Optimierungsverfahren zur Isoflächen-Extraktion in der wissenschaftlichen Visualisierung

Dissertation zur Erlangung des akademischen Grades
“Dr. rer. nat.”

Universität Konstanz

vorgelegt von
Jürgen Toelke

begutachtet von
Prof. Dr. Dietmar Saupe
Prof. Dr. Ulrik Brandes

Tag der mündlichen Prüfung: 24.7.2006

Inhaltsverzeichnis

Zusammenfassung

Die Aufgabenstellung der Isoflächen-Extraktion ist es, für einen vorgegebenen Satz von Volumendaten, der eine Funktion repräsentiert, zu jedem so genannten Isowert die Grenzfläche zu finden, die die Bereiche mit Funktionswerten über und unter diesem Wert voneinander trennt. Mit der Hilfe von zusätzlichen Datenstrukturen lässt sich diese Aufgabe schneller lösen als durch eine vollständige Durchsuchung des Datensatzes. Dafür lassen sich Datenstrukturen verwenden, die mit wenig Speicher eine grobe Übersicht über den Datensatz geben (z. B. Partitionsbäume) oder alle Intervalle in ein schnelles, aber speicherintensives Suchschema eintragen (Intervallbäume, KD-Trees).

Zu jeder Methode, insbesondere zur Brute-Force-Suche, Partitionsbaum-, Intervallbaum-, KD-Tree- und einer hier beschriebenen Out-of-Core-Methode lässt sich ein Verfahren angeben, mit dem man abhängig von den Volumendaten die mittlere Extraktionszeit bei Annahme einer gegebenen Wahrscheinlichkeitsverteilung der Isowerte analytisch und meist rekursiv über den Aufbau der jeweiligen Datenstruktur bestimmen kann. Die Berechnung der Extraktionszeiten ist in dieser Arbeit beschrieben.

Mit Hilfe dieser Extraktionszeiten lässt sich eine parameterabhängige Methode entwickeln, die zu jeder vorgegebenen Hauptspeichergröße einen so genannten Conditioned Tree konstruiert, der den Speicher optimal zugunsten der Extraktionsgeschwindigkeit ausnutzt. Dieser ist eine hybride Datenstruktur, die auf einem Partitionsbaum basiert, der in einigen seiner Blätter Verweise auf andere Datenstrukturen enthält. Das Optimierungsverfahren zur Bildung der besten Conditioned Trees läuft darauf hinaus, eine Kostenfunktion $C_\lambda(\mathcal{T}) = M(\mathcal{T}) + \lambda T(\mathcal{T})$ über alle Bäume $\mathcal{T}$ einer vorgegebenen Klasse zu minimieren, wobei $M(\mathcal{T})$ der Speicherbedarf des Baums und aller angehängten Datenstrukturen ist und $T(\mathcal{T})$ die mittlere Extraktionszeit, die zu diesem Zweck unter Benutzung der erwähnten Näherungsformeln geschätzt wird.

Wie leicht einzusehen ist, lässt sich sowohl $M(\mathcal{T})$, als auch $T(\mathcal{T})$ rekursiv und additiv über den Aufbau des Baums $\mathcal{T}$ bestimmen. Die Minimierung der Kostenfunktion $C_\lambda(\mathcal{T})$ ist daher auch rekursiv möglich, indem man ihr Minimum zunächst für Teilbäume bestimmt und diese dann zu einem großen Baum vereinigt.

Aus einer Arbeit von Everett [16] geht hervor, dass ein mit dieser Minimierungsmethode erhaltener Optimalbaum stets auch die Zeitfunktion $T(\mathcal{T})$ minimiert unter der Nebenbedingung, dass der Speicherbedarf $M(\mathcal{T})$ den Wert des Optimums nicht überschreiten darf. Die Lagrange-Optimierung hat also den Vorteil, dass wir durch Variation des Lagrange-Faktors λ eine Serie von jeweils optimalen Bäumen erhalten, die sich für Rechner-Konfigurationen mit verschiedenem zur Verfügung stehendem Speicher eignen und fast jede Speichervorgabe optimal zugunsten der Extraktionsgeschwindigkeit ausnutzen.

Ebenso wird ein Verfahren beschrieben, mit dem auch die Rechenzeit komplexer Algorithmen - in diesem Fall Extraktionsmethoden - experimentell ermittelt werden kann. Hierfür wird das Vorhandensein einer lineare Formel zur Bestimmung der Rechenzeit vorausgesetzt, in der aber noch unbestimmte Zeitkonstanten vorkommen. Für eine Reihe von Rechendurchläufen mit verschiedenen Datenstrukturen gleichen Typs werden die Koeffizienten in dieser Formel analytisch und die

tatsächliche Rechenzeit durch Messung bestimmt. Dann werden die noch fehlenden Zeitkonstanten in der Formel approximiert, indem der Fehler zwischen dem Wert theoretischen Formel und der praktischen Messung minimiert wird.

Im Speicher-Zeit-Diagramm können noch Lücken auftreten, also Speicherbereiche, für die die Lagrange-Methode auch bei kleinsten Abständen zwischen den verwendeten Lagrange-Werten kein Optimum findet. Die größte dieser Lücken tritt für manche Datensätze zwischen dem KD-Tree und dem Intervallbaum auf. In dieser Arbeit ist am Beispiel dieser Lücke eine Gap-Filling-Methode beschrieben. In dieser Methode werden die Grenzen der Lücke - in diesem Fall der KD-Tree und der Intervallbaum jeweils für den ganzen Datensatz - näher untersucht und versucht, durch Einfügung mehrerer KD-Trees für Teilbereiche den Speicherbedarf des Intervallbaums auf Kosten der Suchzeit stückchenweise zu reduzieren. Dieses Verfahren schließt die Lücke zwar nicht ganz, macht sie jedoch kleiner.

Ein weiterer Beitrag dieser Arbeit ist der Design von Intervallbäumen, die bezüglich Speicherbedarf und Suchzeit bessere Resultate liefern als bisher beschriebene herkömmliche Intervallbäume. Dies geschieht durch die Wahl geeigneter Unterteilungswerte in den Intervallbaum-Knoten mit Hilfe eines zweistufigen Optimierungsverfahrens, mit denen insgesamt weniger Knoten verwendet werden und im Mittel auch weniger Knoten pro Extraktion besucht werden.

In der ersten Stufe des Optimierungsverfahrens wird für den gesuchten Intervallbaum eine Liste von Unterteilungswerten kleinstmöglicher Länge bestimmt, die später in die Intervallbaum-Knoten eingetragen werden. In der zweiten Stufe werden die Knoten mit diesen Unterteilungswerten so angeordnet, dass die mittlere Anzahl der besuchten Knoten pro Extraktion minimal ist.

Dadurch werden gegenüber herkömmlichen Intervallbäumen Speicher- und Suchzeit-Reduktionen im einzigen Bereich erreicht, in dem das möglich ist, da die Min- und Maxlisten eines Intervallbaums zu einer vorgegebenen Intervallliste stets gleich viele Einträge enthalten und auch die Suche in ihnen stets gleich schnell ist, nämlich output-sensitiv. In den Intervallbaum-Knoten hingegen sind noch wesentliche Verbesserungen möglich; die in dieser Arbeit beschriebene Methode reduziert die Anzahl der benötigten Knoten um ca. 30 Prozent und die mittlere Anzahl der besuchten Knoten je nach Größe des Intervallbaums um zwei bis drei pro Extraktion.

Ein weiteres Thema dieser Arbeit, das vor den Conditioned Trees erforscht wurde und deshalb in einem vorangehenden Kapitel beschrieben wird, ist die Organisation von Partitionsbäumen. Dabei werden zu einem gegebenen Volumen-Datensatz eine hierarchische Zerlegung und der zugehörige Partitionsbaum so bestimmt, dass bei einer vorgegebenen zugelassenen Anzahl von Knoten die Extraktionsgeschwindigkeit durch die Verwendung des Partitionsbaums maximiert wird. Hierfür werden Methoden zur Modifizierung der Grenzen der hierarchischen Zerlegung verwendet, sowie auch Methoden, um aus einem gegebenen Baum durch Abschneiden der richtigen Zweige qualitativ hochwertige kleinere Bäume zu erhalten.

Danksagung

Ich bedanke mich bei allen, die mich bei der Fertigstellung meiner Arbeit unterstützt haben.

Insbesondere geht mein Dank an:

- Prof. Dr. Dietmar Saupe, Prof. Dr. Oliver Deussen, Prof. Dr. Ulrik Brandes, Prof. Dr. Daniel Keim, sowie Raouf Hamzaoui, Martin Röder, Jens-Peer Kuska, Andreas Zerbst und viele andere Mitarbeiter der Universitäten von Freiburg, Leipzig und Konstanz, die mir mit Vorschlägen zur Gestaltung der Arbeit sehr geholfen haben.
- meine Mutter und meine leider inzwischen verstorbene Großmutter, die mich sowohl moralisch, als auch finanziell unterstützt haben.
- Besonderer Dank geht an meine Verlobte Marion Meyer, die mich bei der Fertigstellung der Dissertation sehr motiviert hat.
- Ein Dank geht an alle Autoren, die etwas zum Thema Isoflächen-Extraktion geschrieben haben; ferner an Frithjof Kruggel und alle anderen, die mir mit den Volumendaten und Funktionen ausgeholfen haben, die ich teilweise persönlich, teilweise übers Internet erhalten habe.
- Danke auch an alle Administratoren für die Instandhaltung der von mir verwendeten Rechnersysteme in Freiburg, Leipzig und Konstanz.

Hinweis

Diese Version der Dissertation wurde nachbearbeitet und unterscheidet sich daher in einigen Punkten von der offiziellen, zur Bewertung abgegebenen Arbeit.

Kapitel 1

Einführung

1.1 Übersicht

Das Problem der Isoflächen-Extraktion oder auch Isozellen-Extraktion aus Volumen-Datensätzen hat eine einfache, sich direkt aus der Definition der Aufgabenstellung ergebende Lösung, die aber für viele Anwendungen zu langsam ist. Die in dieser Arbeit vorgestellten Lösungsansätze laufen darauf hinaus, über den Platz des Datensatzes hinausgehend Speicherplatz für weitere Datenstrukturen einzusetzen, um durch eine gezieltere Suche in diesen Strukturen den Extraktionsvorgang zu beschleunigen.

Die Isoflächen-Extraktion kann in vielen Forschungsbereichen zur besseren Visualisierung von räumlichen Zusammenhängen verwendet werden. Besonders in der Medizin ist es nützlich, aus anatomischen Volumendaten durch Isoflächen-Extraktion die jeweils gewünschten Teilinformationen zu gewinnen. Aus diesem Grund habe ich mehrere medizinische Datensätze in meine Experiment-reihe übernommen.

Die Dissertation beschränkt sich auf die Isozellen-Extraktion, das heißt, es wird wie in Abbildung 1.1 aus einem Datensatz eine Datenstruktur erzeugt, mit deren Hilfe wiederum durch Extraktion eine Liste von Zellen ermittelt wird. Aus dieser Liste kann im nächsten Arbeitsschritt ein Bild generiert werden, auf die Rendering-Methoden gehe ich in der Arbeit aber nicht näher ein. Abbildung 1.2 zeigt etwas genauer, wie das von mir gelöste Teilproblem in den Gesamtkontext eingebettet sein kann; hier ist auch zu sehen, dass der Rechenzeit-Bedarf der Extraktion je nach angewandter Methode zwischen $O(k)$ (output-sensitiv) und $O(n)$ (Untersuchung des ganzen Datensatzes)

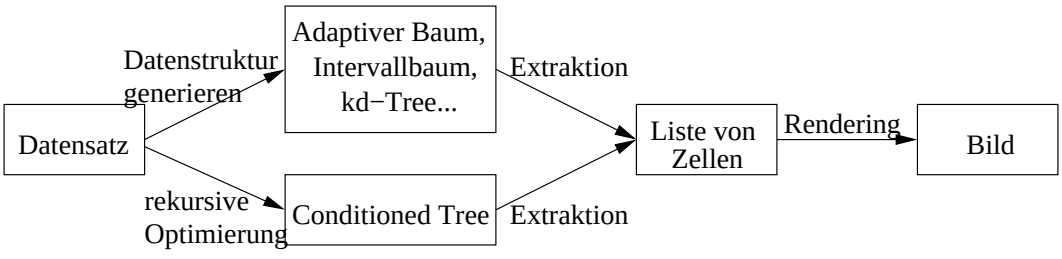


Abbildung 1.1: Die Arbeitsschritte der Isoflächen-Extraktion vom Datensatz bis zum Bild

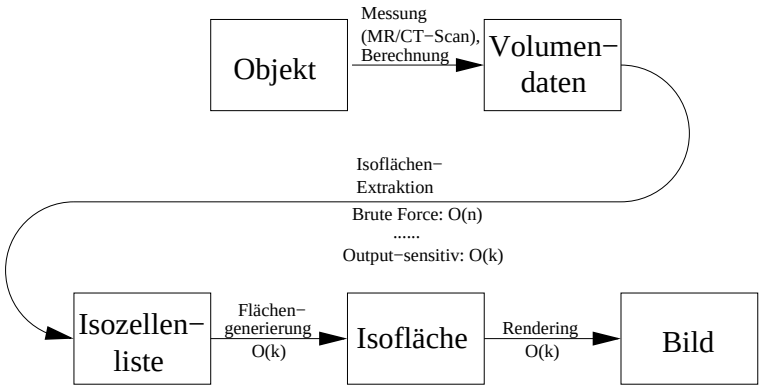


Abbildung 1.2: Die Arbeitsschritte und ihr Rechenzeit-Bedarf im Detail

schwanken kann, daher ist die Wahl der Methode von großer Bedeutung.

Gleich im nächsten Abschnitt werde ich die Isozellen-Extraktion exakt definieren. Abbildung 1.3 zeigt eine Übersicht über die wichtigen Methoden der Isoflächen-Extraktion und ihre Zuordnung zu den Kapiteln dieser Arbeit.

Wie die Abbildung zeigt, lassen sich die meisten bekannten Methoden der Isoflächen-Extraktion in zwei große Gruppen einordnen: die räumlich orientierten und die intervall-orientierten Methoden.

Räumlich orientiert nennen wir alle Methoden, die den Volumen-Datensatz in kleinere Teile zerlegen und diese Zerlegung sowie statistische Eigenschaften der Daten innerhalb jeweils eines dieser Teile in einer Datenstruktur ablegen. Meistens werden als Eigenschaften das Minimum und das Maximum der Volumendaten innerhalb des betroffenen Teils verwendet; diese legen durch einen Vergleich mit dem gegebenen Isowert (Intervalltest) fest, ob der betroffenen Teil des Datensatzes näher untersucht werden muss oder nicht. Solche Methoden machen schon mit wenig zusätzlichem Speicher wesentliche Verkürzungen der Suchzeit möglich. Im Abschnitt 2.1 werden Partitionsbäume als typisches Beispiel der räumlich orientierten Methoden beschrieben. Jane Wilhelms und Allen Van Gelder beschreiben in [32] die Idee der Partitionsbäume anhand von Octrees.

Unter intervall-orientierten Methoden verstehen wir alle Methoden, die die zu untersuchenden Zellen des Volumen-Datensatzes von ihrer räumlichen Anordnung befreien und nach ihren Datenintervall-Grenzen in eine Datenstruktur einsortieren. Diese Art von Methoden kostet viel zusätzlichen Speicher, macht aber auch eine extrem schnelle Extraktion der Isoflächen möglich. Ab Abschnitt 2.2 sind einige gut geeignete Extraktionsmethoden dieser Art dargestellt. Dazu gehören die ISSUE-Methode, die von Shen, Hansen, Livnat und Johnson in [27] beschrieben wird, der KD-Tree, der von den gleichen Autoren ohne Hansen verwendet wird [23], nachdem seine Grundidee 1975 von Bentley [6] beschrieben wurde und nicht zuletzt der 1980 von Edelsbrunner beschriebene Intervallbaum [15], der später von Cignoni et. al. für die Isoflächen-Extraktion angewendet wurde [13].

Im 3. Kapitel stelle ich ein paar weitere Beiträge anderer Autoren zum Thema Isoflächen-Extraktion dar, die von etwas abgewandelten Modell-Voraussetzungen ausgehen und sich daher nicht in die Reihe der hier behandelten Methoden einordnen lassen. Zunächst werden zwei Out-of-Core-Methoden nach Chiang und Silva [10, 11] beschrieben, die eine Datenstruktur auf der Festplatte anstatt im Hauptspeicher ablegen. Diese ermöglichen eine Beschleunigung der Extraktion ohne Verwendung von Hauptspeicher. Dafür wird jedoch Festplattenspeicher gebraucht, auf

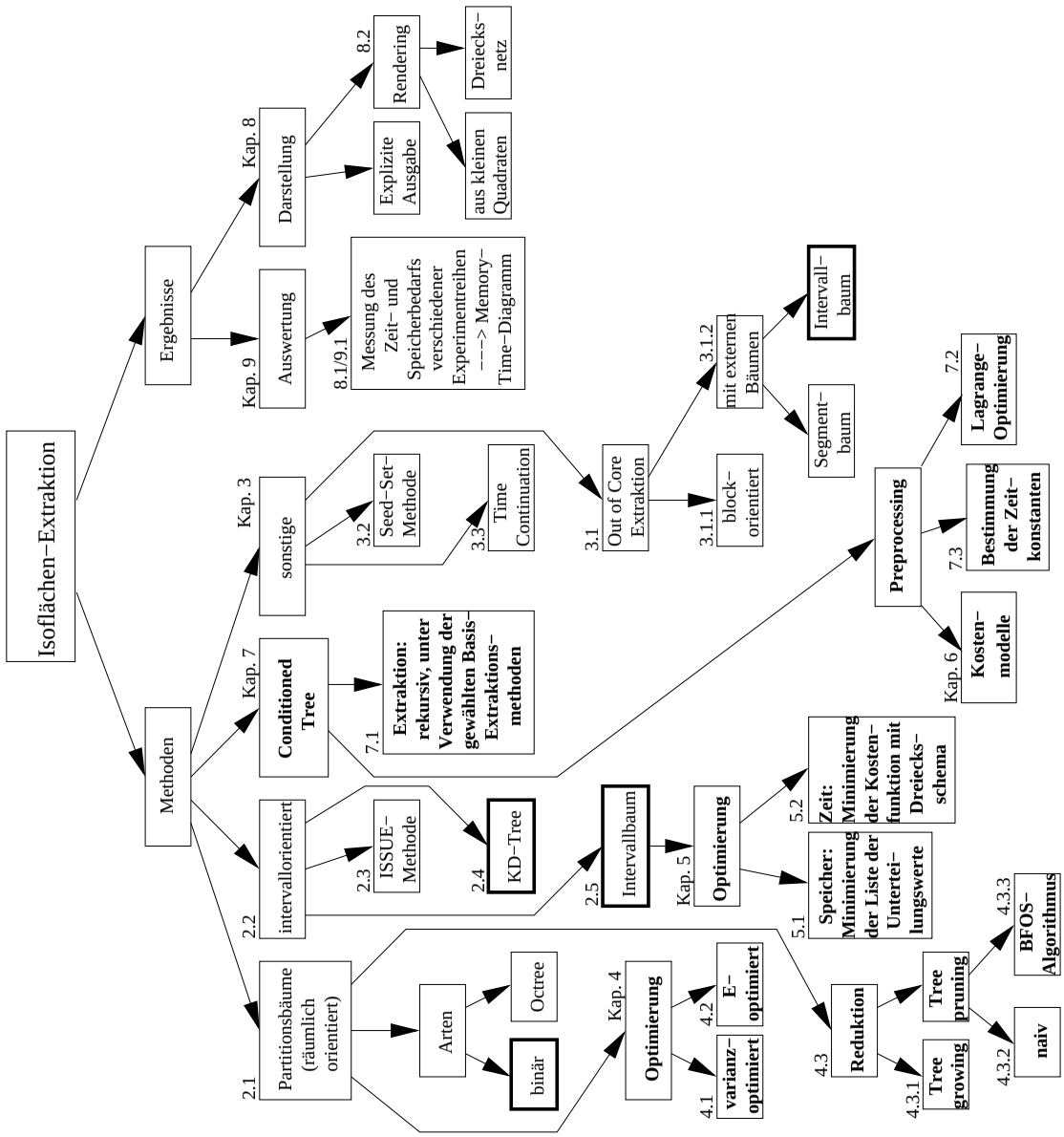


Abbildung 1.3: Die Methoden der Isoflächen-Extraktion. Mein eigener Beitrag ist in dieser Übersicht in Fettschrift dargestellt. Die zum Conditioned-Tree-Verfahren kombinierten Grundmethoden erscheinen in einem stärkeren Rahmen. Die Zahlen geben die Nummern der Abschnitte an, in denen Informationen zum entsprechenden Objekt zu finden sind.

den der Zugriff im allgemeinen langsamer ist als auf den Hauptspeicher. Anschließend folgt eine Beschreibung der Seed-Set-Methode, deren Sinn es ist, die Isofläche aus einer kleinen Menge von Volumen-Zellen zu rekonstruieren. Eine Variante der Seed-Set-Methode wird z.B. von Bajaj, Pascucci und Schikore in [3, 4] beschrieben, später wurde das Thema von anderen Autoren aufgegriffen, da das Grundprinzip der Seed-Set-Methoden bei der Implementierung noch einige Freiheitsgrade offen lässt. Schließlich wird hier eine Time-Continuation-Methode nach Shen und Johnson [28] aufgeführt, die das Problem der Isoflächen-Extraktion dann besonders schnell löst, wenn bereits eine Isofläche gegeben ist und eine andere Fläche zu einem nur wenig geänderten Isowert zu bestimmen ist.

Mit dem 4. Kapitel beginnt mein eigener Beitrag zum Problem der Isoflächen-Extraktion. In diesem Kapitel werden zunächst ein paar Möglichkeiten dargestellt, die die Qualität von Partitionsbäumen verbessern. Das ist möglich, indem die zum Baum gehörende Zerlegung gezielt beeinflusst wird.

Es ist auch möglich, von einem bereits existierenden Baum Unterbäume abzuschneiden und ihn so auf jede gewünschte Speichergröße zu reduzieren. Der daraus resultierende Speedup-Verlust kann durch Optimierungsverfahren minimiert werden, von denen der BFOS-Algorithmus nach der Beschreibung von Chou, Lookabaugh und Gray [12] das beste ist, weil er mathematisch nachweisbar das optimale Ergebnis liefert.

Im 5. Kapitel beschreibe ich ein Verfahren, mit dem die vorher beschriebenen Intervallbäume bezüglich der Speicher- und Zeitkosten optimiert werden können. Es gibt viele Arbeiten wie [15, 13], in denen Intervallbäume beschrieben werden und die in ihre Knoten eingetragenen Unterteilungswerte mit einfacheren Methoden wie z.B. Mittelwert-Bestimmung aus den vorgegebenen Intervallen bestimmt werden. Hier ist eine weitere Beschleunigung des Verfahrens durch geeignete Wahl der Unterteilungswerte möglich; dieser Umstand wurde bisher wegen der sowieso schon hohen Geschwindigkeit des Intervallbaum-Verfahrens außer acht gelassen.

Im Kapitel 6 wird gezeigt, wie für die grundlegenden Verfahren, also für die Brute-Force-Methode (Abschnitt 6.1), hierarchische Zerlegungsbäume (Abschnitt 6.2), Intervallbäume (Abschnitt 6.3), KD-Trees (Abschnitt 6.4 und 6.5) und das Out-of-Core-Verfahren mit extern gespeicherten Intervallbäumen (Abschnitt 6.6) der Erwartungswert der Extraktionszeit analytisch und ohne explizite Zeitmessungen ermittelt werden kann. Für den Erwartungswert wird vorausgesetzt, dass eine Wahrscheinlichkeitsverteilung für die angefragten Isowerte vorgegeben ist. Die Formeln sind für eine Gleichverteilung der Isowerte auf dem ganzen Wertebereich angegeben; sie lassen jedoch völlig unproblematisch eine Änderung des Wahrscheinlichkeitsmodells zu.

Basierend auf diesen Extraktionsmethoden habe ich die Conditioned-Tree-Methode realisiert, deren Ergebnis abhängig von einem vorgegebenen Parameter λ von speichersparend langsam bis speicheraufwändig schnell variiert. Diese Methode, beschrieben im Kapitel 7, ist das Kernthema der Dissertation und hat den Vorzug, dass sie für viele Rechnertypen mit verschiedener Größe des Hauptspeichers diesen optimal zugunsten der Suchgeschwindigkeit ausnutzen kann. Durch ihre Variabilität erhält die Conditioned-Tree-Methode einen großen Vorteil gegenüber Methoden mit von vornherein festgelegter Datenstruktur wie z.B. Partitionsbaum-, Intervallbaum- oder KD-Tree-Verfahren, weil diese jeweils einen festen Speicherbedarf haben und nur dann angewendet werden können, wenn der Hauptspeicher in entsprechender Größe zur Verfügung steht.

Der Conditioned Tree basiert auf einem binären Partitionsbaum des Volumen-Datensatzes, in dessen Blättern jeweils eine andere, bereits bekannte Extraktionsmethode mit ihrer Datenstruktur eingetragen ist. Der Sinn davon ist es, diese Extraktionsmethode für den entsprechenden Teilblock des

Datensatzes anzuwenden. Zu einer vorgegebenen Kostenfunktion lässt sich der Conditioned Tree, der diese minimiert, jeweils rekursiv über seinen Aufbau bestimmen. Die Optimierung erfolgt nach der von Everett in [16] beschriebenen Lagrange-Methode.

