bzgl. s scharf getrennt sind, führt zu folgendem Ergebnis:

5.2.2 Lemma ([25, 26]). *Seien $\varepsilon > 0$ und $s > 0$ zwei reelle positive Konstanten, S eine endliche Punktmenge aus dem $\mathbb{R}^d$, $\mathcal{R} = \{\{A_1, B_1\}, \ldots, \{A_k, B_k\}\}$ eine bzgl. s scharf getrennte Realisation von $S \otimes S$ und $G = (S, E)$ ein Graph aus $\boldsymbol{G}(S, \mathcal{R}, s, \varepsilon)$. Gilt $s > 4$, so existiert zu jedem Paar $\{a, b\}$ mit $a, b \in S$ und $a \neq b$ ein Weg in G von a nach b, dessen Länge höchstens*

$$\frac{1+\varepsilon}{1 - 4s^{-1}} \cdot d(a,b)$$

beträgt.

Beweis. Callahan und Kosaraju [25, 26] beweisen diese Aussage, indem sie eine rekursive Prozedur zur Konstruktion eines Weges zwischen den Punkten a und b angeben und anschließend beweisen, dass der so konstruierte Weg höchstens die angegebene Länge besitzt. Dieser Beweis entspricht der folgenden Induktion über die euklidischen Abstände zwischen Punkten aus S:

I.A.: Sei $\{a, b\}$ ein Paar von Punkten aus S mit $a \neq b$ und minimalem euklidischen Abstand $d(a, b)$. Aufgrund der Eigenschaften der Realisation $\mathcal{R}$ existiert genau ein Paar $\{A_i, B_i\}$ der Realisation $\mathcal{R}$ mit $\{a, b\} \in A_i \otimes B_i$. Weiterhin enthält E nach Definition von $\mathbf{G}(S, \mathcal{R}, s, \varepsilon)$ genau eine ε-approximative Kante $\{\bar{a}, \bar{b}\}$ zwischen A_i und B_i mit $\bar{a} \in A_i$ und $\bar{b} \in B_i$. Für die beiden Endpunkte $\bar{a}$ und $\bar{b}$ dieser Kante muss $\bar{a} = a$ und $\bar{b} = b$ gelten: Da A_i und B_i bzgl. s scharf getrennt sind, gibt es zwei d-Kugeln K_1 und K_2 mit Radius $r \geq 0$, so dass $A_i \subset K_1$, $B_i \subset K_2$ und $d(K_1, K_2) \geq sr$ gilt. Würde nun $\bar{a} \neq a$ gelten, so wäre aufgrund von $s > 4$ und

$$d(\bar{a}, a) \leq 2r < sr \leq d(a, b)$$

der Euklidische Abstand zwischen a und $\bar{a}$ echt kleiner als der zwischen a und b, im Widerspruch zur Minimalität von $d(a, b)$. Analog folgt $\bar{b} = b$. Die Kante $\{\bar{a}, \bar{b}\} = \{a, b\}$ muss somit in E enthalten sein. Es existiert demnach in diesem Fall trivialerweise ein Weg in G von a nach b, dessen Länge höchstens

$$d(a, b) \leq \frac{1 + \varepsilon}{1 - 4s^{-1}} \cdot d(a, b)$$

beträgt.

I.S.: Sei nun ein beliebiges Paar $\{a, b\}$ mit $a, b \in S$ und $a \neq b$ gegeben. Analog zu obiger Argumentation existiert zu diesem ein Paar $\{A_i, B_i\}$ der Realisation $\mathcal{R}$ mit $\{a, b\} \in A_i \otimes B_i$ und eine ε-approximative Kante $\{\bar{a}, \bar{b}\} \in E$ zwischen A_i und B_i mit $\bar{a} \in A_i$ und $\bar{b} \in B_i$. Ebenso gilt aufgrund von $s > 4$ wieder

$$d(\bar{a}, a) \leq 2r < sr \leq d(a, b) \quad \text{bzw.} \quad d(\bar{b}, b) \leq 2r < sr \leq d(a, b),$$

wobei $r \geq 0$ eine entsprechende Konstante ist. Da der Euklidische Abstand zwischen $\bar{a}$ und a bzw. zwischen $\bar{b}$ und b echt kleiner ist als der zwischen a und b, kann induktiv angenommen werden, dass es zwischen $\bar{a}$ und a bzw. zwischen $\bar{b}$ und b einen Weg in G gibt, dessen Länge höchstens

$$\frac{1 + \varepsilon}{1 - 4s^{-1}} \cdot d(a, \bar{a}) \quad \text{bzw.} \quad \frac{1 + \varepsilon}{1 - 4s^{-1}} \cdot d(b, \bar{b})$$

beträgt. Aufgrund von $d(\bar{a}, \bar{b}) \leq (1 + \varepsilon) \cdot \bar{d}(A_i, B_i) \leq (1 + \varepsilon) \cdot d(a, b)$ existiert demnach insgesamt ein Weg in G von a nach b mit einer Länge von maximal

$$(1 + \varepsilon) \cdot d(a, b) + \frac{1 + \varepsilon}{1 - 4s^{-1}} \left(d(a, \bar{a}) + d(b, \bar{b}) \right),$$

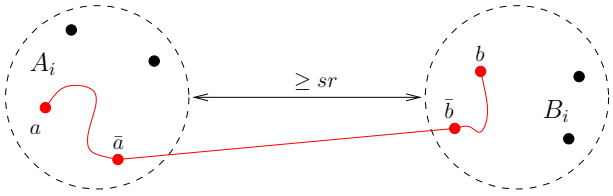


Abbildung 5.3: Ein Weg in G von a nach b mit einer euklidischen Länge von maximal $\frac{1+\varepsilon}{1-4s^{-1}} \cdot d(a,b)$. Der Weg von a nach $\bar{a}$ bzw. der Weg von b nach $\bar{b}$ besitzt eine euklidische Länge von höchstens $\frac{1+\varepsilon}{1-4s^{-1}} \cdot d(a,\bar{a})$ bzw. $\frac{1+\varepsilon}{1-4s^{-1}} \cdot d(b,\bar{b})$

vgl. Abbildung 5.3. Da A_i und B_i bzgl. s scharf getrennt sind, gilt weiter

$$d(a,\bar{a}) \leq 2r = \frac{2rs}{s} \leq \frac{2 \cdot d(a,b)}{s} \quad \text{bzw.} \quad d(b,\bar{b}) \leq 2r = \frac{2rs}{s} \leq \frac{2 \cdot d(a,b)}{s}.$$

Der Weg in G von a nach b besitzt somit maximal eine Länge von

$$\begin{aligned}(1+\varepsilon) \cdot d(a,b) + \frac{1+\varepsilon}{1-4s^{-1}} \left(d(a,\bar{a}) + d(b,\bar{b})\right) &\leq (1+\varepsilon) \cdot d(a,b) \cdot \left(1 + \frac{4}{(1-4s^{-1}) \cdot s}\right) \\ &= (1+\varepsilon) \cdot d(a,b) \cdot \frac{s}{s-4} \\ &= \frac{1+\varepsilon}{1-4s^{-1}} \cdot d(a,b) \qquad \square\end{aligned}$$

Sei nun $G = (S,E)$ ein euklidischer Graph mit einer Kantenmenge E, welche für jedes Paar $\{A_i, B_i\}$ der Realisation $\mathcal{R}$ eine Kante $\{a_i, b_i\}$ mit $a_i \in A_i$ und $b_i \in B_i$ enthält. Für $s = c\varepsilon^{-1}$ mit $c \geq 8 + 4\varepsilon$ ist G dann ein $(1+\varepsilon)$-Spanner von S:

5.2.3 Theorem ([25, 26]). *Seien $\varepsilon > 0$ und $c > 0$ zwei reelle positive Konstanten mit $c \geq 8 + 4\varepsilon$, S eine endliche Punktmenge aus dem $\mathbb{R}^d$, $\mathcal{R} = \{\{A_1, B_1\}, \ldots, \{A_k, B_k\}\}$ eine bzgl. $s = c\varepsilon^{-1}$ scharf getrennte Realisation von $S \otimes S$ und $G = (S,E)$ ein euklidischer Graph mit einer Kantenmenge E, die für jedes Paar $\{A_i, B_i\} \in \mathcal{R}$ eine Kante $\{a_i, b_i\}$ mit $a_i \in A_i$ und $b_i \in B_i$ enthält. Dann ist der Graph G ein $(1+\varepsilon)$-Spanner von S.*

Beweis. Nach Lemma 5.2.1 gilt $G \in \mathbf{G}(S, \mathcal{R}, c\varepsilon^{-1}, 4\varepsilon c^{-1})$. Zu einem beliebigen Paar $\{a,b\}$ mit $a,b \in S$ und $a \neq b$ existiert demnach aufgrund von Lemma 5.2.2 ein Weg in G von a nach b, dessen Länge höchstens

$$\frac{1 + 4c^{-1}\varepsilon}{1 - 4c^{-1}\varepsilon} \cdot d(a,b)$$

beträgt. Mit $c \geq 8 + 4\varepsilon$ folgt dann

$$\begin{aligned}\frac{1 + 4c^{-1}\varepsilon}{1 - 4c^{-1}\varepsilon} \cdot d(a,b) &\leq \left(1 + \frac{4}{8+4\varepsilon} \cdot \varepsilon\right) \cdot \left(1 - \frac{4}{8+4\varepsilon} \cdot \varepsilon\right)^{-1} \cdot d(a,b) \\ &= \left(1 + \frac{\varepsilon}{2+\varepsilon}\right) \cdot \left(1 - \frac{\varepsilon}{2+\varepsilon}\right)^{-1} \cdot d(a,b) \\ &= \frac{2+2\varepsilon}{2} \cdot d(a,b) \\ &= (1+\varepsilon) \cdot d(a,b) \qquad \square\end{aligned}$$

Durch das obige Theorem ergibt sich ein allgemeines Verfahren zur Konstruktion eines $(1+\varepsilon)$-Spanners $G = (S, E)$ einer endlichen Punktmenge $S \subset \mathbb{R}^d$, wobei $\varepsilon > 0$ eine beliebige positive reelle Konstante ist. Dieses Verfahren besteht aus den beiden folgenden Schritten, vgl. [25, 26]:

(1) Konstruktion einer WSPD $\mathcal{D} = (T, \mathcal{R})$ für die Punktmenge S bzgl. $s = c\varepsilon^{-1}$, wobei $c \geq 8 + 4\varepsilon$ eine reelle Konstante ist.

(2) Für jedes Paar $\{A_i, B_i\}$ der Realisation $\mathcal{R} = \{\{A_1, B_1\}, \ldots, \{A_k, B_k\}\}$ muss eine Kante $\{a_i, b_i\}$ mit $a_i \in A_i$ und $b_i \in B_i$ der (zu Beginn leeren) Kantenmenge E hinzugefügt werden.

Zu beachten ist hierbei, dass durch diese Vorgehensweise zwar ein $(1+\varepsilon)$-Spanner $G = (S, E)$ von S konstruiert wird, dieser im Allgemeinen jedoch nicht dünn besetzt ist – die Anzahl der Kanten aus E ist also im Allgemeinen *nicht* linear in der Anzahl der Punkte aus S. Zur Konstruktion eines dünn besetzten $(1+\varepsilon)$-Spanners muss die Größe der im ersten Schritt berechneten WSPD linear in der Anzahl der Punkte sein, d.h. es muss $k \in \mathcal{O}(|S|)$ gelten. Die Existenz einer solchen WSPD linearer Größe und somit die Existenz eines dünn besetzten $(1+\varepsilon)$-Spanners von S ist aufgrund der Ergebnisse von Callahan und Kosaraju [25, 27] gesichert, vgl. Kapitel 4. Weiterhin ist die effiziente Berechnung einer entsprechenden WSPD linearer Größe und somit die effiziente Konstruktion eines dünn besetzten $(1+\varepsilon)$-Spanners von S zumindest im RAM-, I/O- und *cache-oblivious*-Modell nach Kapitel 4 möglich.

In den beiden folgenden Abschnitten werden die im RAM-, I/O- und *cache-oblivious*-Modell effizienten Algorithmen zur Konstruktion eines dünn besetzten $(1+\varepsilon)$-Spanners einer gegebenen endlichen Punktmenge $S \subset \mathbb{R}^d$ näher erläutert.

5.3 Konstruktionsverfahren im RAM- und I/O-Modell

Es werden nun die beiden im RAM- bzw. I/O-Modell effizienten Algorithmen besprochen, welche sich direkt aus dem obigen allgemeinen Konstruktionsverfahren ergeben. Der im RAM-Modell effiziente Algorithmus zur Konstruktion eines dünn besetzten $(1+\varepsilon)$-Spanners einer endlichen Punktmenge aus dem $\mathbb{R}^d$ basiert, wie auch das obige allgemeine Vorgehen, auf den Arbeiten von Callahan und Kosaraju [25, 26]. Der entsprechende im Kontext des I/O-Modells effiziente Konstruktionsalgorithmus stammt von Govindarajan *et al.* [40].

Im ersten Teil dieses Abschnitts werden der RAM- und der I/O-Algorithmus erläutert. Dabei wird nur kurz auf den I/O-Algorithmus eingegangen wird, da dieser die (identisch übernommene) Vorlage für den im nächsten Abschnitt ausführlich diskutierten *cache-oblivious*-Konstruktionsalgorithmus bildet. Im zweiten Teil dieses Abschnitts werden untere Schranken für die Komplexität der Konstruktion von $(1+\varepsilon)$-Spannern im RAM- bzw. I/O-Modell angegeben, durch welche die Optimalität beider Algorithmen gezeigt wird.

Funktion ConstructSpanner-RAM(S, ε)

Eingabe: Eine endliche Punktmenge S aus dem $\mathbb{R}^d$ und eine reelle Konstante $\varepsilon > 0$.

Ausgabe: Ein $(1+\varepsilon)$-Spanner $G = (S, E)$ von S mit $|E| \in \mathcal{O}(|S|)$.

1: $E = \emptyset$
2: Berechne mit Hilfe des Algorithmus aus Abschnitt 4.2.1 eine WSPD $\mathcal{D} = (T, \{\{A_1, B_1\}, \ldots, \{A_k, B_k\}\})$ von S bzgl. des Wertes $s = c\varepsilon^{-1}$, wobei $c \geq 8 + 4\varepsilon$ eine reelle Konstante ist.
3: **for** $i = 1$ to k **do**
4: Sei $\{a_i, b_i\}$ ein beliebiges Paar mit $a_i \in A_i$ und $b_i \in B_i$.
5: $E = E \cup \{a_i, b_i\}$
6: **end for**
7: **return** $G = (S, E)$

Algorithmus 5.1: Konstruktionsalgorithmus im RAM-Modell, vgl. [25, 26]

5.3.1 Algorithmen

Die Funktion ConstructSpanner-RAM (Algorithmus 5.1) berechnet für eine endliche Punktmenge $S \subset \mathbb{R}^d$ einen $(1+\varepsilon)$-Spanner $G = (S, E)$ von S mit $|E| \in \mathcal{O}(|S|)$. Die Analyse dieser Funktion im RAM-Modell wird durch das folgende Theorem gegeben:

5.3.1 Theorem ([25, 26]). *Sei eine endliche Punktmenge $S \subset \mathbb{R}^d$ und eine reelle Konstante $\varepsilon > 0$ gegeben. Dann kann mit Hilfe der Funktion* ConstructSpanner-RAM *(Algorithmus 5.1) in einer Laufzeit von $\mathcal{O}(|S| \log |S| + \varepsilon^{-d}|S|) = \mathcal{O}(|S| \log |S|)$ ein $(1+\varepsilon)$-Spanner $G = (S, E)$ von S mit $|E| \in \mathcal{O}(\varepsilon^{-d} \cdot |S|) = \mathcal{O}(|S|)$ konstruiert werden.*

Beweis. Die Berechnung der WSPD von S bzgl. $s = c\varepsilon^{-1}$ in Zeile 2 der Funktion ConstructSpanner-RAM kann mit Hilfe des Algorithmus aus Abschnitt 4.2 in einer Laufzeit von $\mathcal{O}(|S| \log |S| + s^d \cdot |S|) = \mathcal{O}(|S| \log |S| + \varepsilon^{-d} \cdot |S|)$ vollzogen werden. Für die Größe k der WSPD gilt dabei $k \in \mathcal{O}(\varepsilon^{-d} \cdot |S|)$, d.h. die Größe der WSPD ist linear in der Anzahl der Punkte von S. Ein Paar $\{A_i, B_i\}$ der Realisation wird weiterhin durch ein Paar $\{v_i, w_i\}$ von Knoten des zugehörigen fairen Aufteilungsbaums T dargestellt, vgl. Abschnitt 4.2.

Um die beiden Repräsentanten $a_i \in A_i$ und $b_i \in B_i$ in Zeile 4 effizient auswählen zu können, kann bei der Konstruktion von T jeweils für jeden Knoten $v \in T$ mit $\sigma(v) = A \subseteq S$ ein Punkt $a \in A$ in v explizit gespeichert werden. Der Speicherplatzbedarf eines auf diese Weise modifizierten fairen Aufteilungsbaums sowie die Laufzeit zur Konstruktion eines solchen verändert sich dadurch asymptotisch gesehen nicht. Die Erstellung der Kantenmenge E (Zeilen $3-6$) kann somit durch eine Traversierung der die Realisation darstellenden Liste $\{\{v_1, w_1\}, \ldots, \{v_k, w_k\}\}$ umgesetzt werden, welche insgesamt in einer Laufzeit von $\mathcal{O}(k) \subseteq \mathcal{O}(\varepsilon^{-d} \cdot |S|)$ möglich ist. Der Algorithmus besitzt somit eine Gesamtlaufzeit von

$\mathcal{O}(|S| \log |S| + \varepsilon^{-d}|S|)$. Die Korrektheit des Algorithmus ergibt sich aus Theorem 5.2.3 und der linearen Größe der berechneten WSPD. □

Wie Callahan und Kosaraju bemerken, kann die Anzahl der Kanten eines durch die Funktion ConstructSpanner-RAM konstruierten $(1+\varepsilon)$-Spanners um den Faktor $\frac{1}{\varepsilon}$ verringert werden, wodurch sich die Laufzeit allerdings um den Faktor $\log \frac{1}{\varepsilon}$ vergrößert. Auf diese und weitere Verfeinerungen soll an dieser Stelle jedoch nicht weiter eingegangen werden.

Der im I/O-Modell effiziente Konstruktionsalgorithmus von Govindarajan *et al.* [40] berechnet ebenso zunächst eine WSPD bzgl. $s = c\varepsilon^{-1}$ mit $c \geq 8+4\varepsilon$ und wählt anschließend für jedes Paar $\{A_i, B_i\}$ der WSPD eine Kante $\{a_i, b_i\}$ mit $a_i \in A_i$ und $b_i \in B_i$ aus. Um die Repräsentanten a_i und b_i auszuwählen, wenden Govindarajan *et al.* die *time-forward processing*-Technik (siehe Kapitel 4) an. Da diese Vorgehensweise von dem im nächsten Abschnitt besprochenen *cache-oblivious*-Algorithmus exakt übernommen wird, sei hier nur das Gesamtergebnis im I/O-Modell gegeben und auf weitere Ausführungen verzichtet:

5.3.2 Theorem ([40]). *Sei eine endliche Punktmenge $S \subset \mathbb{R}^d$ und eine reelle Konstante $\varepsilon > 0$ gegeben. Dann kann ein $(1+\varepsilon)$-Spanner $G = (S, E)$ von S mit $|E| \in \mathcal{O}(|S|)$ unter Verwendung von $\mathcal{O}(sort(|S|))$ I/O-Operationen berechnet werden.*

5.3.2 Untere Schranken

Die Optimalität der gerade genannten Verfahren ergibt sich mittels der beiden folgenden Theoreme. Kernidee der Beweise dieser Theoreme ist die Transformation des Elementeindeutigkeitsproblems [52] auf das Problem der Konstruktion eines $(1+\varepsilon)$-Spanners für die entsprechende Punktmenge, wodurch sich jeweils die für das Elementeindeutigkeitsproblem gezeigten unteren Schranken [10, 52] auf die beiden Konstruktionsprobleme übertragen.

5.3.3 Theorem ([28]). *Sei eine endliche nicht-leere Punktmenge $S \subset \mathbb{R}^d$ und eine reelle Konstante $\varepsilon > 0$ gegeben. Dann benötigt die Berechnung eines beliebigen $(1+\varepsilon)$-Spanners von S im algebraischen Entscheidungsbaummodell mindestens $\Omega(|S| \log |S|)$ Zeit.*

Auf den Beweis dieses Theorems wird hier verzichtet. Exemplarisch wird stattdessen der Beweis des entsprechenden Ergebnisses im I/O-Modell durchgeführt:

5.3.4 Theorem ([40]). *Sei eine endliche Punktmenge $S \subset \mathbb{R}^d$ und eine reelle Konstante $\varepsilon > 0$ gegeben. Dann werden mindestens $\Omega(sort(|S|))$ I/O-Operationen zur Berechnung eines $(1+\varepsilon)$-Spanners $G = (S, E)$ von S mit $|E| \in \mathcal{O}(|S|)$ benötigt.*

Beweis. Sei $G = (S, E)$ ein $(1+\varepsilon)$-Spanner von $S = \{p_1, \ldots, p_N\}$ mit $|E| \in \mathcal{O}(|S|)$. Dieser müsste für jedes Paar p_i, p_j mit $p_i = p_j$ und $i \neq j$ die Kante $\{p_i, p_j\}$ enthalten. Mittels einer Traversierung der Kantenmenge E von G könnte somit mit $\mathcal{O}(scan(|S|))$ I/O-Operationen entschieden werden, ob alle Punkte in S paarweise verschieden sind.

Könnte nun ein solcher Graph in $o(sort(|S|))$ I/O-Operationen berechnet werden, so wäre das Elementeindeutigkeitsproblem unter Verwendung von $o(sort(|S|))$ I/O-Operationen lösbar, was jedoch der für dieses Problem gezeigten unteren Schranke von $\Omega(sort(|S|))$ I/O-Operationen [10] widerspricht. □

Zu beachten ist noch folgender Sachverhalt: Bei der Bearbeitung von Punktmengen aus dem $\mathbb{R}^d$ wird in der algorithmischen Geometrie oft die Annahme getroffen, dass alle Punkte paarweise verschieden sind. Für diese Art von Punktmengen funktioniert der obige Beweis nicht. Diese Fälle werden deshalb von Chen *et al.* [28] und Govindarajan *et al.* [40] gesondert betrachtet.

Bei den Konstruktionsalgorithmen im RAM- und I/O-Modell handelt es sich also um asymptotisch optimale Algorithmen. Die untere Schranke für das I/O-Modell überträgt sich weiterhin direkt auf das *cache-oblivious*-Modell.

5.4 Konstruktionsverfahren im *Cache-Oblivious*-Modell

Der I/O-effiziente Algorithmus von Govindarajan *et al.* [40] lässt sich direkt in das *cache-oblivious*-Modell übertragen:

5.4.1 Theorem. *Sei eine endliche Punktmenge $S \subset \mathbb{R}^d$ und eine reelle Konstante $\varepsilon > 0$ gegeben. Dann kann ein $(1+\varepsilon)$-Spanner $G = (S, E)$ von S mit $|E| \in \mathcal{O}(|S|)$ unter Verwendung von $\mathcal{O}(sort(|S|))$ Speichertransfers berechnet werden.*

Beweis. Analog zu Govindarajan *et al.* bzw. in Anlehnung an das allgemeine Verfahren wird für die Konstruktion von G zunächst eine WSPD $\mathcal{D} = (T, \{\{A_1, B_1\}, \ldots, \{A_k, B_k\}\})$ von S bzgl. $s = c\varepsilon^{-1}$ mit $c \geq 8 + 4\varepsilon$ berechnet. Die Realisation wird dabei durch eine Liste $\mathcal{R}_T = \{\{v_1, w_1\}, \ldots, \{v_k, w_k\}\}$ von Knotenpaaren mit Knoten $v_i, w_i \in T$ dargestellt, vgl. Kapitel 4. Anschließend wird für jedes Knotenpaar $\{v_i, w_i\} \in \mathcal{R}_T$ eine Kante $\{r(v_i), r(w_i)\}$ mit $r(v_i) \in A_i$ und $r(w_i) \in B_i$ ausgewählt und der anfangs leeren Kantenmenge E hinzugefügt.