Außer der optimalen Ausnutzung des Speicherplatzes hat dieses hybride Verfahren auch den Vorteil, dass sich darin beliebige Extraktionsmethoden einbauen und verwenden lassen. Die einzige Voraussetzung dafür ist, dass es zu jeder Extraktionsmethode jeweils eine Kostenfunktion zur Berechnung des von ihr belegten Speichers und der erwarteten Rechenzeit wie in Kapitel 6 gibt, die in die Kostenoptimierung des Conditioned Trees eingebaut werden kann.

Im 8. Kapitel wird näher auf die Einzelheiten der Testprogramme sowie der in OpenGL geschriebenen Darstellungsprogramme eingegangen.

Genauere Bewertungen der Methoden folgen jeweils am Ende der im Einzelnen beschriebenen Methoden, sowie im 9. Kapitel über die Ergebnisse. Die abschließende Zusammenfassung steht im Kapitel 10.

Im Anhang A der Arbeit werden zunächst ein paar mit den beschriebenen Algorithmen bearbeitete Beispiel-Datensätze aufgelistet.

Hier ist eine kurze Zusammenfassung meiner Beiträge zur Isoflächen-Extraktion:

- Mathematische Modelle für die erwartete Rechenzeit $T(.)$ der verschiedenen Extraktionsmethoden (Kapitel 6) und ihre empirische Berechnung
- Bildung von so genannten Conditioned Trees, die eine aus der so bestimmten Rechenzeit und dem Speicherbedarf der Datenstruktur gebildeten Kostenfunktion minimieren. (Abschnitt 7.1)
- Optimierung von Intervallbäumen, bestmögliche Wahl der Unterteilungswerte γ^*, bezüglich Speicher- und Rechenzeit-Bedarf (Anzahl der Knoten, mittlere Baumtiefe) (Kapitel 5)
- Verbesserung von Partitionsbäumen durch geeignetes Tree pruning (Kapitel 4)

Zu diesem Thema habe ich zusammen mit D. Saupe die Arbeit [25] geschrieben, die bei der VMV veröffentlicht ist. Bajaj veröffentlichte in [3] und [5] (letzteres zusammen mit Pascucci und Schikore) Zusammenfassungen bekannter Methoden der Isoflächen-Extraktion, darunter die Marching-Cube-Methode, partitions-orientierte Ansätze, Span-Space-orientierte Lösungen wie den Intervallbaum und etwas detaillierter die Möglichkeiten von Seed-Set-Methoden.

Nachstehend habe ich die wichtigsten Extraktionsmethoden mit ihren Qualitätsmerkmalen aufgelistet. In den O-Notationen steht n jeweils für die Größe des Volumen-Datensatzes, bzw. für die Anzahl der darin enthaltenen Zellen. k ist die Länge der Ausgabe, also die Anzahl der Zellen, die in der gesuchten Isofläche enthalten sind. Die Angabe des Speicherbedarfs in O-Notation ist redundant, weil dieser in jedem Fall einschließlich Datensatz $O(n^3)$ ist und sich nur durch konstante Faktoren unterscheidet.

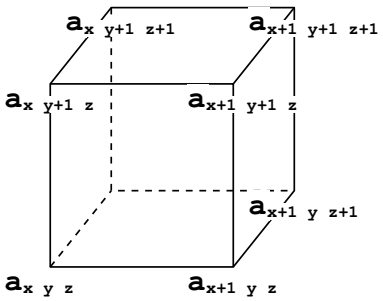


Abbildung 1.4: Eine Zelle c_{xyz} mit ihren Eckvoxeln und den zugeordneten Datenwerten $a_{xyz}, ...$

Methode	Geschwindigkeit	Speicherbedarf	beschrieben in...
Brute Force	$O(n^3)$ (langsam)	niedrig	(trivial)
Adap. Zerlegungsbaum	variabel, $\geq O(k + k \cdot log(n))$ (mittel)	mittel	Abs. 2.1 + Kap. 4
KD-Tree	$O(k + \sqrt{n})$ (schnell)	hoch	Abschnitt 2.4
Intervallbaum	$O(k + log(n))$ (sehr schnell)	hoch	Abs. 2.5 + Kap. 5
Conditioned Tree	variabel, speicherabhängig	variabel	Kapitel 7
Out-of-Core	kann schnell sein (abhängig von der Speichermethode)	Hauptsp. niedrig, Festplattensp. hoch	Abschnitt 3.1

1.2 Die Problemstellung der Isoflächen-Extraktion

Gegeben ist ein Volumen-Datensatz, d. h., ein dreidimensionales Array von Zahlenwerten aus einem geordneten Wertebereich R (z. B. $R = \mathbb{R}$, $[0, 1]$ oder $\mathbb{Z}$) in der Form

$$(a_{xyz})_{x\in\{0,...,x_{\max}\};y\in\{0,...,y_{\max}\};z\in\{0,...,z_{\max}\}}$$

An vielen Stellen dieser Arbeit wird anstelle dieses Datensatzes eine Funktion $f : \mathbb{R}^3 \to \mathbb{R}$ angegeben. Dies ist jedoch keine von der Vorgabe abweichende Form, sondern lediglich eine implizite Definition des Volumen-Datensatzes, der sich aus gitterförmig angeordneten Werten $f(x, y, z)$ zusammensetzt, also wäre z. B. für $R = [0, 1]$ eine Berechnung der Volumendaten durch $a_{xyz} = \frac{r_{xyz}-r_{\min}}{r_{\max}-r_{\min}}$ mit

$$r_{xyz} := f(x_0 + (x_1 - x_0)\frac{x}{x_{\max}}, y_0 + (y_1 - y_0)\frac{y}{y_{\max}}, z_0 + (z_1 - z_0)\frac{z}{z_{\max}})$$

möglich. Dabei können $r_{\min}$ und $r_{\max}$ als Minimum und Maximum aller r_{xyz}-Werte definiert werden oder die Grenzen eines vom Benutzer vorgegebenen Wertebereichs sein. Für $R = \mathbb{R}$ kann einfach $a_{xyz} = r_{xyz}$ verwendet werden, für diskrete Wertebereiche wie $R = \mathbb{Z}$ oder endliche Teilmengen davon wird a_{xyz} aus r_{xyz} durch eine geeignete lineare Abbildung mit anschließender mathematischer Rundung (Quantisierung) gebildet. Wir fassen jeweils acht würfelförmig benachbarte Gitterpunkte (**Voxel**) dieses Datensatzes zu einer Einheit (**Zelle**) zusammen (siehe Abbildung 1.4). Die Eckwerte der Zelle $c_{xyz} := \{x, x + 1\} \times \{y, y + 1\} \times \{z, z + 1\}$ mit $x \in \{0, ..., x_{\max} - 1\}$, $y \in \{0, ..., y_{\max} - 1\}$ und $z \in \{0, ..., z_{\max} - 1\}$ bilden also die Menge

$$M_{xyz} := \{a_{x'y'z'} | x' \in \{x, x + 1\}; y' \in \{y, y + 1\}; z' \in \{z, z + 1\}\}$$

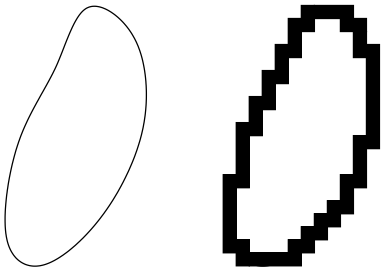


Abbildung 1.5: Links im Bild ist ein Beispiel einer kontinuierlichen Isolinie dargestellt. Wir betrachten jedoch diskrete Isolinien (oder -flächen) wie im rechten Teil des Bildes

Im mathematischen Sinn ist eine **Isofläche** für eine kontinuierliche Funktion $f : \mathbb{R}^3 \to \mathbb{R}$ als die Lösungsmenge der Gleichung $f(x, y, z) = c$ definiert, wobei c der **Isowert**, eine vorgegebene reellwertige Konstante ist. Es hätte keinen Sinn, diese Definition unverändert auf den vom Computer bearbeiteten diskreten Fall zu übertragen, weil die Werte der Funktion an den Gitterpunkten nur zufällig den gegebenen Isowert annehmen können. Es ist aber bei stetigen Funktionen durch den Zwischenwertsatz gesichert, dass Zellen, an deren Eckpunkten sowohl Funktionswerte über als auch unter dem Isowert vorkommen, einen Teil der Isofläche enthalten. In allen anderen Zellen kann natürlich auch ein Stück der Fläche enthalten sein, was aber aus den vorgegebenen Datenwerten weder beweisbar, noch widerlegbar ist.

Basierend auf dieser Betrachtung definieren wir für dreidimensionale Arrays zu jedem vorgegebenen Isowert $\gamma \in R$ eine so genannte **diskrete Isofläche**, die aus den Zellen des Datensatzes besteht, durch die die oben betrachtete kontinuierliche Isofläche nachweislich läuft, die also die **Intervallbedingung** $\min_{xyz} \leq \gamma < \max_{xyz}$ erfüllen, wobei

$$\min_{xyz} := \min(M_{xyz}) \quad ; \qquad \max_{xyz} := \max(M_{xyz})$$

In dieser Arbeit wird im Kontext eines diskreten Datensatzes mit dem Begriff “Isofläche” die diskrete Isofläche bezeichnet (wie in Abbildung 1.5). Es ist nun die Aufgabe dieser Arbeit, wie in Abbildung 1.6 die Liste dieser Zellen mit Hilfe eines gegebenen freien Speicherbereichs innerhalb kürzestmöglicher Suchzeit aus dem Datensatz zu erhalten.

In dieser Dissertation werden Intervalltests stets mit links abgeschlossenen und rechts offenen Intervallen durchgeführt, also z. B. in der Form $\gamma \in [\min_{xyz}, \max_{xyz})$ oder $\gamma \in [a_{\min}, a_{\max})$, wobei $a_{\min}$ und $a_{\max}$ jeweils durch den Kontext des verwendeten Algorithmus vorgegebene Intervallgrenzen sind. Die Wahl dieser Form hat gegenüber beidseitig abgeschlossenen Intervallen einige Vorteile, die ich hier allgemein formuliere, ohne auf Einzelfälle einzugehen:

- Bei Intervall-Zerlegungen passen mehrere Intervalle dieses Typs nahtlos zusammen, ohne dazwischen eine Lücke zu lassen oder gemeinsame Punkte zu haben.

- Für die Berechnung von Wahrscheinlichkeitsmaßen sind im allgemeinen, nicht notwendigerweise stetigen Fall halboffene Intervalle am besten geeignet. Es genügt sowohl für ganzzahlige als auch für kontinuierliche Wertebereiche eine Formel des Typs $P([a_1, a_2)) = P(a_2) - P(a_1)$, wobei angenommen wird, dass das Maß über die Werte einer monoton steigenden Funktion $P(a) = P((-\infty, a))$ gegeben ist. Die Annahme einer gegebenen Häufig-

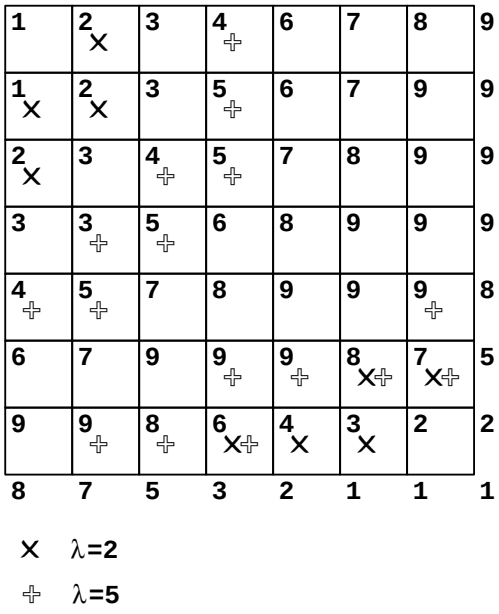


Abbildung 1.6: Ein zweidimensionales Beispiel für die Isozellen-Extraktion. Die Ausgabe des Programms ist in diesem Fall eine Isolinie.

keitsverteilung für die angefragten Isowerte γ ist eine Grundvoraussetzung für die Berechnung der Bewertungsfunktion, die in Kapitel 7 bei der Bestimmung des optimalen Conditioned Trees verwendet wird.

- In den meisten Grenzfällen ist das halboffene Intervall am besten für die Konstruktion einer lückenlosen Isofläche geeignet. Bei Tests mit beidseitig abgeschlossenen Intervallen würde man aus der Menge der extrahierten Zellen Gebilde erhalten, die stellenweise mehr als eine Zelle breit sind, wogegen bei beidseitig offenen Intervallen Löcher in den Flächen entstehen.

Es ist leicht, ein Programm anzugeben, das die Aufgabenstellung der Isoflächen-Extraktion direkt löst. Der für dieses Programm verwendete Algorithmus heißt Brute-Force-Methode und durchläuft alle möglichen Tripel (x, y, z), um festzustellen, für welche die Intervallbedingung zutrifft (beschrieben von Lorensen und Cline in [24]). Das Problem dabei ist allerdings, dass diese Methode $O(n^3)$ Rechenzeit benötigt und dadurch für größere Datensätze kein geeignetes interaktives Extraktionsprogramm ermöglicht. Wir suchen also eine Datenstruktur und eine zugehörige Suchmethode, die das Suchproblem innerhalb von kürzerer Zeit löst.

Es gibt eine weitere triviale Methode, die das Extraktionsproblem löst und immer dann funktioniert, wenn die Volumen-Datenwerte ganze Zahlen sind oder aus einem anderen Grund nur endlich viele verschiedene Isowerte vorgegeben werden können. Hierfür werden im Vorverarbeitungs-Schritt die Isoflächen für alle möglichen Isowerte in einem Array gespeichert (siehe Abbildung 1.7). Im Hauptprogramm kann dann zu jedem Isowert die zugehörige Liste aus diesem Array zurückgegeben werden.

Diese Methode löst das Problem in minimaler Suchzeit, weil zu einem Isowert γ nur die passende Liste aus dem Array auszugeben ist. Ein großer Nachteil ist es allerdings, dass die Liste aller Zellen zu jedem möglichen Isowert im Normalfall wesentlich mehr Speicher benötigt, als selbst der

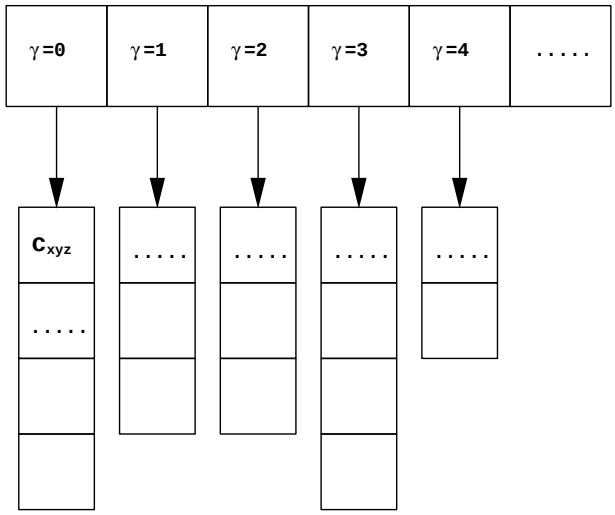


Abbildung 1.7: Die Datenstruktur für die trivial-optimale Methode

leistungsfähigste Rechner bieten kann. Zum Beispiel für einen $256 \times 256 \times 256$-Datensatz mit einem Speicherbedarf von 2 Byte pro Voxel benötigt der Original-Datensatz 32 MB; wenn dieser eine abhängig von der x-Koordinate steigende und den ganzen Wertebereich durchlaufende Funktion beschreibt, dann belegen die Listen der trivial-optimalen Methode insgesamt 12 GB Speicher.

Wir suchen hier also nach Methoden, die die Isoflächen-Extraktion beschleunigen und dafür nicht mehr Speicher brauchen, als vom jeweiligen System vorgegeben ist. Die in dieser Arbeit beschriebene Conditioned-Tree-Methode bietet eine Klasse von Datenstrukturen, die das Problem mit verschiedenem Speicherbedarf und verschiedener Rechenzeit lösen. Zusammenfassend besteht also folgende Problemstellung:

- Es ist ein Problem gegeben, das mit vorgegebenem Speicher innerhalb kürzestmöglicher Zeit zu lösen ist.
- Es steht eine Liste von Lösungsmöglichkeiten $\mathcal{T}_i \ \ (i = 1, ..., m)$ zur Verfügung, die jeweils $M(\mathcal{T}_i)$ Speicher und durchschnittlich $T(\mathcal{T}_i)$ Rechenzeit benötigen.
- Die Lösungen $\mathcal{T}_i$ sind durch hierarchische Datenstrukturen (Bäume) repräsentiert, deren Unterbäume jeweils Lösungen von Teilproblemen sind, deren Speicher- und Zeitaufwand sich additiv zusammenfügen lässt.

1.3 Die verwendeten Volumendaten

Für den Vergleich der Extraktionsmethoden habe ich zwei Arten von Volumen-Datensätzen herangezogen: algebraische Funktionen, die implizite Flächen als Lösungs-Isoflächen haben und gemessene, z. B. medizinische Daten, um die Brauchbarkeit der Methoden für real gemessene Daten zu testen (siehe Abbildung 1.8).

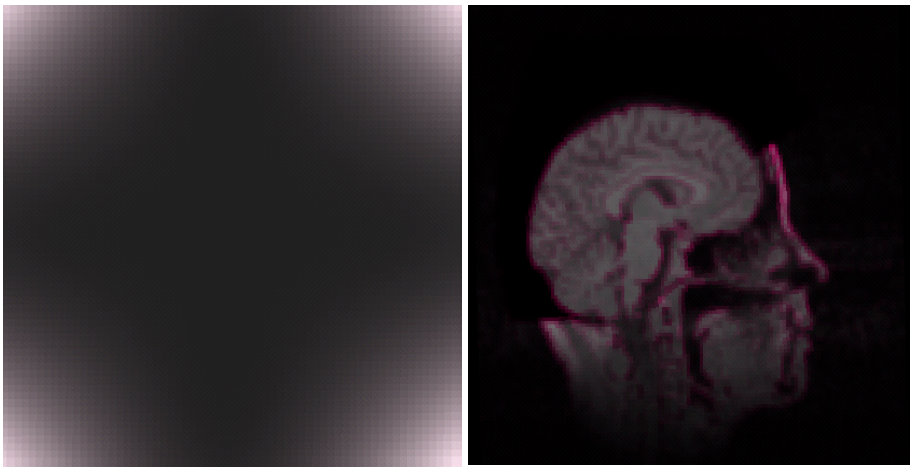


Abbildung 1.8: Beispiele für eine Ebene der Volumendaten in Grauwertdarstellung

Ein Beispiel für die algebraischen Funktionen ist die Suche nach der Kummer-Oberfläche, die die Lösungsmenge der Gleichung $f(x, y, z) = 0$ ist, wobei $f(x, y, z)$ gegeben ist durch:

$$p(u, v) := 1 + v + u\sqrt{2}$$

$$f(x, y, z) := (x^2 + y^2 + z^2 - 2)^2 + 5p(x, z)p(-x, z)p(x, -z)p(-x, -z)$$

Wir untersuchen also nun, wie am schnellsten eine stückweise lineare Approximation der Lösungsmenge der Gleichung $f(x, y, z) = \gamma$ für beliebige Isowerte γ mit Methoden der Isoflächen-Extraktion zu finden ist. Der Definitions- und Wertebereich der Funktion f ist zu diesem Zweck wie in der Definition am Anfang dieses Kapitels transformiert worden, so dass der interessante Teil der Funktion in den Volumendaten-Bereich $(0, 0, 0) - (x_{\max}, y_{\max}, z_{\max})$ passt und die Funktionswerte innerhalb einer vorgegebenen endlichen Menge ganzer Zahlen liegen.

Der Datensatz vom menschlichen Kopf ist zum Beispiel ein MR-Scan aus dem FTP-Verzeichnis unter [1]. Nähere Einzelheiten über diesen und andere medizinische Datensätze sowie die Bilder dazu sind im Anhang A.2 zu finden. Die Idee, diese Daten zu verwenden, stammt von der Website [20], von Levoy und Lacroute. Der Anhang A enthält Beispiele für Datensätze, die für die Isoflächen-Extraktion ausgewertet wurden.

Intern sind sämtliche dreidimensionalen Datensätze als eindimensionale Arrays gespeichert. Das heißt, dass ein Datenwert a_{xyz} von den Programmen als $a[(y_{max}+1)(z_{max}+1)x+(z_{max}+1)y+z]$ angesprochen wird, nicht als $a[x][y][z]$. Der Vorteil dieser Speicherungsweise ist, dass Speicher für insgesamt $(x_{max} + 1)(y_{max} + 2)$ Pointer gespart wird, der in andere Datenstrukturen investiert werden kann. Dadurch wird bei vorgegebenem zur Verfügung gestelltem Hauptspeicher bereits viel eingespart, weil dieser Speicher für einen hierarchischen Zerlegungsbaum verwendet werden kann, der die Extraktion der Isofläche schon um einiges beschleunigt.

An der Zugriffsgeschwindigkeit ändert sich durch diese Speicherungsform nicht viel, weil für einen Zugriff auf $a[x][y][z] = *(*(*(a + x) + y) + z)$ drei Additionen und drei Pointer-Operationen gebraucht werden, während für einen Zugriff auf $a[(y_{max} + 1)(z_{max} + 1)x + (z_{max} + 1)y + z] = *(a + z_m(y_m x + y) + z)$ drei Additionen, zwei Multiplikationen und eine Pointer-Operation nötig sind ($y_m = y_{max} + 1$ und $z_m = z_{max} + 1$ sind als Kantenlängen des Datensatzes bereits gegeben).

Kapitel 2

Die wichtigsten Methoden der Isoflächen-Extraktion

2.1 Das Prinzip der partitionsbaum-orientierten Methoden

Die in diesem Kapitel beschriebenen Methoden zur Beschleunigung der Isoflächen-Extraktion basieren auf Partitionsbäumen, die auf dem Volumen-Datensatz errichtet werden. Die Grundidee dafür stammt aus den Ausführungen von Jane Wilhelms und Allen Van Gelder in [32], wo das Prinzip am Beispiel eines Octrees erklärt wird: das Volumen wird durch eine oder mehrere Ebenen in Teile zerlegt, diese können wieder in kleinere Teile zerlegt werden; dies wird so lange wiederholt, bis die Einzelblöcke klein genug sind. Diese Zerlegung wird durch einen Zerlegungsbaum dargestellt, in dem jeder Knoten einen Teilblock des Volumen-Datensatzes repräsentiert; die Wurzel des Baums entspricht dabei dem ganzen Volumen-Datensatz.

Als Datenstruktur betrachtet enthält jeder Knoten des Baums folgende Informationen des von ihm repräsentierten Blocks (Abbildung 2.1):

- die Information, ob und wie der Block unterteilt wird. Diese besteht z. B. im allgemeinen binären Fall aus der Ausrichtung und der Position der Unterteilungsebene.
- Eine Liste von Zeigern $p_1, p_2, ..., p_n$ (im Fall eines binären Baums p_l und p_r), die auf die Knoten zu den Teilblöcken verweisen.
- Zwei Datenwerte $a_{\min}$ und $a_{\max}$, die das Minimum und das Maximum der Volumendaten innerhalb des Blocks sind.

Wir gehen in der Dissertation davon aus, dass binäre Partitionsbäume verwendet werden, falls nichts Gegensätzliches erwähnt wird.

Die Partitionsbaum-Methoden basieren auf der Grundidee, dass der Funktionsverlauf des gegebenen Datensatzes in lokalen Bereichen nur wenig variiert, wodurch die Größe des Intervalls $[min, max)$ schnell abnimmt, wenn man auf einem Pfad des Baums herunter wandert. Die Isoflächen-Extraktion wird jetzt durchgeführt, indem der Partitionsbaum rekursiv durchlaufen wird. Dabei wird jeder Knoten darauf getestet, ob der vorgegebene Isowert γ im darin eingetragenen Intervall $[min, max)$ enthalten ist. Wenn das nicht der Fall ist, dann kann dieser Knoten und alles, was in der Baumhierarchie darunter liegt, übersprungen werden.

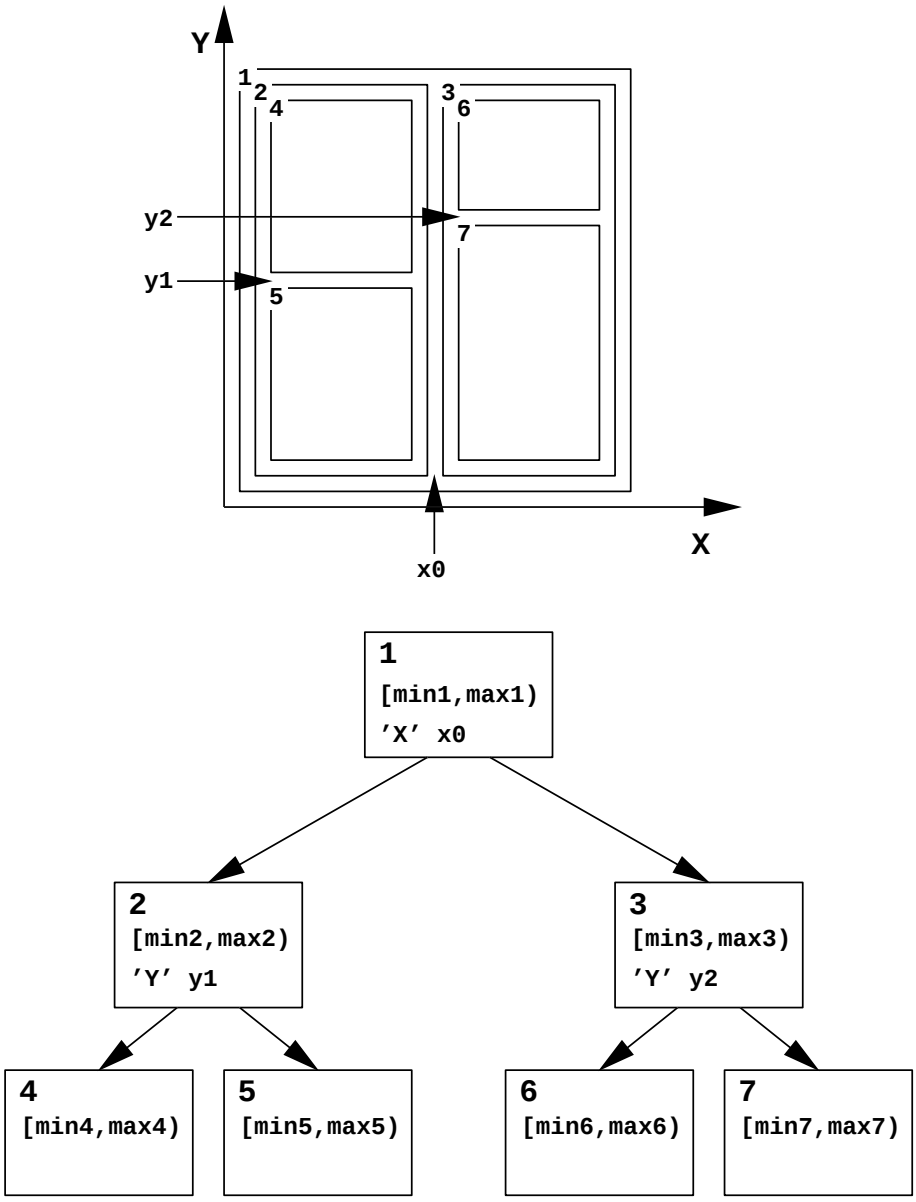


Abbildung 2.1: Struktur des binären adaptiven Baums zu einer gegebenen Partition der Volumendaten, vereinfacht auf den zweidimensionalen Fall

Wenn jedoch $\gamma \in [min, max)$ gilt, dann wird der Knoten weiter untersucht:

- Wenn er ein innerer Knoten ist, dann werden seine Nachfolger p_{left} und p_{right} in der gleichen Weise getestet.
- Wenn er ein Blatt ist, dann wird eine einfache Brute-Force-Suche innerhalb des von ihm repräsentierten Teilblocks gestartet. Die jeweils aktuellen Blockgrenzen sind von der rekursiven Suchfunktion bis hierher durchgegeben worden.

Die einfachste Idee, den Suchbaum zu bilden, ist es, jeden betrachteten Teilblock achsenparallel in der Mitte zu teilen. Dabei wechselt die Richtung der Teilungsebene zyklisch zwischen den drei gegebenen Möglichkeiten. Diese Idee entspricht ungefähr der eines Octrees, der jeden Teilblock in acht (fast) gleiche Unterblöcke zerlegt. Der wesentliche Unterschied zwischen binären Bäumen und Octrees besteht darin, dass Octrees jeweils sieben Knoten der binäre Bäume in einem zusammenfassen, wodurch ein Datenblock mit etwas weniger Speicheraufwand in acht Teilblöcke zerlegt wird. Dafür ist der Octree aber nicht so flexibel und ermöglicht es nicht, in einem Knoten z. B. einen von zwei Teilblöcken von vornherein auszuschließen. Im Octree ist auch eine mehrfache Zerlegung eines langen Datensatzes in der gleichen Richtung nicht vorgesehen.

Aufbauend auf dem erwähnten Bericht von Wilhelms/van Gelder beschreiben Sutton und Hansen in [29], wie sich ein Branch-on-Need-Octree für irreguläre Gitter konstruieren lässt. “Branch on need” heißt dabei, dass jeder Zweig des Octrees nur bis zu der Tiefe geht, an der der vom Blattknoten definierten Bereich genau einem der gegebenen unregelmäßigen Polyeder entspricht.

Ferner zeigen Weinstein und Johnson in [31], wie die Hierarchie eines Octrees eine Hierarchie der Definition der Isofläche erzeugen kann, die in diesem Fall als Multimesh repräsentiert ist.

Im 4. Kapitel wird untersucht, wie das Ergebnis von Partitionsbäumen (Speicher- und Zeitbedarf) durch die Wahl einer geeigneten, datenabhängigen Zerlegung verbessert werden kann.

2.2 Die Grundidee der intervall-basierten Suchmethoden

Der eigentliche Kern des Problems der Isoflächen-Extraktion ist die Intervallsuche: Es ist eine Liste von Volumen-Zellen mit ihrer Position (x, y, z) und ihren Intervallwerten min und max gegeben. Dabei ist für die Suche nach einem gegebenen Isowert γ die Position gar nicht von Bedeutung, weil nur die Bedingung $\gamma \in [min, max)$ zu testen ist.

In den folgenden Abschnitten werden intervall-orientierte Suchmethoden beschrieben, die von dieser Tatsache Gebrauch machen, indem sie alle gegebenen Zellen in einer vergleichsweise großen Datenstruktur systematisch nach min und max sortiert einordnen.

Methoden, die mit den Intervallen arbeiten, können im Span Space grafisch dargestellt werden. Der Span Space ist ein Min-Max-Diagramm, in dem alle Intervalle $[a, b)$ als Punkt (a, b) gezeigt werden und folglich oberhalb der Linie mit der Gleichung $x = y$ liegen. (siehe Abbildung 2.2).

Eine Span-Space-Methode, die in diesem Kapitel nicht gesondert erklärt wird, wird in [7] von Bordoloi und Shen vorgestellt. Diese ähnelt der ISSUE-Methode, weil sie den Span Space ebenfalls durch Linien in Rechtecke zerlegt. Diese Linien sind jedoch nicht mehr achsenparallel im $\min - \max -$Raum, sondern stattdessen achsenparallel im $u - v-$Raum, wobei $u := \min + \max$

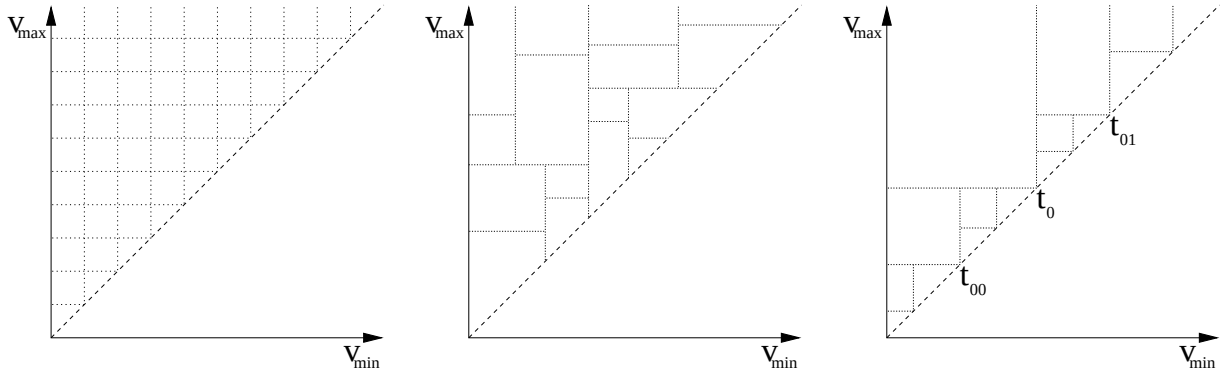


Abbildung 2.2: Hier sind die für die intervall-orientierten Methoden verwendeten Zerlegungen des Span Space (Min-Max-Diagramm) dargestellt. Von links nach rechts: ISSUE-Methode, KD-Tree, Intervallbaum

das doppelte des Intervallzentrums und $v := \max - \min$ die Länge des Intervalls ist. Das Intervall-Kriterium lautet dann $|2\gamma - u| < v$ und ist leichter zu untersuchen, weil die Intervalllänge v meistens einen kleinen Wert hat und somit die Suche nach Intervallen, deren Zentrum $\frac{u}{2}$ weiter von γ entfernt ist, oft abgekürzt werden kann.

2.3 "Iso-surfacing in Span Space with Utmost Efficiency" (ISSUE)

Die ISSUE-Methode ist die erste und einfachste intervall-orientierte Suchmethode und wurde von Shen, Hansen, Livnat und Johnson beschrieben ([27]).

Die Grundidee ist es, dass die vorgegebenen Intervalle als Punkte mit den Koordinaten (min, max) in den Span-Space eingeordnet werden können. Wenn nun ein Isowert γ vorgegeben wird, dann sind im Span-Space alle Punkte im Quadranten oben links vom Punkt (γ, γ) gesucht. In Abbildung 2.3 sind ein paar mögliche Suchbereiche dargestellt.

Eigentlich operieren alle in diesem Kapitel aufgeführten Methoden im Span Space, weil das gerade in der Natur der intervall-orientierten Methoden liegt. Die Arbeit von Shen et al. hat jedoch die Besonderheit, dass sie den Grundgedanken, den Span Space zu benutzen, explizit ausdrückt und ausführlich beschreibt und somit ein Modell zur Verfügung stellt, mit dem man Intervallbäume, KD-Trees und andere intervall-orientierte Methoden vergleichen kann.

Die Suche wird bei der ISSUE-Methode durchgeführt, indem der Span-Space wie in Abbildung 2.2 in Quadrate zerlegt wird. In der Suchphase brauchen dann nur diejenigen unter den Quadrate durchsucht zu werden, die eine nicht leere Schnittmenge mit dem gegebenen Quadranten bilden. Die Intervalle in den Quadraten, die ganz vom Quadranten überdeckt werden, können sogar vollständig in die Lösungsmenge übernommen werden.

Die Effizienz dieser Methode wird zusätzlich gesteigert, indem die nicht leeren Quadrate mit ihren Intervallen in Form von verketteten Listen gespeichert werden, die abhängig von der zur Verfügung stehenden Rechner-Konfiguration nacheinander bei sequentiellen Rechnern oder gleichzeitig bei parallelen Rechnern mit mehreren Prozessoren durchsucht werden.

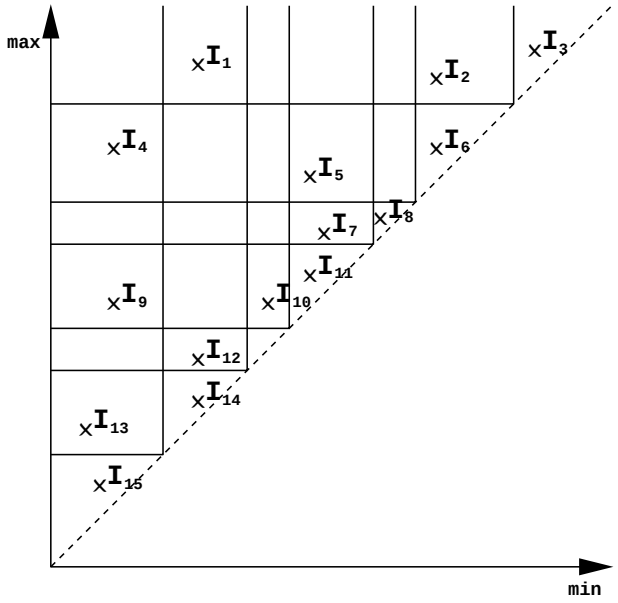


Abbildung 2.3: Der Span-Space, in den bei intervall-basierten Methoden die Intervalle eingeordnet werden. Der Suchbereich zu einem Isowert wird jeweils durch eine Doppellinie begrenzt, die auf der Linie mit $max = min$ ihre Richtung ändert.

2.4 Der KD-Tree

Die Betrachtung weiterer intervall-orientierter Suchmethoden legt zunächst die häufig in der Datenbanksuche verwendete Methode nahe, die gegebenen Intervalle abwechselnd nach min und max vorzusortieren und so einen Indexbaum zu errichten. Diese Sortierung wird durch einen Suchbaum wie in Abbildung 2.4, den so genannten $KD-Tree$ dargestellt, in dem entlang einem Pfad die Intervalle abwechselnd nach einem Entscheidungs-Kriterium der Form $\min < \min_k$ versus $\min \geq \min_k$ bzw. $\max < \max_k$ versus $\max \geq \max_k$ in jeweils zwei Gruppen aufgeteilt werden. $\min_k$ bzw. $\max_k$ ist dabei in den jeweiligen Knoten des Baums eingetragen und so gewählt, dass die betrachtete Intervallliste in zwei ungefähr gleich große Teillisten zerlegt wird (muss nicht genau stimmen). Livnat, Shen und Johnson haben die KD-Tree-Methode in [23] beschrieben. In diesem Artikel steht auch, in Anhang B, wie die Worst-Case-Schätzung von $O(k + \sqrt{n})$ Suchzeit pro Extraktion bestimmt wird. Der ursprüngliche, von Isoflächen unabhängige KD-Tree wurde schon 1975 von Bentley mit [6] eingeführt.

Der rekursive Algorithmus, mit dem in einem so gebildeten KD-Tree nach einem Isowert γ gesucht wird, ist relativ einfach aufgebaut:

- Für $\min$-Knoten vergleichen wir γ mit dem eingetragenen $\min$-Wert. Der linke Nachfolger wird immer untersucht, der rechte nur dann, wenn $\gamma > \min_k$ gilt.
- Für $\max$-Knoten vergleichen wir γ mit dem eingetragenen $\max$-Wert. Der rechte Nachfolger wird immer untersucht, der linke nur dann, wenn $\gamma < \max_k$ gilt.
- Wenn der Knoten ein Blatt ist, dann sind darin ein Intervall $[\min, \max)$ und eine Liste von Tripeln (x, y, z) eingetragen. Für sämtliche Einträge (x, y, z) der Liste gilt $\min_{xyz} = \min$

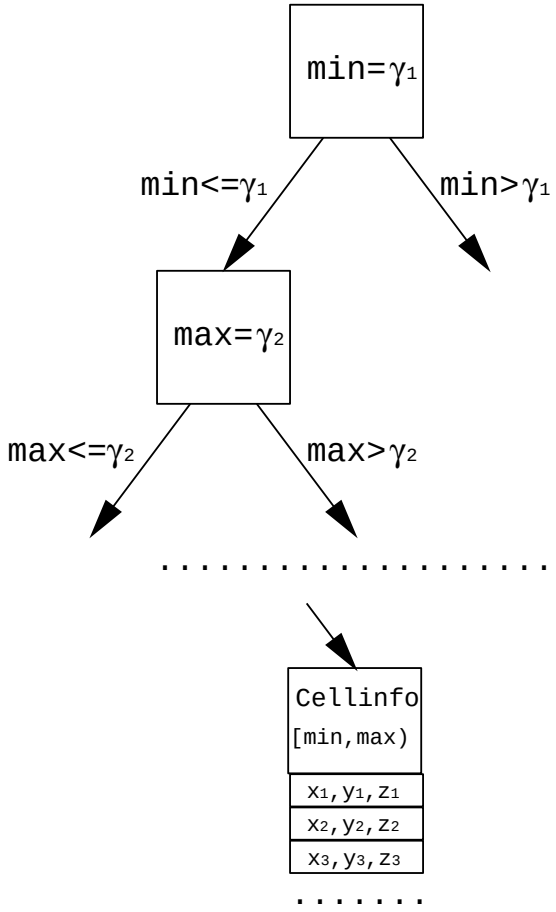


Abbildung 2.4: Der KD-Tree mit seinen nach min und max unterteilenden Knoten. Die Koordinaten (x_1, y_1, z_1), (x_2, y_2, z_2), (x_3, y_3, z_3) ... geben die Positionen an, an denen sich die Zellen mit dem Intervall $[min, max)$ befinden.

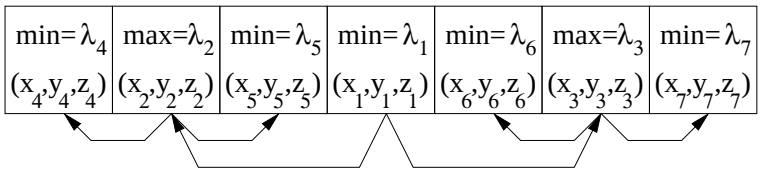


Abbildung 2.5: Der KD-Tree in seiner intern gespeicherten Arrayform. Die Pointer sind nicht in der Datenstruktur enthalten, sondern hier nur zur Orientierung dargestellt.

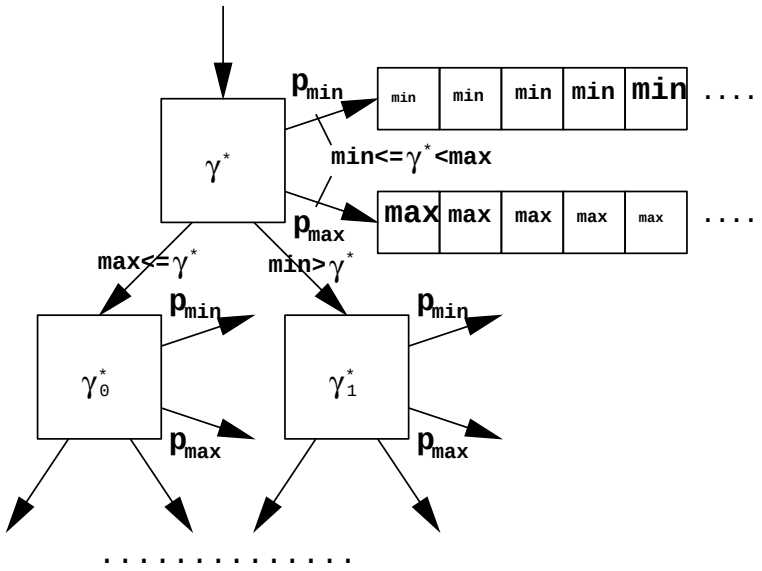


Abbildung 2.6: Der Intervallbaum: jeder innere Knoten hat Zeiger auf jeweils eine Min- und Maxliste. Diese enthalten die gleichen Zellen des Volumen-Datensatzes in verschiedener Anordnung

und $\max_{xyz} = \max$. Es wird also geprüft, ob $\gamma \in [\min, \max)$ gilt. Wenn die Intervallbedingung zutrifft, dann werden alle Tripel der Liste als Ergebniszellen ausgegeben.

Intern wird der KD-Tree nicht als Baum wie in Abbildung 2.4, sondern als Array wie in Abbildung 2.5 gespeichert. Dabei sind die Knoten des Baums durch einfache Indexberechnung zu finden, da sich der Knoten des Baums stets in der Mitte des Array befindet und ihre Nachfolger in der Mitte des jeweils linken bzw. rechten Teilarrays zu finden sind. In jedem dieser "Knoten" ist die Information für jeweils eine Zelle des Volumen-Datensatzes zu finden, die im Span Space genau auf der Grenzlinie der entsprechenden Unterteilung liegt.

2.5 Schnelle Isoflächen-Extraktion mit Intervallbäumen

Als nächstes wird eine weitere Methode der Intervallsuche beschrieben, die wie die KD-Tree-Methode ursprünglich nicht aus dem Gebiet Computergrafik, sondern vielmehr aus dem allgemeineren Bereich "Algorithmen und Datenstrukturen" stammt. Der Intervallbaum (Abbildung 2.6) ermöglicht eine schnelle, fast sogar rechenzeit-optimale Durchführung der Isoflächen-Extraktion. Das Intervallbaum-Verfahren wird von Edelsbrunner in [15] vorgestellt. Cignoni, Marino, Montani, Puppo und Scopigno haben in [13] die Möglichkeit aufgezeigt, Intervallbäume für die Extraktion von Isoflächen zu benutzen. Ferner beschreiben die gleichen Autoren außer Marino in [14], dass sich Intervallbäume auch für irreguläre Gitter verwenden lassen.

Der Intervallbaum ist ein binärer Baum, in dem jeder Knoten einen Iso-Grenzwert γ^*, Zeiger auf je eine Min- und Maxliste von Intervallen ($p_{\min}$ und $p_{\max}$), sowie Zeiger auf die beiden Nachfolger bzw. leere Zeiger enthält.

Der Intervallbaum kann rekursiv konstruiert werden, indem die vorgegebene Liste der Intervalle wiederholt in drei Teile zerlegt wird: die linke Teilliste enthält alle Intervalle $[min, max)$ mit

$max \leq \gamma^*$, die rechte Teilliste alle Intervalle mit $min > \gamma^*$ und die mittlere Teilliste alle Intervalle, die γ^* enthalten. Dabei kann γ^* im ersten Lösungsansatz so gewählt werden, dass die linke und die rechte Teilliste ungefähr gleich viele Intervalle enthalten; später werde ich erklären, wie der Intervallbaum durch eine ausgefeiltere Wahl von γ^* weiter optimiert werden kann. Nun wird die linke Teilliste an den linken Nachfolger und die rechte Teilliste an den rechten Nachfolger des Knotens übergeben und dort in gleicher Weise weiterverarbeitet. Die Intervalle der mittleren Teilliste werden in zwei Listen abgelegt: in der Minliste $l_{\min}$ nach aufsteigendem Minimum und in der Maxliste $l_{\max}$ nach fallendem Maximum sortiert.

Wenn der Intervallbaum einmal konstruiert ist, dann kann in diesem Baum die Suche nach einem Isowert γ folgendermaßen, ausgehend von der Wurzel des Baums, durchgeführt werden:

- Vergleiche γ mit dem eingetragenen γ^* des bearbeiteten Knotens
- Wenn $\gamma < \gamma^*$, dann
  - Gib die in der Minliste eingetragenen Intervalle so weit aus, wie die Ungleichung $\min \leq \gamma$ für diese Intervalle noch erfüllt ist. Die obere Intervallgrenze braucht wegen $\gamma < \gamma^* < \max$ nicht geprüft zu werden.
  - Wiederhole diesen Vorgang für der linken Nachfolger des Knotens, wenn dieser existiert. Der rechte Nachfolger braucht nicht mehr betrachtet zu werden, weil $\gamma < \gamma^* \leq \min$ für alle darin enthaltenen Intervalle gilt.
- ansonsten
  - Gib die in der Maxliste eingetragenen Intervalle so weit aus, wie die Ungleichung $\max > \gamma$ erfüllt ist
  - Wenn $\gamma = \gamma^*$ gilt, dann ist die Suche beendet
  - Ansonsten wiederhole diesen Vorgang für den rechten Nachfolger des Knotens, wenn dieser existiert.

Wie leicht zu sehen ist, ist diese Funktion nur scheinbar rekursiv, weil sie sich selbst jeweils nur einmal ganz am Ende aufruft. Sie kann also durch eine einfache $while$-Schleife ausgeführt werden, die einen Suchpfad im Intervallbaum durchläuft. Wir können also leicht die erwartete Suchzeit abschätzen, weil sie sich zusammensetzt aus der Zeit, die benötigt wird, um einen Suchpfad im Intervallbaum runter zu laufen, und der Zeit, die für die Ausgabe der Ergebniszellen gebraucht wird. Bei hinreichend geschickter Wahl des Intervallbaums, wenn also die Länge sämtlicher Pfade von $O(log(n))$ beschränkt ist, wird eine Extraktion in $O(k + log(n))$ durchgeführt.

Eine Variation der Idee mit dem Intervallbaum stammt von Chiang und Silva. Diese Autoren haben eine Methode beschrieben, bei der ein (nicht mehr unbedingt binärer) Segmentbaum abgespeichert wird, um den Platzbedarf im Hauptspeicher minimal zu halten. Durch die Abweichung von der Forderung eines binären Baums sind in jedem Knoten anstelle eines Grenzwerts γ^* mehrere Grenzwerte gespeichert, außerdem wird zu jedem dieser Grenzwerte eine Min- und Maxliste und zu jedem Intervall zwischen zwei Grenzwerten eine Multiliste benötigt, um eine reibungslose Suche zu ermöglichen. Genaueres zu dieser Methode steht im Abschnitt über Out-of-Core-Methoden dieser Arbeit sowie in [10].

Bajaj, Pascucci und Schikore veröffentlichten in der Zusammenfassung [5] einen Beweis dafür, dass die von Intervallbäumen erreichte Suchzeit $O(k + log(n))$ die bestmögliche Worst-Case-Suchzeit unter allen vergleichs-basierten Extraktionsalgorithmen ist. Der Beweis dafür basiert auf der Konstruktion einer durch den Datensatz laufenden Kette aus $O(n)$ Zellen, die paarweise disjunkte charakteristische Intervalle haben. Die Suche nach dem passenden Intervall für einen beliebigen Isowert γ benötigt dann mindestens $O(log(n))$ Verzweigungen, dazu kommen $O(k)$ Rechenschritte für den output-sensitiven Teil des Programms.