Die Berechnung der WSPD verursacht nach Abschnitt 4.3 höchstens $\mathcal{O}(sort(|S|))$ Speichertransfers. Dabei ist die Größe k der WSPD linear in der Anzahl der Punkte aus S. Die Auswahl einer Kante $\{r(v_i), r(w_i)\}$ für jedes Knotenpaar $\{v_i, w_i\} \in \mathcal{R}_T$ gelingt wie auch im I/O-Modell mit Hilfe der *time-forward processing*-Technik: Die Knoten des fairen Aufteilungsbaums T können dazu, wie am Ende des Kapitels 4 beschrieben, topologisch sortiert und anschließend von den Blättern aus zur Wurzel hin durchlaufen werden. Im Zuge dieser Traversierung kann für jeden Knoten v von T ein Repräsentant $r(v) \in \sigma(v)$ ausgewählt und im Knoten v gespeichert werden. Ein Blatt $v \in T$ mit $\sigma(v) = p \in S$ erhält dabei den schon in v gespeicherten Punkt p und jeder innere Knoten $v \in T$ einen der beiden Repräsentanten seiner Kinder als Repräsentant $r(v)$. Die Auswahl aller Repräsentanten kann somit mit $\mathcal{O}(sort(|S|))$ realisiert werden.

Um die Kantenmenge E zu erstellen, wird zunächst durch eine Traversierung der Knotenmenge von T eine Liste L erstellt, welche für jeden Knoten $v \in T$ einen Eintrag der Form $(v, r(v))$ enthält. Die Liste L sowie die Liste $\mathcal{R}_T$ der Knotenpaare werden anschließend jeweils bzgl. der ersten Komponente sortiert und nach dieser Sortierung simultan durchlaufen. Bei diesem Durchlauf können die ersten Komponenten v_i aller Listeneinträge von $\mathcal{R}_T$ durch die Repräsentanten $r(v_i)$ ersetzt werden. Entsprechend können die zweiten Komponenten w_i von $\mathcal{R}_T$ durch die Repräsentanten $r(w_i)$ ersetzt werden. Die Kantenmenge E besteht nach Ausführung dieser Schritte aus den Einträgen der Liste $\mathcal{R}_T$. Insgesamt werden durch diesen Vorgang $\mathcal{O}(sort(|S|))$ Speichertransfers verursacht. □

Kapitel 6

Ausdünnung dichter Spanner-Graphen

Im vorherigen Kapitel wurde die Konstruktion von dünn besetzten $(1+\varepsilon)$-Spannern für endliche Punktmengen aus dem $\mathbb{R}^d$ behandelt und gezeigt, wie diese im Kontext des RAM-, I/O- bzw. *cache-oblivious*-Modell effizient berechnet werden können. Dieses Kapitel befasst sich mit einem mit der Konstruktion solcher Spanner-Graphen verwandten Problem – der sogenannten „Ausdünnung dichter Spanner-Graphen“.

Das Kapitel ist wie folgt aufgebaut: In Abschnitt 6.1 wird der Begriff des „Ausdünnens“ von Spanner-Graphen präzisiert. In Abschnitt 6.2 wird ein allgemeines von Gudmundsson *et al.* [44] vorgestelltes Verfahren beschrieben, mit dessen Hilfe ein t-Spanner ($t \geq 1$) ausgedünnt werden kann. Dieses allgemeine Verfahren führt mit Hilfe der effizienten Berechnung der in Kapitel 4 vorgestellten WSPD-Datenstruktur unmittelbar zu effizienten Algorithmen im RAM- und I/O-Modell. Diese beiden Ausdünnungsalgorithmen werden in Abschnitt 6.3 kurz erläutert. In Abschnitt 6.4 wird der I/O-effiziente Ausdünnungsalgorithmus in das *cache-oblivious*-Modell übertragen. Diese Übertragung ist das Kernanliegen des Kapitels.

6.1 Vorbemerkungen

Ein Teilgraph G' eines gegebenen euklidischen Graphen $G = (S, E)$ mit konstanter Dilatation $t \geq 1$ ist ein t'-*Spanner* von G, wenn

$$\delta_{G'}(p, q) \leq t' \cdot \delta_G(p, q)$$

für alle Punkte $p, q \in S$ gilt. Ein dünn besetzter t'-Spanner eines euklidischen Graphen $G = (S, E)$ mit konstanter Dilatation $t \geq 1$ ist demnach ein t'-Spanner $G' = (S, E')$ von G mit $|E'| \in \mathcal{O}(|S|)$.

Das Ziel der in Kapitel 5 vorgestellten Algorithmen war jeweils die Konstruktion eines dünn besetzten $(1+\varepsilon)$-Spanners für eine endliche Punktmenge S aus dem $\mathbb{R}^d$ und eine beliebige reelle Konstante $\varepsilon > 0$. Die mittels dieser Konstruktionsalgorithmen berechneten Spanner-Graphen besitzen jeweils eine in der Anzahl der Ecken lineare Anzahl von Kanten. Allgemein kann ein (beliebiger) Spanner-Graph jedoch superlinear viele Kanten besitzen. Der vollständige euklidische Graph einer endlichen Punktmenge besitzt z.B.

$$\binom{|S|}{2} \in \mathcal{O}(|S|^2)$$

Kanten. Ein euklidischer Graph mit einer superlinearen Anzahl von Kanten wird im Folgenden als *dicht* bezeichnet. Das Ziel der in diesem Kapitel vorgestellten Algorithmen ist es, einen gegebenen (dichten) euklidischen Graphen $G = (S, E)$ mit konstanter Dilatation $t \geq 1$ *auszudünnen*, d.h. einen dünn besetzten $(1+\varepsilon)$-Spanner $G' = (S, E')$ von G zu konstruieren. Aufgrund der konstanten Dilatation des ursprünglichen Graphen ist ein solcher dünn besetzter Teilgraph G' insgesamt ein $(t(1+\varepsilon))$-Spanner der Eckenmenge S.

Der *greedy*-Algorithmus [33, 42] kann zur Ausdünnung eines gegebenen (dichten) t-Spanners $G = (S, E)$ genutzt werden. Wie Gudmundsson *et al.* [44] bemerken, besitzt dieser jedoch im RAM-Modell eine Laufzeit von $\Omega(|E| \log |S|)$. Ein im Kontext des RAM-Modells effizienterer Algorithmus zur Ausdünnung eines solchen t-Spanners ist der von Gudmundsson *et al.* [44] vorgestellte Ausdünnungsalgorithmus, welcher einen entsprechenden dünn besetzten Teilgraphen in einer Laufzeit von $\mathcal{O}(|E| + |S| \log |S|)$ berechnet. Kernkomponente dieses Algorithmus ist die in Kapitel 4 vorgestellte WSPD-Datenstruktur. Weiterhin bildet dieser Algorithmus die Vorlage für den im I/O-Modell effizienten Algorithmus von Gudmundsson und Vahrenhold [45], welcher für die Berechnung eines dünn besetzten $(1+\varepsilon)$-Spanners G' von G insgesamt $\mathcal{O}(sort(|E|))$ I/O-Operationen benötigt.

Im Rest dieses Kapitels wird dieses I/O-effiziente Verfahren in das *cache-oblivious*-Modell übertragen. Dazu wird im nächsten Abschnitt zunächst ein allgemeines auf den Arbeiten von Gudmundsson *et al.* [44] basierendes Verfahren zur Ausdünnung eines t-Spanners gegeben, welches unmittelbar zu den im RAM-, I/O- und *cache-oblivious*-Modell effizienten Algorithmen führt.

6.2 Allgemeines Verfahren

Zur Beschreibung des allgemeinen Verfahrens sei eine endliche Punktmenge $S \subset \mathbb{R}^d$, ein t-Spanner $G = (S, E)$ mit $t \geq 1$ und eine reelle Konstante $\varepsilon > 0$ gegeben. Das Ziel des Verfahrens besteht in der Ausdünnung des Graphen G, d.h. in der Konstruktion eines $(1+\varepsilon)$-Spanners $G' = (S, E')$ von G mit $|E'| \in \mathcal{O}(|S|)$.

Die grundlegende Idee des Verfahrens ist die Konstruktion einer *Approximation* der Kantenmenge E von G. Eine Approximation von E bzgl. eines Wertes $s \in \mathbb{R}^+$ ist eine

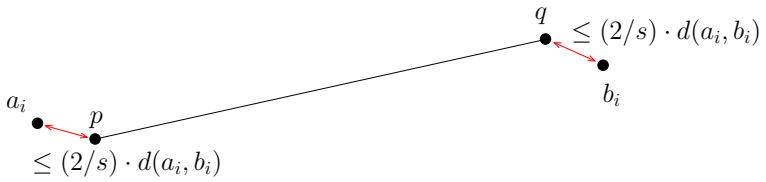


Abbildung 6.1: „Approximatives“ Paar $\{a_i, b_i\}$ für eine Kante $\{p, q\} \in E$, vgl. [44]

Menge $P = \{\{a_1, b_1\}, \ldots, \{a_m, b_m\}\}$ von $m \geq 1$ Paaren mit $a_j, b_j \in \mathbb{R}^d, 1 \leq j \leq m$, so dass für jede Kante $\{p, q\}$ aus E ein Index $i \in \{1, \ldots, m\}$ existiert mit

(i) $d(p, a_i) \leq (2/s) \cdot d(a_i, b_i)$ und $d(q, b_i) \leq (2/s) \cdot d(a_i, b_i)$ oder

(ii) $d(p, b_i) \leq (2/s) \cdot d(a_i, b_i)$ und $d(q, a_i) \leq (2/s) \cdot d(a_i, b_i)$.

Durch die beiden Bedingungen (i) und (ii) wird also gefordert, dass es zu jeder Kante $\{p, q\}$ aus E ein „approximatives“ Paar $\{a_i, b_i\}$ in P gibt, vgl. Abbildung 6.1.

Seien nun eine solche Approximation $P = \{\{a_1, b_1\}, \ldots, \{a_m, b_m\}\}$ $(m \geq 1)$ von E bzgl. eines Wertes $s \in \mathbb{R}^+$ und m anfangs leere Listen $C_1, \ldots, C_m$ gegeben. Aufgrund der Eigenschaften der Approximation P existiert für jede Kante $\{p, q\} \in E$ ein Index $i \in \{1, \ldots, m\}$, so dass für diesen eine der beiden obigen Bedingungen erfüllt ist. Für jede Kante aus E sei ein solcher Index i fest gewählt und die Kante der Liste C_i hinzugefügt. Mit Hilfe dieser Listen kann anschließend der Graph $G' = (S, E')$ erstellt werden, dessen Kantenmenge E' genau eine Kante aus jeder der *nicht-leeren* Listen $C_1, \ldots, C_m$ enthält. Bei Wahl von $s = \frac{1}{\varepsilon}((1 + \varepsilon)(8t + 4) + 4)$ folgt, dass G' ein $(1 + \varepsilon)$-Spanner von G ist:

6.2.1 Lemma ([44]). *Gilt $s = \frac{1}{\varepsilon}((1 + \varepsilon)(8t + 4) + 4)$, so ist der obige Graph $G' = (S, E')$ ein $(1 + \varepsilon)$-Spanner von G.*

Beweis. Es ist zu zeigen, dass es für jede Kante $\{p, q\}$ aus E einen Weg in G' von p nach q gibt, dessen Länge höchstens $(1 + \varepsilon) \cdot d(p, q)$ beträgt. Gudmundsson *et al.* beweisen dies induktiv über die Länge der Kanten in E:

I.A.: Sei $\{p, q\}$ eine Kante in E mit minimaler Länge. Für diese Kante existiert ein Index $i \in \{1, \ldots, m\}$ mit $\{p, q\} \in C_i$. Weiterhin ist für diese Kante und den Index i eine der beiden obigen Bedingungen erfüllt. O. B. d. A. kann angenommen werden, dass die Bedingung (i) erfüllt ist und somit $d(p, a_i) \leq (2/s) \cdot d(a_i, b_i)$ und $d(q, b_i) \leq (2/s) \cdot d(a_i, b_i)$ gilt. Die Kantenmenge E' von G' enthält genau eine Kante $\{x, y\}$ aus der nicht-leeren Liste C_i. Für diese gelte wiederum o. B. d. A. die Bedingung (i) (ansonsten können die Rollen von x und y vertauscht werden). Da G ein t-Spanner ist, existiert ein Weg zwischen p und x und es gilt

$$\delta_G(p, x) \leq t \cdot d(p, x) \leq t \cdot (d(p, a_i) + d(a_i, x)) \leq (4t/s) \cdot d(a_i, b_i).$$

Weiterhin folgt aus Bedingung (i)

$$d(a_i, b_i) \leq d(a_i, p) + d(p, q) + d(q, b_i) \leq (4/s) \cdot d(a_i, b_i) + d(p, q),$$

was äquivalent zu

$$d(a_i, b_i) \leq \frac{s}{s-4} \cdot d(p, q)$$

ist. Nach Wahl von s gilt $s = 4 \cdot \left(\frac{(1+\varepsilon)}{\varepsilon}(2t+1) + \frac{1}{\varepsilon} \right) > 4(t+1)$ und somit

$$\delta_G(p, x) \leq \frac{4t}{s} \frac{s}{s-4} \cdot d(p, q) = \frac{4t}{s-4} \cdot d(p, q) < d(p, q).$$

Die Länge einer jeden Kante auf dem Weg von p nach x ist also echt kleiner als $d(p, q)$. Da $d(p, q)$ nach Voraussetzung minimal ist, muss demnach $p = x$ gelten. Analog kann $q = y$ gezeigt werden. Die Kante $\{p, q\}$ muss also in der Kantenmenge E' enthalten sein; es gilt somit $\delta_{G'}(p, q) = d(p, q) \leq (1 + \varepsilon) \cdot d(p, q)$.

I.S.: Sei nun $\{p, q\}$ eine beliebige Kante aus E. Induktiv kann angenommen werden, dass $\delta_{G'}(u, v) \leq (1+\varepsilon) \cdot d(u, v)$ für alle Kanten $\{u, v\} \in E$ mit $d(u, v) < d(p, q)$ gilt. Analog zum obigen Vorgehen gibt es für die Kante $\{p, q\}$ einen Index i mit $\{p, q\} \in C_i$ und eine Kante $\{x, y\} \in C_i$ mit $\{x, y\} \in E'$. Für beide Kanten gelte wieder o. B. d. A. die Bedingung (i). Erneut folgt

$$\delta_G(p, x) \leq \frac{4t}{s-4} \cdot d(p, q) < d(p, q).$$

Alle Kanten auf dem Weg von p nach x in G besitzen also eine Länge, die echt kleiner als $d(p, q)$ ist. Induktiv ergibt sich also

$$\delta_{G'}(p, x) \leq (1 + \varepsilon) \cdot \delta_G(p, x) \leq (1 + \varepsilon) \frac{4t}{s-4} \cdot d(p, q).$$

Entsprechend kann

$$\delta_{G'}(q, y) \leq (1 + \varepsilon) \cdot \delta_G(q, y) \leq (1 + \varepsilon) \frac{4t}{s-4} \cdot d(p, q)$$

gezeigt werden. Insgesamt existiert demnach in G' ein Weg von p nach q mit

$$\delta_{G'}(p, q) \leq \delta_{G'}(p, x) + d(x, y) + \delta_{G'}(y, q) \leq (1 + \varepsilon) \frac{8t}{s-4} \cdot d(p, q) + d(x, y),$$

vgl. Abbildung 6.2. Für die Länge der Kante $\{x, y\}$ folgt unter Verwendung von Bedingung (i):

$$\begin{aligned} d(x, y) &\leq d(x, a_i) + d(a_i, b_i) + d(b_i, y) \\ &\leq (1 + 4/s) \cdot d(a_i, b_i) \\ &\leq (1 + 4/s) \cdot \frac{s}{s-4} \cdot d(p, q) \\ &= \frac{s+4}{s-4} \cdot d(p, q) \end{aligned}$$

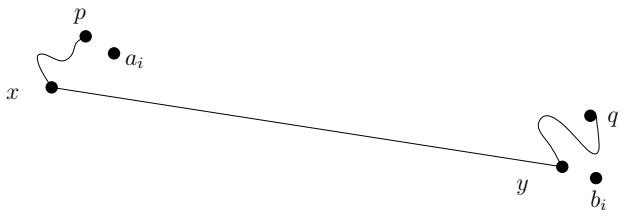


Abbildung 6.2: Der Weg von p nach q in G', vgl. [44]

Die Länge des Weges von p nach q in G' lässt sich somit durch

$$\begin{aligned}\delta_{G'}(p,q) &\leq (1+\varepsilon)\frac{8t}{s-4}\cdot d(p,q) + \frac{s+4}{s-4}\cdot d(p,q)\\ &= (1+\varepsilon)\cdot d(p,q)\cdot\left(\frac{8t}{s-4}+\frac{s+4}{(s-4)(1+\varepsilon)}\right)\end{aligned}$$

abschätzen. Da

$$\begin{aligned} & s && = \frac{1}{\varepsilon}((1+\varepsilon)(8t+4)+4)\\ \Leftrightarrow\quad & (1+\varepsilon)8t && = \varepsilon(s-4)-8\\ \Leftrightarrow\quad & (1+\varepsilon)8t+(s+4) && = (s-4)(1+\varepsilon)\\ \Leftrightarrow\quad & 1 && = \frac{8t}{s-4}+\frac{s+4}{(s-4)(1+\varepsilon)}\end{aligned}$$

gilt, folgt nach Wahl von s die Behauptung. □

Wie Gudmundsson *et al.* bemerken, wird durch die Konstruktion von G' jede Kante aus E einem Paar aus P zugeordnet. Für jedes Paar $\{a_i, b_i\}$, welchem mindestens eine Kante zugeordnet wurde, enthält der Graph G' genau eine dieser Kanten. Aufgrund des obigen Lemmas ist der Graph G' ein $(1+\varepsilon)$-Spanner von G. Ein auf diese Weise konstruierter $(1+\varepsilon)$-Spanner von G kann jedoch dicht sein. Um mit Hilfe dieses Ansatzes einen dünn besetzten Spanner-Graphen zu erhalten, muss demnach zusätzlich $m \in \mathcal{O}(|S|)$ gelten.

Das allgemeine Verfahren zur Ausdünnung eines t-Spanners $G = (S, E)$ setzt sich also aus den folgenden Schritten zusammen, vgl. Gudmundsson *et al.* [44]:

(1) Berechnung einer Approximation $P = \{\{a_1, b_1\}, \ldots, \{a_m, b_m\}\}$ von E bzgl. des Wertes $s = \frac{1}{\varepsilon}((1+\varepsilon)(8t+4)+4)$ mit $m \in \mathcal{O}(|S|)$.

(2) Berechnung eines Indexes $i \in \{1, \ldots, m\}$ für jede Kante $\{p, q\} \in E$, so dass die Bedingung (i) oder (ii) erfüllt ist.

(3) Auswahl einer Kante für jeden Index $i \in \{1, \ldots, m\}$, dem in Schritt (2) mindestens eine Kante zugeordnet wurde.

Nach Kapitel 4 existiert zu der Eckenmenge S von G eine Realisation $\{\{A_1, B_1\}, \ldots, \{A_m, B_m\}\}$ von $S \otimes S$ mit $m \in \mathcal{O}(|S|)$. Die Existenz einer Approximation P von E bzgl. $s = \frac{1}{\varepsilon}((1+\varepsilon)(8t+4)+4)$ mit $m \in \mathcal{O}(|S|)$ und somit die Existenz eines dünn besetzten $(1+\varepsilon)$-Spanners von G ergibt sich deshalb aus dem folgenden Lemma:

6.2.2 Lemma. *Sei eine endliche Punktmenge $S \subset \mathbb{R}^d$, ein euklidischer Graph $G = (S, E)$, eine bzgl. eines Wertes $s \in \mathbb{R}^+$ scharf getrennte Realisation $\mathcal{R} = \{\{A_1, B_1\}, \ldots, \{A_m, B_m\}\}$ von $S \otimes S$ und eine Menge $P = \{\{a_1, b_1\}, \ldots \{a_m, b_m\}\}$, welche für jedes Paar $\{A_i, B_i\}$ der Realisation genau ein Paar $\{a_i, b_i\}$ mit $a_i \in A_i$ und $b_i \in B_i$ enthält, gegeben. Dann ist die Menge P eine Approximation von E bzgl. s.*

Beweis. Aufgrund der Eigenschaften der Realisation $\mathcal{R}$ gibt es zu jeder Kante $\{p, q\} \in E$ ein bzgl. s scharf getrenntes Paar $\{A_i, B_i\} \in \mathcal{R}$ mit $\{p, q\} \in A_i \otimes B_i$. Da A_i und B_i scharf getrennt sind, existieren weiterhin zwei d-Kugeln K_1 und K_2 mit Radius $r \geq 0$, so dass $A_i \subset K_1$, $B_i \subset K_2$ und $d(K_1, K_2) \geq sr$ gilt. Somit folgt für die beiden Repräsentanten a_i und b_i entweder

$$\begin{aligned} d(p, a_i) &\leq 2r = (2/s)sr \leq (2/s) \cdot d(a_i, b_i) \quad \text{und} \\ d(q, b_i) &\leq 2r = (2/s)sr \leq (2/s) \cdot d(a_i, b_i) \end{aligned}$$

oder

$$\begin{aligned} d(p, b_i) &\leq 2r = (2/s)sr \leq (2/s) \cdot d(a_i, b_i) \quad \text{und} \\ d(q, a_i) &\leq 2r = (2/s)sr \leq (2/s) \cdot d(a_i, b_i). \end{aligned}$$

□

Die WSPD kann also, *muss* jedoch *nicht* zur Konstruktion einer Approximation P der Kantenmenge E des gegebenen Graphen $G = (S, E)$ verwendet werden. In den folgenden beiden Abschnitten wird nun gezeigt, wie diese allgemeine Vorgehensweise effizient im RAM-, I/O- bzw. *cache-oblivious*-Modell umgesetzt werden kann.

6.3 Ausdünnungsverfahren im RAM- und I/O-Modell

Das allgemeine Verfahren führt mit Hilfe der effizienten Berechnung einer WSPD direkt zu einem effizienten Ausdünnungsalgorithmus im RAM-Modell, welcher nun beschrieben und analysiert werden soll.

Die Prozedur PruneSpanner-RAM (Algorithmus 6.1) berechnet zunächst in Zeile 2 mit Hilfe des Algorithmus von Callahan und Kosaraju (Abschnitt 4.2.1) in einer Laufzeit von $\mathcal{O}(|S| \log |S| + s^d|S|)$ eine WSPD von S bzgl. $s = \frac{1}{\varepsilon}((1+\varepsilon)(8t+4)+4)$. Die Größe dieser WSPD ist linear in der Anzahl der Punkte aus S, d. h. es gilt $m \in \mathcal{O}(s^d|S|) = \mathcal{O}(|S|)$.[1] Um eine Approximation der Kantenmenge E bzgl. des Wertes s zu konstruieren, wird

[1] Nach Wahl von s geht der Parameter ε in die „versteckten“ Konstanten beider $\mathcal{O}$-Notationen ein.

Funktion PRUNESPANNER-RAM(G, ε)

Eingabe: Ein t-Spanner $G = (S, E)$ $(t \geq 1)$ mit endlicher Eckenmenge $S \subset \mathbb{R}^d$ und eine reelle Konstante $\varepsilon > 0$.

Ausgabe: Ein $(1+\varepsilon)$-Spanner $G' = (S, E')$ von G mit $|E'| \in \mathcal{O}(|S|)$.

1: $P = \emptyset, E' = \emptyset$

2: Berechne mit Hilfe des Algorithmus aus Abschnitt 4.2.1 eine WSPD $\mathcal{D} = (T, \{\{A_1, B_1\}, \ldots, \{A_m, B_m\}\})$ von S bzgl. des Wertes $s = \frac{1}{\varepsilon}((1+\varepsilon)(8t+4)+4)$.

3: Wähle für jedes Paar $\{A_i, B_i\}$ ein beliebiges Paar $\{a_i, b_i\}$ mit $a_i \in A_i$ und $b_i \in B_i$ aus und füge es der Menge P hinzu.

4: Markiere jede Kante $\{p, q\}$ aus E mit einem Index $i \in \{1, \ldots, m\}$, für den die Bedingung (i) oder (ii) erfüllt ist.

5: Traversiere die Kantenmenge E und füge für jeden auftretenden Index i eine der ihm insgesamt zugeordneten Kanten der Menge E' hinzu.

6: **return** $G' = (S, E')$

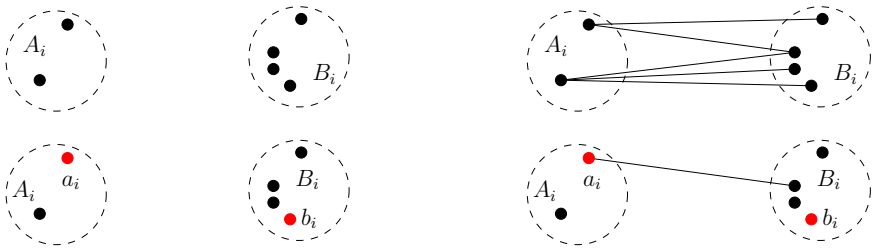
Algorithmus 6.1: Ausdünnungsalgorithmus im RAM-Modell, vgl. [43, 44]

anschließend in Zeile 3 für jedes scharf getrennte Paar $\{A_i, B_i\}$ der WSPD genau ein Paar $\{a_i, b_i\}$ mit $a_i \in A_i$ und $b_i \in B_i$ ausgewählt und der Menge P hinzugefügt, vgl. Abbildung 6.3 (a). Die so konstruierte Menge P stellt nach Lemma 6.2.2 eine Approximation von E bzgl. s dar. Der erste Schritt des allgemeinen Verfahrens lässt sich also im RAM-Modell in einer Laufzeit von $\mathcal{O}(|S| \log |S|)$ realisieren.

Zur Umsetzung des zweiten Schritts des allgemeinen Verfahrens stellen Gudmundsson *et al.* [43, 44] in Zeile 4 pro Kante aus E eine Anfrage an den fairen Aufteilungsbaum T, der zur berechneten WSPD gehört. Diese Anfragen sind mit Ergebnissen von Arya *et al.* [12] nach einer Vorverarbeitungsphase effizient möglich und benötigen *pro* Kante $\mathcal{O}(\log |S|)$ Zeit. Die Laufzeit der Vorverarbeitungsphase beträgt $\mathcal{O}(|S| \log |S|)$. Insgesamt ist die Zuordnung der Indizes in Zeile 4 somit in einer Laufzeit von $\mathcal{O}((|S| + |E|) \log |S|)$ möglich.

In Zeile 5 findet anschließend die Konstruktion der Kantenmenge E' des ausgedünnten Graphen G' statt: Für jeden Index $i \in \{1, \ldots, m\}$, dem mindestens eine Kante zugeordnet wurde, muss eine der ihm zugeordneten Kanten der Kantenmenge E' von G' hinzugefügt werden. Dies ist insgesamt in einer Laufzeit von $\mathcal{O}(|E| + m) = \mathcal{O}(|E| + |S|)$ realisierbar. In Abbildung 6.3 (b) wird die Auswahl einer solchen Kante schematisch dargestellt. Jede der dargestellten Kanten ist mit dem Index i markiert; die Kantenmenge von G' enthält jedoch nur eine dieser Kanten.

Insgesamt besitzt der obige Ausdünnungsalgorithmus somit eine Laufzeit von $\mathcal{O}((|S| + |E|) \log |S|) = \mathcal{O}(|E| \log |S|)$.



(a) Auswahl der Repräsentanten (b) Auswahl einer Kante

Abbildung 6.3: Ausdünnung des Graphen mit Hilfe der WSPD, vgl. [44]

6.3.1 Theorem ([44]). *Sei eine endliche Punktmenge $S \subset \mathbb{R}^d$ und ein t-Spanner $G = (S, E)$ von S mit $t \geq 1$ gegeben. Dann kann für jede reelle Konstante $\varepsilon > 0$ in einer Laufzeit von $\mathcal{O}(|E| \log |S|)$ ein $(1+\varepsilon)$-Spanner $G' = (S, E')$ von G mit $|E'| \in \mathcal{O}(|S|)$ berechnet werden.*

Gudmundsson *et al.* [44] zeigen zudem, wie die benötigte Laufzeit zur Konstruktion des Graphen G' zu einer Laufzeit von $\mathcal{O}(|E| + |S| \log |S|)$ verbessert werden kann. Auf diese Variante soll hier aber nicht näher eingegangen werden.

Im I/O-Modell lässt sich der obige allgemeine Ansatz ebenfalls effizient verwirklichen: Gudmundsson und Vahrenhold [45] zeigen anhand der Ergebnisse von Govindarajan *et al.* [40], wie ein ausgedünnter Graph G' von G mit $\mathcal{O}(sort(|E|))$ I/O-Operationen konstruiert werden kann. Da der im folgenden Abschnitt beschriebene *cache-oblivious*-Algorithmus fast ausschließlich mit dem I/O-effizienten Ausdünnungsalgorithmus von Gudmundsson und Vahrenhold übereinstimmt, wird hier auf weitere Erläuterungen verzichtet und nur das Ergebnis gegeben:

6.3.2 Theorem ([45]). *Sei eine endliche Punktmenge $S \subset \mathbb{R}^d$ und ein t-Spanner $G = (S, E)$ von S mit $t \geq 1$ gegeben. Dann kann für jede reelle Konstante $\varepsilon > 0$ mit $\mathcal{O}(sort(|E|))$ I/O-Operationen ein $(1+\varepsilon)$-Spanner $G' = (S, E')$ mit $|E'| \in \mathcal{O}(|S|)$ berechnet werden.*

Im folgenden Abschnitt wird das I/O-effiziente Verfahren in das *cache-oblivious*-Modell übertragen.

6.4 Ausdünnungsverfahren im *Cache-Oblivious*-Modell

Als Grundlage für das im *cache-oblivious*-Modell effiziente Verfahren dient der im I/O-Modell konzipierte Algorithmus von Gudmundsson und Vahrenhold [45], welcher bis auf die Berechnung spezieller Bereichsanfragen übernommen werden kann.

Der folgende Abschnitt gliedert sich in drei Unterabschnitte. Im ersten Unterabschnitt wird ein im *cache-oblivious*-Modell effizientes Verfahren zur Bearbeitung (spezieller) orthogonaler Bereichsanfragen vorgestellt. Der zweite Unterabschnitt behandelt von Gudmundsson und Vahrenhold [45] eingeführte Markierungstechniken für Bäume. Diese Markierungstechniken werden, ebenso wie das effiziente Verfahren zur Bearbeitung von orthogonalen Bereichsanfragen, für die im dritten Unterabschnitt beschriebene effiziente Umsetzung des obigen allgemeinen Ausdünnungsverfahrens im Kontext des *cache-oblivious*-Modells verwendet.

6.4.1 Orthogonale Bereichsanfragen

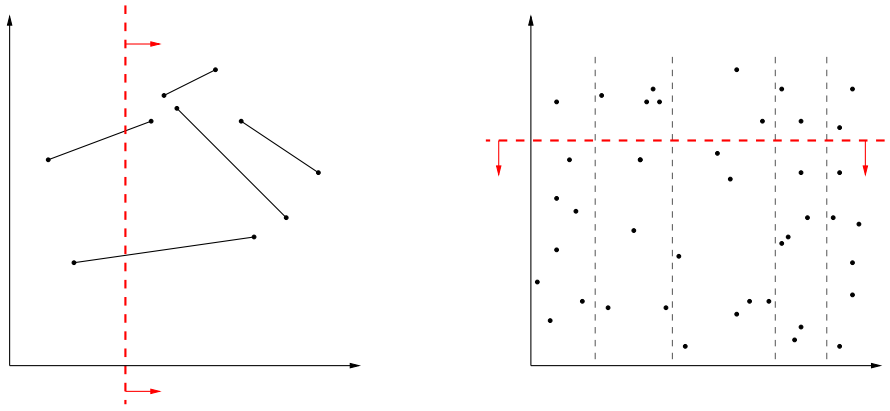
Das im *cache-oblivious*-Modell effiziente Verfahren zur Bearbeitung von orthogonalen Bereichsanfragen stützt sich auf eine Technik namens *distribution sweeping*. Diese soll zunächst beschrieben werden. Anschließend wird das effiziente Verfahren zur Beantwortung orthogonaler Bereichsanfragen entwickelt.

Distribution Sweeping

Das *plane sweep*-Paradigma [49, 52] ist ein einfaches Hilfsmittel zur Lösung geometrischer Probleme im d-dimensionalen Raum. Durch diese Technik wird ein statisches d-dimensionales Problem in eine endliche Anzahl von Instanzen eines dynamischen $(d-1)$-dimensionalen Problems transformiert, vgl. Breimann und Vahrenhold [19]. Die Kernidee des *plane sweep*-Konzepts besteht in der Annahme einer imaginären Hyperebene, welche über den Datensatz bewegt wird und durch die inkrementell Informationen hinsichtlich der Lösung des gegebenen Problems gesammelt werden, vgl. Abbildung 6.4 (a). Auf eine detaillierte Beschreibung des *plane sweep*-Paradigmas wird hier verzichtet und stattdessen auf andere Arbeiten verwiesen [19, 49, 52].

Das *plane sweep*-Konzept eignet sich gut für den Entwurf von Algorithmen im RAM-Modell und führt dort in vielen Fällen zu optimalen Laufzeiten. Wie Breimann und Vahrenhold [19] bemerken, ergibt eine direkte Anwendung dieser Technik im I/O-Modell, anhand derer die für die Verwaltung der ermittelten Informationen verwendete interne Datenstruktur durch eine externe ersetzt wird, im Allgemeinen keine effizienten Algorithmen. Wählt man z. B. einen B-Baum [13] als externe Datenstruktur zur Speicherung der durch die *sweep*-Hyperebene gesammelten Informationen, so benötigt ein Algorithmus, der im RAM-Modell eine Laufzeit von $\mathcal{O}(N \log N)$ aufweist, insgesamt $\mathcal{O}(N \log_B N)$ I/O-Operationen. Diese (obere) Grenze stellt jedoch für Probleme mit einer I/O-Komplexität von $\Omega(\frac{N}{B} \log_{M/B} \frac{N}{B})$ keine optimale Grenze dar, vgl. Breimann und Vahrenhold [19].

Ein Ansatz zur effizienten Lösung geometrischer Probleme im I/O- bzw. anderen Mehrspeichermodellen wird durch die von Goodrich *et al.* [39] vorgestellte *distribution sweeping*-Technik gegeben. Diese kombiniert das *plane sweep*-Konzept mit einem *divide and conquer*-



(a) *Plane sweep* (b) *Distribution sweeping*

Abbildung 6.4: Lösung geometrischer Probleme mit Hilfe des *plane sweep*- bzw. *distribution sweeping*-Paradigmas, vgl. [19]

Vorgehen: Informell beschrieben wird durch Anwendung dieser Technik im I/O-Modell ein zweidimensionales geometrisches Problem zunächst in $\Theta(\frac{M}{B})$ parallele (vertikale) „Streifen“ unterteilt, die jeweils ungefähr gleich viele geometrische Datenobjekte (z. B. Punkte oder Endpunkte von Liniensegmenten) enthalten.[2] Jeder dieser Streifen wird anschließend anhand eines *sweeps* von oben nach unten bearbeitet und ggf. rekursiv weiter betrachtet, vgl. Abbildung 6.4 (b). Die generelle Vorgehensweise des *distribution sweeping* wurde von Goodrich *et al.* [39] für das I/O-Modell konzipiert.

Breimann und Vahrenhold [19] beschreiben die Anwendung der Technik im I/O-Modell auf ein zweidimensionales Problem wie folgt: Bevor das rekursive Verfahren gestartet wird, werden alle geometrischen Datenobjekte bzgl. der *sweep*-Richtung sortiert. Für die effiziente (rekursive) Aufteilung in Streifen wird zudem die Menge aller x-Koordinaten in aufsteigender Reihenfolge sortiert. Zu Beginn eines jeden rekursiven Aufrufs findet eine Aufteilung der aktuellen Datenmenge in $\Theta(M/B)$ Streifen statt. Anschließend werden mittels eines *sweeps* über die Streifen Informationen hinsichtlich der Beziehungen zwischen Objekten aus unterschiedlichen Streifen gesammelt. Beziehungen zwischen Objekten aus dem gleichen Streifen werden rekursiv ermittelt. Die Rekursion bricht ab, sobald ein (rekursives) Teilproblem in den internen Speicher passt und somit effizient gelöst werden kann. Die für die Aufteilung einer Datenmenge in Teilprobleme benötigten Partitionierungselemente können mit Hilfe der zu Beginn erstellten sortierten Menge der x-Koordinaten aller Datenobjekte pro Rekursionsebene in $\Theta(N/B)$ I/O-Operationen bestimmt werden. Besitzt der *sweep* zur Bestimmung von Beziehungen zwischen Objekten aus verschiedenen Strei-

[2] Geometrische Objekte können ggf. mehrere Streifen überdecken. In diesem Fall werden sie dem maximalen Intervall zusammenhängender Streifen zugeordnet, mit denen sie jeweils interagieren, vgl. Breimann und Vahrenhold [19].

fen ebenso eine in der Anzahl der involvierten Datenelemente lineare I/O-Komplexität, d. h. $\Theta(N/B)$ I/O-Operationen pro Rekursionsebene, so ergibt sich für die Gesamtkomplexität des Verfahrens eine obere Grenze von $\mathcal{O}(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/O-Operationen.

Brodal und Fagerberg [20] zeigen, wie die *distribution sweeping*-Technik im Kontext des *cache-oblivious*-Modells effizient umgesetzt werden kann. Sie betrachten die Anwendung dieser Technik als einen Verschmelzungsprozess: Die geometrischen Datenobjekte werden zunächst bzgl. einer Dimension sortiert und anschließend mittels eines *divide and conquer*-Ansatzes in Teilprobleme aufgeteilt, deren Lösungen nach Bearbeitung aller rekursiven Aufrufe „verschmolzen", d. h. zu einer Gesamtlösung kombiniert werden. Die genauen Details der Verschmelzung von Teilproblemen variieren dabei im Allgemeinen von Problem zu Problem. Der zentrale Baustein ihrer Technik besteht in dem in Kapitel 3 vorgestellten *lazy funnelsort*-Verfahren, mit dessen Hilfe der Verschmelzungsprozess durchgeführt wird.

Als Anwendungen ihrer Technik geben Brodal und Fagerberg [20] effiziente Verfahren zur Lösung einiger Probleme aus der algorithmischen Geometrie an. Für das weitere Vorgehen ist eines dieser Beispiele von besonderem Interesse und soll nun detailliert beschrieben werden. Ebenso wird anhand dieses Beispiels die im *cache-oblivious*-Modell effiziente Umsetzung der *distribution sweeping*-Technik genauer erläutert.

Gebündelte orthogonale Bereichsanfragen

Der von Gudmundsson und Vahrenhold [45] vorgestellte Ausdünnungsalgorithmus verwendet (spezielle) orthogonale Bereichsanfragen, um die Kanten des gesuchten ausgedünnten Graphen G' zu bestimmen. Die formale Definition des im Folgenden untersuchten Problems lautet wie folgt:

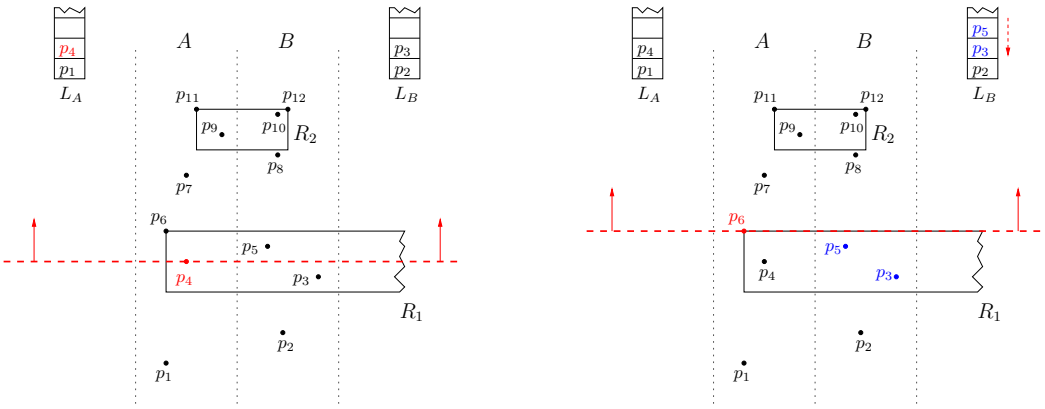
Problem. Sei eine Menge $\mathcal{E}$ von Punkten in der Ebene und eine Menge $\mathcal{Q}$ von achsenparallelen Rechtecken mit $|\mathcal{Q}| \in \mathcal{O}(|\mathcal{E}|)$ gegeben. Pro Rechteck $R \in \mathcal{Q}$ soll jeder Punkt $p \in \mathcal{E}$ mit $p \in R \cap \mathcal{E}$ gefunden und die Punkt-Rechteck-Kombination (R, p) ausgegeben werden.

Bevor das im Kontext des *cache-oblivious*-Modells effiziente Verfahren zur Lösung des obigen Problems angegeben wird, soll die zugrundeliegende allgemeine Vorgehensweise besprochen werden, welche unmittelbar zu effizienten Algorithmen im RAM- und I/O-Modell führt.

Allgemeine Vorgehensweise Brodal und Fagerberg [20] beschreiben den allgemeinen *distribution sweeping*-Ansatz zur Lösung des obigen Problems wie folgt: Zunächst wird die Menge bestehend aus allen $|\mathcal{E}|$ Punkten und allen $2|\mathcal{Q}|$ oberen linken und oberen rechten Eckpunkten der $|\mathcal{Q}|$ Rechtecke bzgl. der ersten Koordinate sortiert (die Punkte aus $\mathcal{E}$ werden im Folgenden als *Eingabepunkte* bezeichnet). Jeder der Eckpunkte wird dabei um eine vollständige Darstellung des dazugehörigen Rechtecks ergänzt. Nach diesem Vorsortierungsschritt wird analog zu obiger Beschreibung der *distribution sweeping*-Technik

ein *divide and conquer*-Ansatz bzgl. der ersten Koordinate angewendet, durch den eine Aufteilung der Menge in vertikale „Streifen“ stattfindet. Nach der rekursiven Bearbeitung der Streifen werden diese wieder „verschmolzen“, d. h. die Punktsequenzen aus zwei benachbarten Streifen A und B zu einer bzgl. der zweiten Koordinate sortierten Sequenz von Punkten aus dem Streifen $A \cup B$ vereinigt. Dieser Verschmelzungsprozess kann als *sweep* von unten nach oben angesehen werden, durch welchen die Punkte sukzessiv abgearbeitet werden.

Während des Verschmelzungsprozesses zweier Streifen A und B werden zwei Listen L_A und L_B erstellt: L_A (bzw. L_B) enthält diejenigen Eingabepunkte aus A (bzw. B), welche sich unterhalb der (imaginären) *sweep*-Linie befinden. Die sukzessive Abarbeitung der Punkte durch den *sweep* geschieht wie folgt: Ist der nächste zu bearbeitende Punkt p ein Eingabepunkt aus A (bzw. B), so wird dieser in die Liste L_A (bzw. L_B) eingefügt. Ist p ein zu einem Rechteck R gehörender oberer Eckpunkt aus A (bzw. B) und überspannt R den Streifen B (bzw. A) komplett bzgl. der ersten Koordinate, so werden alle Punkt-Rechteck-Kombinationen (R, p) mit $p \in R \cap B$ (bzw. $p \in R \cap A$) ausgegeben, indem die Liste L_B (bzw. L_A) bis zum ersten Punkt unterhalb von R traversiert wird. Überspannt das Rechteck R den Streifen B (bzw. A) nur teilweise, so sind alle Punkt-Rechteck-Kombinationen (R, p) mit $p \in R \cap B$ (bzw. $p \in R \cap A$) schon durch die (zuvor abgearbeiteten) rekursiven Aufrufe ausgegeben worden. In Abbildung 6.5 wird der Verschmelzungsprozess zweier Streifen veranschaulicht.



(a) Bearbeitung eines Eingabepunktes

(b) Bearbeitung eines oberen Eckpunktes

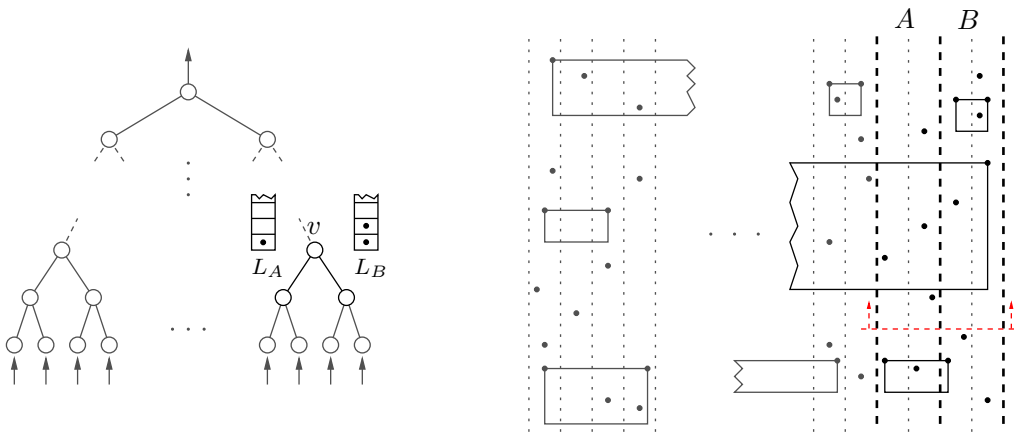
Abbildung 6.5: Verschmelzungsprozess zweier Streifen A und B. Abbildung (a) zeigt die Bearbeitung des Eingabepunktes p_4, durch welche dieser in die Liste L_A eingefügt wird. Abbildung (b) veranschaulicht die Bearbeitung des oberen Eckpunktes p_6. Durch eine Traversierung der Liste L_B können die Punkt-Rechteck-Kombinationen (R_1, p_3) und (R_1, p_5) ausgegeben werden. Die Punkt-Rechteck-Kombinationen $(R_1, p_4), (R_2, p_9)$ und (R_2, p_{10}) wurden schon durch die rekursive Bearbeitung der Streifen ausgegeben.

Wie Brodal und Fagerberg [20] bemerken, ergibt dieses Verfahren unmittelbar effiziente Algorithmen im RAM- und I/O-Modell: In beiden Fällen müssen die L-Listen nur für jeweils *einen* aktiven Verschmelzungsprozess gespeichert werden. Die rekursive Aufteilung der Datenmenge in jeweils zwei Streifen führt somit im RAM-Modell zu einem Algorithmus mit einer Laufzeit von $\mathcal{O}(|\mathcal{E}| \log |\mathcal{E}|)$ und einem Speicherplatzbedarf von $\mathcal{O}(|\mathcal{E}|)$. Im I/O-Modell ergibt sich durch eine rekursive Aufteilung der Datenmenge in $\Theta(\frac{M}{B})$ Streifen ein Algorithmus mit einer I/O-Komplexität von $\mathcal{O}(sort(|\mathcal{E}|))$ I/O-Operationen und einem Speicherplatzbedarf von $\mathcal{O}(|\mathcal{E}|)$.

Der Schlüssel zu einem im Kontext des *cache-oblivious*-Modells effizienten Algorithmus zur Lösung des obigen Problems wird durch das in Kapitel 3 vorgestellte *lazy funnelsort*-Verfahren gegeben. Eine direkte Anwendung obiger Ideen unter Verwendung dieses Sortierverfahrens führt jedoch zu einem ineffizienten Verfahren. Dieses wird zunächst kurz erläutert. Anschließend werden die von Brodal und Fagerberg [20] entwickelten Veränderungen besprochen, die letztendlich den gewünschten effizienten Algorithmus ergeben.