Anmerkung: Der Beweis ist leichter nachvollziehbar, wenn er mit $O(\sqrt[3]{n})$ Zellen entlang einer Kante rekonstruiert wird. Die Abschätzung von $O(\log(n))$ Verzweigungen ergibt sich daraus dennoch, da $\log(\sqrt[3]{n}) = \frac{1}{3}\log(n) = O(\log(n))$ gilt.

Kapitel 3

Weitere Extraktionsmethoden

In diesem Kapitel werden ein paar von anderen Autoren beschriebene Methoden der Isoflächen-Extraktion beschrieben.

Im ersten Abschnitt des Kapitels beschreibe ich die Out-of-Core-Methoden von Chiang und Silva ([10, 11]), die die Datenstruktur nicht im Hauptspeicher, sondern auf einem externen Speichermedium ablegen. Dadurch wird der oft beschränkte Hauptspeicher entlastet und trotzdem eine kleinere Extraktionszeit erreicht. Diese wird jedoch durch Leseoperationen auf dem Speichermedium eingeschränkt, die oft viel mehr Zeit benötigen als Operationen innerhalb des Hauptspeichers.

Der zweite Abschnitt beschreibt das Prinzip der Seed-Set-Methode, die von Bajaj, Pascucci und Schikore eingeführt wird ([3, 4]). Eigentlich handelt es sich nicht um **die** Seed-Set-Methode, sondern um eine Klasse von Methoden, da die beiden Teilaufgaben Saatmengen-Reduktion und Rekonstruktion einer Isofläche aus der Saatmenge unabhängig voneinander auf verschiedene Weise gelöst und beliebig miteinander kombiniert werden können.

Im dritten Abschnitt wird die Time-Continuation-Methode von Shen und Johnson ([28]) beschrieben, bei der vorausgesetzt wird, dass sich der Isowert nur in kleinen Schritten verändert, wie das z. B. bei einer interaktiven Steuerung über einen Schieberegler der Fall wäre. Auch diese Methode passt nicht in das Modell der Dissertation, in deren Kostenanalysen davon ausgegangen wird, dass die angeforderten Isowerte unabhängig identisch verteilt sind.

3.1 Out-of-Core-Methoden

Chiang und Silva haben in [10] und [11] beschrieben, wie man eine Isoflächen-Extraktion bei knappem Speicherplatz durchführen kann. Ferner zeigen die beiden Autoren zusammen mit Farias und Wei in [9], wie sich das dafür benutzte Prinzip der Out-of-Core-Extraktion auf einem Rechner mit mehreren Prozessoren anwenden lässt.

Die Grundidee der Out-of-Core-Extraktion ist es, die Volumendaten und die Suchstruktur auf der Festplatte oder einem ähnlichen Datenträger zu speichern und jeweils nur den Teil in den Hauptspeicher einzulesen, der für die Suche gebraucht wird. Unter Verwendung dieses Prinzips haben Chiang/Silva drei verschiedene Suchprinzipien realisiert, nämlich eine volumendaten-orientierte Blocksuche und eine Suche im Intervallbaum sowie im Segmentbaum, einer verallgemeinerten, nicht binären Version des Intervallbaums.

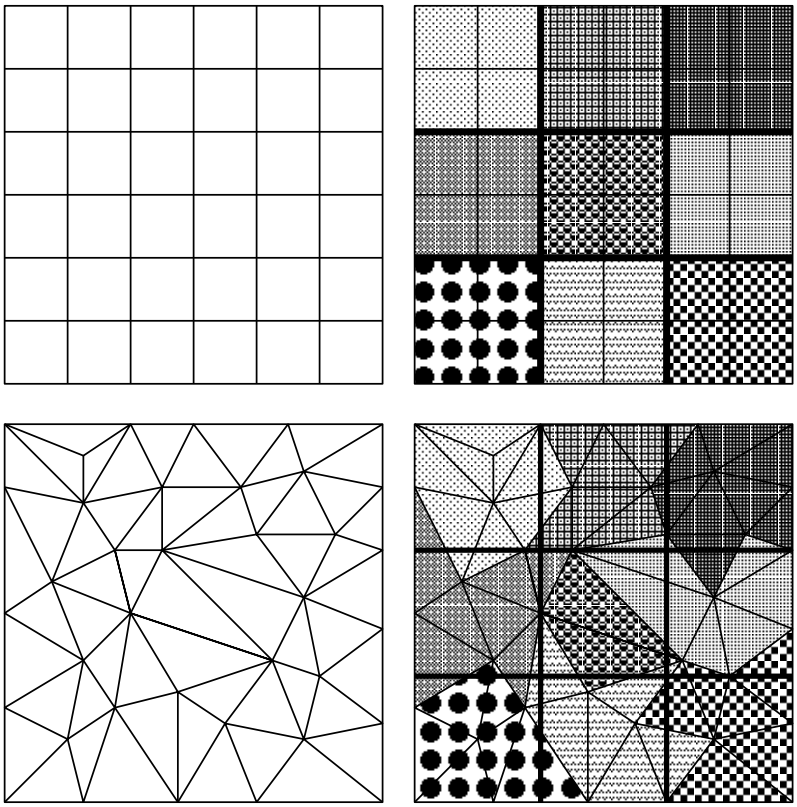


Abbildung 3.1: Ein Netz von Polygonen oder Polyedern wird in Metazellen zerlegt, die in einem gemeinsamen Block der Festplatte abgelegt sind. Im oberen Bild sind die Polygone die bei der üblichen Extraktion verwendeten Zellen. Im unteren Beispiel sieht man, dass diese Methode auch auf ungleichmäßige Netze anwendbar ist.

3.1.1 Blockorientierte Out-of-Core-Extraktion

Bei der blockorientierten Out-of-Core-Extraktion wird die Liste aller Zellen des Volumen-Datensatzes wie in Abbildung 3.1 in Blöcke, so genannte Metazellen verteilt, deren Informationen auf jeweils einem Speicherblock der Festplatte abgelegt sind. Ferner wird ein Index angelegt, der zu jedem Block das Minimum und das Maximum der darin enthaltenen Volumendaten beinhaltet und so zu jedem Isowert die Suche nach den Metazellen erleichtert, die Teile der Isofläche enthalten.

3.1.2 Intervallbaum-orientierte Out-of-Core-Extraktion

Von der intervallbaum-orientierten Out-of-Core-Extraktion haben Chiang und Silva zwei Versionen realisiert. Bei der ersten Version benutzen sie keine echten Intervallbäume, sondern so genannte Segmentbäume. Diese unterscheiden sich von den Intervallbäumen dadurch, dass jeder Knoten nicht nur zwei, sondern mehr Nachfolger haben kann. Ein Knoten des Segmentbaums enthält jeweils n Unterteilungswerte $\gamma(0), ..., \gamma(n-1)$, $n+1$ Zeiger auf Nachfolgerknoten, je n Zeiger auf left- und right-Listen und $\frac{n(n-1)}{2}$ Zeiger auf multi-Listen. Die left- und right-Listen entsprechen der min- und max-Liste im Intervallbaum und fassen jeweils die Intervalle zusammen, deren linke

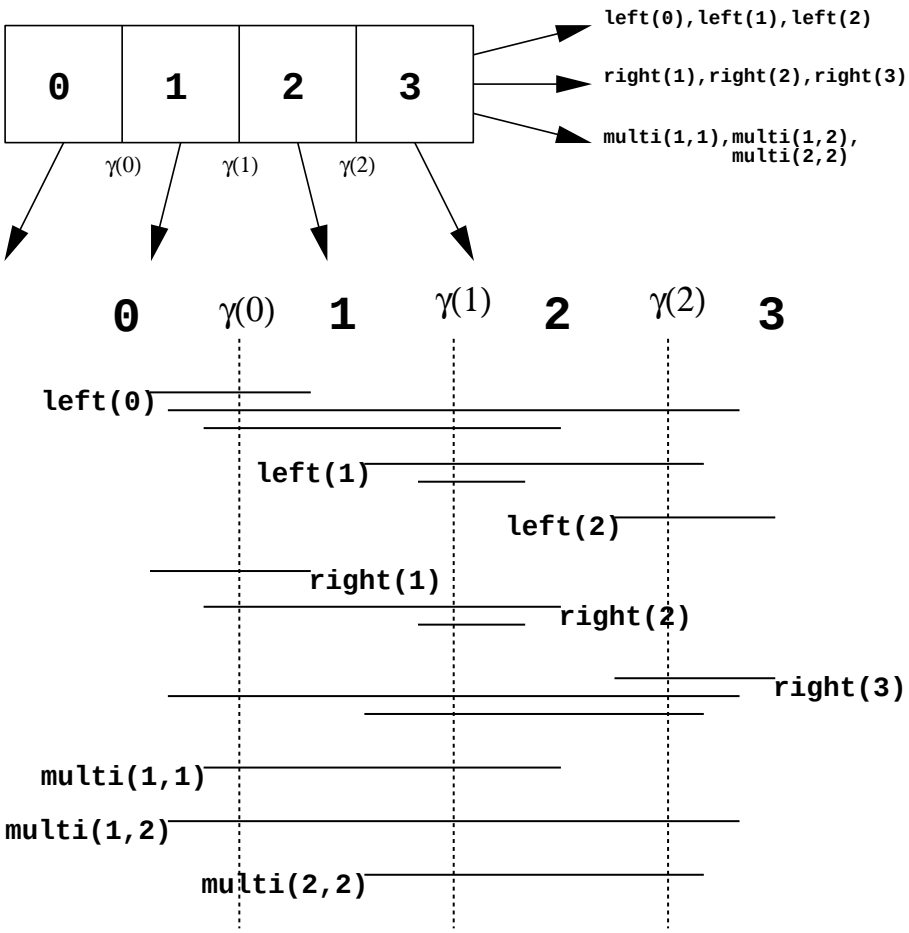


Abbildung 3.2: Ein Knoten des Segmentbaums. Darunter ist die zugehörige Einordnung von Beispielintervallen in die left-, right- und multi-Listen dargestellt, wobei jedes Intervall als Linie dargestellt ist.

bzw. rechte Grenze im selben Bereich (Slab) der Form $[\gamma(i), \gamma(i+1))$ liegen. Die multi-Listen fassen die Intervalle zusammen, die einen oder mehrere dieser Bereiche vollständig überdecken (Multislabs). Unberücksichtigt bleiben dabei nur die Intervalle, die ganz in einem Bereich enthalten sind. Diese werden an den entsprechenden Nachfolgerknoten weitergegeben. In Abbildung 3.2 ist ein Beispielknoten eines Segmentbaums mit der zugehörigen Verteilung der Intervalle auf die Slabs und Multislabs dargestellt.

Bei der zweiten Version werden wirklich Intervallbäume auf der Festplatte gespeichert. Damit ein Datenblock des Speichermediums voll ausgenutzt werden kann, wird darin wie auf Abbildung 3.3 ein Ausschnitt des Baums mit mehreren Knoten gespeichert. Außer den Blöcken, die Intervallbaum-Knoten enthalten, gibt es bei diesem Prinzip auch Min- und Maxlistenblöcke, die nach Bedarf sequentiell gelesen werden, sowie Restzellenblöcke, die dann verwendet werden, wenn eine Liste von Zellen in einem Teil des Intervallbaums vollständig in einen Datenblock passt. In diesem Fall werden die Zellen als Liste in einen Block gespeichert, anstatt die Konstruktion des Intervallbaums fortzusetzen.

Ich habe die Out-of-Core-Methode mit den echten Intervallbäumen ebenfalls realisiert und sowohl

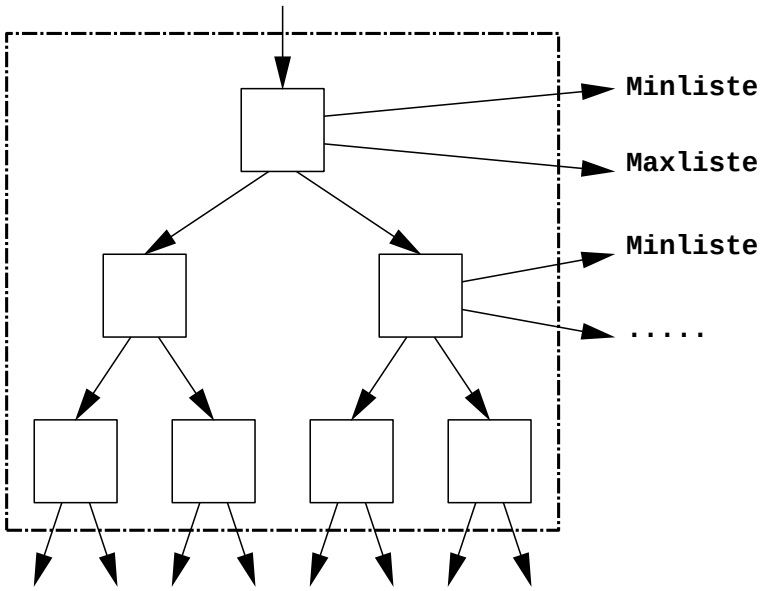


Abbildung 3.3: Ein Ausschnitt des Intervallbaums, der nach Chiang/Silva in einem Block der Festplatte gespeichert wird. Im dargestellten Beispiel werden 7 Knoten in 3 Leveln gespeichert, was bei einem Speicherbedarf von 24 Byte pro Knoten zu einem gesamten Speicherbedarf von 168 Byte pro Block führt.

einzeln als auch als Wahlmöglichkeit für die Conditioned Trees ausgetestet. Hier folgen nun einige numerische Ergebnisse, in denen der Speedup-Faktor gegenüber der Brute-Force-Methode und der benötigte Festplattenspeicher gemessen wurden.

Datensatz (Format)	Speedup-Faktor	Festplattenspeicher (MB)
Engine_256 ($256 \times 256 \times 110$)	31	85
Bighead_256 ($256 \times 256 \times 225$)	47	161
ScannedBrain400 ($384 \times 400 \times 276$)	21	641
Bunny ($360 \times 512 \times 512$)	97	1131

Von Prof. Dr. Daniel Keim stammte die Idee, bei der Out-of-Core-Extraktion mehrere Datenblöcke unmittelbar nacheinander zu lesen, um die Suchzeit zu beschleunigen. Ich habe dieses Experiment für den ScannedBrain-Datensatz durchgeführt und herausgefunden, dass tatsächlich geringfügige Unterschiede in der Extraktionszeit bestehen, nämlich dass die bestmögliche Suchzeit bei ca. 5 Datenblöcken (2,5 kB) pro Leseoperation erreicht wird. Da der größte Teil der Suchzeit für den Aufbau der Zellenliste benötigt wird, ist der Unterschied jedoch geringfügig und liegt bei einem Prozent, also für mein Beispiel im Bereich der Messgenauigkeit. Eine zufällige Messung dieses Unterschieds ist jedoch auszuschließen da jeder der aufgelisteten Werte mehrmals gemessen wurde, um die Tendenz zu bestätigen.

Leseoperation (kB)	Extraktionszeit (s)
0.5	2.74
1	2.73
1.5	2.73
2	2.73
2.5	2.73
3	2.73
3.5	2.73
4	2.73
4.5	2.74
5	2.74

Leseoperation (kB)	Extraktionszeit (s)
6	2.74
7	2.74
8	2.74
9	2.74
10	2.74
12.5	2.75
15	2.75
17.5	2.75
20	2.75
25	2.75

Die Suche nach der optimalen Verwendung der Out-of-Core-Methode in Conditioned Trees habe ich ebenfalls durchgeführt. Das Problem dabei ist allerdings, dass die Out-of-Core-Extraktion außer dem Hauptspeicher eine weitere Hardware-Komponente verwendet, nämlich den Festplattenspeicher. In Berichten, die sich hauptsächlich mit Out-of-Core-Methoden beschäftigen, wird meistens stillschweigend vorausgesetzt, dass Festplattenspeicher kostenlos und in beliebiger Menge zur Verfügung steht, was ich für unrealistisch halte. Aus diesem Grund habe ich den verwendeten Festplattenspeicher in die Kostenformel einbezogen, jedoch nicht genauso stark bewertet wie den Hauptspeicher, sondern mit einem Kostenfaktor $\mathcal{H}$ versehen, der dem Programm als Parameter übergeben werden kann.

Out-of-Core-Methoden werden in vielen Arbeiten über Isoflächen-Extraktion beschrieben. Ich persönlich glaube, dass Out-of-Core-Verfahren keine gute Zukunft haben, weil die Prozessoren der modernen Computermodelle immer schneller werden, während die Entwicklung bei externen Speichermedien nicht so schnell ist. Ein Grund dafür liegt darin, dass es einfach ist, einen Prozessor gegen einen schnelleren auszutauschen, ohne dafür viel an der Hardwareumgebung zu verändern. Wenn hingegen das externe Medium beschleunigt wird, ist das mit größerem Aufwand verbunden, weil dafür die Schnittstelle des Rechners angepasst werden muss (ein Speichermedium kann immer nur so schnell sein, wie der Zugriff auf ihn erfolgt). Oft muss dafür auch eine ganz neue Norm eingeführt werden, weil den häufig benutzten Speichermedien wie Disketten, CDs und Festplatten einer vorgegebenen Architektur, schon durch die physikalischen Gegebenheiten natürliche Grenzen gesetzt werden.

3.2 Die Seed-Set-Methode

In [3] und [4] haben Bajaj, Pascucci und Schikore die Seed-Set-Methode beschrieben, deren Zweck es ist, in einer kleinen, repräsentativen Menge von Zellen zu suchen und so eine schnelle Übersicht über alle Zusammenhangskomponenten der gesuchten Isofläche zu erhalten. Die Seed-Set-Methode wird ebenfalls von Kreveld, Oostrum und den eben genannten Autoren in [19] beschrieben, wo die Saatmenge durch Analyse der Extremwerte des Funktionsverlaufs ermittelt wird; diese werden ermittelt und durch einen so genannten Extremwertgraphen verbunden, der alle Zellen der ermittelten Saatmenge durchläuft. Der Algorithmus der Seed-Set-Methode ist nicht fest vorgegeben; er besteht vielmehr aus zwei Teilaufgaben, die sich unabhängig voneinander lösen lassen:

- **Preprocessing:** Aus der Menge aller Zellen des Datensatzes wird eine geeignete Teilmenge extrahiert. Diese muss zwei Eigenschaften haben, nämlich eine Saatmenge sein und zum

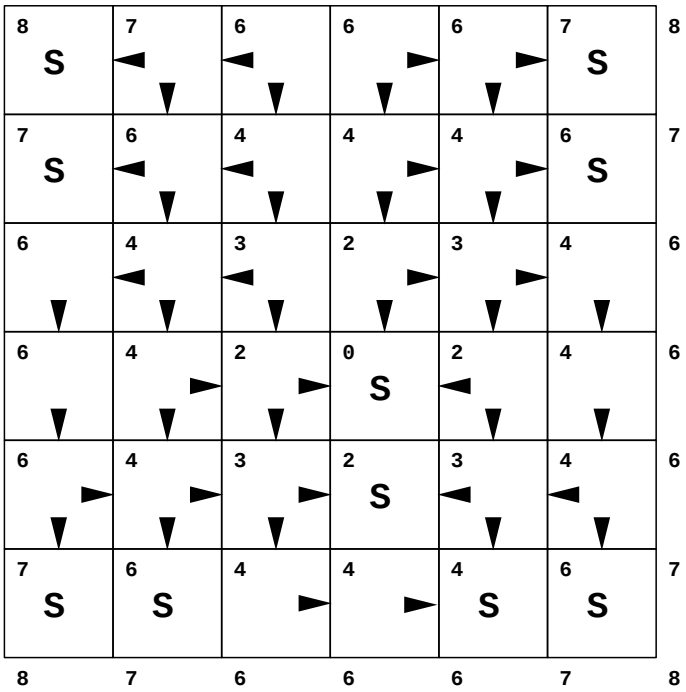


Abbildung 3.4: Ein Beispiel für die Saatmengen-Reduktion durch die Sweep-Filtering-Methode

Zweck der Effizienz so wenig Elemente wie möglich enthalten. Bei geschickter Wahl des Algorithmus ist für algebraische Volumendaten aus n Zellen eine Reduktion auf $O(n^{1/3})$ Zellen möglich. Die Saatmengen-Eigenschaft wird in diesem Abschnitt definiert. Ein Beispiel für eine geeignete Saatmengen-Reduktion ist die Sweep-Filtering-Methode, die in Abbildung 3.4 dargestellt ist. Die Abbildung zeigt einen Satz von Volumendaten, deren Werte die Zahlen an den entsprechenden Gitterpunkten sind. Bei zeilenweiser Bearbeitung von oben nach unten fallen die Zellen mit den Pfeilen durch die lokale Reduktionseigenschaft (wird später genau formuliert) nacheinander weg. Dabei verweisen die Pfeile auf die Nachbarzellen, die für die Voraussetzung der lokalen Reduktionseigenschaft benutzt werden, wobei der untere Nachbar der bearbeiteten Zelle stets zuerst untersucht wird. Am Ende des Reduktionsalgorithmus verbleiben nur noch die mit einem **S** markierten Zellen in der Saatmenge.

Im Artikel [18] ist eine weitere Möglichkeit dargestellt, die darauf basiert, dass die gegebene Menge unter Erhaltung der lokalen Topologie und der Zellen, die lokale Minima und Maxima der Volumendaten enthalten, reduziert wird. Das Ergebnis dieser Reduktion ist stets eine Saatmenge.

- **Extraktion:** Zunächst wird die Saatmenge entweder vollständig oder mit einer intervallorientierten Methode nach einem gegebenen Isowert durchsucht. Ausgehend von den gefundenen Zellen der Saatmenge werden dann durch eine Suche im Nachbarschaftsgraphen des Datensatzes alle Zellen der Isofläche durchlaufen. Für diesen Teil des Algorithmus kann z. B. eine Breiten- oder Tiefensuche verwendet werden.

Um die Rolle klar zu machen, die die Saatmengen in dieser Methode spielen, folgen hier noch einige Definitionen und Regeln:

Definitionen:

- Es sei c eine Zelle des gegebenen Gitters. Dann ist $\min(c)$ das Minimum und $\max(c)$ das Maximum aller Datenwerte in den Eckpunkten von c.
- $R(c) := [\min(c), \max(c))$ wird als der in c von der Funktion durchlaufene Wertebereich angenommen.
- Für einen gegebenen Isowert w ist c genau dann eine aktive Zelle, wenn $w \in R(c)$ gilt.
- Es seien c_1 und c_2 zwei benachbarte Zellen. Dann ist $R(c_1, c_2) := R(c_1) \cap R(c_2)$ der gemeinsame Wertebereich von c_1 und c_2.
- $\mathcal{G} = (G, Z)$ ist der Zusammenhangsgraph der Zellen, d. h. G ist die Menge aller Zellen und $Z \subseteq G \times G$ die Menge aller Paare von Zellen, die benachbart sind. Die genaue Definition der Nachbarschaftsrelation spielt dabei nur für Fragestellungen der Effizienz eine Rolle, nicht für das prinzipielle Funktionieren des Algorithmus.
- Für jede gegebene reelle Zahl w ist $\mathcal{G}_w$ definiert als der Subgraph von $\mathcal{G}$, der zur Isofläche von w gehört, d. h., $\mathcal{G}_w = (G_w, Z_w)$ mit $G_w := \{c \in G | w \in R(c)\}$ und $Z_w := \{(c_1, c_2) \in Z | w \in R(c_1, c_2)\}$
- Zwei Zellen c_1, c_2 heißen genau dann miteinander w-verbunden, wenn $c_1, c_2 \in G_w$ gilt und es einen Weg von c_1 nach c_2 innerhalb des Graphen $\mathcal{G}_w$ gibt.
- Es sei $c \in G$ eine Zelle und $S \subseteq G$ eine Menge von Zellen. Dann heißt c mit S verbunden, wenn für alle $w \in R(c)$ eine Zelle $c' \in S$ existiert, so dass c mit c' w-verbunden ist.
- $S \subseteq G$ heißt **Saatmenge** genau dann, wenn alle Zellen $c \in G$ mit S verbunden sind.

Eigenschaften von Saatmengen:

- **Saatmengen-Eigenschaft:** Ausgehend von den Zellen einer Saatmenge lassen sich zu jedem Isowert w alle Elemente von G_w durch eine lokale Graphensuche bestimmen. Dies ist die zentrale Eigenschaft, die die Benutzung von Saatmengen für die Isoflächen-Extraktion nahelegt.
- **Existenzeigenschaft:** Die Menge aller im Datensatz enthaltenen Zellen G ist eine Saatmenge.
- **Reduktionseigenschaft:** Wenn $S \subseteq G$ eine Saatmenge und $c \in S$ mit $S \backslash \{c\}$ verbunden ist, dann ist auch $S \backslash \{c\}$ eine Saatmenge.
- **Lokale Reduktionseigenschaft:** Wenn $S \subseteq G$ eine Saatmenge ist und $c \in S$ sowie eine Liste $c_1, ..., c_k \in S$ von Nachbarzellen von c gegeben ist, so dass die Eigenschaft $R(c) \subseteq \bigcup_{i=1}^{k} R(c_i)$ gilt, dann ist auch $S \backslash \{c\}$ eine Saatmenge. Die lokale Reduktionseigenschaft wird oft für die Konstruktion kleiner Saatmengen verwendet.