Ineffizientes *Cache-Oblivious*-Verfahren Eine zu der obigen Beschreibung analoge Vorgehensweise unter Verwendung des *lazy funnelsort*-Verfahrens hat den folgenden Aufbau: Zunächst wird die Menge bestehend aus allen $N = |\mathcal{E}|$ Eingabepunkten und allen $2K = 2|\mathcal{Q}|$ oberen linken und oberen rechten Eckpunkten der K Rechtecke in einem Vorsortierungsschritt bzgl. der ersten Koordinate sortiert. Die bzgl. der ersten Koordinate sortierte Liste wird anschließend mit Hilfe des *lazy funnelsort*-Verfahrens bzgl. der zweiten Koordinate sortiert. Die Liste wird dadurch in $(N + 2K)^{1/d}$ bzgl. der ersten Koordinate sortierte Segmente aufgeteilt, welche rekursiv bearbeitet und nach Abarbeitung aller rekursiven Aufrufe mit Hilfe eines $(N + 2K)^{1/d}$-Verschmelzers zu einer bzgl. der zweiten Koordinate sortierten Liste verschmolzen werden ($d \geq 2$). Wird bei dem Verschmelzungsprozess in einem (inneren) Knoten des $(N + 2K)^{1/d}$-Verschmelzers zuerst derjenige Punkt der beiden zugehörigen Eingabeströme in den Ausgabepuffer des Knotens bewegt, welcher den kleineren y-Wert aufweist, so entspricht der gesamte Verschmelzungsprozess in diesem Knoten einem *sweep* von unten nach oben. Bei diesem *sweep* kann analog zum obigen Vorgehen die Ausgabe der Punkt-Rechteck-Kombinationen stattfinden: Dazu werden in einem Knoten des $(N + 2K)^{1/d}$-Verschmelzers mit Hilfe von L-Listen Informationen über die Eingabepunkte unterhalb der imaginären *sweep*-Linie aufrecht erhalten und bei Bearbeitung eines oberen Eckpunkts entsprechende Informationen ausgegeben. Der um diese Bearbeitung ergänzte Verschmelzungsprozess in einem Knoten des $(N + 2K)^{1/d}$-Verschmelzers entspricht dann der obigen Verschmelzung zweier Streifen A und B, vgl. Abbildung 6.6.

Dieser Ansatz führt jedoch *nicht* direkt zu einem optimalen *cache-oblivious*-Algorithmus. Wie Brodal und Fagerberg bemerken, besteht das Problem in der Speicherung der für die Ausgabe der entsprechenden Punkt-Rechteck-Kombinationen benötigten L-Listen. Durch die wechselweise Ausführung der einzelnen Verschmelzungsprozesse in den jeweiligen Kno-



(a) k-Verschmelzer (b) Verschmelzung zweier Streifen

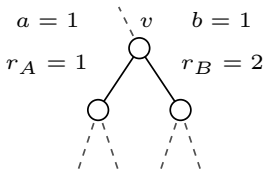
Abbildung 6.6: Verschmelzung von Streifen mit Hilfe eines k-Verschmelzers. In Abbildung (a) ist ein Knoten v des Verschmelzers hervorgehoben. Der Verschmelzungsprozess in diesem Knoten korrespondiert mit der Verschmelzung der beiden hervorgehobenen Streifen in Abbildung (b).

ten des $(N + 2K)^{1/d}$-Verschmelzers ist im Allgemeinen *nicht* nur ein Verschmelzungsprozess aktiv (wie dies der Fall im RAM- und I/O-Modell ist). Jeder der N Eingabepunkte müsste somit im schlimmsten Fall auf jeder Ebene des $(N + 2K)^{1/d}$-Verschmelzers in mindestens einer der L-Listen gespeichert werden, wodurch sich die Speicherkomplexität des $(N + 2K)^{1/d}$-Verschmelzers auf $\Theta(N \log N)$ belaufen würde. Dieser Speicherplatzbedarf, so Brodal und Fagerberg, ist suboptimal und stellt weiterhin ein Problem in der Analyse von Theorem 3.2.3 dar, da in dieser ein sublinearer Speicherplatzbedarf von k-Verschmelzern angenommen wird.

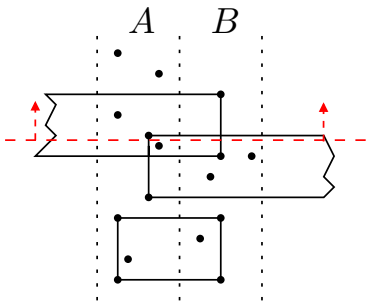
Effizientes *Cache-Oblivious*-Verfahren Brodal und Fagerberg [20] verringern diesen suboptimalen Speicherplatzbedarf, indem sie zum Einen nicht nur die $2K$ oberen Eckpunkte sondern alle $4K$ Eckpunkte der K Rechtecke betrachten und zum Anderen die Funktionsweise der k-Verschmelzer verändern. Dabei teilen sie die Anwendung eines solchen in zwei Durchläufe auf. Im ersten Durchlauf wird *ohne* Speicherung der L-Listen für jeden Knoten des k-Verschmelzers die Anzahl der Eingabepunkte bestimmt, die bei Abarbeitung des Verschmelzungsprozesses in diesem Knoten jeweils mindestens einmal (zusammen mit einem entsprechenden Rechteck als Punkt-Rechteck-Kombination) ausgegeben werden. Im zweiten Durchlauf werden zwar die L-Listen gespeichert, jedoch wird ein Eingabepunkt nur dann in eine L-Liste eingefügt, wenn dieser im Laufe des Verschmelzungsprozesses mindestens einmal ausgegeben wird. Zudem zerlegen Brodal und Fagerberg diesen zweiten Durchlauf in Iterationen, um linearen Speicherplatzbedarf zu erreichen. Die Details dieser

beiden Durchläufe werden nun genauer beschrieben. Anschließend wird der effiziente *cache-oblivious*-Algorithmus zur Bearbeitung des obigen Problems angegeben und analysiert.

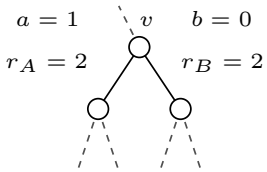
1. Durchlauf eines k-Verschmelzers Im ersten Durchlauf werden Informationen über die Eingabepunkte und Rechtecke gesammelt. Die obigen L-Listen werden noch *nicht* gespeichert und es werden auch keine Punkt-Rechteck-Kombinationen ausgegeben. Stattdessen wird ein Knoten des k-Verschmelzers um die folgenden vier Variablen erweitert: Die Variablen a bzw. b geben jeweils die Anzahl der Rechtecke mit zwei Eckpunkten in B bzw. A an, welche die *sweep*-Linie schneiden und den Streifen A bzw. B bzgl. der ersten Koordinate komplett überdecken. Die Variablen r_A bzw. r_B geben jeweils die Anzahl der Eingabepunkte in A bzw. B unterhalb der *sweep*-Linie an, welche im Laufe des Verschmelzungsprozesses in diesem Knoten mindestens einmal ausgegeben werden.



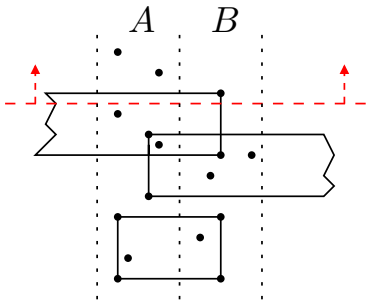
(a) Knoten v des k-Verschmelzers



(b) Verschmelzung zweier Streifen



(c) Knoten v des k-Verschmelzers



(d) Verschmelzung zweier Streifen

Abbildung 6.7: Verschiedene Zustände bei der Verschmelzung zweier Streifen während des ersten Durchlauf eines veränderten k-Verschmelzers. Die in dem korrespondierenden Knoten v gespeicherten konstant vielen Informationen werden während dieses Vorgangs entsprechend aktualisiert.

Diese Informationen können während des Verschmelzungsprozesses in einem Knoten leicht ermittelt bzw. aufrecht erhalten werden, vgl. Abbildung 6.7: Ist der nächste zu bearbeitende Punkt ein zu einem Rechteck R gehörender unterer Eckpunkt aus A (bzw. B)

und überspannt R den Streifen B (bzw. A) komplett bzgl. der ersten Dimension, so wird b (bzw. a) um eins erhöht. Entsprechend können bei Bearbeitung eines oberen Eckpunkts diese Variablen um eins verringert werden, d. h. b (bzw. a) wird bei Bearbeitung eines oberen Eckpunktes aus A (bzw. B) um eins verringert, wenn dieser zu einem Rechteck R gehört, welches den Streifen B (bzw. A) komplett bzgl. der ersten Dimension überspannt. Ist der nächste zu bearbeitende Punkt ein Eingabepunkt aus A (bzw. B), so wird r_A (bzw. r_B) genau dann um eins erhöht, wenn a (bzw. b) größer null ist.

2. Durchlauf eines k-Verschmelzers Im zweiten Durchlauf vollzieht sich die eigentliche Ausgabe der Punkt-Rechteck-Kombinationen. Dazu werden im Gegensatz zum ersten Durchlauf die L-Listen tatsächlich erstellt. Anhand der $4K$ Eckpunkte können analog zum ersten Durchlauf während des Verschmelzungsprozesses für jeden Knoten die beiden obigen Variablen a und b aufrecht erhalten werden. Mit Hilfe der beiden Variablen ist es dann möglich, einen Eingabepunkt nur dann in eine der L-Listen des Knotens einzufügen, wenn dieser während des Verschmelzungsprozesses in diesem Knoten auch mindestens einmal ausgegeben wird, d. h. ein Eingabepunkt aus einem Streifen A bzw. B wird nur dann in die Liste L_A bzw. L_B des Knotens eingefügt, wenn a bzw. b größer null ist. Durch diese Änderung wird für die beiden L-Listen des Knotens höchstens $\mathcal{O}(r_A + r_B)$ Speicherplatz benötigt. Da jeder in einer dieser Listen eingefügte Punkt mindestens einmal ausgegeben wird, ist dieser Speicherplatzbedarf weiterhin durch den Speicherplatzbedarf beschränkt, der für die Ausgabe der Punkt-Rechteck-Kombinationen in diesem Knoten benötigt wird.

Um insgesamt linearen Speicherplatzbedarf für einen (veränderten) k-Verschmelzer zu erreichen, teilen Brodal und Fagerberg diesen zweiten Durchlauf in Iterationen ein. Durch jede dieser Iterationen (mit Ausnahme der letzten Iteration) werden unter Verwendung von $\mathcal{O}(k^d)$ Speicherplatz jeweils $\Omega(k^d)$ Punkt-Rechteck-Kombinationen ausgegeben. Die Aufteilung in Iterationen geschieht mit Hilfe der im ersten Durchlauf für jeden Knoten berechneten Werte r_A und r_B: Zu Beginn jeder Iteration wird entweder ein (beliebiger) Knoten v, für den die Summe der r_A- und r_B-Werte seiner Nachfolger zwischen k^d und $3k^d$ liegt (für jeden Knoten gilt $r_A + r_B \leq k^d$), oder, falls ein solcher Knoten nicht existiert, die Wurzel des k-Verschmelzers ausgewählt. Die Auswahl eines solchen Knotens gelingt mit Hilfe einer Postorder-Traversierung des k-Verschmelzers. Dazu wird die Funktion TRAVERSE (Algorithmus 6.2) zu Beginn jeder Iteration auf den Wurzelknoten des k-Verschmelzers angewendet. Die Hilfsprozedur NEXT-ITERATION(v) markiert einen Knoten v und beendet alle aktuellen Instanzen der Funktion TRAVERSE.

Nach Auswahl eines solchen Knotens v wird ein Array der Größe $3k^d$ erzeugt, um die L_A- und L_B-Listen der Nachfolger von v speichern zu können. Anschließend wird der Verschmelzungsprozess in diesem Knoten vollständig durchgeführt, d. h. die Prozedur FILL solange auf v angewendet, bis die beiden Eingabeströme von v aufgebraucht sind. Bei diesem Vorgang wird die Ausgabe des Knotens v nicht in dem (evtl. zu kleinen) Ausgabepuffer

Funktion TRAVERSE(v)

Eingabe: Ein Knoten v eines k-Verschmelzers.

Ausgabe: Summe der r_A- und r_B-Werte der Nachfolger von v.

```
1: if v ist ein Blatt then
2:    return 0
3: else
4:    S_L = TRAVERSE(Linkes Kind von v)
5:    S_R = TRAVERSE(Rechtes Kind von v)
6:    if (S_L + S_R ∈ [k^d, 3k^d]) then
7:       NEXT-ITERATION(v)
8:    end if
9:    return S_L + S_R
10: end if
```

Algorithmus 6.2: Auswahl eines geeigneten Knotens des k-Verschmelzers

von v sondern in einem temporären Array der Größe $\mathcal{O}(k^d)$ gespeichert. Nachdem der Verschmelzungsprozess in v komplett abgearbeitet wurde, werden die r_A- und r_B-Werte aller Nachfolger von v auf null gesetzt und der Inhalt des temporären Arrays in ein globales Array der Größe $\mathcal{O}(k^d)$ transferiert. Dieses globale Array enthält die verschmolzenen Listen, welche aus den Verschmelzungsprozessen der verschiedenen Iterationen hervorgegangen sind. Haben die Eingabeströme $E_1, \ldots, E_k$ des k-Verschmelzers die Größen $N_1, \ldots, N_k$ und besitzt der in v gewurzelte Teilbaum die Eingabeströme $E_i, \ldots, E_j$, so wird der Inhalt des temporären Arrays an die Stelle $1 + \sum_{l=1}^{i-1} N_l$ und die darauf folgenden kopiert und danach mit der nächsten Iteration fortgefahren.

Algorithmus und Analyse Die Anwendung eines veränderten k-Verschmelzers besteht also aus den beiden gerade beschriebenen Durchläufen. Aufgrund dieser Änderungen besitzt die Anwendung eines solchen k-Verschmelzers nur einen linearen Speicherplatzbedarf, wodurch insgesamt der folgende *cache-oblivious*-Algorithmus zur effizienten Bearbeitung des eingangs gestellten Problems ermöglicht wird: Im ersten Schritt wird die Menge bestehend aus den $N = |\mathcal{E}|$ Eingabepunkten und den $4K$ Eckpunkten der $K = |\mathcal{Q}|$ Rechtecke bzgl. der ersten Koordinate sortiert. Die resultierende Liste wird anschließend in $(N + 4K)^{1/d}$ zusammenhängende Segmente aufgeteilt, welche rekursiv bearbeitet und anschließend mit Hilfe eines veränderten $(N + 4K)^{1/d}$-Verschmelzers zu einer bzgl. der zweiten Koordinate sortierten Liste verschmolzen werden ($d \geq 2$). Die Anwendung des veränderten $(N+4K)^{1/d}$-Verschmelzers setzt sich aus den beiden obigen Durchläufen zusammen: Im ersten Durchlauf werden Informationen über die Eingabepunkte und Rechtecke gesammelt, welche die im zweiten Durchlauf stattfindende effiziente Ausgabe der Punkt-

Rechteck-Kombinationen ermöglichen. Die Analyse dieses Verfahrens wird durch das folgende Lemma gegeben:

6.4.1 Lemma ([20]). *Sei eine Menge $\mathcal{E}$ von Punkten in der Ebene und eine Menge $\mathcal{Q}$ von orthogonalen Bereichsanfragen auf $\mathcal{E}$ mit $|\mathcal{Q}| \in \mathcal{O}(|\mathcal{E}|)$ gegeben. Unter der Annahme von $M \in \Omega(B^{(d+1)/(d-1)})$ für ein $d \geq 2$ verursacht die Bearbeitung aller Anfragen durch das obige Verfahren höchstens $\mathcal{O}(sort(|\mathcal{E}|) + \frac{T}{B})$ Speichertransfers, wobei T die Anzahl der insgesamt ausgegebenen Punkt-Rechteck-Kombinationen ist.*

Beweis. Die Sortierung der $N + 4K$ Punkte bzgl. der ersten Koordinate verursacht

$$\mathcal{O}\left(sort(N + 4K)\right) = \mathcal{O}\left(sort(|\mathcal{E}|)\right)$$

Speichertransfers. Da in den ersten Durchläufen der veränderten k-Verschmelzer die Knoten jeweils nur um konstant viele Informationen ergänzt werden, verursacht die Abarbeitung aller ersten Durchläufe insgesamt ebenso höchstens $\mathcal{O}\left(sort(|\mathcal{E}|)\right)$ Speichertransfers. Es müssen somit nur die zweiten Durchläufe der involvierten k-Verschmelzer analysiert werden.

Die Analyse der zweiten Durchläufe orientiert sich an den Beweisen zu den Theoremen 3.2.3 und 3.3.2. Der zweite Durchlauf eines k-Verschmelzers ist in Iterationen eingeteilt, von denen jede nach Konstruktion einen Speicherplatzbedarf von $\Theta(k^d)$ besitzt. Weiterhin werden durch jede Iteration, mit Ausnahme der letzten, mindestens $\Omega(k^d)$ Punkt-Rechteck-Kombinationen ausgegeben. Der Speicherplatzbedarf $S(k)$ für den zweiten Durchlauf eines k-Verschmelzers ist also insgesamt linear in der Anzahl der bearbeiteten Punkte, d. h. $S(k) \leq c \cdot k^d$ für eine passende Konstante $c > 0$. Die Anzahl der durch Anwendung eines veränderten k-Verschmelzers verursachten Speichertransfers kann somit analog zum Beweis des Theorems 3.3.2 analysiert werden:

Ist der k-Verschmelzer selbst ein Basisbaum, so werden durch Anwendung desselben $\mathcal{O}(k + \frac{k^d}{B} + \frac{\bar{T}}{B})$ Speichertransfers verursacht, wobei $\bar{T}$ die Anzahl der durch die Anwendung des k-Verschmelzers ausgegebenen Punkt-Rechteck-Kombinationen sei. Ansonsten kann wieder die Einteilung des k-Verschmelzers in Basisbäume betrachtet werden. Sei dazu $\bar{k}$ die Anzahl der Blätter eines solchen Basisbaums. Zusätzlich zu den ursprünglichen Speichertransfers fallen bei Anwendung eines Basisbaums nun die Speichertransfers für die Bearbeitung der L-Listen der Knoten des Basisbaums an. Dabei können jedoch die durch das Schreiben von Elementen in die L-Listen verursachten Speichertransfers den Kosten für das Einlesen der Elemente aus den Eingabeströmen des Basisbaums zugeschrieben werden. Ebenso können die durch das Lesen von Elementen aus den L-Listen verursachten Speichertransfers den Kosten für die Ausgabe der Punkt-Rechteck-Kombinationen auferlegt werden. Unter der Annahme, dass kein direkt unter dem Basisbaum liegender großer Puffer geleert wird, fallen bei Anwendung des Basisbaums also wieder höchstens $\mathcal{O}(\frac{\bar{k}^d}{B})$ Speichertransfers an, was $\mathcal{O}(\frac{1}{B})$ amortisierten Speichertransfers pro Element entspricht.

Insgesamt werden deshalb *pro* Iteration höchstens $\mathcal{O}(k + \frac{k^d}{B} + \frac{T'}{B})$ Speichertransfers verursacht, wobei T' die Anzahl ausgegebenen Punkt-Recht-Kombinationen plus die Anzahl der durch den Transfer von Elementen zwischen den Basisbäumen anfallenden Speichertransfers sei. Die Anwendung der Prozedur TRAVERSE auf den Wurzelknoten des k-Verschmelzers zu Beginn einer jeden Iteration könnte evtl. involvierte Blöcke aus dem internen Speicher verdrängen, was zum erneuten Laden dieser Blöcke führen würde. Da pro Iteration aber mindestens $\Omega(\frac{k^d}{B})$ Punkt-Rechteck-Kombinationen ausgegeben werden, können die Kosten für das erneute Laden diesen Ausgabeelementen auferlegt werden.

Für *alle* Iterationen beträgt die Anzahl der durch den Transfer von Elementen zwischen den Basisbäumen verursachten Speichertransfers insgesamt höchstens $\mathcal{O}(\frac{k^d}{B} \log_M k^d)$. Die Anwendung eines veränderten k-Verschmelzers verursacht deshalb insgesamt $\mathcal{O}(\frac{k^d}{B} \log_M k^d + \frac{T}{B})$ Speichertransfers und benötigt $\mathcal{O}(\frac{k^d}{B})$ Speicherblöcke. Analog zur Analyse des *lazy funnelsort*-Verfahrens (Theorem 3.2.3) folgt dann, dass der gesamte Algorithmus höchstens

$$\mathcal{O}\left(d \cdot \left(\frac{N+4K}{B}\right) \log_{M/B}\left(\frac{N+4K}{B}\right) + \frac{T}{B}\right) = \mathcal{O}\left(sort(|\mathcal{E}|) + \frac{T}{B}\right)$$

Speichertransfers verursacht. □

In Anlehnung an die von Gudmundsson und Vahrenhold [45] vorgestellte Version des Verfahrens von Arge *et al.* [11] zur effizienten Bearbeitung von (speziellen) orthogonalen Bereichsanfragen im I/O-Modell kann die folgende Variante des obigen Lemmas gezeigt werden:

6.4.2 Lemma. *Sei eine Menge $\mathcal{E}$ von Punkten in der Ebene und eine Menge $\mathcal{Q}$ von orthogonalen Bereichsanfragen auf $\mathcal{E}$ mit $|\mathcal{Q}| \in \mathcal{O}(|\mathcal{E}|)$ gegeben. Soll pro Rechteck R aus $\mathcal{Q}$ mit $R \cap \mathcal{E} \neq \emptyset$ genau eine Punkt-Rechteck-Kombination (R, p) mit $p \in R \cap \mathcal{E}$ ausgegeben werden, so verursacht die Bearbeitung aller Anfragen unter der Annahme von $M \in \Omega(B^{(d+1)/(d-1)})$ für ein $d \geq 2$ insgesamt höchstens $\mathcal{O}(sort(|\mathcal{E}|))$ Speichertransfers.*

Beweis. Die Bearbeitung dieser speziellen Bereichsanfragen gestaltet sich als wesentlich einfacher als die Bearbeitung der allgemeinen Bereichsanfragen, bei denen pro Anfragerechteck $R \in \mathcal{Q}$ *jede* Punkt-Rechteck-Kombination (R, p) mit $p \in R \cap \mathcal{E}$ ausgegeben werden muss. Das Problem bei Bearbeitung der allgemeinen Bereichsanfragen besteht in der Speicherung der für die Ausgabe der Punkt-Rechteck-Kombinationen benötigten L-Listen. Dieses Problem tritt bei Bearbeitung der speziellen Bereichsanfragen nicht auf. Da pro Rechteck $R \in \mathcal{Q}$ jeweils nur maximal eine Punkt-Rechteck-Kombination (R, p) mit $p \in R \cap \mathcal{E}$ ausgegeben werden muss, kann die folgende Variante des im Kontext der allgemeinen Bereichsanfragen ineffizienten *cache-oblivious*-Verfahrens verwendet werden:

In einem Vorsortierungsschritt werden die $|\mathcal{E}| + 2|\mathcal{Q}|$ Eingabe- und oberen Eckpunkte bzgl. der ersten Koordinate sortiert. Anschließend wird die bzgl. der ersten Koordinate sortierte Liste mit Hilfe des *lazy funnelsort*-Verfahrens bzgl. der zweiten Koordinate

sortiert. Während der Verschmelzung zweier Streifen in einem Knoten eines involvierten k-Verschmelzers findet analog zum ursprünglichen (ineffizienten) Verfahren die Ausgabe der Punkt-Rechteck-Kombinationen statt. Im Gegensatz zum ursprünglichen Verfahren wird aber jeweils nur der sich direkt unter der (imaginären) *sweep*-Linie befindende Eingabepunkt aus dem Streifen A bzw. B in der L_A- bzw. L_B-Liste gespeichert. Bei Abarbeitung eines zu einem Rechteck R gehörenden oberen Eckpunkts aus A bzw. B wird dann geprüft, ob das zugehörige Rechteck R den Streifen B bzw. A überspannt und falls ja, ob der in der Liste L_B bzw. L_A gespeicherte Punkt p in R liegt. Sind beide Bedingungen erfüllt, so wird die Punkt-Rechteck-Kombination (R, p) ausgegeben. Der obere Eckpunkt wird in diesem Fall *nicht* weiter bearbeitet, d. h. nicht in den Ausgabepuffer des aktuellen Knotens bewegt. Ist eine der beiden Bedingungen nicht erfüllt, so wird der Eckpunkt zur weiteren Bearbeitung in den Ausgabepuffer des Knotens bewegt. Da für jedes Rechteck insgesamt zwei obere Eckpunkte bearbeitet werden, können durch den obigen Sortierschritt höchstens zwei Punkt-Rechteck-Kombinationen pro Rechteck ausgegeben werden. Diese Duplikate lassen sich nach Beendigung des Sortierschritts durch einen weiteren Sortierschritt mit anschließender Traversierung entfernen.

Der Vorsortierungsschritt und das Entfernen von Duplikaten am Ende des Algorithmus verursacht jeweils höchstens $\mathcal{O}(sort(|\mathcal{E}|))$ Speichertransfers. Da die Knoten eines bei dem zweiten Sortierschritt involvierten k-Verschmelzers jeweils nur konstant viele Informationen speichern, werden durch diesen Schritt ebenso höchstens $\mathcal{O}(sort(|\mathcal{E}|))$ Speichertransfers verursacht. □

6.4.2 Markierungstechniken für Bäume

Für die I/O-effiziente Ausdünnung eines Spanner-Graphen führen Gudmundsson und Vahrenhold [45] allgemeine Markierungstechniken für Bäume ein. Die folgenden drei Lemmata fassen diese Markierungstechniken, angepasst an die Bedürfnisse des *cache-oblivious*-Modells, zusammen.

6.4.3 Lemma ([45]). *Sei ein Baum T mit N Knoten gegeben. Dann können alle $\mathcal{O}(N)$ Blätter des Baums mit $\mathcal{O}(sort(N))$ Speichertransfers in einer von-links-nach-rechts-Ordnung markiert werden.*

Beweis. Gudmundsson und Vahrenhold berechnen zunächst eine Euler-Tour für T, aus der sie anschließend das BFS-Level für jeden Knoten von T ermitteln. Dieser Prozess lässt sich im *cache-oblivious*-Modell ebenso mit höchstens $\mathcal{O}(sort(N))$ Speichertransfers durchführen [7, 24]. Während eines erneuten Durchlaufs der Knotenmenge anhand der Euler-Tour kann ein Knoten von T als Blatt erkannt werden, da das BFS-Level eines solchen mit einem lokalen Maximum korrespondiert. Im Zuge dieser Traversierung lassen sich die Blätter zudem in der gewünschten Ordnung markieren. Die Korrektheit dieser von-links-nach-rechts-Markierung folgt aus der Tatsache, dass die Knoten in Präorder traversiert werden

und somit jeder Knoten vor seinem rechten Geschwisterknoten betrachtet wird. Wie auch im I/O-Modell werden die Kosten für den gesamten Vorgang durch die Kosten für die Berechnung der Euler-Tour dominiert; es werden also insgesamt höchstens $\mathcal{O}(sort(N))$ Speichertransfers verursacht. □

6.4.4 Lemma ([45]). *Sei ein Baum T mit N Knoten gegeben, dessen Blätter in einer von-links-nach-rechts-Ordnung markiert sind. Dann kann jeder innere Knoten v mit einem Intervall $[l_v, r_v], l_v, r_v \in \mathbb{N}$, markiert werden, so dass die folgenden Bedingungen erfüllt sind:*

(1) Jedes Blatt in dem in v gewurzelten Teilbaum ist mit einer ganzen Zahl $l(v) \in [l_v, r_v]$ markiert.

(2) Es existiert mindestens ein Blatt in dem in v gewurzelten Teilbaum, welches mit einer Zahl $l(v) \in [l_v, r_v]$ markiert ist.

(3) Das Intervall $[l_v, r_v]$ ist das kleinste Intervall, welches diese Eigenschaften erfüllt.

Die Markierung aller inneren Knoten verursacht $\mathcal{O}(sort(N))$ Speichertransfers.

Beweis. Der Algorithmus von Gudmundsson und Vahrenhold berechnet die gesuchte Markierung von den Blättern ausgehend, indem er jedem Knoten das minimale Intervall zuordnet, welches die den Kindern des Knotens zugeordneten Intervalle umschließt. Jedem Blatt v mit Markierung $l(v)$ wird dabei zu Beginn das Intervall $[l(v), l(v)]$ zugewiesen. Die dadurch entstehende Markierung aller Knoten erfüllt die drei oben angegebenen Eigenschaften.

Um diesen Vorgang effizient umzusetzen, berechnen Gudmundsson und Vahrenhold zunächst für jeden Knoten sein BFS-Level, seine BFS-Nummer und die BFS-Nummer seines Vaterknotens. Diese Berechnungen können analog zum Vorgehen im I/O-Modell mit Hilfe von Euler-Tour-Techniken im *cache-oblivious*-Modell mit $\mathcal{O}(sort(N))$ Speichertransfers ausgeführt werden [7, 24]. Um nun die Knoten des Baums von den Blättern aus zur Wurzel hin mit den entsprechenden Intervallen zu markieren, werden diese in absteigender Reihenfolge bzgl. ihrer BFS-Levels sortiert und anschließend durchlaufen. Sei i_0 das maximale BFS-Level. Beginnend mit $i = i_0$ können während des Durchlaufs iterativ alle Knoten mit BFS-Level i und $i-1$ aus dieser Liste entnommen werden. Die Knoten mit BFS-Level i werden anschließend bzgl. der BFS-Nummer ihres Vaterknotens und die Knoten mit BFS-Level $i-1$ bzgl. ihrer BFS-Nummer sortiert. Durch eine simultane Traversierung beider sortierten Listen kann dann jeder Knoten v auf BFS-Level $i-1$ mit dem kleinsten Intervall markiert werden, welches die den Kindern von v zugeordneten Intervalle umschließt.

Die Korrektheit der Markierung folgt induktiv. Der Aufwand für diese Markierung beläuft sich auf $\mathcal{O}(sort(N))$ Speichertransfers, da insgesamt konstant viele Euler-Touren

berechnet werden und jeder Knoten an konstant vielen Sortierschritten beteiligt ist, vgl. Gudmundsson und Vahrenhold [45]. □

Die in den Lemmata 6.4.3 und 6.4.4 vorgestellten Markierungstechniken können auf einen fairen Aufteilungsbaum T, der für die Eckenmenge S eines t-Spanners $G = (S, E)$ konstruiert wurde, angewendet werden. Die Markierung der Blätter von T impliziert dann eine Ummarkierung der Ecken aus S: Zu jeder Ecke $p \in S$ existiert in T genau ein Blatt v mit $\sigma(v) = \{p\}$. Mit $l(p) = l(v)$ wird dann der Ecke p der Wert $l(p) \in \{1, \ldots, |S|\}$ zugeordnet.

Gudmundsson und Vahrenhold [45] zeigen, wie die Ummarkierung der Eckenmenge S des t-Spanners G I/O-effizient auf die Kantenmenge E von G abgebildet werden kann. Dieser Vorgang lässt sich im Kontext des *cache-oblivious*-Modells ebenso effizient realisieren:

6.4.5 Lemma ([45]). *Sei eine eindeutige Ummarkierung der Eckenmenge $S \subset \mathbb{R}^d$ eines t-Spanners $G = (S, E)$ gegeben. Dann kann jede gerichtete Kante $e = (p, q) \in E$ mit der Markierung $(l(p), l(q))$ gekennzeichnet werden, wobei $l(p)$ und $l(q)$ die beiden neuen Markierungen der Ecken p und q sind. Liegen die Kantenmenge E und ein Baum vor, welcher die ummarkierten Ecken des t-Spanners in seinen Blättern speichert, so können alle Kanten mit $\mathcal{O}(sort(|E|))$ Speichertransfers ummarkiert werden.*

Beweis. Die ummarkierten Ecken aus S können mit $\mathcal{O}(sort(|S|))$ Speichertransfers mit Hilfe von Euler-Tour-Techniken aus dem Baum extrahiert und anschließend bzgl. ihrer ursprünglichen Markierung sortiert werden. Indem alle Kanten aus E bzgl. der (ursprünglichen) Markierungen ihrer Anfangsecken sortiert und anschließend zusammen mit der zuvor erstellten Liste simultan durchlaufen werden, können die Anfangsecken der Kanten ummarkiert werden. Analog kann mit den Endecken verfahren werden. Der gesamte Vorgang verursacht somit höchstens $\mathcal{O}(sort(|S|+|E|)) = \mathcal{O}(sort(|E|))$ Speichertransfers.□

6.4.3 Der Ausdünnungsalgorithmus

Anhand der in den vorherigen Unterabschnitten vorgestellten Techniken kann nun der im *cache-oblivious*-Modell effiziente Ausdünnungsalgorithmus beschrieben werden. Dieser folgt zwar der in Abschnitt 6.2 vorgestellten allgemeinen Vorgehensweise, setzt diese jedoch in einer leicht modifizierten Form um.

Analog zu Gudmundsson und Vahrenhold [45] wird zunächst eine WSPD $\mathcal{D} = (T, \mathcal{R})$ für die Eckenmenge S des gegebenen t-Spanners $G = (S, E)$ bzgl. des Wertes $s = \frac{1}{\varepsilon}((1 + \varepsilon)(8t + 4) + 4)$ berechnet. Die effiziente Berechnung der WSPD für S wird durch den in Abschnitt 4.3 vorgestellten Algorithmus ermöglicht, welcher insgesamt $\mathcal{O}(sort(|S|))$ Speichertransfers verursacht. Wie in diesem Abschnitt beschrieben, wird die Realisation $\mathcal{R} = \{\{A_1, B_1\}, \ldots, \{A_m, B_m\}\}$ der WSPD durch eine Menge $\mathcal{R}_T = \{\{v_1, w_1\}, \ldots, \{v_m, w_m\}\}$ von Knotenpaaren $\{v_i, w_i\}$ mit Knoten aus T repräsentiert.

Die Knoten von T können anschließend mit Hilfe der in den Lemmata 6.4.3 und 6.4.4 vorgestellten Techniken markiert werden. Jedem inneren Knoten v wird dadurch ein Intervall $[l_v, r_v] \subset [1, |S|]$ und jedem Blatt eine Zahl aus $\{1, \ldots, |S|\}$ zugeordnet. Die Markierung der Knoten impliziert dann, wie oben beschrieben, eine Ummarkierung der Eckenmenge S; jede Ecke $p \in S$ erhält dadurch eine neue Markierung $l(p)$. Die Ummarkierung der Eckenmenge S erfüllt dabei die folgende Eigenschaft:

6.4.6 Lemma ([45]). *Die obige Ummarkierung der Eckenmenge S erfüllt die Eigenschaft, dass jede Komponente $C \in \{A_i, B_i\}$ eines scharf getrennten Paars $\{A_i, B_i\}$ der WSPD mit einem Intervall $[l(C), r(C)]$ korrespondiert, so dass die neue Markierung $l(p)$ einer Ecke $p \in S$ genau dann in $[l(C), r(C)]$ liegt, wenn $p \in C$ gilt.*

Beweis. Sei $\{A_i, B_i\}$ ein scharf getrenntes Paar der WSPD und C eine Komponente von $\{A_i, B_i\}$. Nach Definition der WSPD gibt es in dem dazugehörigen fairen Aufteilungsbaum T einen Knoten v mit $\sigma(v) = C$. Da der faire Aufteilungsbaum ein mit der Menge S verknüpfter Binärbaum ist, sind die durch die Blätter des in v gewurzelten Teilbaums von T dargestellten Punkte genau die Punkte aus C. Weiterhin wurde dem Knoten v durch den im Beweis zu Lemma 6.4.4 verwendeten Algorithmus ein Intervall $[l_v, r_v]$ zugeordnet. Mit $[l(C), r(C)] = [l_v, r_v]$ bleibt dann

$$p \in C \quad \Leftrightarrow \quad l(p) \in [l_v, r_v]$$

zu zeigen.

Sei zunächst $p \in C$. Nach Wahl von $[l(C), r(C)]$ wird p durch ein Blatt in dem in v gewurzelten Teilbaum von T dargestellt, dessen Markierung $l(p)$ in $[l_v, r_v]$ liegt, vgl. Lemma 6.4.4. Es folgt also $p \in C \Rightarrow l(p) \in [l_v, r_v]$.

Die Rückrichtung kann durch Kontraposition, also durch $p \notin C \Rightarrow l(p) \notin [l_v, r_v]$ gezeigt werden. Dies gelingt durch einen Widerspruchsbeweis: Gelte also $p \notin C$ und $l(p) \in [l_v, r_v]$. Da die Blätter nach Lemma 6.4.3 in einer von-links-nach-rechts-Ordnung markiert sind, muss für alle $q \in C$ entweder $l(q) > l(p)$ oder $l(q) < l(p)$ gelten. O. B. d. A. gelte $l(q) < l(p)$ für alle $q \in C$. Mit l_{max} sei die maximale Markierung aller Punkte aus C bezeichnet, d. h. $l_{max} = \max_{q \in C} l(q)$. Es liegen somit alle Markierungen $l(q)$ in dem Intervall $[l_v, l_{max}]$. Da $l_{max} < l(p) \leq r_v$ gilt, führt dies zu einem Widerspruch zur Minimalität des Intervalls $[l_v, r_v]$, vgl. Lemma 6.4.4. Es gilt also $p \notin C \Rightarrow l(p) \notin [l_v, r_v]$ und somit $l(p) \in [l_v, r_v] \Rightarrow p \in C$. □

Die Ummarkierung der Eckenmenge S kann nach Lemma 6.4.5 mit $\mathcal{O}(sort(|E|))$ Speichertransfers auf die Kantenmenge E abgebildet werden. Dadurch lässt sich jede (gerichtete) Kante $(p, q) \in E$ mit einem Punkt $M_{(p,q)} = (l(p), l(q)) \in \{1, \ldots, |S|\} \times \{1, \ldots, |S|\}$ identifizieren.

Um die Kantenmenge des gesuchten ausgedünnten Graphen G' zu konstruieren, wählen Gudmundsson und Vahrenhold [45] anschließend für jedes scharf getrennte Paar $\{A_i, B_i\}$

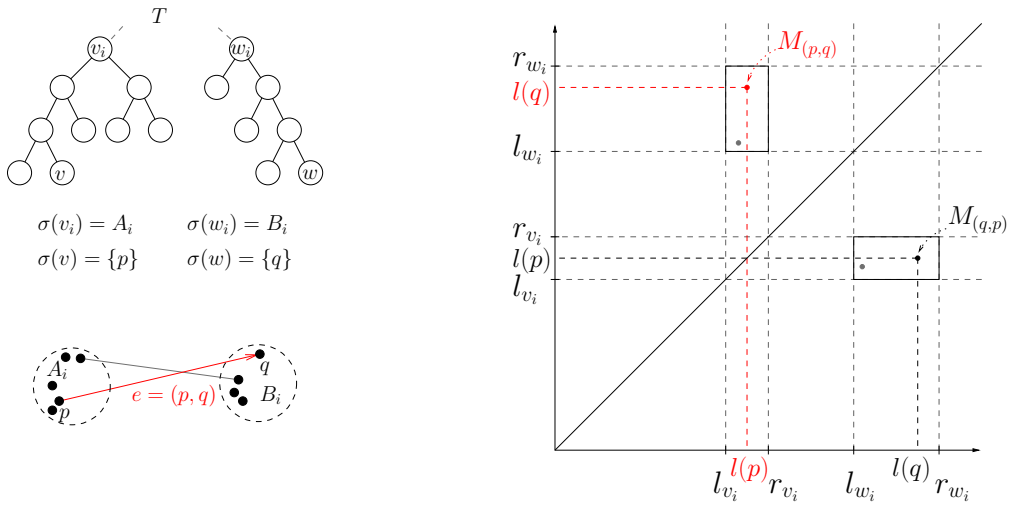


Abbildung 6.8: Abbildung (a) zeigt eine (gerichtete) Kante $e = (p, q)$, deren Endecken durch ein scharf getrenntes Paar $\{A_i, B_i\}$ getrennt werden. Das Paar $\{A_i, B_i\}$ wird dabei durch ein ungeordnetes Knotenpaar $\{v_i, w_i\}$ und die Ecken p und q durch zwei Blätter v und w des zugehörigen fairen Aufteilungsbaums T repräsentiert. Abbildung (b) zeigt die durch die Markierungstechniken induzierten Rechtecke und Punkte. Das ungeordnete Knotenpaar $\{v_i, w_i\}$ korrespondiert dabei mit den beiden Rechtecken $[l_{v_i}, r_{v_i}] \times [l_{w_i}, r_{w_i}]$ und $[l_{w_i}, r_{w_i}] \times [l_{v_i}, r_{v_i}]$. Die rot gekennzeichnete Kante $e = (p, q)$ wird anhand der Markierungstechniken auf den rot gekennzeichneten Punkt $M_{(p,q)} = (l(p), l(q))$ und die Kante $\bar{e} = (q, p)$ auf den Punkt $M_{(q,p)} = (l(q), l(p))$ abgebildet.

der WSPD jeweils genau eine der Kanten von E aus, deren Endpunkte durch das Paar getrennt werden (falls eine solche existiert). Die Kantenmenge E' von G' besteht dann aus allen ausgewählten Kanten. Da die Größe der Realisation in $\mathcal{O}(|S|)$ liegt, gilt wie gewünscht $|E'| \in \mathcal{O}(|S|)$. Die I/O-effiziente Auswahl der Kanten gelingt Gudmundsson und Vahrenhold [45] mit Hilfe von speziellen Bereichsanfragen, die durch das folgende Korollar motiviert werden, vgl. Abbildung 6.8:

6.4.7 Korollar ([45]). *Sei $S \subset \mathbb{R}^d$ eine endliche Punktmenge aus dem $\mathbb{R}^d$, eine bzgl. eines Wertes $s > 0$ scharf getrennte Realisation $\mathcal{R} = \{\{A_1, B_1\}, \ldots, \{A_k, B_k\}\}$ von $S \otimes S$, ein t-Spanner $G = (S, E)$ mit $t \geq 1$ und ein fairer Aufteilungsbaum T von S gegeben. Sind die Knoten aus T und die Ecken aus S wie oben beschrieben markiert, dann gilt für ein Knotenpaar $\{v_i, w_i\}$ mit $\sigma(v_i) = A_i$ und $\sigma(w_i) = B_i$ und eine gerichtete Kante $e = (p, q) \in E$:*

$$p \in A_i \quad \wedge \quad q \in B_i \quad \Longleftrightarrow \quad l(p) \in [l_{v_i}, r_{v_i}] \quad \wedge \quad l(q) \in [l_{w_i}, r_{w_i}]$$

Sei nun die Menge $\mathcal{E}$ mit $\mathcal{E} = \{M_{(p,q)} \mid (p, q) \in E\}$ mit $M_{(p,q)} = (l(p), l(q)) \in \{1, \ldots, |S|\} \times \{1, \ldots, |S|\}$ gegeben. Ein Paar $\{A_i, B_i\}$ der berechneten WSPD wird durch ein ungeordnetes Paar $\{v_i, w_i\}$ von Knoten des zugehörigen fairen Aufteilungsbaums repräsentiert. Das

Funktion PRUNESPANNER-CACHE(G, ε)

Eingabe: Ein t-Spanner $G = (S, E)$ $(t \geq 1)$ mit endlicher Eckenmenge $S \subset \mathbb{R}^d$ und eine reelle Konstante $\varepsilon > 0$.

Ausgabe: Ein $(1+\varepsilon)$-Spanner $G' = (S, E')$ von G mit $|E'| \in \mathcal{O}(|S|)$.

1: $E' = \emptyset$
2: Berechne mit Hilfe des Algorithmus aus Abschnitt 4.3 eine WSPD $\mathcal{D} = (T, \{\{A_1, B_1\}, \ldots, \{A_m, B_m\}\})$ von S bzgl. des Wertes $s = \frac{1}{\varepsilon}((1+\varepsilon)(8t+4)+4)$.
3: Markiere die Knoten des fairen Aufteilungsbaums T mit Hilfe der in den Beweisen zu Lemma 6.4.3 und Lemma 6.4.4 verwendeten Algorithmen um. Durch die Markierung der Blätter von T wird jeder Ecke $p \in S$ eine Zahl $l(p) \in \{1, \ldots, S\}$ zugeordnet.
4: Bilde mit Hilfe des Algorithmus aus Lemma 6.4.5 die Markierung der Eckenmenge S auf die Kantenmenge E ab. Jede gerichtete Kante $e = (p, q) \in E$ kann dadurch mit einem Punkt $M_{(p,q)} = (l(p), l(q)) \in \{1, \ldots, S\} \times \{1, \ldots, S\}$ identifiziert werden.
5: Konstruiere die Punktmenge $\mathcal{E} = \{(l(p), l(q)) \in \{1, \ldots, |S|\} \times \{1, \ldots, S\} \mid (p, q) \in E\}$.
6: Konstruiere die Anfragemenge $\mathcal{Q} = \{[l_{v_i}, r_{v_i}] \times [l_{w_i}, r_{w_i}] \mid i \in \{1, \ldots, m\}\}$.
7: Wähle für jedes Anfragerechteck $R \in \mathcal{Q}$ mit $R \cap \mathcal{E} \neq \emptyset$ genau einen Punkt $M_{(p,q)} \in R \cap \mathcal{E}$ aus.
8: Füge für jeden in Schritt 7 ausgewählten Punkt $M_{(p,q)}$ die zugehörige ungerichtete Kante $\{p, q\} \in E$ der Kantenmenge E' hinzu.
9: **return** $G' = (S, E')$

Algorithmus 6.3: Ausdünnungsalgorithmus im *cache-oblivious*-Modell

geordnete Paar (v_i, w_i) korrespondiert weiterhin nach obigem Korollar mit dem Bereich $[l_{v_i}, r_{v_i}] \times [l_{w_i}, r_{w_i}] \subset [1, |S|] \times [1, |S|]$. Führt man nun für die durch die geordneten Paare $\{(v_1, w_1), \ldots, (v_k, w_k)\}$ induzierten Rechtecke $\mathcal{Q} = \{[l_{v_i}, r_{v_i}] \times [l_{w_i}, r_{w_i}] \mid i \in \{1, \ldots, m\}\}$ jeweils eine Bereichsanfrage auf der Menge $\mathcal{E}$ durch und gibt für jedes Anfragerechteck $R \in \mathcal{Q}$ mit $R \cap \mathcal{E} \neq \emptyset$ genau einen Punkt $M_{(p,q)} \in \mathcal{E} \cap R$ aus, so stellt die durch die Menge A aller ausgegebenen Punkte dargestellte Kantenmenge $\{\{p, q\} \mid M_{(p,q)} \in A\}$ die Kantenmenge E' des gesuchten dünn besetzten $(1+\varepsilon)$-Spanners $G' = (S, E')$ von G dar.[3]

Die für dieses Verfahren benötigte Menge $\mathcal{E}$ kann dabei durch den in Lemma 6.4.5 vorgestellten Algorithmus mit $\mathcal{O}(sort(|E|))$ Speichertransfers berechnet werden. Entsprechend kann die Anfragemenge $\mathcal{Q} = \{[l_{v_i}, r_{v_i}] \times [l_{w_i}, r_{w_i}] \mid i \in \{1, \ldots, m\}\}$ effizient konstruiert werden: Durch die Markierung von T wurde jedem Knoten v ein Intervall $[l_v, r_v]$ zugeordnet. Diese Markierung kann analog zu Lemma 6.4.5 auf die Menge $\mathcal{R}_T = \{\{v_1, w_1\}, \ldots, \{v_m, w_m\}\}$ der die Realisation darstellenden Menge von Knotenpaaren mit Hilfe von zwei Sortier- und Traversierungsschritten übertragen werden. Da $m \in \mathcal{O}(|S|)$ gilt, verursacht die

[3] Zu beachten ist hierbei, dass die Kantenmenge E des t-Spanners G nach Definition eines ungerichteten Graphen zu jeder (gerichteten) Kante (p, q) auch die entsprechende Zwillingskante (q, p) enthält, vgl. Abbildung 6.8. Für jede durch einen Ausgabepunkt $M_{(p,q)} \in \mathcal{E} \cap R$ dargestellte gerichtete Kante $(p, q) \in E$ wird die ungerichtete Kante $\{p, q\}$ der Kantenmenge von E' hinzugefügt.