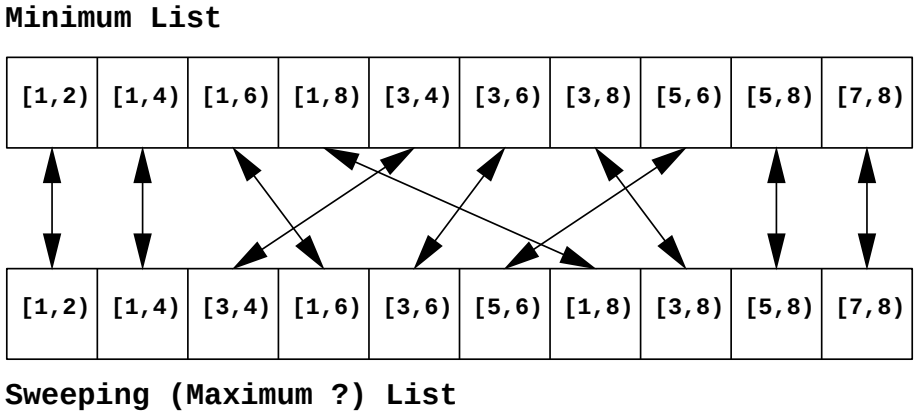


Abbildung 3.5: Die Minimum- und Sweeping-Liste der Time-Continuation-Methode

3.3 Die Time-Continuation-Methode

Shen und Johnson zeigen in [28], wie mit der Time-Continuation-Methode die Isofläche bei kleinen Änderungen des Isowerts innerhalb kurzer Zeit angepasst werden kann. Sie präsentieren einen Algorithmus, der bei einer bereits gegebenen Isofläche zum Isowert γ die Fläche zum Wert $\gamma + \epsilon$ innerhalb einer Zeit findet, die sich linear zum Volumen zwischen den beiden Flächen verhält, also im Mittel proportional zum Wert ϵ ist. Dadurch ist die Time-Continuation-Methode besonders gut für interaktive Programme geeignet, bei denen der Isowert meistens nur in kleinen Schritten geändert wird.

Das Prinzip der Time-Continuation-Methode ist es, alle Zellen zusammen mit ihren Intervallen in zwei Listen abzuspeichern, nämlich jeweils ein Mal nach dem Intervall-Minimum sortiert in der Minimum-Liste und nach dem Maximum sortiert in der Sweeping- oder Maximum-Liste (siehe Abbildung 3.5). In diesen beiden Listen sind einander entsprechende Einträge miteinander verzeigert. Eventuell enthält nicht jede der beiden Listen alle Informationen zu den Zellen, sondern nur die Teilinformationen, die für einen hinreichend schnellen Zugriff gebraucht werden.

Ferner sind noch die beiden Zeiger oder Indizes p_L (Last) auf einen Eintrag der Minimum-Liste und p_F (First) auf einen Eintrag der Sweeping-Liste gegeben: wenn als letztes die Isofläche zum Isowert γ bestimmt worden ist, dann zeigt p_L auf das letzte Element der Minimum-Liste, dessen Intervall-Minimum kleiner oder gleich γ ist; entsprechend zeigt p_F auf das erste Element der Maximum-Liste, dessen Maximum größer als γ ist.

Wenn der Wert von γ geändert wird, dann ändern sich dadurch auch die Zeiger p_F und p_L (siehe Abbildung 3.6). Angenommen, der Isowert $\gamma = \gamma_1$ wird größer und erhält den neuen Wert γ_2. Dann rücken dadurch die Zeiger p_F und p_L nach rechts und die Isofläche zu γ_2 kann außer den schon bekannten Zellen der Isofläche zu γ_1 nur Zellen enthalten, die zwischen der alten und neuen Position des Zeigers p_L eingetragen sind. Analog dazu werden bei einer Verkleinerung von γ neue Zellen nur zwischen der alten und neuen Position des Zeigers p_F gesucht.

Umgekehrt können die wegfallenden Zellen bei einer Vergrößerung von γ zwischen der alten und neuen Position des Zeigers p_F in der Maximum-Liste gefunden werden; analog dazu findet man sie bei einer Verkleinerung von γ zwischen der alten und neuen Position von p_L in der Minimum-Liste. Aus der Anzahl der Positionen, um die die beiden Zeiger p_F und p_L geändert werden, ergibt

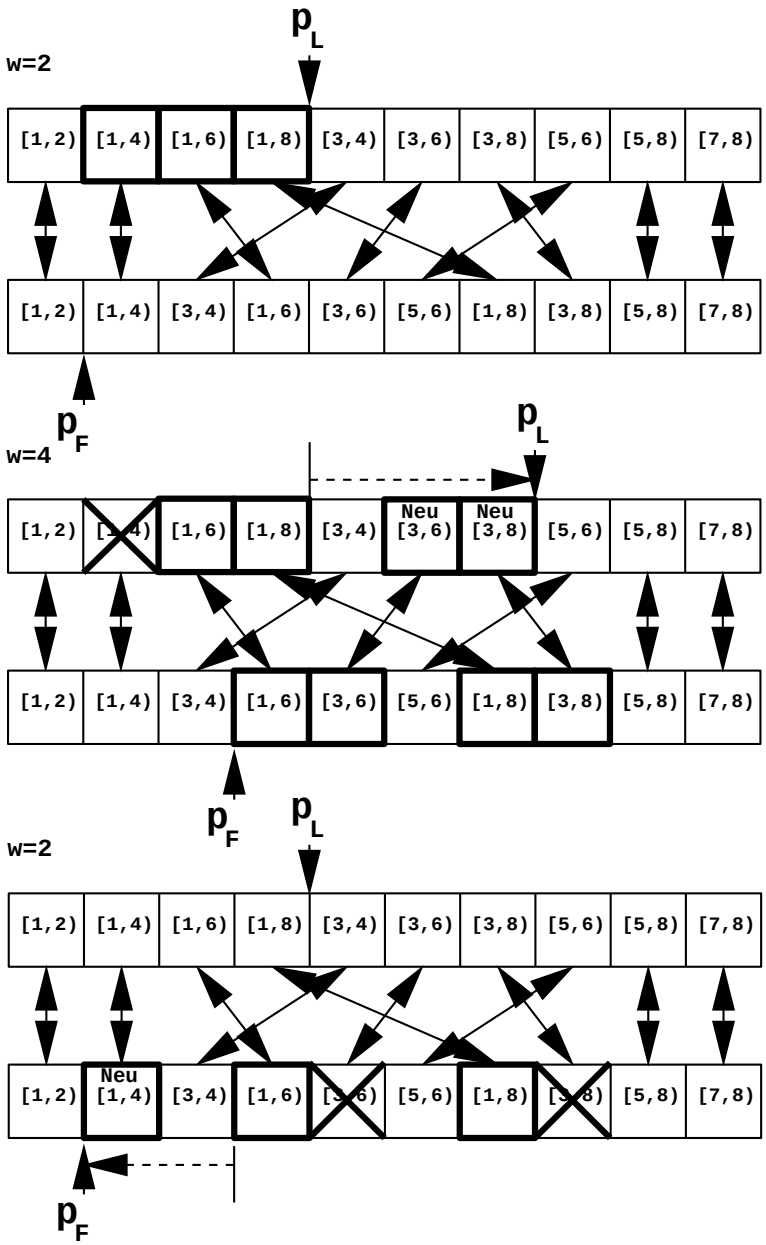


Abbildung 3.6: Ein Beispiel der Time-Continuation-Methode. Hier wird der Isowert erst von 2 auf 4 vergrößert, dann wieder auf 2 verkleinert. Die fett umrahmten Intervalle bilden jeweils die aktuelle Isofläche; die neu hinzugekommenden Zellen sind in der verwendeten Suchliste mit **Neu** gekennzeichnet und die wegfallenden Zellen sind durchgestrichen.

sich die am Anfang dieses Abschnitts angegebene Suchzeit.

3.4 Literatur zu anderen Methoden der Isoflächen-Extraktion

Diverse Autoren beschreiben weitere Lösungsansätze für abweichende Aufgabenstellungen der Isoflächen-Extraktion.

Chiang geht in [8] davon aus, dass es nicht nur einen Volumen-Datensatz gibt, sondern eine Folge von Datensätzen, die abhängig von einem Zeitparameter gewechselt werden. Diese werden in einer partitionsbaumartigen Datenstruktur gespeichert, in der jeder Knoten nicht nur den Volumen- und Intervallbereich, sondern auch den Zeitbereich einschränkt. Darauf aufbauend beschreibt Shen in [26] eine span-space-orientierte Methode, die mit zeitabhängigen Volumendaten arbeitet.

Livnat und Hansen beschreiben in [21] und [22] eine Extraktionsmethode, die abhängig von der jeweiligen Perspektive nur den sichtbaren Teil der Isofläche extrahiert. Dies geschieht mit der Shear-Warp-Methode, die durch eine Transformation (Shear) die auf einer Blicklinie enthaltenen Teile des Volumens miteinander zur Deckung bringt und die so erhaltene Ebene durch eine weitere Transformation (Warp) in das gesehene Bild umwandelt.

Wilhelms und Gelder beschreiben in [33] eine Methode, mit der die gesuchte Isofläche durch Interpolation innerhalb einzelner Zellen ermittelt werden kann. Fujishiro, Maeda, Sato und Takeshima zeigen in [17], wie ein $\alpha-\beta-$Bereich zellenweise aus einem Datensatz erhalten werden kann. Dieser ist eine etwas breitere Version der Isofläche, für den der Datenwert zwischen zwei gegebenen Werten α und β liegt.

Weigle und Banks erweitern das Problem der Isoflächen-Extraktion in [30] auf den n-dimensionalen Fall. Sie zeigen, wie sich ein Hyperwürfel durch eine kanonische Methode in $2^n n!$ Simplizes oder durch eine auf Permutation basierende Methode in $n!$ Simplizes zerlegen lässt, in denen die für die Iso-Hyperflächenermittlung durch Interpolation einfacher ist als im Hyperwürfel.

Kapitel 4

Qualitätsverbesserung bei Partitionsbäumen

In diesem Kapitel werden einige Möglichkeiten der Verbesserung der in Abschnitt 2.1 beschriebenen Partitionsbäume gezeigt. Als Maß für diese Verbesserung dienen der Speicherbedarf des Baums und der Erwartungswert der benötigten Rechenzeit für eine Extraktion innerhalb des Baums.

In den beiden nächsten Abschnitten wird beschrieben, wie der Erwartungswert der benötigten Suchzeit für einen gegebenen Partitionsbaum sehr zeitaufwändig berechnet werden kann, und wie eine Approximation innerhalb kürzerer Zeit möglich ist. Indem der Erwartungswert der Suchzeit minimiert wird, wird der bestmögliche Baum einer vorgegebenen Tiefe erreicht. Im darauf folgenden Abschnitt werden Möglichkeiten beschrieben, einen gegebenen Partitionsbaum in für den Speicher- und Zeitbedarf sinnvoller Weise zu reduzieren.

4.1 Varianz-optimierte Bäume

Wie leicht zu erkennen ist, ist die normierte Länge des Intervalls, das einem Knoten des adaptiven Baums zugeordnet ist, ein mögliches Wahrscheinlichkeitsmaß dafür, dass die Nachfolger dieses Knotens untersucht werden müssen. Aus diesem Grund bilden wir einen Baum, der diese Intervalllängen minimiert, oder - statistisch geschickter - die Varianzen oder mittleren quadratischen Abweichungen der Daten innerhalb der Teilblöcke minimiert. Der Vorteil der Varianzen gegenüber den absoluten Intervalllängen ist es, dass einzelne Ausreißer unter den Datenwerten, die bei entsprechender Verfeinerung in den meisten Teilblöcken verschwinden, auch bei der Bewertung nicht so stark ins Gewicht fallen.

Natürlich ist es letztendlich die Summe der Intervalllängen aller inneren Knoten, die minimal werden sollte. Diese zu bestimmen und zu minimieren, ist zwar durch eine rekursive Formel möglich, aber deren Realisierung ist eindeutig zu zeitaufwändig. Dafür muss nämlich jeder der möglichen $O(n^6)$ Teilblöcke des Datensatzes untersucht werden, von denen jeder durchschnittlich $O(n)$ weitere Zerlegungen hat; der Vergleich dieser Zerlegungen benötigt also $O(n^7)$ Rechenzeit. Aus diesem Grund minimieren wir bei jeder Unterteilung jeweils die Summe der Varianzen der beiden Teilblöcke. Diese ist auch ein Maß dafür, wie schnell die Intervalllänge bei einer weiteren Zerlegung des Volumen-Datensatzes abnimmt, denn es ist dadurch möglich festzustellen, wie stark die

Daten innerhalb des betrachteten Blocks variieren und wie wahrscheinlich es ist, dass eine weitere Zerlegung des Blocks zu einer Reduktion der Intervall-Längen führt.

Wir bilden den Zerlegungsbaum auf den Volumendaten rekursiv, indem wir für jeden betrachteten Teilblock, der laut Abbruchbedingung weiter unterteilt werden muss, folgende Schritte durchführen:

- Es sei $\delta_{\min} := +\infty$ (Es ist noch keine Zerlegung des betrachteten Blocks gefunden, also wird für den zu minimierenden Kostenterm ein Wert eingesetzt, der auf jeden Fall unterboten werden kann)
- Für jede der drei achsenparallelen Richtungen
  - Setze $s_{\text{res}} := 0$
  - Zerteile den Block in dieser Richtung in Scheiben. Eine Scheibe ist dabei die Menge aller Datenpunkte, für die die Koordinate der betrachteten Richtung einen festgelegten Wert hat. Es sei k die Anzahl der Scheiben.
  - Für jede dieser Scheiben $pl = 1...k$ tue
    - Berechne avg_{pl}, den Durchschnitt der Datenwerte in pl
    - Berechne mit Hilfe von avg_{pl} den Wert res_{pl}, die Varianz der Datenwerte von pl
    - Sei $s_{res} := s_{res} + res_{pl}$ (Summe der Einzelscheiben-Varianzen)
  - Für jeden der Werte $pl = 2...k-1$ tue
    - Es sei h_1 die reduzierte Summe der Quadrate der Werte avg_{ps} mit $ps \leq pl$ ('linker' Teil des Blocks), also $h_1 := \sum_{ps=1}^{pl} (avg_{ps} - a_1)^2$ mit $a_1 := \frac{1}{pl} \sum_{ps=1}^{pl} avg_{ps}$
    - Es sei h_2 analog dazu die reduzierte Summe der Quadrate der Werte avg_{ps} mit $ps \geq pl$ ('rechter' Teil des Blocks) Der Fall $ps = pl$ wird dabei sowohl im diesem, als auch im vorangehenden Schritt betrachtet, weil er genau die Scheibe darstellt, die den linken und den rechten Teil des Blocks voneinander trennt.
    - Berechne $\delta := \frac{1}{k}(s_{res} + res_{pl} + h_1 + h_2)$, das gewichtete Mittel der Varianzen der betrachteten Blockpartition
    - Wenn $\delta < \delta_{\min}$, dann setze $\delta_{\min} := \delta$ und betrachte die Scheibe pl als Trennungsebene der momentan besten Blockzerlegung
- Alloziere einen adaptiven Baumknoten
- Trage die Informationen über die Ausrichtung und Lage der optimalen Trennungsebene in den Knoten ein
- Trage die Werte des Datenmaximums $a_{\max}$ und des Minimums $a_{\min}$ des betrachteten Blocks in den Knoten ein
- Rufe diese Funktion rekursiv für den linken und den rechten Teilblock der Zerlegung auf und trage die zurückgegebenen Zeiger in den Knoten ein
- Gib einen Zeiger auf den Knoten als Funktionswert zurück

Der Sinn des rechnerischen Teils dieses Algorithmus ist es, für jede mögliche Zerlegung des Blocks in Teilblöcke die Summe der entsprechenden Teilblock-Varianzen zu bestimmen, um anschließend über diese zu minimieren. Um die Varianz im linken Block zu bestimmen, wird die Summe der Varianzen seiner Scheiben genommen und dazu die Varianz der Durchschnittswerte dieser Scheiben addiert, also gilt:

$$var_{\text{left}} = \sum_{ps=1}^{pl} res_{ps} + \sum_{ps=1}^{pl} (avg_{ps} - a_1)^2 =$$

$$\sum_{ps=1}^{pl} res_{ps} + h_1$$

und entsprechendes für den rechten Block:

$$var_{\text{right}} = \sum_{ps=pl}^{k} res_{ps} + \sum_{ps=pl}^{k} (avg_{ps} - a_1)^2 =$$

$$\sum_{ps=pl}^{k} res_{ps} + h_2$$

Daraus folgt für die Summe der Varianzen der beiden Blöcke:

$$\delta = var_{\text{left}} + var_{\text{right}} =$$

$$\sum_{ps=1}^{k} res_{pl} + res_{ps} + h_1 + h_2 =$$

$$s_{res} + res_{ps} + h_1 + h_2$$

wobei die Hilfsvariablen wie im Algorithmus definiert sind.

Für Blöcke, die klein genug sind und nicht mehr zerlegt werden müssen, sind dagegen nur folgende Schritte durchzuführen:

- Alloziere einen adaptiven Baumknoten und kennzeichne ihn als Blatt
- Bestimme die Werte max und min für den betrachteten Block und trage sie in den Knoten ein
- Gib einen Zeiger auf den Knoten als Funktionswert zurück

Wie an diesem Algorithmus zu erkennen ist, werden aus Gründen der Zeitersparnis die Varianzen der einzelnen Scheiben berechnet und in ein Array eingetragen. Das ermöglicht es, sie schnell zu den Gesamtvarianzen der Teilblöcke zu vereinen und dadurch die Rechenzeit des Vorverarbeitungs-Schritts zu verkürzen.

4.2 Erwartungswert-optimierte Bäume

Wie bereits erwähnt wurde, werden die Nachfolger eines Knotens K bei der Suche nach einem Isowert γ genau dann untersucht, wenn $\gamma \in [\min(K), \max(K))$ gilt. Diese Tatsache führt zu folgender, im letzten Abschnitt bereits verbal angedeuteter Formel für die Berechnung der erwarteten Suchzeit im adaptiven Baum T. In dieser Formel werden keine rechnerabhängigen Zeitkonstanten berechnet; es wird dadurch vielmehr der Erwartungswert der Anzahl der besuchten Knoten und Zellen bestimmt und dabei angenommen, dass das Wahrscheinlichkeitsmaß P die Verteilung der angeforderten Isowerte angibt. Die zweite Gleichheit dieser Formel trifft zu, wenn die zufällige Wahl von γ einer Gleichverteilung im Daten-Intervall unterliegt.

$$
\begin{aligned}
E(t_T) = \mathrm{Zeit}(\mathrm{Knoten}) + \sum_{K \in \mathrm{Knotenliste}(T)} & [\mathrm{AnzahlNachfolger}(K) \\
& \mathrm{ZeitProNachfolger}(K) P([\min(K), \max(K)))] = \\
\mathrm{Zeit}(\mathrm{Wurzel}(T) + \sum_{K \in \mathrm{Knotenliste}(T)} & \Big[\mathrm{AnzahlNachfolger}(K) \\
& \mathrm{ZeitProNachfolger}(K) \frac{\max(K) - \min(K)}{\max(\mathrm{Wurzel}(T)) - \min(\mathrm{Wurzel}(T))} \Big]
\end{aligned}
$$

Dabei ist die Anzahl der Nachfolger eines inneren Knotens in dieser Formel bei binären Bäumen immer 2; die Anzahl der Nachfolger eines Blatts ist die Anzahl der Zellen, die der dazugehörige Datenblock enthält. Die Zeit pro Nachfolger ist für innere Knoten die Zeit, die für die Untersuchung eines Knotens gebraucht wird und für Blätter die Zeit für die Bearbeitung einer Zelle des Datensatzes. $\mathrm{Zeit}(\mathrm{Knoten})$ ist ein konstanter Wert, der für die in der Summe nicht berücksichtigte Bearbeitung der Wurzel angerechnet wird. Diese Formel gibt uns die Möglichkeit, die Qualität eines gegebenen adaptiven Baums zu bewerten, ohne für diesen Baum experimentelle Zeitmessungen durchführen zu müssen.

Theoretisch ist es auch möglich, ein Programm zu schreiben, das mit Hilfe dieser Kostenformel rekursiv die optimale Partition eines gegebenen Volumen-Datenblocks ausarbeitet. Praktisch ist dieses Programm aber nicht zu realisieren, weil seine Rechenzeit mit wachsender Blockgröße extrem stark ansteigt, denn ein $X \times Y \times Z$ großer Datenblock hat $O(X^2 Y^2 Z^2)$ mögliche Teilblöcke, die alle in mindestens einem Zerlegungsbaum vorkommen und deshalb ausgewertet werden müssen. Im später beschriebenen Conditioned-Tree-Verfahren ändert sich übrigens die Situation, weil dann nicht alle, sondern nur die zentral unterteilten Zerlegungsbäume zum Vergleich verwendet werden.

Eine besser realisierbare Version dieser Methode ist es, bei der Berechnung der Kostenformel in jedem untersuchten Knoten anzunehmen, dass die Unterblöcke des jeweils zu zerlegenden Blocks durch einen zentral zerlegten Partitionsbaum verwaltet werden. Der zentral zerlegte Baum zu einem Block ist eindeutig definiert, nämlich als der vollständige Baum, dessen Wurzel die größte Kantenlänge dieses Blocks genau in der (nach unten gerundeten) Mitte teilt und deren Nachfolger mit ihren zugeordneten Teilblöcken entsprechend verfahren. Vollständig heißt dabei, dass die Blätter des Baums dadurch charakterisiert sind, dass sie Teilblöcke mit genau einer Zelle verwalten. Das ergibt eine gute Kostenschätzung, die aber nicht zu zeitaufwändig ist, weil für jeden Baumknoten nur ein weiterführender Baum untersucht wird, nämlich der vom Algorithmus kanonisch festgelegte Baum. Daraus entsteht für die Konstruktion des Erwartungswert-optimalen Baums folgender, zum vorangehenden ähnlicher Algorithmus:

- Es sei $\delta_{\min} := \infty$
- Für jede mögliche Zerlegung des Blocks tue
  - Bilde den eindeutig festgelegten zentral-partitionierten Baum jeweils für den linken und den rechten Teil der Zerlegung
  - Bilde einen adaptiven Baumknoten, der auf die beiden Norm-Bäume verweist
  - Werte den so gebildeten Baum mit der Kostenformel für $E(t_T)$ aus, δ sei das Ergebnis der Berechnung
  - Lösche die Baumstruktur, um den Speicher wieder freizugeben
  - Wenn $\delta < \delta_{\min}$, dann setze $\delta_{\min} := \delta$ und betrachte die bearbeitete Zerlegung als die bisher beste
- Alloziere einen adaptiven Baumknoten
- Trage die Informationen der besten Zerlegung in den Knoten ein
- Trage $a_{\min}$ und $a_{\max}$, das Minimum und das Maximum der Daten des betrachteten Blocks in den Knoten ein
- Rufe diese Funktion rekursiv für die beiden Teilblöcke auf und trage die zurückgegebenen Zeiger in p_l und p_r des Knotens ein
- Gib den Zeiger auf den Knoten als Funktionswert zurück

Wenn ein Block klein genug ist und nicht mehr weiter unterteilt zu werden braucht, dann wird wie im vorangehenden Abschnitt ein Blattknoten für diesen Block gebildet. Die Entscheidung, wann ein Block klein genug ist, hängt von der verfügbaren Rechenzeit ab. Im günstigen Fall, wenn genügend Zeit für eine vollständige Optimierung vorhanden ist, ist es sinnvoll, den Baum bis zu einer Blockgröße von $2 \times 2 \times 2$ zu zerlegen.

4.3 Baumreduktion durch Tree growing und Tree pruning

Wir setzen in diesem Abschnitt voraus, dass bereits ein guter Partitionsbaum vorliegt, der aber für die Benutzung zu groß ist. Dann können wir zwischen zwei Methoden wählen, um den Baum auf einen kleineren zu reduzieren: Tree growing und Tree pruning (Abbildung 4.1). Beim Tree growing geht es - entgegen dem ersten Eindruck bei diesem Begriff - nicht darum, den ursprünglichen Baum noch größer zu machen; es wird vielmehr ausgehend von der Wurzel des Baums ein anderer aufgebaut, der aber trotz Vergrößerung immer ein Teilbaum des ursprünglichen bleibt.

Den Abschluss dieses Abschnitts bildet der BFOS-Algorithmus, der eine Verallgemeinerung des Tree-pruning-Algorithmus ist und mathematisch nachweisbar stets eine Liste von Lösungsbäumen erzeugt, die in den Sinn optimal sind, dass sie das Extraktionsproblem unter allen Bäumen der gleichen Größe am schnellsten lösen.

Zur Vereinfachung ist es auch möglich, anstatt den ganzen so verkleinerten Baum nur die Blockdaten von seinen Blättern zu speichern. Bei der Extraktion werden dann die Blöcke einer nach dem anderen durchgegangen und auf das Kriterium $\gamma \in [min, max)$ geprüft. In den Blöcken, bei denen

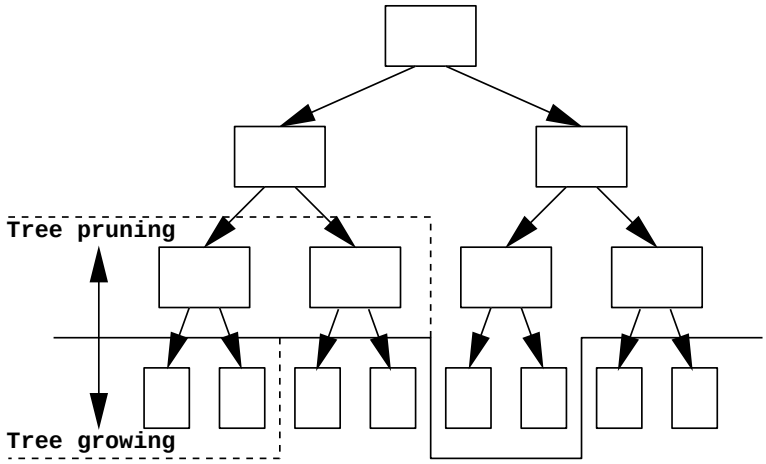


Abbildung 4.1: Ein adaptiver Baum wird durch Tree growing und -pruning reduziert

diese Beziehung zutrifft, kann dann eine nähere Untersuchung der einzelnen Zellen vorgenommen werden. Der Suchbaum, der zur Bestimmung des Zeitaufwands dieser Methode ausgewertet wird, besteht aus drei Ebenen: in der ersten steht der Knoten, der den ganzen Datensatz verkörpert, in der zweiten steht ein Knoten für jeden Teilblock und in der dritten steht ein Knoten für jede Zelle, als Nachfolger des Teilblock-Knotens, in dem die Zelle enthalten ist.

4.3.1 Tree growing

Das Prinzip der Tree-growing-Methode ist es, mit der Wurzel des ursprünglichen Baums zu starten, und diesen als im neuen Baum enthalten zu markieren. Dann werden wiederholt alle markierten Knoten miteinander verglichen, deren Nachfolger noch nicht markiert sind. Dieser Schritt kann mit Hilfe von Heaps, die die Liste der bearbeiteten Knoten verwalten, beschleunigt werden. Wir untersuchen diese Knoten darauf, wie viel Suchzeit dadurch gewonnen würde, dass ihre Nachfolger ebenfalls markiert und somit in den neuen Baum übernommen werden. Dieser Zeitgewinn ist zu berechnen durch:

$$\begin{aligned}\Delta t(p) = &[\max(p) - \min(p)]\mathrm{size}(p)\\ -&[\max(\mathrm{left}(p)) - \min(\mathrm{left}(p))]\mathrm{size}(\mathrm{left}(p))\\ -&[\max(\mathrm{right}(p)) - \min(\mathrm{right}(p))]\mathrm{size}(\mathrm{right}(p))\end{aligned}$$

$\mathrm{size}(p)$ ist dabei die Anzahl der Zellen, die im durch p dargestellten Teilblock enthalten sind. Es werden stets die Nachfolger des Knotens p markiert, für den der Zeitgewinn $\Delta t(p)$ den größten Wert erreicht. Dieser Vorgang wird so lange wiederholt, bis der Suchbaum die gewünschte Größe erreicht hat. Zu bemerken ist zu dieser Methode, dass sie zu keinem Zeitpunkt die modellierte Rechenzeit t kennt, sondern nur ihre Änderung Δt, wenn dem Baum Knoten angehängt werden.

Durch diese Methode kann der Speicherbedarf stets auf zwei Knotengrößen genau vorgegeben werden. Der Algorithmus kann durch regelmäßige Zwischenspeicherung eine ganze Folge von guten Teilbäumen erzeugen. Durch wiederholten Abzug des jeweils neuen Werts von $\Delta t(p)$ vom aktuell geschätzten Zeitbedarf können wir diesen aktualisieren und für weitere Auswertungen verwenden.

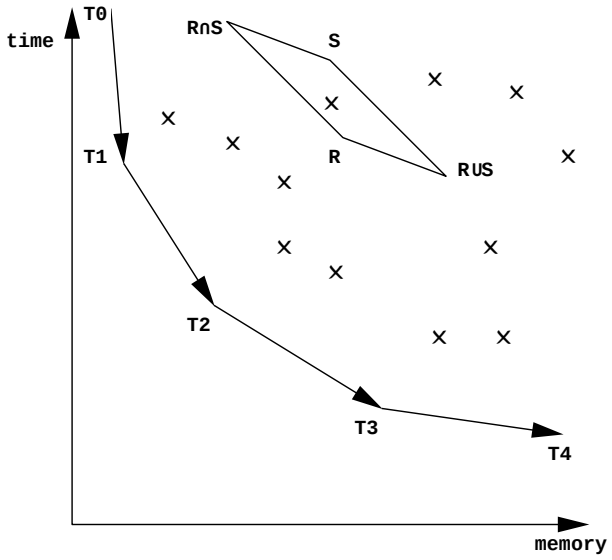


Abbildung 4.2: Speicher-Zeit-Diagramm mit der Klasse aller Unterbäume eines gegebenen adaptiven Zerlegungsbaums. Betrachte das Parallelogramm mit den Ecken R, $R \cup S$, S und $R \cap S$, um den Beweis von $T_0 \subset T_1 \subset T_2 \subset ...$ zu rekonstruieren (siehe Text)

4.3.2 Tree pruning

Der Tree-pruning-Algorithmus basiert auf der gleichen Formel wie der Tree-growing-Algorithmus, mit dem Unterschied, dass der Wert von $\Delta t(p)$ minimiert werden muss.

Dabei gehen wir wieder von einem großen Partitionsbaum aus und markieren alle darin enthaltenen Knoten. Dann werden wiederholt alle Knoten p miteinander verglichen, deren Nachfolger alle markiert sind, die aber keinen markierten Nach-Nachfolger haben. $\Delta t(p)$ mißt den Erwartungswert der Zeit, die durch das Streichen der Nachfolger von p verloren gehen würde. Es wird also die Markierung in den Nachfolgern des Knotens p gelöscht, für den $\Delta t(p)$ den kleinsten Wert hat.

Dieser Minimierungsschritt wird so lange wiederholt, bis der Baum aller noch markierten Knoten klein genug ist, um den Vorgaben zu entsprechen.

Eine Methode, die dieser Tree-pruning-Methode ähnelt, aber wesentlich besser ist, ist der BFOS-Algorithmus, der im nächsten Unterabschnitt beschrieben wird.

4.3.3 Der BFOS-Algorithmus

Der BFOS-Algorithmus ist eine verallgemeinerte Version des Tree-pruning-Algorithmus, die nicht nur zwei Nachfolger eines Knotens entfernen, sondern ganze Teilbäume abschneiden kann. Der so gewonnene Speicher ist größer, deshalb muss der daraus resultierende Zeitverlust durch den Speichergewinn dividiert werden, um den so erhaltenen Kosten-Erfolg-Quotienten zu minimieren. Das Grundprinzip des BFOS-Algorithmus wird in [12] von Philip A. Chou, Tom Lookabaugh und Robert M. Gray beschrieben; ich habe die Idee ausgearbeitet, diese Methode auf Partitionsbäume anzuwenden. Im Anhang des Artikels [12] wird mathematisch bewiesen, dass die aus dem BFOS-

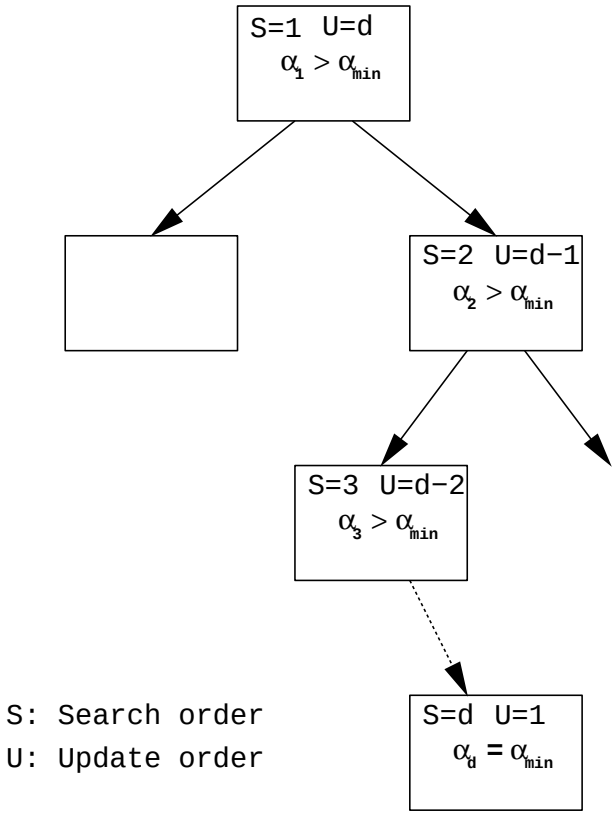


Abbildung 4.3: Das Prinzip des BFOS-Algorithmus: erst den abzuschneidenden Unterbaum suchen, dann den Suchpfad rückwärts durchlaufen, um die eingetragenen Zeitkosten zu aktualisieren

Algorithmus resultierenden Unterbäume jeweils die optimalen Bäume ihrer Größe sind. Die Idee des Beweises basiert auf folgenden Fakten:

- Jedes Element der Klasse aller Unterbäume des vorgegebenen Zerlegungsbaums lässt sich durch einen Punkt im Speicher-Zeit-Diagramm darstellen (siehe Abbildung 4.2)
- Unter diesen Punkten suchen wir diejenigen, die auf der unteren Grenze der konvexen Hülle der Punktmenge liegen.
- Die Kernaussage des Beweises ist, dass die auf den Ecken dieser Linie liegenden Bäume durch die Teilbaum-Relation verschachtelt sind. Diese lässt sich durch das Parallelogramm-Kriterium im Diagramm nachweisen (Abbildung 4.2): Wenn zwei Teilbäume R und S auf der optimalen Kurve liegen, von denen keiner ein Teilbaum des anderen ist, dann müssen $R \cap S$ (immer links von R und S) und $R \cup S$ (rechts von R und S) sowohl auf der optimalen Kurve als auch auf der Geraden RS liegen, weil sonst wegen Konvexität der optimalen Kurve einer der beiden Punkte unter der Kurve wäre. Also liegen R, S auf der Strecke von $R \cap S$ nach $R \cup S$ und sind in diesem Spezialfall für die Bildung der optimalen Kurve unnötig.

Zu einem gegebenen Zerlegungsbaum ist der Verbesserungs-Quotienten (VQ) eines Knotens definiert als der Quotient des zusätzlich erwarteten Suchzeit-Bedarfs, wenn alles unterhalb dieses

Knotens abgeschnitten würde und des dadurch wiedergewonnenen Speicherplatzes. Der BFOS-Algorithmus lässt sich beschleunigen, indem alle VQ-Werte ganz am Anfang des Programms berechnet und in den jeweiligen inneren Knoten gespeichert werden, in der Abbildung 4.3 sind diese als $\alpha_1, \alpha_2 ...$ dargestellt. Es werden in den inneren Knoten auch die minimalen VQ des ganzen darunter liegenden Teilbaums abgelegt (in der Abbildung als $\alpha_{\min}$), um schnell den jeweils kleinsten VQ des ganzen Baums finden zu können.

Im Hauptprogramm wird ausgehend von der Wurzel des Baums die Stelle des kleinsten VQs gesucht. Dafür braucht nur ein Pfad durchlaufen werden, da der kleinste VQ bereits als $\alpha_{\min}$ in der Wurzel und auf dem ganzen Pfad bis zur gesuchten Stelle eingetragen ist. Dann wird der Baum unterhalb dieser Stelle abgeschnitten und die Eintragungen der VQ und minimalen VQ werden an den neuen Zerlegungsbaum angepasst. Dieser Update ist sehr schnell möglich, weil die VQ sich nur in den Knoten des durchlaufenen Suchpfads ändern.

Diese Methode der Zeitersparnis kann auch für die einfache Tree-Pruning-Methode vom vorangehenden Abschnitt verwendet werden, indem die Menge der zugelassenen Tree-Pruning-Knoten auf diejenigen eingeschränkt wird, deren Nachfolger Blätter sind. Diese Variante ist jedoch im Vergleich zum BFOS-Algorithmus ineffizient und wird nur zum Zweck des schnelleren Kostenvergleichs verwendet.

4.4 Ergebnisse der adaptiven Methoden

Wie aus den Memory-Speedup-Graphen in den Abbildungen 4.4 und 4.5 zu sehen ist, kann die Extraktion mit einem adaptiven Baum beschleunigt werden; die gewonnene Rechenzeit wird durch die in diesem Kapitel beschriebenen Methoden noch weiter verbessert.

Trotz der Ungenauigkeiten der Zeitmessung im Bild erkennt man, dass die Tree-pruning-Methode und der BFOS-Algorithmus geringfügig besser als die Tree-growing-Methode sind. Ein Unterschied zwischen der Tree-pruning-Methode und dem BFOS-Algorithmus lässt sich mit diesen Graphen nicht nachweisen, es ist jedoch mathematisch gezeigt worden, dass der BFOS-Algorithmus stets das beste Ergebnis liefert.

Ferner sieht man, dass die Verwendung von Partitionsbäumen schon für den Einsatz von wenig zusätzlichem Speicher sichtbare Verbesserungen der Rechenzeit liefert, der Nutzen des Speichers (gewonnene Rechenzeit pro kB) nimmt aber bei zunehmender Größe des Baums immer weiter ab. Bei großen Partitionsbäumen wird die Zerlegung der Volumendaten so fein, dass eine weitere Zerlegung die Rechenzeit sogar erhöhen kann.

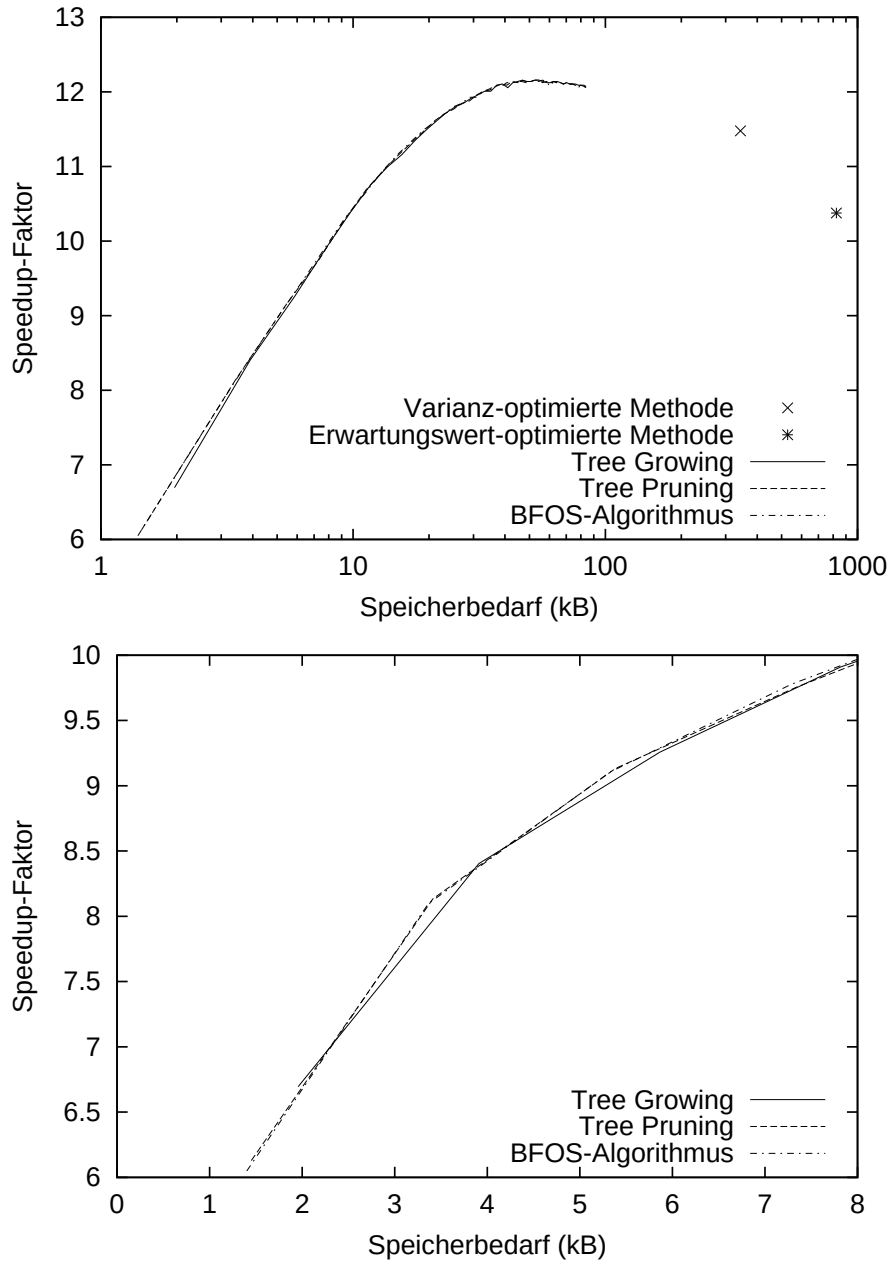


Abbildung 4.4: Das Memory-Speed-Diagramm der adaptiven Methoden, angewandt auf den Datensatz Kummer $100 \times 100 \times 100$, darunter ein Ausschnitt aus dem Diagramm, an dem der Vergleich der speicherabhängigen Methoden verdeutlicht wird.

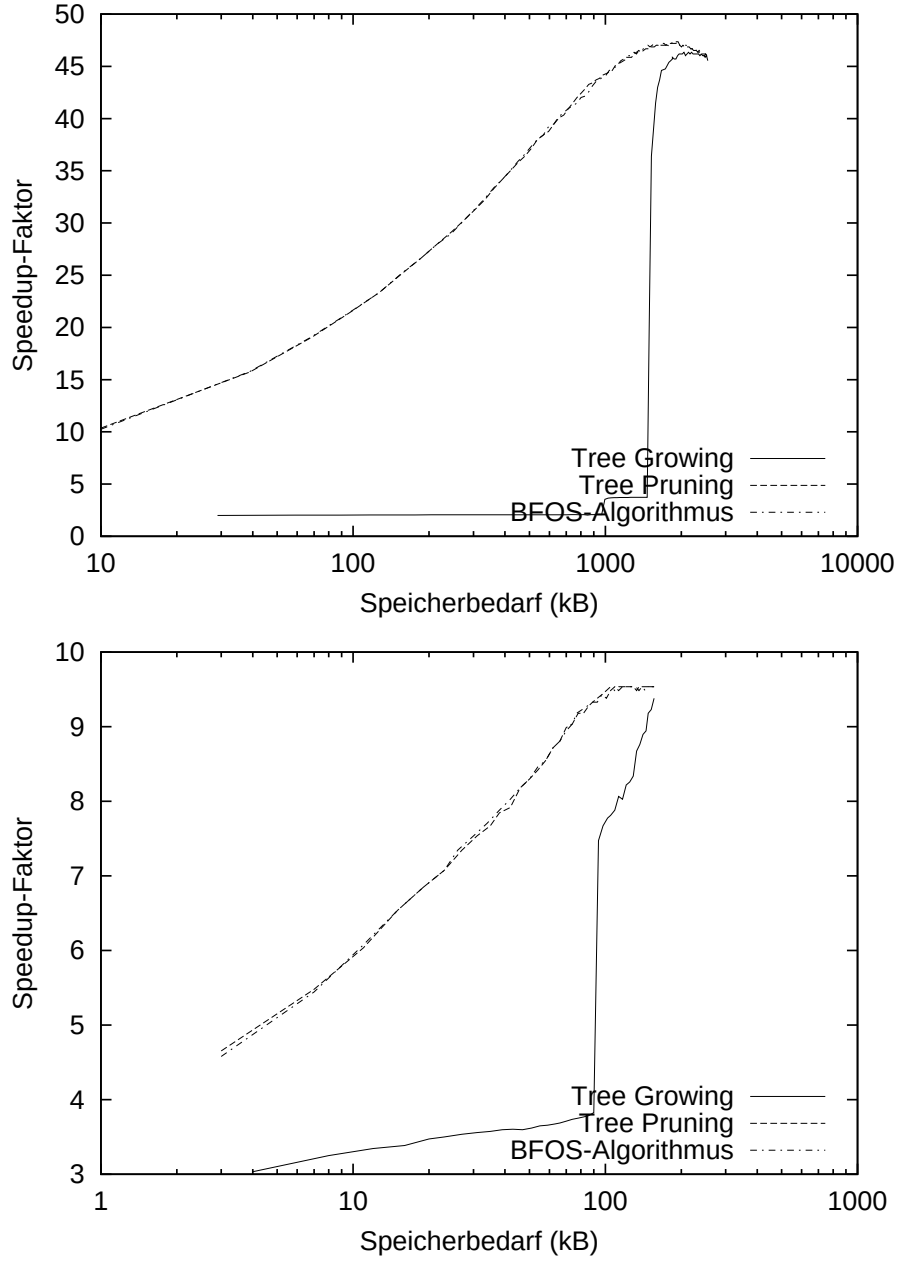


Abbildung 4.5: Weitere Memory-Speed-Diagramme für die Datensätze Kummer $250 \times 250 \times 250$ (oben) und Brain $128 \times 128 \times 128$ (unten).

Kapitel 5

Optimierung der Kosten von Intervallbäumen

In diesem Kapitel stelle ich ein Verfahren vor, mit dem sich für eine gegebene Liste von Intervallen ein optimaler Intervallbaum bestimmen lässt. Bei dieser Optimierung wird sowohl der Speicherbedarf des Intervallbaums als auch die erwartete Suchzeit eines vorgegebenen Werts berücksichtigt. Der Einfachheit halber bezeichne ich diesen Wert weiterhin als Isowert, obwohl sich die Überlegungen dieses Abschnitts auf beliebige Problemstellungen anwenden lassen, die die Verwendung eines Intervallbaums sinnvoll machen.

Das Problem in mathematischer Notation ist:

Gegeben:
Eine Liste von Intervallen $\{[\min_i, \max_i) | i = 1, ..., n\}$

Gesucht:
Ein Speicher-Suchzeit-optimaler Intervallbaum I, der die Intervalle dieser Liste enthält. Ich verzichte hier bewusst auf eine mathematische Formulierung der Optimalität wie z. B. die Forderung, die Kostenfunktion $C_\lambda(I) = M(I) + \lambda T(I)$ zu minimieren. Im Laufe dieses Abschnitts wird sich herausstellen, dass die genaue Optimalitäts-Bedingung keine Rolle spielt und dadurch hinfällig ist.