Konstruktion der Anfragemenge $\mathcal{Q}$ somit höchstens $\mathcal{O}(sort(|S|))$ Speichertransfers. Nach Lemma 6.4.2 werden weiterhin für die Bearbeitung aller $|\mathcal{Q}|$ Anfragen auf der Menge $\mathcal{E}$ höchstens $\mathcal{O}(sort(|\mathcal{E}|)) = \mathcal{O}(sort(|E|))$ Speichertransfers benötigt.

Der Aufbau des gesamten Algorithmus zur Ausdünnung eines t-Spanners G wird durch die Funktion PruneSpanner-Cache (Algorithmus 6.3) zusammengefasst. Die Anwendung dieser Funktion auf einen t-Spanner $G = (S, E)$ mit $t \geq 1$ und eine beliebige reelle Konstante $\varepsilon > 0$ verursacht höchstens $\mathcal{O}(sort(|E|))$ Speichertransfers und produziert einen $(1+\varepsilon)$-Spanner $G' = (S, E')$ von G mit $|E'| \in \mathcal{O}(|S|)$.

Insgesamt erhält man also im Kontext des *cache-oblivious*-Modells analog zum Ergebnis von Gudmundsson und Vahrenhold [45] das folgende Ergebnis:

6.4.8 Theorem. *Sei eine endliche Punktmenge $S \subset \mathbb{R}^d$ und ein t-Spanner $G = (S, E)$ mit $t \geq 1$ gegeben. Dann kann für jede reelle Konstante $\varepsilon > 0$ mit $\mathcal{O}(sort(|E|))$ Speichertransfers ein $(1+\varepsilon)$-Spanner $G' = (S, E')$ von G mit $|E'| \in \mathcal{O}(|S|)$ berechnet werden.*

Kapitel 7

Implementierung

Im Rahmen dieser Arbeit wurde eine programmtechnische Realisierung der in den Kapiteln 5 und 6 vorgestellten Verfahren zur Konstruktion und Ausdünnung von Spanner-Graphen durchgeführt. Diese Umsetzung der Konstruktions- und Ausdünnungsverfahren wird im Folgenden kurz erläutert.

In Abschnitt 7.1 wird zunächst auf die Zielsetzung der programmtechnischen Realisierung und auf die bei der Umsetzung verwendeten (externen) Softwarebibliotheken eingegangen. In Abschnitt 7.2 werden einige Details der Implementierung der relevanten Algorithmen gegeben. Der Funktionsumfang der programmtechnischen Realisierung wird anschließend in Abschnitt 7.3 besprochen. In Abschnitt 7.4 werden abschließend einige Spanner-Graphen präsentiert, die mit Hilfe des Programms erzeugt werden können.

7.1 Grundlegendes

Das Programm `GraphAnalyser` stellt die programmtechnische Realisierung dar und wurde in der Programmiersprache `Java` verfasst. Mit Hilfe dieses Programms kann die Arbeitsweise der in dieser Arbeit besprochenen Konstruktions- und Ausdünnungsverfahren bei Anwendung auf Punktmengen aus dem $\mathbb{R}^2$ bzw. auf Graphen mit Eckenmengen aus dem $\mathbb{R}^2$ visualisiert werden. Eine ausführbare `jar`-Datei, der zugehörige Programmcode, verwendete Softwarebibliotheken sowie weitere relevante Dateien liegen der Arbeit in elektronischer Form bei.

7.1.1 Zielsetzung

Das Ziel bei der Umsetzung der Konstruktions- und Ausdünnungsverfahren bestand darin, durch eine programmtechnische Realisierung eine qualitative und quantitative Beurteilung der Arbeitsweise der Konstruktions- und Ausdünnungsverfahren und der mit Hilfe dieser Verfahren konstruierbaren Spanner-Graphen zu erhalten. Eine asymptotisch optimale Umsetzung der entsprechenden Verfahren war nicht erforderlich.

7.1.2 Verwendete Bibliotheken

Das Programm greift auf die Softwarebibliothek *Java Universal Network/Graph (JUNG)* [46] zurück. Die JUNG-Bibliothek ist eine freie Softwarebibliothek und vereinfacht die Modellierung, Analyse und Visualisierung von Daten, die als Graphen oder Netzwerke dargestellt werden können. Eine Beschreibung dieser Bibliothek und deren Eigenschaften findet sich z. B. in der Arbeit von Madadhain *et al.* [50].

Die JUNG-Bibliothek greift selbst auf drei weitere Bibliotheken zurück: Die erste ist die *Commons Collections*-Bibliothek [3], welche die `Java`-API für Sammlungen von Objekten erweitert. Die zweite Bibliothek ist die *Colt*-Bibliothek [31], welche aus einer Reihe von kleineren Bibliotheken besteht, die für wissenschaftliche und technische Berechnungen genutzt werden können. Die dritte Bibliothek ist die *Xerces*-Bibliothek [4]. Diese stellt Funktionen für das Parsen einiger Datenstrukturen der JUNG-Bibliothek in das XML-Format [58] und vice versa bereit.

Sowohl für den Export von erstellten Graphen in das `EPS`- bzw. `JPG`-Format als auch für den Aufbau der Menüstruktur des Programms `GraphAnalyser` wurde in Teilen zusätzlich zu den oben angeführten Bibliotheken von dritter Seite erstellter Programmcode übernommen bzw. adaptiert. Auf diese übernommenen bzw. adaptierten Programmcode-Fragmente wird jeweils explizit in den jeweiligen Kopfzeilen der Programmcode-Dateien hingewiesen.

7.2 Implementierungsdetails

In diesem Abschnitt sollen kurz einige Details der Implementierung der Algorithmen beschrieben werden. Dazu wird zunächst auf die Datenstrukturen eingegangen, die diesen Implementierungen zugrunde liegen.

7.2.1 Datenstrukturen

Die JUNG-Bibliothek stellt unter anderem Datenstrukturen zur Speicherung von Graphen einschließlich deren Ecken und Kanten bereit. Zudem kann mit Hilfe sogenannter *Prädikate* (*predicates*) die Konsistenz der Graphen, Ecken und Kanten überprüft bzw. aufrecht erhalten werden.[1]

Im Kontext der JUNG-Bibliothek ist ein Prädikat ein Ausdruck, der bei Anwendung auf ein Objekt entweder *wahr* oder *falsch* ist. Informell beschrieben könnte ein Prädikat für Kanten von der Form „`e.isDirected()`“ sein, wobei `e` in diesem Fall das entsprechende an das Prädikat übergebene Objekt ist. In diesem Fall würde eine Kante anhand des Prädikats also daraufhin überprüft werden, ob sie gerichtet ist.

[1] Mit Hilfe dieser Prädikate wird z. B. sichergestellt, dass in einem ungerichteten Graphen keine gerichteten Kanten gespeichert werden können.

Für die Repräsentation geometrischer bzw. euklidischer Graphen wurde die Datenstruktur der JUNG-Bibliothek für ungerichtete Graphen um die entsprechenden geometrischen Informationen für die Ecken erweitert. Die Länge einer Kante eines euklidischen Graphen wird dabei implizit durch die geometrischen Informationen ihrer beiden Endecken gespeichert. Die Konsistenz geometrischer bzw. euklidischer Graphen wird dabei ebenfalls mit Hilfe spezieller Prädikate sichergestellt.

7.2.2 Algorithmen

Alle in diesem Abschnitt vorgestellten Implementierungen operieren auf Punktmengen aus dem $\mathbb{R}^2$.

Fairer Aufteilungsbaum

Die Implementierung zur Berechnung eines fairen Aufteilungsbaums einer endlichen Punktmenge P aus dem $\mathbb{R}^2$ orientiert sich an dem asymptotisch suboptimalen Algorithmus aus Abschnitt 4.2.1: Gilt $|P| = 1$, so wird der aus einem Knoten bestehende faire Aufteilungsbaum T konstruiert. Ansonsten wird in einem ersten Schritt das Begrenzungsrechteck $R(P)$ bestimmt. Dieses Rechteck wird anschließend durch diejenige Hyperebene in zwei geometrisch gleich große Hälften R_1 und R_2 aufgeteilt, welche senkrecht zur $i_{\max}(P)$-ten Koordinatenachse steht. Für die beiden Punktmengen $P_1 = P \cap R_1$ und $P_2 = P \cap R_2$ werden danach rekursiv zwei faire Aufteilungsbäume T_1 und T_2 bestimmt. Der durch die Implementierung berechnete faire Aufteilungsbaum T besteht dann aus einem Wurzelknoten v und den beiden in v gewurzelten Bäumen T_1 und T_2.

Well-Separated Pair Decomposition

Die Implementierung zur Berechnung einer WSPD $\mathcal{D} = (T, \mathcal{R})$ einer endlichen Punktmenge $P \subset \mathbb{R}^2$ bzgl. eines Wertes $s \in \mathbb{R}^+$ stimmt weitgehend mit dem in Abschnitt 4.2 vorgestellten Algorithmus überein: Es wird zunächst ein fairer Aufteilungsbaum T der Punktmenge P berechnet, mit dessen Hilfe anschließend eine bzgl. s scharf getrennte Realisation $\mathcal{R}$ von $P \otimes P$ bestimmt wird. Analog zu Callahan [25] wird bei der Berechnung der Realisation auf eine alternative Definition des Begriffs „scharf getrennt“ zurückgegriffen. Nach dieser alternativen Definition sind zwei Punktmengen $A \subset \mathbb{R}^2$ und $B \subset \mathbb{R}^2$ *scharf getrennt* bzgl. eines Wertes $s \in \mathbb{R}^+$, wenn

$$d(a, b) - (r_1 + r_2) > s \cdot \max(r_1, r_2)$$

gilt. Dabei ist a bzw. b das Zentrum von $R(A)$ bzw. $R(B)$ und r_1 bzw. r_2 der Radius der kleinsten 2-Kugel mit Mittelpunkt a bzw. b, welche $R(A)$ bzw. $R(B)$ enthält, vgl. Abbildung 7.1.

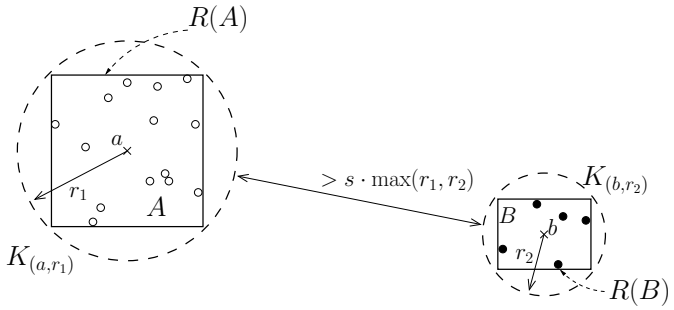


Abbildung 7.1: Alternative Definition des Begriffs „scharf getrennt“

Wie Callahan bemerkt, sind zwei Punktmengen, welche nach dieser neuen Definition scharf getrennt sind, ebenfalls nach der alten Definition dieses Begriffs scharf getrennt. Die mittels dieser alternativen Definition berechneten Realisationen erfüllen dabei dieselben (gewünschten) Eigenschaften wie die Realisationen, welche mittels der ursprünglichen Definition entstehen, vgl. Callahan [25].

Dilatation

Für die Implementierung des Ausdünnungsverfahrens wird ein Verfahren benötigt, welches die Dilatation eines gegebenen Spanner-Graphen bestimmt. Dieses Verfahren ist wie folgt aufgebaut: Um die Dilatation eines gegebenen Graphen $G = (S, E)$ zu bestimmen, wird für jedes Paar $p, q \in S$ verschiedener Ecken der kürzeste Weg zwischen p und q bestimmt und mit Hilfe der Länge $\delta_G(p, q)$ dieses Weges und dem Euklidischen Abstand $d(p, q)$ beider Ecken eine Variable t aktualisiert: Der initiale Wert von t beträgt (negativ) unendlich. Gilt $\delta_G(p, q)/d(p, q) > t$, so erhält t den (neuen) Wert $\delta_G(p, q)/d(p, q)$.

Nach diesem verschachtelten Durchlauf ist in der Variablen t die Dilatation des Graphen gespeichert. Für die Berechnung des kürzesten Weges zwischen zwei voneinander verschiedenen Ecken aus S wird dabei auf die Implementierung des Dijkstra-Algorithmus [32] der JUNG-Bibliothek zurückgegriffen.

Konstruktionsalgorithmus

Die Implementierung zur Konstruktion eines dünn besetzten $(1 + \varepsilon)$-Spanners ($\varepsilon > 0$) für eine endliche Punktmenge $S \subset \mathbb{R}^2$ orientiert sich an den in Kapitel 5 vorgestellten Konstruktionsverfahren: Im ersten Schritt wird anhand der obigen Implementierung eine WSPD $\mathcal{D} = (T, \mathcal{R})$ von S bzgl. des Wertes $s = \frac{8+4\varepsilon}{\varepsilon}$ berechnet. Im zweiten Schritt wird dann für jedes Paar $\{A_i, B_i\}$ der Realisation $\mathcal{R} = \{\{A_1, B_1\}, \ldots, \{A_k, B_k\}\}$ ein (beliebiges)

Paar $\{a_i, b_i\}$ mit $a_i \in A_i$ und $b_i \in B_i$ der Kantenmengen des zu berechnenden Graphen hinzugefügt.[2]

Ausdünnungsalgorithmus

Der Aufbau der Implementierung zur Ausdünnung eines gegebenen t-Spanners $G = (S, E)$ ($t \geq 1$), d.h. zur Konstruktion eines dünn besetzten $(1+\varepsilon)$-Spanners ($\varepsilon > 0$) $G' = (S, E')$ von G, entspricht dem in Kapitel 6 vorgestellten Algorithmus 6.1: In einem ersten Schritt wird mittels der obigen Implementierung die Dilatation von G bestimmt. Anschließend wird eine WSPD $\mathcal{D} = (T, \mathcal{R})$ von S bzgl. des Wertes $s = \frac{1}{\varepsilon}((1+\varepsilon)(8t+4)+4)$ berechnet. Für jedes Paar $\{A_i, B_i\}$ der Realisation $\mathcal{R} = \{\{A_1, B_1\}, \ldots, \{A_k, B_k\}\}$ wird hiernach ein Paar $\{a_i, b_i\}$ mit $a_i \in A_i$ und $b_i \in B_i$ ausgewählt und die Kantenmenge E traversiert. Erfüllt dabei eine Kante $e \in E$ die Bedingung (i) oder (ii) aus Abschnitt 6.1, so wird die Kante e einer globalen Menge hinzugefügt, der aktuelle Durchlauf der Kantenmenge E abgebrochen und mit dem nächsten Paar der Realisation fortgefahren. Die nach dieser verschachtelten Traversierung in der globalen Menge gespeicherten Kanten entsprechen dann den Kanten des ausgedünnten Graphen.

7.3 Programm

Das Programm `GraphAnalyser` kann auf jedem System ausgeführt werden, auf dem eine *Java-Laufzeitumgebung* (JRE) der Version 5.0 installiert ist. Ein Programmaufruf in der Kommandozeile des Systems hat im Allgemeinen die Form `java -jar GraphAnalyser.jar`.

7.3.1 Funktionsumfang

In Abbildung 7.2 wird ein Bildschirmfoto des Programms `GraphAnalyser` gegeben. Mit Hilfe des Programms können Graphen (automatisch) erzeugt, bearbeitet, gespeichert und geladen werden. Das Hauptaugenmerk des Programms liegt dabei in der Anwendung der oben beschriebenen Algorithmen auf entsprechende Graphen.

Erzeugung von Graphen

Das Programm `GraphAnalyser` stellt zwei verschiedene Möglichkeiten zur Erzeugung von neuen Graphen bereit: Zum Einen können Graphen „per Hand“ erstellt werden, d.h. neue Graphen können durch sukzessives Hinzufügen von Ecken und Kanten konstruiert werden. Zum Anderen können neue Graphen mit Hilfe von *Generatoren* automatisch erzeugt wer-

[2] Entsprechende Repräsentanten sind dabei in den jeweiligen Objekten gespeichert. Weiterhin wird der Implementierung ein Parameter $t = 1 + \varepsilon$ mit $\varepsilon > 0$ übergeben und anschließend eine WSPD bzgl. $s = 4 \cdot \frac{t+1}{t-1} = \frac{8+4\varepsilon}{\varepsilon}$ berechnet.

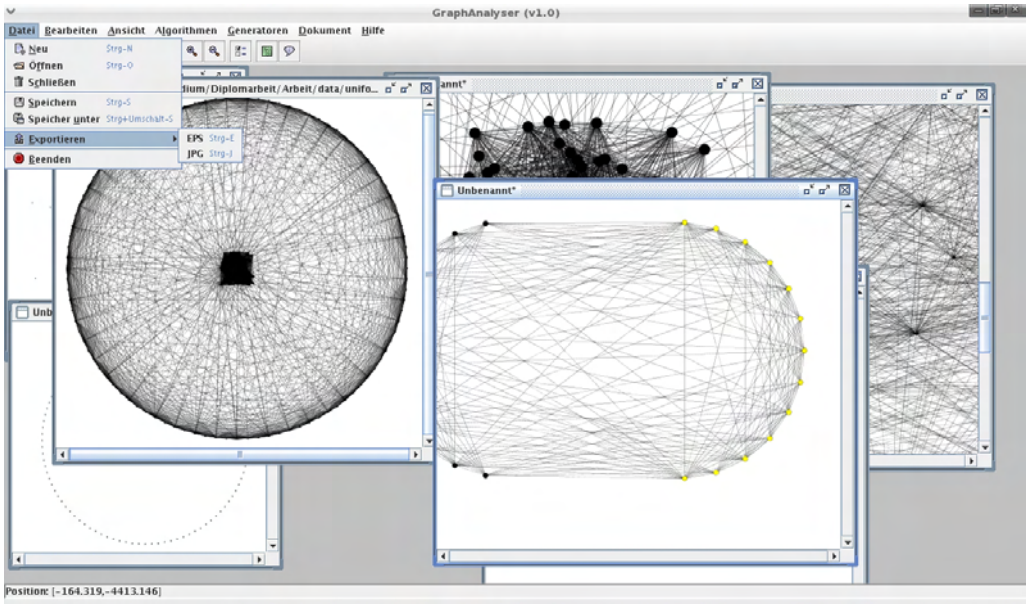


Abbildung 7.2: Bildschirmfoto des Programms

den. Das Programm stellt dabei zwei Generatoren zur automatischen Konstruktion von Graphen bereit, deren Ecken

- gleichmäßig verteilt auf dem Rand eines vorgegebenen Kreises liegen, bzw.
- zufallsbasiert verteilt in einem vorgegebenen Rechteck liegen.

Zudem kann bei beiden Generatoren durch einen Parameter bestimmt werden, wie viele Kanten der zu konstruierende Graph besitzen soll. Die entsprechende Menge an Kanten wird dann automatisch und zufallsbasiert vom Generator erzeugt.

Ein- und Ausgabe

Erzeugte Graphen können geladen und gespeichert werden. Das Programm greift dabei auf das `GraphML`-Format [41] zurück, welches ein XML-Format zur Speicherung von Graphen ist.

Algorithmen

Auf einen erzeugten Graphen können die oben beschriebenen Algorithmen angewendet werden. Dabei kann ein Graph analysiert (Fairer Aufteilungsbaum, WSPD, Dilatation) und ausgedünnt (Ausdünnungsalgorithmus) oder aus einer Menge von Punkten ein (dünn besetzter) Spanner-Graph konstruiert (Konstruktionsalgorithmus) werden.

Export von Ergebnissen

Für den Export von (grafischen) Ergebnissen stellt das Programm `GraphAnalyser` zwei Formate zur Verfügung: Das `EPS`- und das `JPG`-Format.

Mit Hilfe des `EPS`-Formats können die Visualisierungen von Graphen als Vektorgrafik gespeichert werden. Die anhand dieses Formats erzeugten Grafiken können ohne Qualitätsverlust stufenlos skaliert werden, jedoch gehen grafische Effekte der Visualisierungen der Graphen wie z. B. Antialiasing verloren.

Im Gegensatz zum `EPS`-Format werden beim Export von Graphen mittels des `JPG`-Formats die grafischen Effekte der Visualisierungen zwar erfasst, jedoch hängt die Qualität der Grafik von der gewählten Auflösung ab.

7.3.2 Bedienung

Die Bedienung des Programms `GraphAnalyser` ist weitgehend intuitiv und soll hier nicht besprochen werden. Eine Bedienungsanleitung für die grundlegenden Funktionen des Programms findet sich unter dem Menüpunkt `Hilfe->Erste Schritte`.

7.4 Ergebnisse

In diesem Abschnitt sollen einige Spanner-Graphen genauer untersucht werden, welche mit Hilfe des Programms `GraphAnalyser` konstruiert wurden. Weitere Beispiele sind im Anhang dieser Arbeit zu finden.

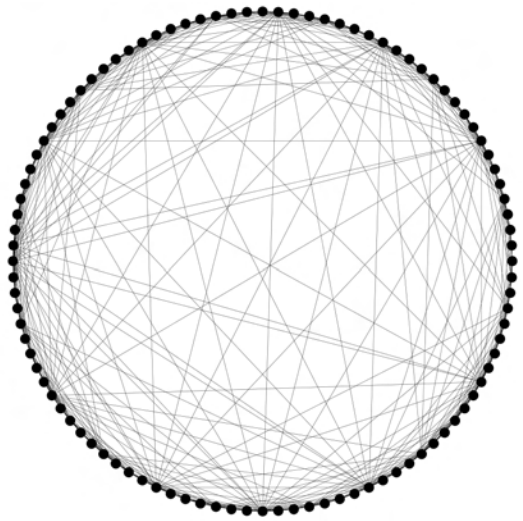
7.4.1 Konstruktion von Spanner-Graphen

In den Abbildungen 7.3 (a)–(d) werden vier Spanner-Graphen gezeigt, die durch Anwendung des Konstruktionsalgorithmus auf dieselbe Punktmenge jedoch mit verschiedenen Parametern $t > 1$ (maximal erlaubte Dilatation des erzeugten Spanner-Graphen) erzeugt wurden. Die Punktmenge $S \subset \mathbb{R}^2$ besteht dabei aus 100 Punkten, die gleichmäßig auf dem Rand eines Kreises angeordnet sind. In Abbildung 7.4 (a) bzw. 7.4 (b) wird die zugehörige Partitionierung der Punktmenge bzw. der zugehörige faire Aufteilungsbaum der Punktmenge dargestellt, welche bzw. welcher durch Anwendung der Implementierung zur Konstruktion eines fairen Aufteilungsbaums entsteht.

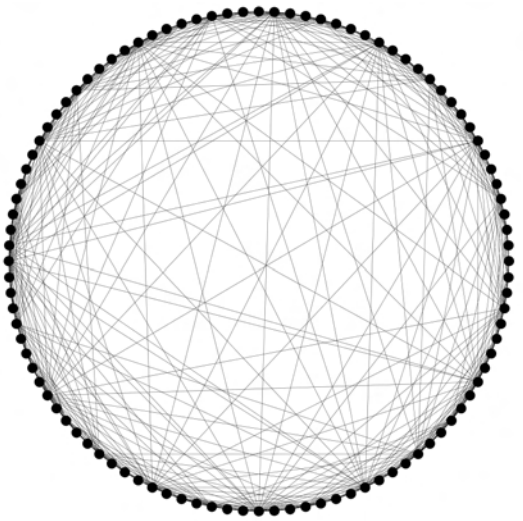
7.4.2 Ausdünnung von Spanner-Graphen

In den Abbildungen 7.5 (a)–(e) wird die Anwendung des Ausdünnungsalgorithmus auf einen vollständigen Graphen für eine Punktmenge $S \subset \mathbb{R}^2$ dargestellt. Der vollständige Graph besitzt 780 Kanten. Die Anwendung des Ausdünnungsalgorithmus auf den vollständigen Graphen (Abbildung 7.5 (b)) mit Parameter $\varepsilon = 4$ liefert z. B. einen Spanner-Graphen, welcher 319 Kanten und eine Dilatation von $\triangle(G) \approx 1.181$ besitzt (Abbildung

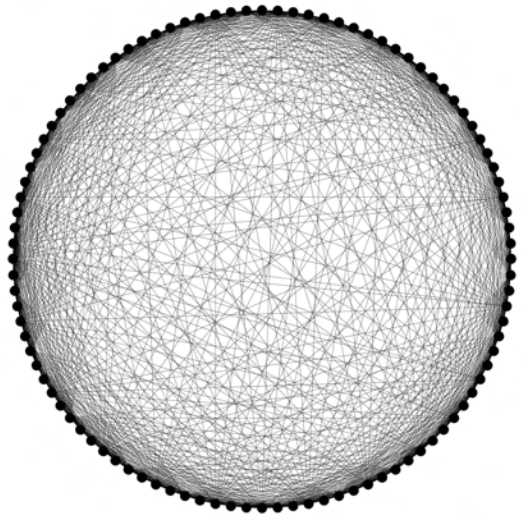
7.5 (d)). Der ausgedünnte Spanner-Graph besitzt also (nur) $\approx 40,1\%$ der ursprünglichen Kanten.