5.1 Optimierung des Speichers

Wir isolieren die beiden Teilprobleme und betrachten die Aufgabe, den Speicherbedarf des Intervallbaums zu minimieren. Der Speicher, den ein Intervallbaum belegt, umfasst zwei Typen von Datenstrukturen:

- **Intervalllisten**
  Die Min- und Max-Listen des Intervallbaums enthalten jeweils genau eine Eintragung gleicher Größe für jedes vorgegebene Intervall. Da die Größe des so reservierten Speicherbereichs nur von der Anzahl n der Intervalle abhängt, ist sie für das vorliegende Optimierungsproblem irrelevant.

$$M_{\text{Listen}}(I) = 2n \cdot M_{\text{Listeneintrag}}$$

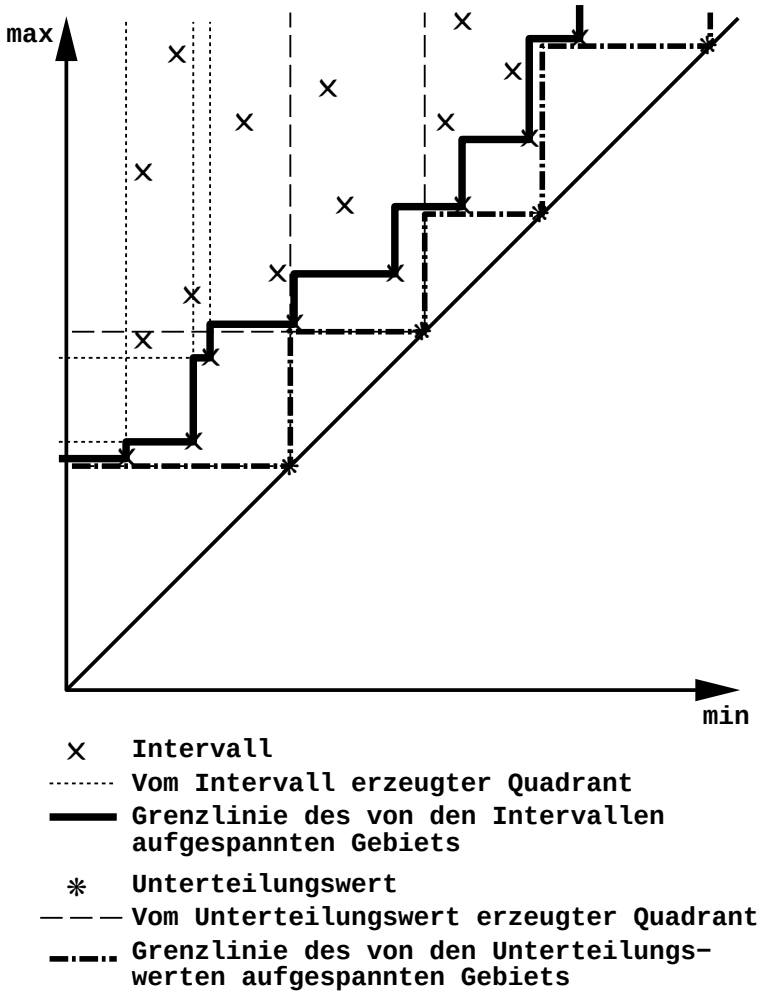


Abbildung 5.1: Die Optimierung der Liste der Unterteilungswerte $\gamma_1^*, ..., \gamma_m^*$, dargestellt im Span-Space

- **Knoten des Intervallbaums**
  Jeder Knoten des Intervallbaums enthält einen Unterteilungswert γ^*, durch den die zugeordneten Intervalle in die bekannten Teilgruppen für den linken und rechten Teilbaum des Knotens und die Min- und Max-Listen des Knotens selbst aufgeteilt werden. Der Intervallbaum lässt sich aus einem binären Baum als Grundgerüst konstruieren, der in jedem seiner Knoten dessen Unterteilungswert γ^*, aber ansonsten keine weiteren Informationen enthält. Dieser Binärbaum muss natürlich geordnet sein, d. h., für einen gegebenen inneren Knoten mit eingetragenem Wert γ^* sind alle Unterteilungswerte im linken Teilbaum kleiner als γ^*, und für seinen rechten Teilbaum entsprechend größer als γ^*. Es sei $\gamma_1^*, ..., \gamma_m^*$ die in aufsteigender Reihenfolge geordnete Liste aller Unterteilungswerte des Baums.
  Wie leicht einzusehen ist, ist eine Eintragung sämtlicher Intervalle in dieses Grundgerüst genau dann möglich, wenn zu jedem der Intervalle $[\min_i, \max_i)$ mindestens ein Wert γ_k^* aus der Liste existiert, der darin enthalten ist. Da der Speicherbedarf der Baumknoten proportional zu der Anzahl dieser Knoten bzw. ihrer Unterteilungswerte ist, wird also zur Optimierung des Speichers eine Liste $\gamma_1^*, ..., \gamma_m^*$ minimaler Länge gesucht, die diese Bedingung erfüllt. Die hier vorgeschlagene Lösung entspricht einer bekannten Methode zur Bestimmung einer minimalen Cliquenüberdeckung im Intervallgraphen; das heißt, wenn wir einen Graphen betrachten, in dem jedes Intervall einen Knoten repräsentiert und zwei Knoten genau dann miteinander verbunden sind, wenn sich die entsprechenden Intervalle überschneiden, dann entspricht die gesuchte Liste der Unterteilungswerte einer Liste von Untergraphen, in denen alle Knoten miteinander verbunden sind, so daß jeder Knoten in dieser Liste vertreten ist.
  Über den Aufbau des Intervallbaums aus der so gewonnenen Zahlenliste kann aus der reinen Forderung, den Speicherbedarf zu minimieren, noch keine Schlussfolgerung gezogen werden.
  $$M_{\text{Baumstruktur}}(I) = m \cdot M_{\text{Knoten}}$$

Um die kürzestmögliche Liste $\gamma_1^*, ..., \gamma_m^*$ zu erhalten, betrachten wir alle gegebenen Intervalle im Span Space. Diese liegen vollständig über der Hauptdiagonalen. Gesucht ist jetzt eine minimale Liste von Quadranten der Form $\{(x, y)|x \leq \gamma^* < y\}$, die diese Liste von Punkten vollständig überdeckt. Zur Vereinfachung kann jeder Punkt $(\min_i, \max_i)$ auch durch die Menge $\{(x, y)|x \leq \min_i \wedge y \geq \max_i\}$ ersetzt werden, was am gegebenen Problem nichts ändert. Die Vereinigung dieser Mengen, die durch die Unterteilungswert-Quadranten abzudecken ist, ist ein massives Gebiet, das von unten durch eine treppenstufenförmige Linie begrenzt ist. Auf den konvexen Ecken dieser Treppe liegen die Intervalle, die die abzudeckende Menge vollständig repräsentieren; alle anderen Intervalle liegen im Inneren des Gebiets und haben auf die Problemstellung keinen Einfluss.

Um die Liste der Abdeckungspunkte in aufsteigender Reihenfolge zu konstruieren, gehen wir nach der Stufenmethode wie in Abbildung 5.1 vor, für eine Liste von nicht leeren Intervallen $[\min_i, \max_i) \quad (i = 1...n)$ mit ganzzahligen Grenzen sei also:

$$\gamma_1^* := \min\{\max_i | i = 1...n\} - 1$$

$$\gamma_{k+1}^* := \min\{\max_i | i = 1...n, \min_i > \gamma_k^*\} - 1$$

Die rekursive Berechnung wird so lange wie möglich fortgesetzt, also bis die darin verwendete Menge $\{\max_i \,|\, \min_i > \gamma_k^*\}$ leer ist. Sobald das der Fall ist, wird $m := k$ gesetzt und wir haben eine Liste $\gamma_1^*, ..., \gamma_m^*$. Wir weisen nun nach, dass sich diese Definition für die Konstruktion einer optimalen Liste eignet:

Satz:

1. Die mit vorstehender Definition konstruierte Liste $\gamma_1^*, \gamma_2^* ...$ ist streng monoton steigend.
2. Die Liste ist endlich lang. (das bedeutet, es existiert ein m, für das das Abbruch-Kriterium, dass $\{\max_i \,|\, \min_i > \gamma_m^*\}$ leer ist, erreicht wird)
3. Für jedes $i \in \{1, ..., n\}$ existiert ein $k \in \{1, ..., m\}$, so dass $\gamma_k^* \in [\min_i, \max_i)$ gilt.
4. $\gamma_1^*, ..., \gamma_m^*$ ist eine kürzestmögliche Liste ganzzahliger Werte, für die Bedingung **3.** erfüllt ist.

Beweis:

1. Es seien $M_1 := \{\max_i \,|i = 1...n\}$ und $M_{k+1} := \{\max_i \,|i = 1...n, \min_i > \gamma_k^*\}$. Dann ergibt sich durch Induktion, dass für alle k $M_{k+1} \subseteq M_k$ und $\gamma_{k+1}^* \geq \gamma_k^*$ gilt:

Für $k = 1$ gilt $M_2 \subseteq M_1$, weil M_2 durch eine zusätzliche Bedingung aus M_1 hervorgeht. Daraus folgt auch $\gamma_2^* \geq \gamma_1^*$, weil in der Definition von γ_2^* über eine kleinere Menge minimiert wird.

Für den Induktionsschritt $k \to k + 1$ setzen wir voraus, dass $\gamma_{k+1}^* \geq \gamma_k^*$ bereits nachgewiesen ist. Daraus folgt nach Definition der Mengen $M_{k+2} \subseteq M_{k+1}$, weil die Bedingung verstärkt wurde, und somit $\gamma_{k+2}^* \geq \gamma_{k+1}^*$, weil in der Definition von γ_{k+2}^* über eine kleinere Menge minimiert wird.

Die Verstärkung dieser Aussage zur echten Monotonie ergibt sich aus der Überlegung, dass sich in der Menge M_{k+1} nur Maxima von Intervallen befinden, deren Minima größer als γ_k^* sind. Folglich sind alle Elemente der Menge größer als $\gamma_k^* + 1$, also auch ihr Minimum. Wenn schließlich laut Definition 1 abgezogen wird, kommen wir auf den Wert von $\gamma_{k+1}^* > \gamma_k^*$.

2. Aus der strengen Monotonie der Folge laut **1.** und ihrer Ganzzahligkeit folgt, dass die Menge M_{k+1} leer wird, sobald γ_k^* einen hinreichend großen Wert erreicht.

3. Es sei $i \in \{1, ..., n\}$ gegeben. Gesucht ist nun ein Index k, so dass $\min_i \leq \gamma_k^* < \max_i$ erfüllt ist. Wir untersuchen also das kleinste k, für das $\gamma_k^* \geq \min_i$ gilt, was bereits eine der geforderten Ungleichungen ist. Dieses existiert auch in jedem Fall, da $\gamma_m^* \geq \min_i$ als Abbruchbedingung für alle i gelten muss. Da wir das kleinste k haben, das die Bedingung erfüllt, gilt sie für $k - 1$ nicht, es ist also $\gamma_{k-1}^* < \min_i$ oder $k = 1$; in beiden Fällen ist $\max_i \in M_k$. Daraus folgt die zweite Ungleichung $\gamma_k^* < \max_i$.

4. Nachdem mit **1.** bis **3.** gezeigt wurde, dass die Definition eine geeignete Liste von Unterteilungswerten liefert. muss noch bewiesen werden, dass es keine kürzere Liste gibt.

Es sei also $\beta_1, ..., \beta_{m'}$ eine Liste von Werten, von denen jedes vorgegebene Intervall $[\min_i, \max_i)$ mindestens einen enthält. O.b.d.A. kann angenommen werden, dass die Liste streng monoton steigend ist, weil andere Listen durch Sortierung und Entfernung von doppelten Werten geeignet umgeformt werden können. Durch Induktion wird gezeigt, dass für alle $k \in \{1, ..., m\}$ gilt: β_k existiert (es gilt also $k \leq m'$) und erfüllt die Ungleichung $\beta_k \leq \gamma_k$. Wenn die Induktion bis m fortgesetzt wird, erhalten wir die gewünschte Ungleichung $m \leq m'$.

Für $k = 1$ betrachten wir den kleinsten Wert der Folge β_1, der kleiner als das kleinstmögliche $\max_i$ sein muss, damit β_1 oder einer der darauf folgenden größeren Werte im entsprechenden Intervall $[\min_i, \max_i)$ enthalten sein kann. Nach der Definition von γ_1 folgt daraus $\beta_1 \leq \gamma_1$.

Es sei nun für ein $k < m$ die Ungleichung $\beta_k \leq \gamma_k$ erfüllt. Dann gilt für alle Intervall-Minima, die $\min_i > \gamma_k$ erfüllen, auch $\min_i > \beta_k$. Es sei $\max_j$ das kleinste Maximum unter diesen Intervallen. Dann gilt laut Definition $\gamma_{k+1} = \max_j - 1$. Ferner muss β_{k+1} existieren und $\beta_{k+1} \leq \gamma_{k+1}$ gelten, damit das Intervall $[\min_j, \max_j)$ einen Wert der β-Liste enthält: Für $\beta_1, ..., \beta_k$ ist das wegen $\min_j > \beta_k$ nicht möglich, und damit β_{k+1} oder einer der größeren Werte im Intervall enthalten sein kann, muss $\beta_{k+1} < \max_j$ gelten, und γ_{k+1} ist der größtmögliche ganzzahlige Wert, der diese Ungleichung erfüllt.

5.2 Optimierung der Extraktionszeit

Der Erwartungswert des Zeitbedarfs für eine Isowert-Suche im Intervallbaum lässt sich wieder in zwei Typen von Zeitkosten zerlegen:

- **Ausgabezeit**
  Diese Zeit umfasst für jedes Intervall, das während der Extraktion ausgegeben wird, die Zeit, die benötigt wird, um dieses Intervall in der Min- oder Max-Liste zu untersuchen und auszugeben. Die jeweils letzte Untersuchung eines Listeneintrags, der nicht mehr ausgegeben wird, fällt nicht in diese Kategorie. Der erwartete Zeitbedarf dieses Typs ist für jedes in den Baum eingetragene Intervall proportional zur Wahrscheinlichkeit, dass der Isowert γ in diesem Intervall enthalten ist. Die Formel lautet also:

  $$T_{\text{Output}}(I) = \sum_{i=1}^{n} p(\gamma \in [\min_i, \max_i)) \cdot t_{\text{Output}}$$

  Dieser Teil der Rechenzeit hängt nicht vom Aufbau des Intervallbaums, sondern nur von der vorgegebenen Liste von Intervallen ab. Er braucht also bei der Minimierung der erwarteten Rechenzeit nicht berücksichtigt zu werden.

- **Knotengebundene Rechenzeit**
  Der zweite Zeitkostentyp umfasst die Operationen, die für jeden untersuchten Knoten des Intervallbaums nur einmal ausgeführt werden. Diese umfassen im Wesentlichen:

  – den Vergleich des Isowerts γ mit dem Eintrag γ^*
  – das Starten der Suche in der Min- oder Max-Liste
  – die Untersuchung des jeweils letzten, nicht ausgegebenen Intervalls der Min- oder Max-Liste
  – die fallabhängige Verzweigung zum linken oder rechten Nachfolger des Knotens (Also **nicht** dessen gesamte Bearbeitung, sondern nur der Aufruf der Bearbeitungsroutine)

  Die Bearbeitungszeit dieser Vorgänge lässt sich durch eine Konstante t_{Knoten} abschätzen, die Formel für die Berechnung des Zeitbedarfs ist also

  $$T_{\text{Baum}}(I) = E(\text{Anzahl der untersuchten Knoten}) \cdot t_{\text{Knoten}}$$

  Sie ist also nun zu optimieren, indem der Erwartungswert der Anzahl der untersuchten Knoten minimiert wird.

Für einen ausgewogenen Intervallbaum verhält sich unter der Bedingung, dass der Baum untersucht wird, die Anzahl der untersuchten Knoten wie $\log(m)$, wobei m die bereits für den Speicherbedarf minimierte Anzahl der Knoten des Intervallbaums ist. Die Methode zur Minimierung der Anzahl der Intervallbaum-Knoten bietet also auch eine günstige Voraussetzung für die Reduktion des durchschnittlichen Zeitbedarfs.

Um aus den so erhaltenen Unterteilungswerten $\gamma_1^*, ..., \gamma_m^*$ einen optimalen Intervallbaum zu bilden, lautet die Aufgabe nun: Konstruiere das Grundgerüst für den Intervallbaum, das für einen

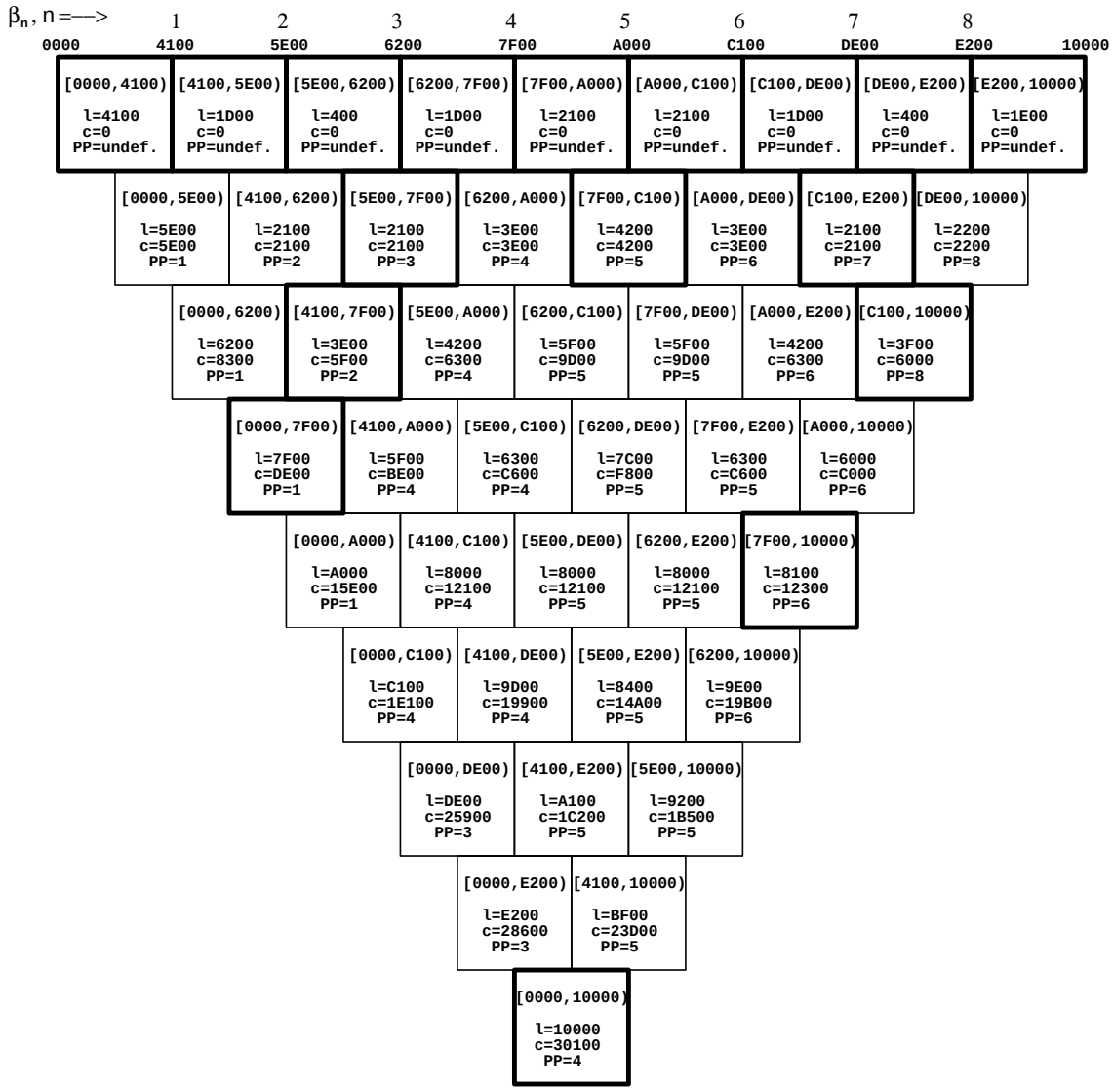


Abbildung 5.2: Das Dreiecksschema, mit dem der optimale Intervallbaum zu den γ^*-Werten $\beta_1, ..., \beta_8$ gefunden wird.

zufällig vorgegebenen Isowert γ den kürzestmöglichen Suchpfad liefert. Diese Optimierungsaufgabe lässt sich nach dem Bottom-Up-Schema zunächst für kleine Teillisten von Unterteilungswerten lösen, die dann zu größeren zusammengesetzt werden. In den Abbildungen 5.2 und 5.3 wird die Optimierung an einem Beispiel dargestellt. Jedes Rechteck entspricht dabei einem möglichen Intervallbaum-Knoten, mit Ausnahme der Rechtecke der obersten Reihe, die einem nicht mehr zerteilbaren Intervall entsprechen. Die Einträge sind dabei von oben nach unten folgendermaßen bestimmt:

- Die oberste Zeile gibt das betrachtete Intervall von Isowerten in Hexadezimal-Form an.
- l ist die Länge dieses Intervalls.
- c ist der rekursiv bestimmte minimale Kostenwert und ein Maß für die mittlere Anzahl der besuchten Knoten innerhalb dieses Bereichs. Ein Wert von $10000(hex)$ entspricht dabei einem besuchten Knoten.
- PP gibt die optimale Unterteilungsstelle im betrachteten Knoten an und wird zur Rekonstruktion des optimalen Intervallbaums in Abbildung 5.3 verwendet.

Die vorliegende Aufgabenstellung ähnelt der Aufgabenstellung der Huffman-Codierung, da jeder Pfad zu einem Teilintervall durch eine Bitfolge (links:0, rechts:1) codiert werden kann und der Erwartungswert der Länge dieser Bitfolge zu minimieren ist. Die dafür angebotene Lösung, jeweils die beiden kleinsten Bereiche zu vereinen, ist hier jedoch nicht anwendbar, da diese Bereiche nicht notwendigerweise benachbart sind.

Fazit

Die Wahl des optimalen Intervallbaums zu einer gegebenen Liste von Intervallen hängt nicht von der Wahl des Parameters λ ab, auch nicht von der Wahl der genauen Optimierungsvorschrift. Das folgt daraus, dass sich der Zeit- und Speicherbedarf mit geeigneten Methoden gemeinsam verbessern lassen.

Um die Kostenfunktion $C_\lambda(I) = M(I) + \lambda T(I)$ des optimalen Intervallbaums I zu bestimmen, braucht dieser nicht aufgebaut zu werden. Es ist dafür - abgesehen von trivialen Zähl- und Summierungsoperationen - nur notwendig, durch die beschriebene Stufenmethode die Grenz-Isowerte $\gamma_1^*, ..., \gamma_m^*$ zu bestimmen und für diese die mittlere Suchpfadlänge der optimalen Zerlegung zu berechnen. Diese Zerlegung braucht dafür auch nicht aufgebaut zu werden, da die Berechnung rekursiv über Teilstücke des Arrays $[\gamma_1^*, ..., \gamma_m^*]$ möglich ist.

Das hier beschriebene Optimierungsverfahren ist eine echte Verbesserung gegenüber den bisher verwendeten Verfahren, den Mittelwert oder den Durchschnitt der gegebenen Intervallgrenzen als Unterteilungswert zu verwenden. Ein Beleg dafür ist das in Abbildung 5.4 aufgeführte Beispiel, das die beiden Methoden an einer Liste aus drei Intervallen vergleicht. Die Intervalle in dem Beispiel überschneiden sich, um dem Umstand gerecht zu werden, dass es sich um eine Anwendung aus der Isoflächen-Extraktion handelt. Die zugehörige Konfiguration aus drei Zellen lässt sich leicht konstruieren.

Das Ergebnis zeigt, dass sich durch das Optimierungsverfahren ein Intervallbaum aus zwei Knoten ergibt, von denen bei einer Suchanfrage durchschnittlich 1,7 Knoten besucht werden. Gegenüber den drei Knoten bei den bisherigen Verfahren, von denen durchschnittlich 2 Knoten besucht werden, ist das sowohl bezüglich Speicherbedarf als auch bezüglich mittlerer Suchzeit eine Verbesserung.

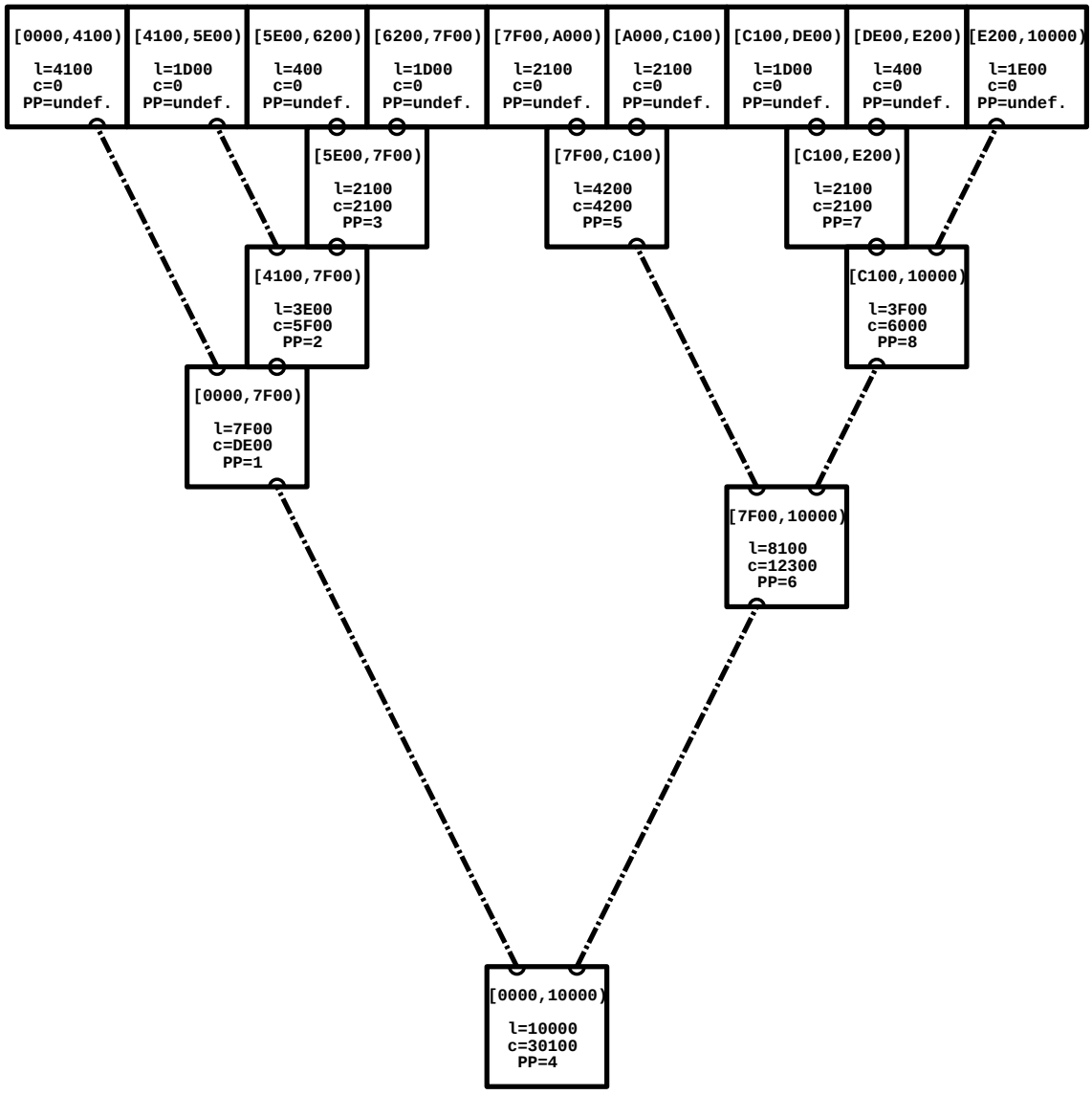
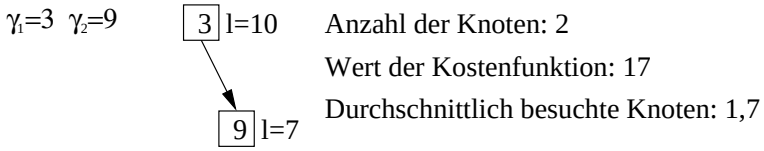


Abbildung 5.3: Hier sind noch einmal die (im vorherigen Bild fett umrahmten) Intervalle dargestellt, aus denen der optimale Intervallbaum gebildet wird.