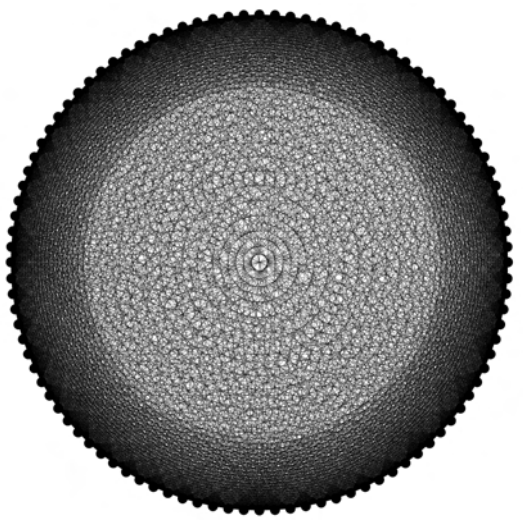
(a) $t = 1000$, $|S| = 100$, $|E| = 533$, $\triangle(G) \approx 1.304$



(b) $t = 10$, $|S| = 100$, $|E| = 565$, $\triangle(G) \approx 1.287$

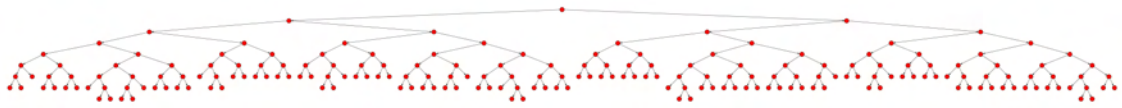


(c) $t = 2$, $|S| = 100$, $|E| = 1249$, $\triangle(G) \approx 1.164$

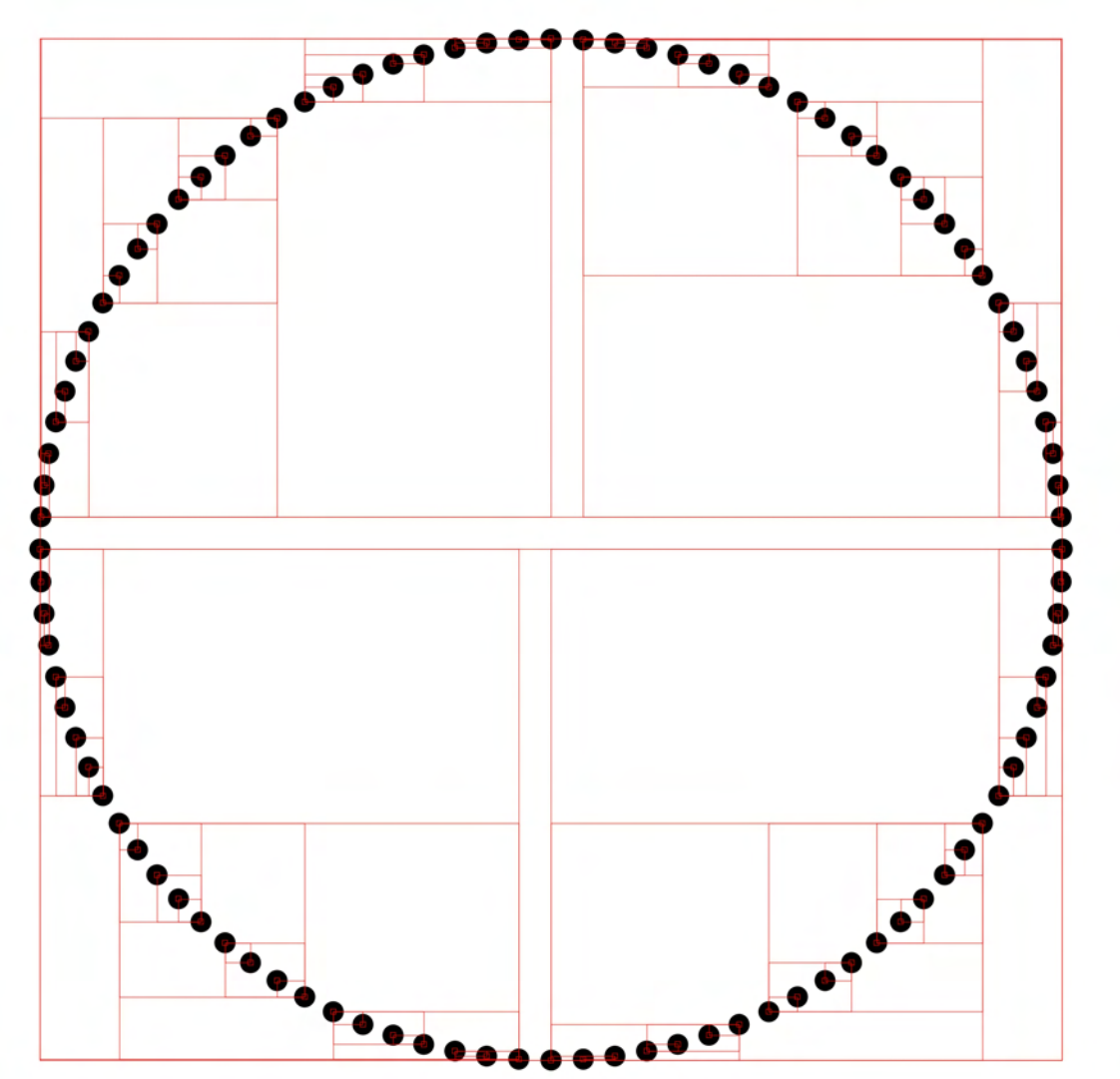


(d) $t = 1.2$, $|S| = 100$, $|E| = 3251$, $\triangle(G) \approx 1.065$

Abbildung 7.3: Die Eckenmenge S der obigen Spanner-Graphen besteht jeweils aus 100 Punkten, die gleichmäßig auf dem Rand eines Kreises verteilt liegen. Die Kantenanzahl wie auch die Dilatation der jeweiligen Spanner-Graphen variieren bei unterschiedlicher Wahl des Parameters $t > 1$.



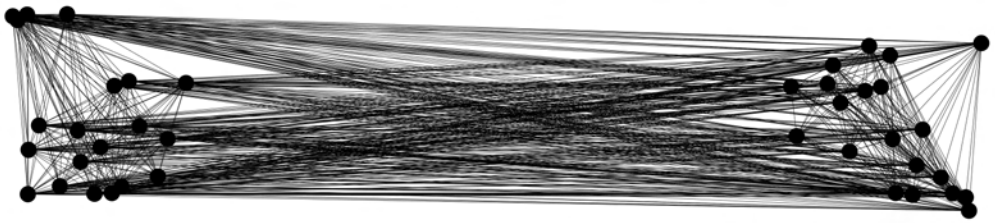
(a)



(b)

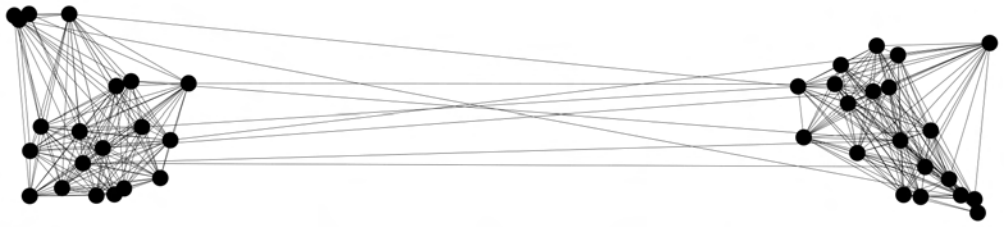
Abbildung 7.4: Ein zu der Punktmenge aus den Abbildungen 7.3 (a)–(d) gehörender fairer Aufteilungsbaum (a) und die entsprechende Partitionierung der Punktmenge (b).

(a) Punktmenge $S \subset \mathbb{R}^2$ mit $|S| = 40$ Punkten

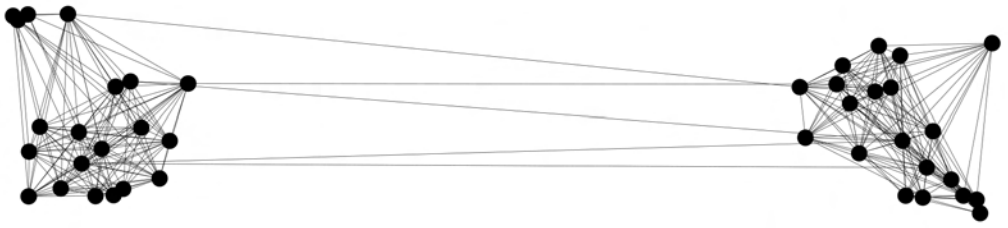


(b) Der vollständige Graph für S: $|S| = 40$, $|E| = 780$, $\triangle(G) = 1.0$

(c) Anwendung des Ausdünnungsalgorithmus auf den vollständigen Graphen mit Parameter $\varepsilon = 1$: $|S| = 40$, $|E| = 396$, $\triangle(G) \approx 1.108$



(d) Anwendung des Ausdünnungsalgorithmus auf den vollständigen Graphen mit Parameter $\varepsilon = 4$: $|S| = 40$, $|E| = 319$, $\triangle(G) \approx 1.181$



(e) Anwendung des Ausdünnungsalgorithmus auf den vollständigen Graphen mit Parameter $\varepsilon = 1000$: $|S| = 40$, $|E| = 287$, $\triangle(G) \approx 1.250$

Abbildung 7.5: Anwendung des Ausdünnungsalgorithmus auf einen vollständigen Graphen

Kapitel 8

Zusammenfassung und Ausblick

8.1 Zusammenfassung

Das Hauptanliegen der Arbeit war die Übertragung der im RAM- bzw. I/O-Modell formulierten Algorithmen zur Konstruktion dünn besetzter bzw. zur Ausdünnung dichter Spanner-Graphen in das *cache-oblivious*-Modell. Die entsprechenden Komponenten der einzelnen Algorithmen sollten dazu so realisiert werden, dass sie im Kontext des *cache-oblivious*-Modells eine möglichst gute asymptotische Komplexität besitzen. Zur Veranschaulichung der Arbeitsweise der Konstruktions- und Ausdünnungsalgorithmen sollte zudem eine programmtechnische Realisierung durchgeführt werden, der die Konstruktions- und Ausdünnungsverfahren zugrunde liegen.

Im Folgenden werden die Inhalte und Ergebnisse dieser Arbeit kurz zusammengefasst:

- In Kapitel 2 wurde der konzeptionelle Aufbau der Speicherhierarchien moderner Computerarchitekturen behandelt. Dabei wurde begründet, weshalb bei Bearbeitung großer Datenmengen der Transfer von Daten und nicht die Anzahl der von der CPU ausgeführten Operationen für die tatsächliche Laufzeit eines Algorithmus verantwortlich sein kann.

  Im Gegensatz zum RAM-Modell erfassen sowohl das I/O- als auch das *cache-oblivious*-Modell diese Problematik beim Umgang mit massiven Datenmengen, d. h. sie berücksichtigen im Gegensatz zum RAM-Modell die charakteristischen Eigenschaften moderner Speicherhierarchien. Zusätzlich zu einer Beschreibung beider Modelle wurde in Kapitel 2 auf die Vorteile des *cache-oblivious*-Modells im Vergleich zum I/O-Modell und zu anderen Mehrspeichermodellen eingegangen.

- Diese Vorteile des *cache-oblivious*-Modells gegenüber anderen Mehrspeichermodellen führten dazu, dass sich viele Autoren mit im Kontext dieses Modells effizienten Algorithmen und Datenstrukturen beschäftigt haben.

In Kapitel 3 wurden einige dieser Algorithmen und Datenstrukturen, die im Rahmen der Arbeit von Bedeutung waren, ausführlich beschrieben und analysiert. Zudem wurde auf allgemeine Entwurfs- und Analysetechniken eingegangen, die im Kontext des *cache-oblivious*-Modells häufig Anwendung finden.

- Das Ergebnis der Überlegungen in Kapitel 4 stellt ein im Kontext des *cache-oblivious*-Modells effizientes Verfahren dar, mit dessen Hilfe für eine endliche nicht-leere Punktmenge aus dem $\mathbb{R}^d$ eine WSPD linearer Größe berechnet werden kann. In Hinblick auf dieses effiziente Verfahren wurden zu Beginn des Kapitels 4 formale Grundlagen geschaffen und der von Callahan und Kosaraju [25, 27] entwickelte und im RAM-Modell effiziente Algorithmus beschrieben und analysiert.

  Der überwiegende Teil dieses Kapitels beschäftigte sich mit der Übertragung des I/O-effizienten Verfahrens von Govindarajan *et al.* [40] in das *cache-oblivious*-Modell. Weite Teile des I/O-effizienten Verfahrens konnten dabei ohne Veränderung übernommen werden. Einige Änderungen mussten jedoch an denjenigen Stellen vorgenommen werden, an denen die im I/O-Modell bekannten Belegungen der Parameter M und B verwendet wurden bzw. an denen Datenstrukturen eingesetzt wurden, zu denen (noch) kein entsprechendes Pendant im *cache-oblivious*-Modell existiert.

- Alle in Kapitel 4 vorgestellten Algorithmen zur Berechnung einer WSPD berechnen eine WSPD linearer Größe. Der Nutzen einer linearen Größe dieser Datenstruktur begründet sich darin, dass zur Lösung einiger geometrischer Probleme jeweils eine konstante Anzahl von Berechnungen auf einem Paar $\{A_i, B_i\}$ einer Realisation für die zu untersuchende Punktmenge aus dem $\mathbb{R}^d$ ausreicht, und nicht $|A_i| \cdot |B_i|$ Berechnungen auf der Menge $A_i \otimes B_i$ durchgeführt werden müssen.

  Eines dieser geometrischen Probleme ist die Konstruktion von dünn besetzten $(1+\varepsilon)$-Spannern ($\varepsilon > 0$) für endliche Punktmengen aus dem $\mathbb{R}^d$. Dieses Problem wurde ausführlich in Kapitel 5 diskutiert. Ergebnis der Überlegungen in diesem Kapitel war ein effizientes *cache-oblivious*-Verfahren zur Konstruktion eines solchen Spanner-Graphen, welches sich direkt aus dem effizienten *cache-oblivious*-Verfahren zur Berechnung einer WSPD ergab.

- Ein mit der Konstruktion von dünn besetzten $(1 + \varepsilon)$-Spannern verwandtes Problem stellt die Ausdünnung von dichten Spanner-Graphen dar. Dieses Problem wurde in Kapitel 6 thematisiert. Der größte Teil dieses Kapitels befasste sich dabei mit der Übertragung des I/O-effizienten Verfahrens von Gudmundsson und Vahrenhold [45] in das *cache-oblivious*-Modell. Die algorithmischen Bausteine des I/O-effizienten Algorithmus konnten dazu weitgehend übernommen werden. Einer gesonderten Betrachtung musste man lediglich die Bearbeitung von speziellen Bereichsanfragen unterziehen.

Wie auch bei der Konstruktion von dünn besetzten Spanner-Graphen erwies sich die Berechnung der WSPD bei der Ausdünnung dichter Spanner-Graphen als wesentlicher Bestandteil aller in Kapitel 6 vorgestellten Algorithmen.

- Zur Veranschaulichung der Arbeitsweise der Konstruktions- und Ausdünnungsalgorithmen wurde eine programmtechnische Realisierung umgesetzt. Diese Realisierung wurde in Kapitel 7 kurz beschrieben. Dabei wurde neben der Beschreibung einiger Details bei der Implementierung relevanter Algorithmen auch auf die Programmfunktionen des Programms und auf einige mittels der implementierten Konstruktions- und Ausdünnungsalgorithmen erhaltenen Ergebnisse eingegangen.

8.2 Ausblick

Abschließend sollen kurz zwei mögliche Themenbereiche skizziert werden, die durch die Ergebnisse dieser Arbeit motiviert werden:

- Wie viele Arbeiten [25, 36, 37, 43, 48] zeigen, erweist sich bei der Bearbeitung von geometrischen Problemen die WSPD-Datenstruktur häufig als Schlüssel zur effizienten Lösung dieser Probleme. Da die meisten dieser Probleme bisher nur im Kontext des RAM- bzw. I/O-Modells betrachtet wurden, führt das in dieser Arbeit vorgestellte effiziente *cache-oblivious*-Verfahren zur Berechnung einer WSPD linearer Größe direkt zu der Frage, ob sich die im RAM- bzw. I/O-Modell vorhandenen Algorithmen zur Lösung dieser Probleme in das *cache-oblivious*-Modell übertragen lassen.

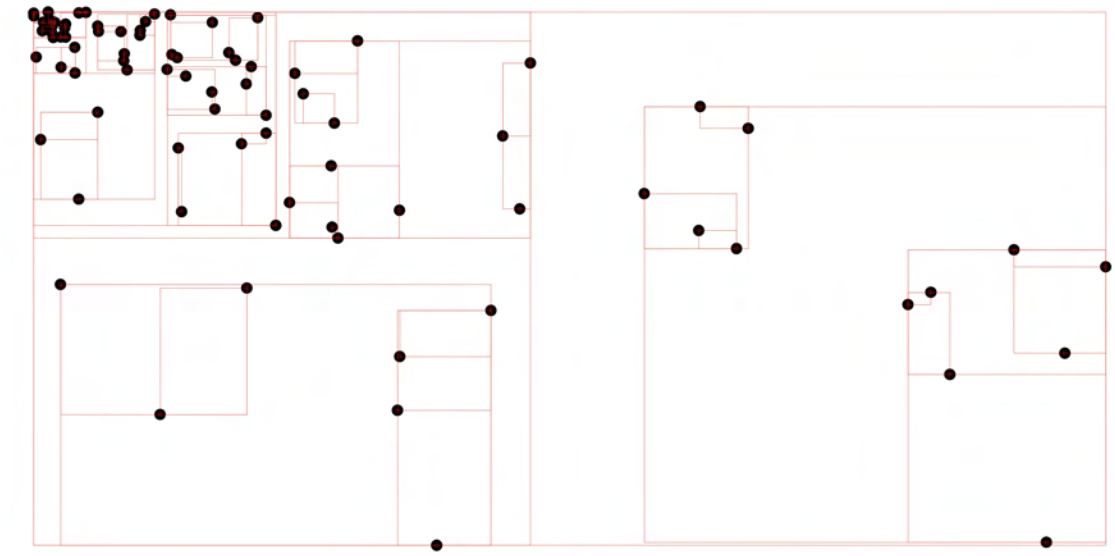
- Ein weiterer Themenbereich ergibt sich aus den Ergebnissen des Kapitels 6. Bei der Übertragung des I/O-effizienten Verfahrens von Gudmundsson und Vahrenhold [45] zur Ausdünnung eines Spanner-Graphen wurde die Bearbeitung von (speziellen) orthogonalen Bereichsanfragen ausführlich diskutiert. Das effiziente *cache-oblivious*-Verfahren zur Bearbeitung der speziellen orthogonalen Bereichsanfragen war dabei Bestandteil des am Ende des Kapitels 6 vorgestellten *cache-oblivious*-Ausdünnungsalgorithmus: In Schritt 7 des Algorithmus 6.3 wurde für jedes Anfragerechteck $R \in \mathcal{Q}$ mit $R \cap \mathcal{E} \neq \emptyset$ genau *ein* Punkt $M_{(p,q)}$ aus $R \cap \mathcal{E}$ ausgewählt.

  Diese spezielle Auswahl scheint auf den ersten Blick nötig zu sein, da bei Verwendung des ursprünglichen Algorithmus zur Bearbeitung der Bereichsanfragen, welcher für jedes Rechteck $R \in \mathcal{Q}$ *alle* Punkt-Rechteck-Kombinationen (R, p) mit $p \in R \cap \mathcal{E}$ ausgibt, die Mächtigkeit der Ausgabemenge solcher Anfragen i. A. nicht besser als durch die Schranke $\mathcal{O}(|\mathcal{Q}| \cdot |\mathcal{E}|)$ begrenzt werden kann. Bei genauerer Betrachtung der Menge $\mathcal{Q}$ stellt sich jedoch heraus, dass aufgrund der Eigenschaften der WSPD die Rechtecke aus $\mathcal{Q}$ disjunkt sein müssen. Demnach kann die Ausgabemenge auch bei Verwendung des ursprünglichen Algorithmus höchstens eine Mächtigkeit von $\mathcal{O}(|\mathcal{E}|)$ besitzen.

Die Verwendung des speziellen Algorithmus zur Bearbeitung der Bereichsanfragen ist also in diesem Fall nicht zwingend erforderlich, kann jedoch als Vorbereitung auf die Frage gesehen werden, ob die Komplexität des Ausdünnungsalgorithmus im Kontext des *cache-oblivious*-Modells zu $\mathcal{O}(scan(|E|) + sort(|S|))$ Speichertransfers verbessert werden kann, vgl. auch Gudmundsson und Vahrenhold [45].

Anhang A

Weitere Beispiele



(a) Partitionierung

(b) Fairer Aufteilungsbaum

Abbildung A.1: In Abbildung (a) wird die Partitionierung einer Punktmenge in der Ebene dargestellt. Der zugehörige faire Aufteilungsbaum wird in Abbildung (b) gezeigt.

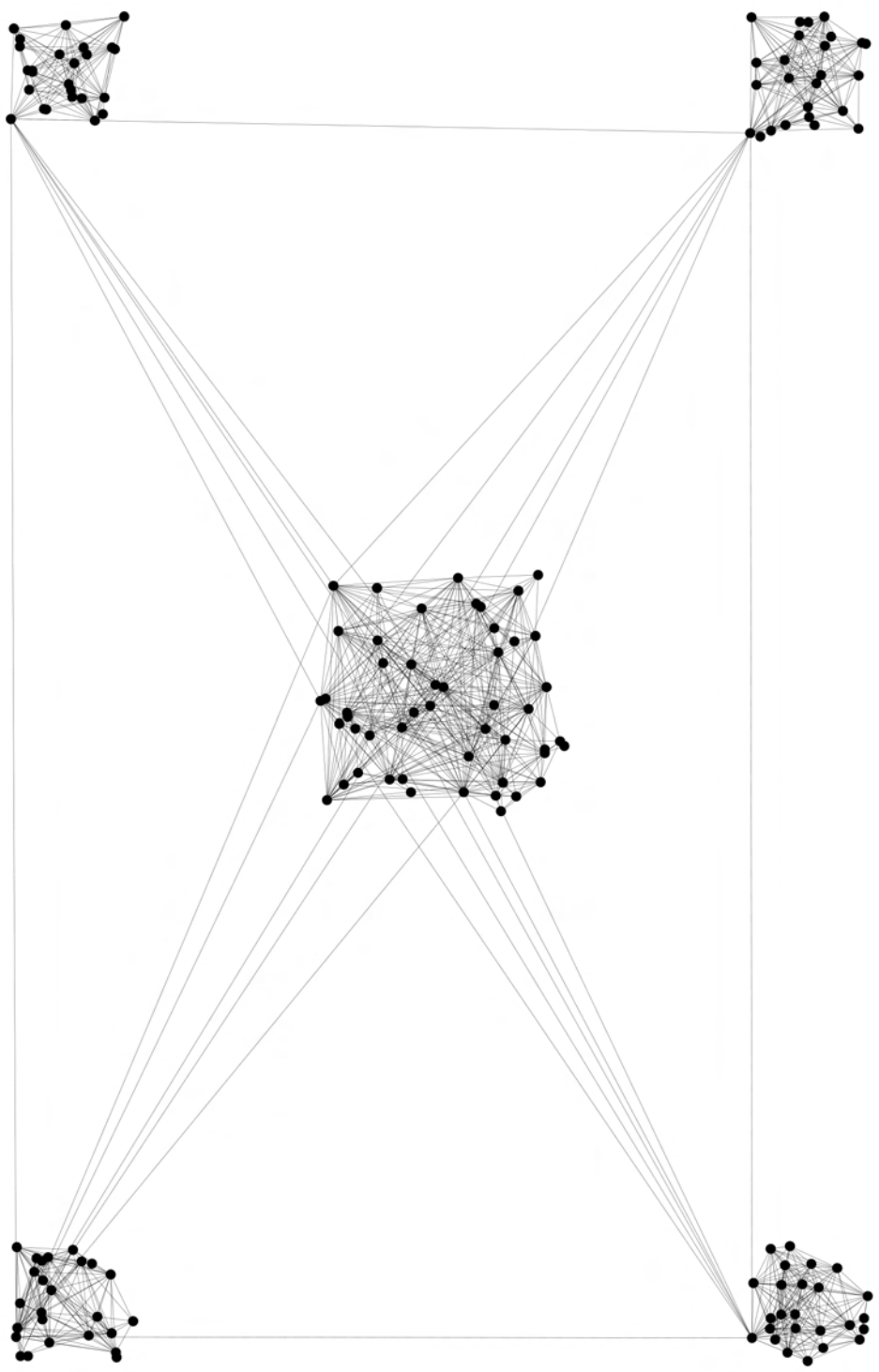
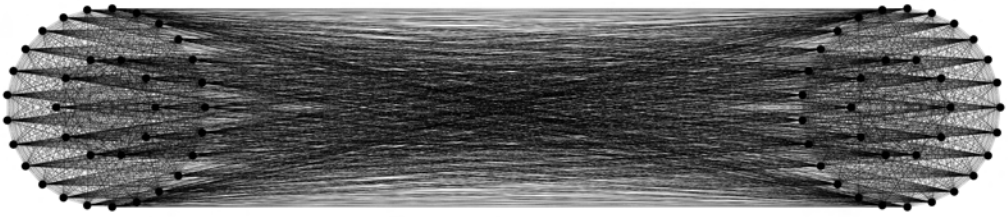
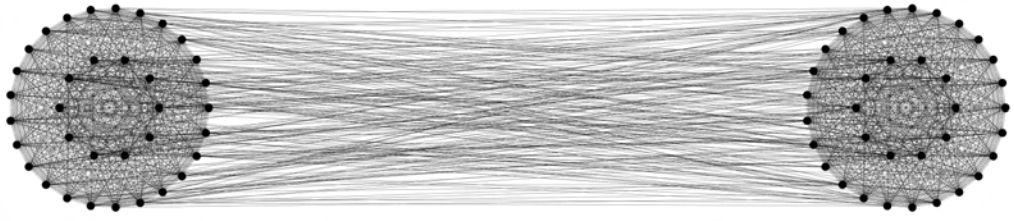


Abbildung A.2: Die Abbildung zeigt einen dünn besetzten Spanner-Graphen für eine Punktmenge in der Ebene. Der Spanner-Graph besitzt 150 Ecken und 1027 Kanten. Die Dilatation beträgt $\triangle(G) \approx 1.759$. Die Kantenmenge des vollständigen Graphen für diese Eckenmenge besteht aus $\binom{150}{2} = 11175$ Kanten. Der dünn besetzte Spanner-Graph besitzt somit nur $\approx 9.2\%$ der Kanten des vollständigen Graphen.

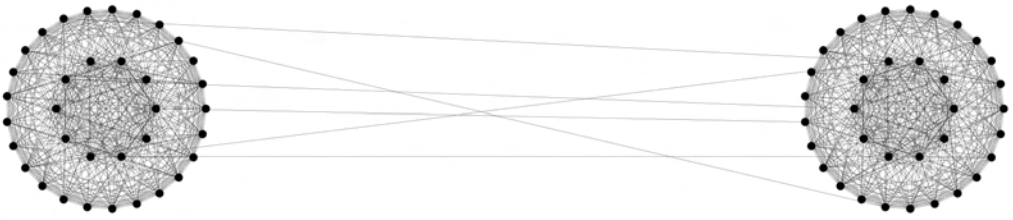
(a) Punktmenge $S \subset \mathbb{R}^2$ mit $|S| = 70$ Punkten



(b) Der vollständige Graph für S: $|S| = 70$, $|E| = 2415$, $\triangle(G) = 1.0$



(c) Anwendung des Ausdünnungsalgorithmus auf den vollständigen Graphen mit Parameter $\varepsilon = 0.5$: $|S| = 70$, $|E| = 1416$, $\triangle(G) \approx 1.010$



(d) Anwendung des Ausdünnungsalgorithmus auf den vollständigen Graphen mit Parameter $\varepsilon = 100$: $|S| = 70$, $|E| = 950$, $\triangle(G) \approx 1.248$

Abbildung A.3: Anwendung des Ausdünnungsalgorithmus auf einen vollständigen Graphen

Abbildungsverzeichnis

1.1 Der Entwurf eines euklidischen Graphen einer Punktmenge in der Ebene als vollständiger Graph und als Spannbaum. 2
1.2 Ein Graph G und zwei Teilgraphen G' und G'' von G, vgl. Diestel [35] . . . 6
1.3 Ein Weg P und ein Kreis C in G 7
1.4 Ein Spannbaum bzw. Spanner-Graph eines Graphen 8

2.1 Typische Speicherhierarchie 10
2.2 Inklusionseigenschaft von Speicherhierarchien 11
2.3 Das RAM-Modell 13
2.4 Das I/O-Modell von Aggarwal und Vitter [2, 34] 15
2.5 Das *cache-oblivious*-Modell von Frigo *et al.* [38, 53] 17

3.1 Das *van Emde Boas layout* 23
3.2 Traversierung von Datenelementen im *cache-oblivious*-Modell 26
3.3 k-Verschmelzer als Black Box 29
3.4 Verschmelzungsbaum und k-Verschmelzer 34
3.5 Rekursive Definition der Puffergrößen und Speicherplatzbelegung eines k-Verschmelzers 35
3.6 Die *funnel heap*-Prioritätswarteschlange 42

4.1 Zwei bzgl. eines reellen Wertes $s > 0$ scharf getrennte Punktmengen A und B in der Ebene, vgl. [25] 55
4.2 Zwei Beispiele für faire Aufteilungen von Punktmengen in der Ebene. Die gepunkteten Linien skizzieren die Aufteilungshyperebenen, vgl. [25]. 57
4.3 Darstellung eines fairen Aufteilungsbaums einer Punktmenge in der Ebene . 58
4.4 Zerlegungen zweier Punktmengen in der Ebene mit Hilfe eines *quadtrees* bzw. *kd*-Baums, vgl. [25] 58
4.5 Schematische Darstellung eines durch den Aufteilungsprozess entstehenden unvollständigen fairen Aufteilungsbaums einer Punktmenge in der Ebene . . 62
4.6 Aufteilung des Rechtecks R mittels Fall 1 77
4.7 Aufteilung des Rechtecks R mittels der Fälle 2 und 3 78

4.8 Verteilung von Punkten auf Zellen bzw. Verteilung von Zellen auf Rechtecke und Regionen. 85
4.9 Erweiterung der Tupelliste 86
4.10 Ersetzung einer komprimierten Kante (R, C) durch den Baum $T(R, C)$. . . 88

5.1 Drei euklidische Graphen mit zugehöriger Dilatation 96
5.2 Eine ε-approximative Kante $\{\bar{a}, \bar{b}\}$ zwischen zwei Punktmengen A_i und B_i $(\varepsilon > 0)$ 98
5.3 Ein Weg in G von a nach b mit einer euklidischen Länge von maximal $\frac{1+\varepsilon}{1-4s^{-1}} \cdot d(a, b)$ 100

6.1 „Approximatives“ Paar $\{a_i, b_i\}$ für eine Kante $\{p, q\} \in E$, vgl. [44] 109
6.2 Der Weg von p nach q in G', vgl. [44] 111
6.3 Ausdünnung des Graphen mit Hilfe der WSPD, vgl. [44] 114
6.4 Lösung geometrischer Probleme mit Hilfe des *plane sweep*- bzw. *distribution sweeping*-Paradigmas, vgl. [19] 116
6.5 Verschmelzungsprozess zweier Streifen A und B 118
6.6 Verschmelzung von Streifen mit Hilfe eines k-Verschmelzers 120
6.7 Verschiedene Zustände bei der Verschmelzung zweier Streifen während des ersten Durchlaufs eines veränderten k-Verschmelzers 121
6.8 Abbildung einer Kante (p, q) auf den Punkt $M_{(p,q)}$ 130

7.1 Alternative Definition des Begriffs „scharf getrennt“ 136
7.2 Bildschirmfoto des Programms 138
7.3 Vier Spanner-Graphen, deren Ecken jeweils auf dem Rand eines Kreises verteilt liegen. 141
7.4 Fairer Aufteilungsbaum und Partitionierung der Ebene 142
7.5 Anwendung des Ausdünnungsalgorithmus auf einen vollständigen Graphen . 143

A.1 Partitionierung einer Punktmenge in der Ebene und zugehöriger fairer Aufteilungsbaum 149
A.2 Dünn besetzter Spanner-Graph für eine Punktmenge in der Ebene 150
A.3 Anwendung des Ausdünnungsalgorithmus auf einen vollständigen Graphen . 151

Algorithmenverzeichnis

3.1 Das Zwei-Wege-*Mergesort*-Sortierverfahren, vgl. [51, 53] 28
3.2 Das *lazy funnelsort*-Verfahren, vgl. [20, 54] 30
3.3 Die Verschmelzungsprozedur, vgl. [20, 21] 36
4.1 Berechnung eines fairen Aufteilungsbaums im RAM-Modell, vgl. [25] 63
4.2 Berechnung einer scharf getrennten Realisation im RAM-Modell, vgl. [40, 59] 66
4.3 Berechnung eines fairen Schnittbaums im *cache-oblivious*-Modell, vgl. [40] . 73
4.4 Berechnung eines unvollständigen fairen Schnittbaums im *cache-oblivious*-Modell, vgl. [40] 75
5.1 Konstruktionsalgorithmus im RAM-Modell, vgl. [25, 26] 102
6.1 Ausdünnungsalgorithmus im RAM-Modell, vgl. [43, 44] 113
6.2 Auswahl eines geeigneten Knotens des k-Verschmelzers 123
6.3 Ausdünnungsalgorithmus im *cache-oblivious*-Modell 131

Literaturverzeichnis

[1] Agarwal, Pankaj K., Lars Arge, Andrew Danner und Bryan Holland-Minkley: *Cache-oblivious data structures for orthogonal range searching.* In: *Proceedings of the 19th ACM Symposium on Computational Geometry, June 8-10, 2003, San Diego, CA, USA*, Seiten 237–245, 2003.

[2] Aggarwal, Alok und Jeffrey Scott Vitter: *The Input/Output Complexity of Sorting and Related Problems.* Communications of the ACM, 31(9):1116–1127, 1988.

[3] Apache Jakarta Project: *Commons Collections 3.1.* `http://jakarta.apache.org/commons/collections/`, Dezember 2006. Webseite.

[4] Apache XML Project: *Xerces2 Java Parser.* `http://xerces.apache.org/xerces2-j/`, Dezember 2006. Webseite.

[5] Arge, Lars: *External Memory Data Structures.* In: *Algorithms - ESA 2001, 9th Annual European Symposium, Aarhus, Denmark, August 28-31, 2001, Proceedings*, Band 2161 der Reihe *Lecture Notes in Computer Science*, Seiten 1–29. Springer, 2001.

[6] Arge, Lars: *The Buffer Tree: A Technique for Designing Batched External Data Structures.* Algorithmica, 37(1):1–24, 2003.

[7] Arge, Lars, Michael A. Bender, Erik D. Demaine, Bryan E. Holland-Minkley und J. Ian Munro: *An Optimal Cache-Oblivious Priority Queue and its Application to Graph Algorithms.* SIAM Journal on Computing, 2006. to appear.

[8] Arge, Lars, Gerth Stølting Brodal und Rolf Fagerberg: *Cache-Oblivious Data Structures.* In: Mehta, Dinesh und Sartaj Sahni (Herausgeber): *Handbook of Data Structures and Applications*, Kapitel 34, Seite 27. CRC Press, 2005.

[9] Arge, Lars, Gerth Stølting Brodal, Rolf Fagerberg und Morten Laustsen: *Cache-oblivious planar orthogonal range searching and counting.* In: *Proceedings of the 21st ACM Symposium on Computational Geometry, Pisa, Italy, June 6-8, 2005*, Seiten 160–169, 2005.

[10] Arge, Lars und Peter Bro Miltersen: *On showing lower bounds for external-memory computational geometry problems.* In: Abello, James und Jeffrey Scott Vitter (Herausgeber): *External Memory Algorithms and Visualization*, Seiten 139–160. American Mathematical Society Press, Providence, RI, 1999.

[11] Arge, Lars, Octavian Procopiuc, Sridhar Ramaswamy, Torsten Suel und Jeffrey Scott Vitter: *Theory and Practice of I/O-Efficient Algorithms for Multidimensional Batched Searching Problems (Extended Abstract).* In: *SODA*, Seiten 685–694, 1998.

[12] Arya, Sunil, David M. Mount und Michiel H. M. Smid: *Randomized and deterministic algorithms for geometric spanners of small diameter.* In: *35th Annual Symposium on Foundations of Computer Science, 20-22 November 1994, Santa Fe, New Mexico, USA*, Seiten 703–712, 1994.

[13] Bayer, Rudolf und Edward M. McCreight: *Organization and Maintenance of Large Ordered Indices.* Acta Informatica, 1:173–189, 1972.

[14] Bender, Michael A., Richard Cole und Rajeev Raman: *Exponential Structures for Efficient Cache-Oblivious Algorithms.* In: *Automata, Languages and Programming, 29th International Colloquium, ICALP 2002, Malaga, Spain, July 8-13, 2002, Proceedings*, Seiten 195–207, 2002.

[15] Bender, Michael A., Erik D. Demaine und Martin Farach-Colton: *Cache-Oblivious B-Trees.* In: *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, Seiten 399–409, 2000.

[16] Bender, Michael A., Ziyang Duan, John Iacono und Jing Wu: *A locality-preserving cache-oblivious dynamic dictionary.* J. Algorithms, 53(2):115–136, 2004.

[17] Berg, Mark de, Marc van Kreveld, Mark Overmars und Otfried Schwarzkopf: *Computational Geometry - Algorithms and Applications.* Springer, 2 Auflage, 2000.

[18] Bollobás, Béla: *Modern Graph Theory.* Springer, 1 Auflage, August 2002.

[19] Breimann, Christian und Jan Vahrenhold: *External Memory Computational Geometry Revisited.* In: Meyer, Ulrich, Peter Sanders und Jop Sibeyn (Herausgeber): *Algorithms for Memory Hierarchies*, Band 2625 der Reihe *Lecture Notes in Computer Science*, Seiten 110–148. Springer, 2003.

[20] Brodal, Gerth Stølting und Rolf Fagerberg: *Cache Oblivious Distribution Sweeping.* In: *Proc. 29th International Colloquium on Automata, Languages, and*

Programming, Band 2380 der Reihe *Lecture Notes in Computer Science*, Seiten 426–438. Springer Verlag, Berlin, 2002.

[21] Brodal, Gerth Stølting und Rolf Fagerberg: *Funnel Heap - A Cache Oblivious Priority Queue.* In: *Proc. 13th Annual International Symposium on Algorithms and Computation*, Band 2518 der Reihe *Lecture Notes in Computer Science*, Seiten 219–228. Springer Verlag, Berlin, 2002.

[22] Brodal, Gerth Stølting und Rolf Fagerberg: *On the Limits of Cache-Obliviousness.* In: *Proc. 35th Annual ACM Symposium on Theory of Computing*, Seiten 307–315, 2003.

[23] Brodal, Gerth Stølting, Rolf Fagerberg und Riko Jacob: *Cache oblivious search trees via binary trees of small height.* In: *SODA*, Seiten 39–48, 2002.

[24] Brodal, Gerth Stølting, Rolf Fagerberg, Ulrich Meyer und Norbert Zeh: *Cache-Oblivious Data Structures and Algorithms for Undirected Breadth-First Search and Shortest Paths.* In: *Algorithm Theory - SWAT 2004, 9th Scandinavian Workshop on Algorithm Theory, Humlebaek, Denmark, July 8-10, 2004, Proceedings*, Band 3111, Seiten 480–492. Springer, 2004.

[25] Callahan, Paul B.: *Dealing with Higher Dimensions: The Well-Separated Pair Decomposition and Its Applications.* Doktorarbeit, The Johns Hopkins University, Baltimore, Maryland, 1996.

[26] Callahan, Paul B. und S. Rao Kosaraju: *Faster Algorithms for Some Geometric Graph Problems in Higher Dimensions.* In: *SODA*, Seiten 291–300, 1993.

[27] Callahan, Paul B. und S. Rao Kosaraju: *A Decomposition of Multidimensional Point Sets with Applications to k-Nearest-Neighbors and n-Body Potential Fields.* Journal of the ACM, 42(1):67–90, 1995.

[28] Chen, Danny Z., Gautam Das und Michiel H. M. Smid: *Lower bounds for computing geometric spanners and approximate shortest paths.* Discrete Applied Mathematics, 110(2-3):151–167, 2001.

[29] Chiang, Yi-Jen, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff und Jeffrey Scott Vitter: *External-Memory Graph Algorithms.* In: *SODA*, Seiten 139–149, 1995.

[30] Chowdhury, Rezaul Alam und Vijaya Ramachandran: *Cache-oblivious shortest paths in graphs using buffer heap.* In: *SPAA 2004: Proceedings of the Sixteenth Annual ACM Symposium on Parallel Algorithms, June 27-30, 2004, Barcelona, Spain*, Seiten 245–254, 2004.

[31] Colt Project: *Cern Colt Scientific Library 1.20.* `http://dsd.lbl.gov/~hoschek/colt/`, Dezember 2006. Webseite.

[32] Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest und Clifford Stein: *Introduction to Algorithms, Second Edition.* The MIT Press and McGraw-Hill Book Company, 2001.

[33] Das, Gautam und Giri Narasimhan: *A Fast Algorithm for Constructing Sparse Euclidean Spanners.* Int. J. Comput. Geometry Appl., 7(4):297–315, 1997.

[34] Demaine, Erik D.: *Cache-Oblivious Algorithms and Data Structures.* In: *Lecture Notes from the EEF Summer School on Massive Data Sets*, Lecture Notes in Computer Science. BRICS, University of Aarhus, Denmark, June 2002.

[35] Diestel, Reinhard: *Graphentheorie.* Springer, Heidelberg, 2 Auflage, Juni 2000.

[36] Eppstein, David: *Spanning trees and spanners.* Technischer Bericht 96-16, Univ. of California, Irvine, Dept. of Information and Computer Science, Irvine, CA, 92697-3425, USA, 1996.

[37] Farshi, Mohammad, Panos Giannopoulos und Joachim Gudmundsson: *Finding the best shortcut in a geometric network.* In: *Proceedings of the 21st ACM Symposium on Computational Geometry*, Seiten 327–335. ACM, 2005.

[38] Frigo, Matteo, Charles E. Leiserson, Harald Prokop und Sridhar Ramachandran: *Cache-Oblivious Algorithms.* In: *FOCS*, Seiten 285–298, 1999.

[39] Goodrich, Michael T., Jyh-Jong Tsay, Darren Erik Vengroff und Jeffrey Scott Vitter: *External-Memory Computational Geometry (Preliminary Version).* In: *Proceedings of the 34th Annual Symposium on Foundations of Computer Science*, Seiten 714–723, 1993.

[40] Govindarajan, Satish, Tamás Lukovszki, Anil Maheshwari und Norbert Zeh: *I/O-Efficient Well-Separated Pair Decomposition and its Applications.* Algorithmica, August 2005.

[41] GraphML: *Graph Markup Language.* `http://graphml.graphdrawing.org/`, Dezember 2006. Webseite.

[42] Gudmundsson, Joachim, Christos Levcopoulos und Giri Narasimhan: *Improved Greedy Algorithms for Constructing Sparse Geometric Spanners.* In: *Scandinavian Workshop on Algorithm Theory*, Seiten 314–327, 2000.

[43] Gudmundsson, Joachim, Christos Levcopoulos, Giri Narasimhan und Michiel H. M. Smid: *Approximate distance oracles for geometric graphs.* In: *SODA*, Seiten 828–837, 2002.

[44] Gudmundsson, Joachim, Giri Narasimhan und Michiel H. M. Smid: *Fast Pruning of Geometric Spanners.* In: Diekert, Volker und Bruno Durand (Herausgeber): *STACS 2005, 22nd Annual Symposium on Theoretical Aspects of Computer Science, Stuttgart, Germany, February 24-26, 2005, Proceedings*, Band 3404 der Reihe *Lecture Notes in Computer Science*, Seiten 508–520. Springer, 2005.

[45] Gudmundsson, Joachim und Jan Vahrenhold: *I/O-Efficiently Pruning Dense Spanners.* In: Akiyama, Jin, Mikio Kano und Xuehou Tan (Herausgeber): *Revised Selected Papers of the Japanese Conference on Discrete and Computational Geometry (JCDCG 2004)*, Band 3742 der Reihe *Lecture Notes in Computer Science*, Seiten 106–116. Springer, 2004.

[46] JUNG: *JUNG – Java Unviversal Network/Graph Framework.* `http//jung.sourceforge.net`, Dezember 2006. Webseite.

[47] Jungnickel, Dieter: *Graphs, Networks and Algorithms.*, Band 5 der Reihe *Algorithms and Computation in Mathematics.* Springer, 2005.

[48] Narasimhan, Giri und Michiel H. M. Smid: *Approximating the Stretch Factor of Euclidean Graphs.* SIAM J. Comput., 30(3):978–989, 2000.

[49] Nievergelt, Jurg und Klaus H. Hinrichs: *Algorithms and Data Structures - With Applications to Graphics and Geometry.* Prentice Hall, 1993.

[50] O'Madadhain, Joshua, Danyel Fisher, Padhraic Smyth, Scott White und Yan-Biao Boey: *Analysis and visualization of network data using JUNG.* `http://jung.sourceforge.net/doc/JUNG_journal.pdf`.

[51] Ottmann, Thomas und Peter Widmayer: *Algorithmen und Datenstrukturen*, Band 70 der Reihe *Reihe Informatik.* Bibliographisches Institut, 1990.

[52] Preparata, Franco P. und Michael Ian Shamos: *Computational Geometry - An Introduction.* Springer, 1985.

[53] Prokop, Harald: *Cache-Oblivious Algorithms.* Diplomarbeit, Massachusetts Institute of Technology, Cambridge, Juni 1999.

[54] Rønn, Frederik: *Cache-Oblivious Searching and Sorting.* Diplomarbeit, Department of Computer Science (University of Copenhagen), Juni 2003.

[55] Salowe, Jeffrey S.: *Construction of Multidimensional Spanner Graphs, with Applications to Minimum Spanning Trees.* In: *Symposium on Computational Geometry*, Seiten 256–261, 1991.

[56] Vaidya, Pravin M.: *A sparse Graph Almost as Good as the Complete Graph on Points in K Dimensions.* Discrete & Computational Geometry, 6:369–381, 1991.

[57] Vitter, Jeffrey Scott: *External memory algorithms and data structures: Dealing with massive data.* ACM Computing Surveys, 33(2):209–271, 2001.

[58] XML: *Extensible Markup Language.* `http://www.w3.org/XML/`, Dezember 2006. Webseite.

[59] Zeh, Norbert: *I/O-Efficient Algorithms for Shortest Path Related Problems.* Doktorarbeit, School of Computer Science, Carleton University, Ottawa, Canada, April 2002.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Münster, den 7. Dezember 2006

Fabian Gieseke