Gegeben:

Definitionsbereich: [0,10)
Intervalle: [0,4) [3,7) [6,10)

Optimierung nach dem hier beschriebenen Verfahren:



Altes Verfahren mit Mittelwert− oder Durchschnittsbildung:
(aus Symmetriegründen gleiche Ergebnisse)

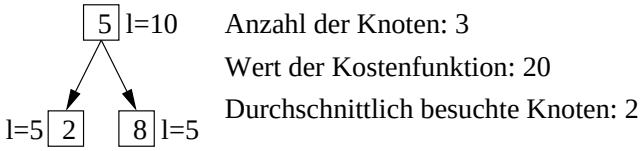


Abbildung 5.4: Ein Beispiel dafür, dass das beschriebene Optimierungsverfahren eine Verbesserung gegenüber bisherigen Verfahren ist.

5.3 Genauere Analyse der Intervallbaum-Optimierung

Wie in den beiden vorangehenden Abschnitten beschrieben ist, ist es möglich, einen Intervallbaum mit minimalem Speicherbedarf zu bestimmen, der unter allen Intervallbäumen mit den gleichen Unterteilungswerten auch die kürzeste mittlere Suchzeit aufweist. Dieser muss allerdings nicht notwendigerweise die kürzeste Suchzeit unter allen möglichen Intervallbäumen haben. Es existiert auch nicht in jedem Fall überhaupt ein Intervallbaum, der gleichzeitig den Speicherbedarf und die Suchzeit minimiert.

Ich stelle hier einen künstlich entwickelten Fall vor, in dem die bezüglich der erwarteten Suchpfadlänge optimale Folge $\lambda_1^*, ..., \lambda_m^*$ nicht gleichzeitig den Wert m minimiert. Hierfür machen wir folgende Zuweisungen von hexadezimal notierten Werten:

$$\gamma_1 := 4000 \qquad \beta_1 := 4100$$

$$\gamma_2 := 5000 \qquad \beta_2 := 5E00$$

$$\gamma_3 := 6000 \qquad \beta_3 := 6200$$

$$\gamma_4 := 7000 \qquad \beta_4 := 7F00$$

$$\gamma_5 := 8000 \qquad \beta_5 := A000$$

$$\gamma_6 := C000 \qquad \beta_6 := C100$$

$$\gamma_7 := D000 \qquad \beta_7 := DE00$$

$$\gamma_8 := E000 \qquad \beta_8 := E200$$

$$\gamma_9 := F000$$

Die vorgegebenen Intervalle seien $[\gamma_i, \beta_i + 1)$ und $[\beta_i, \gamma_{i+1} + 1)$ mit $i = 1, ..., 8$. Wegen $\gamma_1 < \beta_1 < \gamma_2 < \beta_2 < ..... < \beta_8 < \gamma_9$ besteht diese Liste bereits nur aus optimalen Intervallen.

Für die Liste der Unterteilungswerte betrachten wir zwei Alternativen, die nun genauer analysiert und miteinander verglichen werden: Alt. 1: $(\gamma_i)_{i=1,...,9}$ und Alt. 2: $(\beta_i)_{i=1,...,8}$. Wie leicht einzusehen ist, gilt:

- Beide Alternativen liefern eine Liste von γ^*-Werten, von denen in jedem der vorgegebenen Intervalle mindestens einer enthalten ist. Sie sind also für die Konstruktion eines Intervallbaums zulässig.
- Alt. 2 ist die nach dem bereits beschriebenen Optimierungsschema gebildete Liste minimaler Länge. Eine rückwärtige Anwendung des Optimierungsschemas liefert die selbe Lösung und zeigt dadurch, dass die Liste minimaler Länge eindeutig bestimmt ist.
- Für Alternative 1 lässt sich ein zentral unterteilter Intervallbaum konstruieren, der den beiden $4000(\mathrm{hex})$ langen Intervallen die Suchpfadlänge 2 und alles anderen, $1000(\mathrm{hex})$ langen Intervallen die Suchpfadlänge 4 gibt. Beide Intervalltypen belegen jeweils die Hälfte des Gesamtbereichs, was zu einer erwarteten Suchpfadlänge von 3 führt.
- Dass Alternative 2 diesen Wert nicht erreichen oder verbessern kann, wird im folgenden gezeigt, indem die durch die Unterteilung entstehenden Intervalle durch ein rekursives Optimierungsschema wieder vereint werden. Dieses ist in den Abbildungen 5.2 und 5.3 dargestellt und führt zu einer optimalen Kostenfunktion von $30100(\mathrm{hex})$, was einer mittleren Suchpfadlänge von $3\frac{1}{256}$ entspricht.

Anmerkung:
Im Dreiecksschema von Abbildung 5.2 steht in jedem Feld das davon repräsentierte Teilintervall, gefolgt von seiner Länge l, seinen minimalen Kosten c und dem Index seines optimalen Zerlegungspunkts PP. Die Zahlenwerte sind im Hexadezimal-System angegeben. Die Formeln für die Berechnung sind für alle Indexpaare (m, n) mit $m < n$: $l_{mn} := \beta_n - \beta_m$, $c_{m\ m+1} := 0$, sowie für alle Paare mit $m + 1 < n$: $c_{mn} := l_{mn} + \min\limits_{m<k<n}(c_{mk} + c_{kn})$ und $PP_{mn} := \arg\min\limits_{m<k<n}(c_{mk} + c_{kn})$

An diesem Gegenbeispiel sehen wir, dass es schwer ist, eine geeignete Intervall-Zerlegung zu finden, weil es nicht notwendigerweise eine Zerlegung geben muss, die gleichzeitig speicher- und zeitoptimal ist. Ein paar vorteilhafte Voraussetzungen sind bei der Optimierung von Intervallbäumen aber doch gegeben:

- Wie in den vorhergehenden Abschnitten zu sehen war, hängen Speicher- und Zeitbedarf stark miteinander zusammen. Die dort konstruierte Lösung unterscheidet sich also nur geringfügig von der/einer optimalen Lösung zu einem gegebenen Parameter γ.
- Eine geringfügige Uneindeutigkeit besteht noch, weil das Treppenstufenschema von unten nach oben oder von oben nach unten angewendet werden kann. Dies liefert zwei γ^*-Listen gleicher Länge (weil sie ja längenoptimal sind). Die eine enthält die maximal, die andere die minimal möglichen γ^*-Werte. Die tatsächlich optimalen Werte müssen also irgendwo dazwischen liegen. Eine kurze Überlegung liefert das Ergebnis, dass es sehr günstig ist, einem Anfangsstück der ersten Liste ein dazu passendes Endstück der zweiten Liste folgen zu lassen, so dass die Länge des dazwischen liegenden Lückenintervalls minimal ist.

- Wenn die Liste der γ^*-Werte und ihre Intervalle einmal festliegen, dann ist der dazu gehörende optimale Intervallbaum bzw. die optimale Zerlegung leicht zu finden. Man betrachte einfach nacheinander alle Gruppierungen von 2,3,...,k benachbarten Intervallen und konstruiere die dazu gehörende optimale Zerlegung mit ihrem Kostenwert.

  Für zwei Intervalle ist diese trivial, da es nur eine sinnvolle Zerlegung gibt. Für m Intervalle ($m > 2$) betrachte man für jede mögliche Zerlegung in m' und $m - m'$ Intervalle deren Kostenwerte und kombiniere diese zum Kostenwert für die Zerlegung aller m Intervalle. Der kleinste dieser Werte wird dann mit seiner Zerlegungsstelle eingetragen.

  Die rekursive Formel dafür lautet:

$$C(I_j, I_{j+1}) = l_j + l_{j+1}$$

$$C(I_j, ..., I_{j+m-1}) = l_j + ... + l_{j+m-1} + \min_{p=1}^{m-1}[C(I_j, ..., I_{j+p-1}) + C(I_{j+p}, ..., I_{j+m-1})]$$

  Die so erhaltenen Werte und die zugehörige beste Unterteilungsstelle werden in ein Dreiecksarray eingetragen, aus dem sich danach der optimale Zerlegungsbaum leicht von oben nach unten ablesen lässt.

Anmerkung: Dies ist eine formelle Darstellung des in den Abbildungen 5.2 und 5.3 gezeigten Optimierungsschemas. Dabei sind:

$$I_j = [\beta_j, \beta_{j+1})$$

$$l_j = |I_j| = \beta_{j+1} - \beta_j = l_{j\ j+1}$$

$$C(I_j, ..., I_{j+m-1}) = c_{j\ j+m}$$

was zu einer Übereinstimmung der rekursiven Kostenformeln führt.

5.4 Beschleunigung der Optimierung im Conditioned Tree

Wie in Abbildung 5.1 zu sehen ist, werden für die Untersuchung der Speicher- und Zeitkosten eines Intervallbaums nicht alle Intervalle benötigt, sondern nur eine aufsteigend geordnete Liste, die einen repräsentativen Teil dieser Intervalle enthält. In der Abbildung sind das alle Intervalle, die auf den nach unten rechts gerichteten Ecken der durchgezogenen Grenzlinie liegen.

Diese haben die Eigenschaft, dass jedes andere Intervall mindestens eins der Intervalle dieser Liste vollständig umfasst. Dadurch kann die Suche nach einer geeigneten Liste von Unterteilungswerten, von denen jedes Intervall mindestens einen enthalten muss, auf die vergleichsweise kleine Teilliste von Intervallen eingeschränkt werden.

Ein zusätzlicher Vorteil dieses Modells ist es, dass die Analyse der Intervallbäume beim Vorliegen einer hierarchischen Zerlegung beschleunigt werden kann. Wir werden im Kapitel 7 sehen, dass beim Conditioned Tree eine hierarchische Zerlegung gegeben ist. Wenn wir zwei Listen von Intervallen betrachten, zu denen die Grenzlinien bereits bekannt sind, können wir diese beiden Linien einfach und zeitsparend miteinander verknüpfen, um die Grenzlinie zur Vereinigung der beiden Listen zu erhalten (Abbildung 5.5). In einer hierarchischen Zerlegung einer Liste von Intervallen lässt sich dieses Prinzip rekursiv für jede Teilliste anwenden.

Die Vereinigung zweier Grenzlinien ist die Grenzlinie der Vereinigungsmenge der darüber liegenden Bereiche, die in Abbildung 5.5 dargestellt ist. Die dazugehörige Liste von Intervallen kann aus den beiden Listen der Nachfolger mit folgender Methode vereinigt werden:

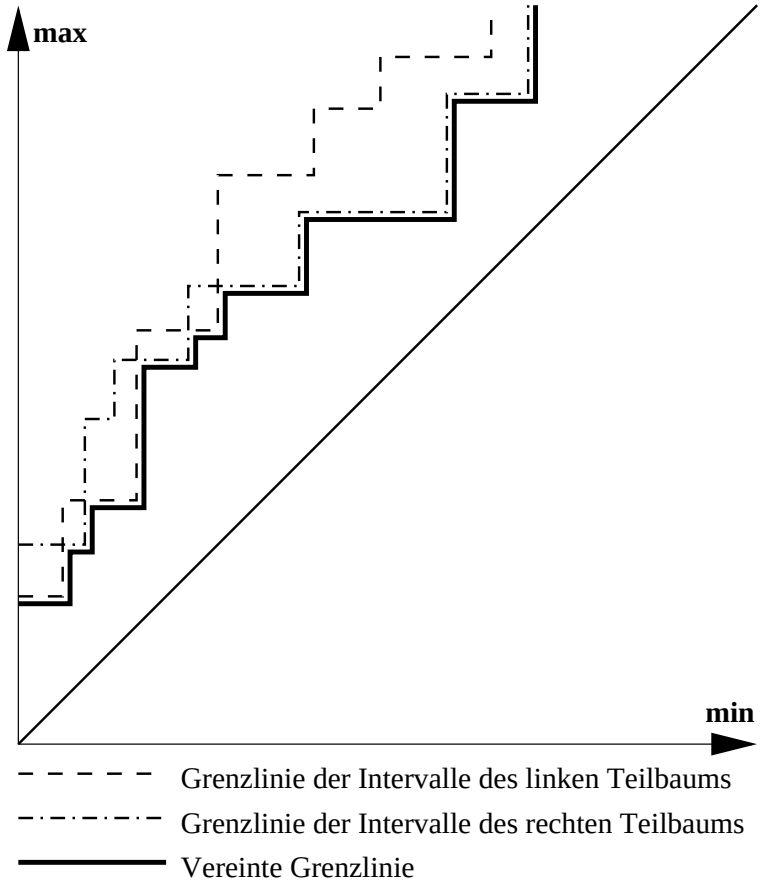


Abbildung 5.5: Die Grenzlinien zweier Intervallbäume werden zur Grenzlinie des vereinten Intervallbaums zusammengefügt. Der Algorithmus dazu ist im Text beschrieben.

- **Input:** Zwei jeweils einen Intervallbaum repräsentierende, aufsteigend geordnete Intervalllisten
  $[\min_{11}, \max_{11}), [\min_{12}, \max_{12})...[\min_{1m}, \max_{1m})$ und
  $[\min_{21}, \max_{21}), [\min_{22}, \max_{22})...[\min_{2n}, \max_{2n})$
- Es sei $i := 0$, $j := 1$ und $k := 1$.
- Solange $j \leq m \vee k \leq n$ wiederhole
  - $i := i + 1$
  - Wenn $k > n \vee \{j \leq m \wedge [\max_{1j} < \max_{2k} \vee (\max_{1j} = \max_{2k} \wedge \min_{1j} > \min_{2k})]\}$ dann
    - $\min_i := \min_{1j}$, $\max_i := \max_{1j}$
    - $j := j + 1$
    - Solange $k \leq n \wedge \min_{2k} \leq \min_i$ wiederhole $k := k + 1$
  - ansonsten
    - $\min_i := \min_{2k}$, $\max_i := \max_{2k}$
    - $k := k + 1$
    - Solange $j \leq n \wedge \min_{2j} \leq \min_i$ wiederhole $j := j + 1$
- **Output:** Die vereinte Liste $[\min_1, \max_1), [\min_2, \max_2)...[\min_i, \max_i)$

Erklärung des Programms: j ist ein Pointer auf die erste, k auf die zweite Intervallliste. Entsprechend ist i ein Pointer auf die Ergebnisliste, die noch zu bilden ist.

Abhängig vom Ergebnis eines längeren boolschen Terms erhält die Ergebnisliste das nächste Intervall aus der ersten oder zweiten Liste. Der boolsche Term macht folgende Vergleiche:

- $k > n$ prüft, ob der Pointer k die zweite Liste schon durchlaufen hat. In diesem Fall ist das nächste Intervall in der ersten Liste.
- $j \leq m$ testet, ob der Pointer j die erste Liste durchlaufen hat. In diesem Fall ist das nächste Intervall in der zweiten Liste.
- $\max_{1j} < \max_{2k}$ vergleicht die betrachteten Intervalle der beiden Listen. Das Intervall mit dem kleineren Maximum wird in die Ergebnisliste genommen.
- Im Fall von Gleichheit, geprüft durch $\max_{1j} = \max_{2k}$, wird das Intervall mit dem kleineren Minimum durch den Vergleich $\min_{1j} > \min_{2k}$ ermittelt.

Nach der Ergänzung der Ergebnisliste werden ein paar Einträge der nicht benutzten Inputliste übersprungen, so dass das Minimum des nächsten betrachteten Intervalls größer als das Minimum des letzten eingetragenen Intervalls ist.

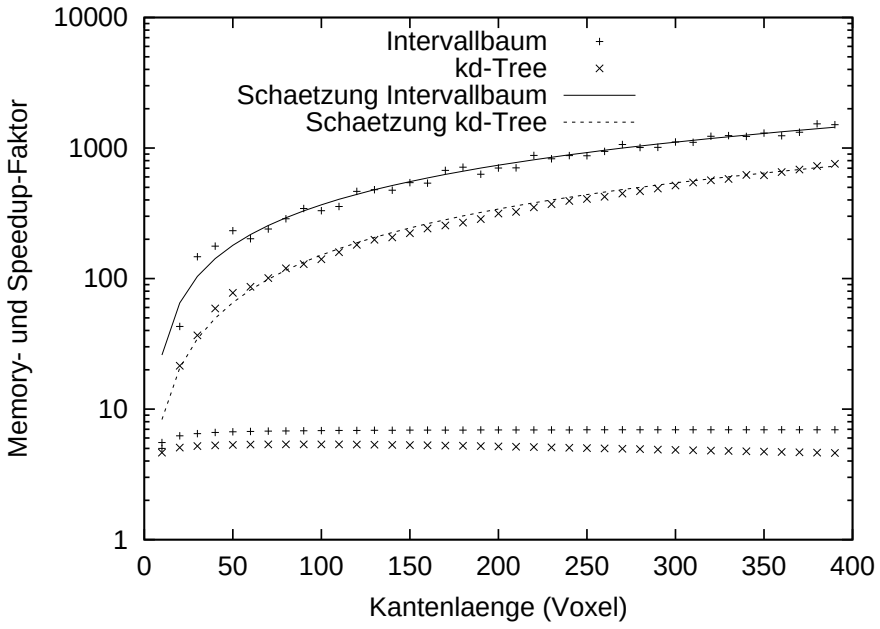


Abbildung 5.6: Der Speedup- (oben) und Memory-Faktor (unten) der KD-Tree- und Intervallbaum-Methode für Kummer-Datensätze verschiedener Größe. Es wird eine Serie von kubischen Datensätzen verwendet; die Dateigröße gibt jeweils die Kantenlänge an. Der Memory-Faktor ist der Quotient des für die Methode insgesamt benötigten Speicherbedarfs mit dem Speicherbedarf des Datensatzes alleine, dabei wird eine Auflösung von 2 Byte/Voxel angenommen. Die durchgehenden Linien ergeben sich aus der theoretischen Schätzung der Rechenzeit.

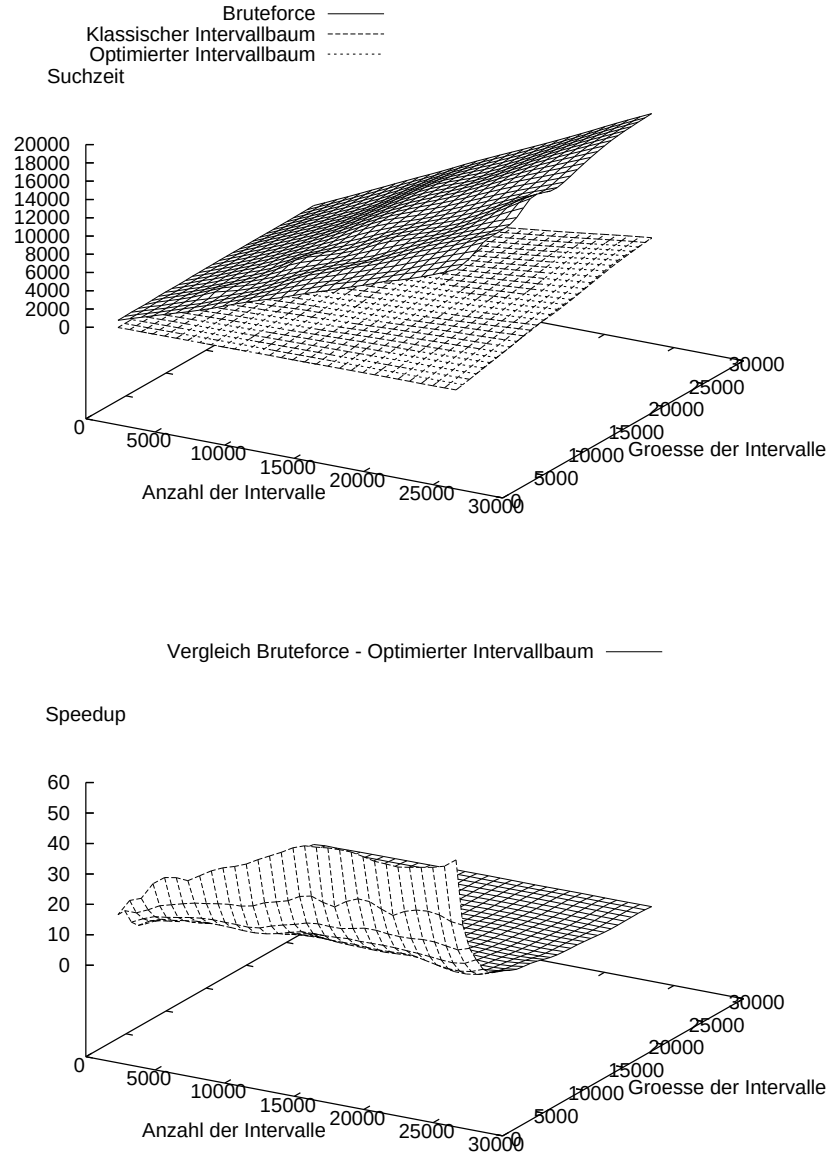


Abbildung 5.7: Die durchschnittlichen Suchzeiten und Speedup-Faktoren der Intervallbaum-Methode, abhängig von der Anzahl der Intervalle und ihrer durchschnittlichen Länge.

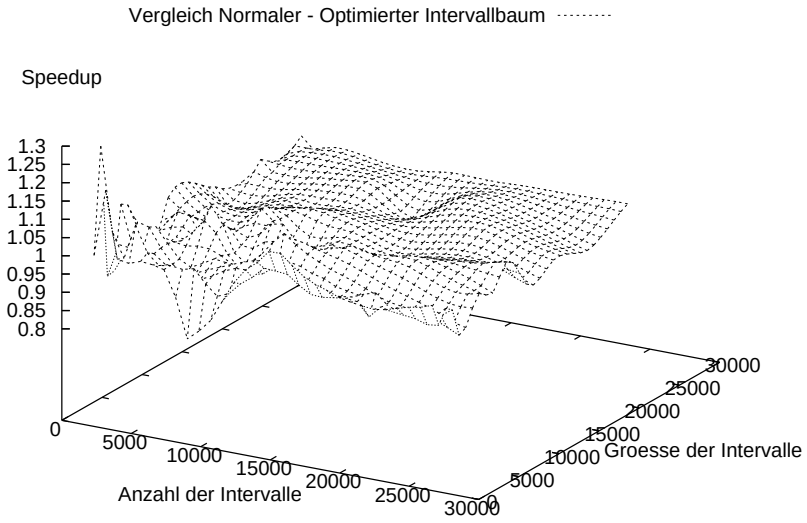


Abbildung 5.8: Der Quotient der gemessenen Suchzeiten eines normalen und eines nach der beschriebenen Methode optimierten Intervallbaums nimmt sowohl Werte über als auch unter 1 an. Die Ungenauigkeit der Messungen überschreitet somit den durch die Optimierung erhaltenen Gewinn.

5.5 Ergebnisse

Nach der theoretischen Analyse benötigt eine Extraktion mit dem KD-Tree $O(k + \sqrt{n})$ und mit dem Intervallbaum $O(k + log(n))$ Rechenzeit, wobei k die Anzahl der ausgegebenen Zellen und n die gesamte Anzahl der Zellen im Datensatz ist. Abbildung 5.6 zeigt für eine Folge von Kummer-Datensätzen die Abhängigkeit des Speicherbedarfs sowie des Rechenzeit-Gewinns der beiden Methoden gegenüber der Brute-Force-Suche.

Abbildung 5.7 zeigt die Ergebnisse eines Experiments, bei dem eine vorgegebene Anzahl von Teilintervallen des Bereichs von 0 bis 65535 ($2^{16} - 1$) zufällig gewählt wurden, wobei der Erwartungswert der Intervall-Länge ebenfalls als Parameter verwendet wird. Abhängig von diesen beiden Parametern wird die mittlere Extraktionszeit für die Brute-Force-Methode, die klassische Intervallbaum-Suche und die Intervallbaum-Suche mit der von mir eingeführten Optimierung der Anzahl der Knoten gemessen. Im ersten Teilbild sind die gemessenen Suchzeiten dreidimensional dargestellt, im zweiten entsprechend die Quotienten von zwei Suchzeiten, also der Speedup-Faktor der neuen Intervallbaum-Methode gegenüber der Brute-Force-Methode.

Bedauerlicherweise ist der Unterschied zwischen der klassischen und der neuen Intervallbaum-Methode nicht relevant; auch ein manueller Vergleich in den Tabellen ergibt keinen Unterschied, der nicht auch mit zufälligen Messfehlern zu erklären wäre. Obwohl Theorie beweist, dass die in diesem Kapitel beschriebene Intervallbaum-Optimierung stets das beste Ergebnis liefert, ist der daraus gezogene Gewinn so gering, dass der Einsatz des optimalen Intervallbaums nur in seltenen Fällen ein lohnenswerter Programmieraufwand ist (Abbildung 5.8).

Eine wesentliche Verbesserung konnte hingegen bei der Anzahl der Intervallbaum-Knoten und

bei der mittleren Anzahl der besuchten Knoten erreicht werden, wie in Abbildung 5.9 zu sehen ist. Als herkömmliche Methode wurde hier das häufig verwendete Prinzip eingesetzt, bei dem als Unterteilungswert jeweils der Mittelwert aller gegebenen Intervallgrenzen verwendet wird.

Bemerkenswert an den Graphen ist auch, dass die Anzahl aller Knoten sich nur um etwa 30 Prozent reduziert hat, während die mittlere Anzahl der besuchten Knoten um 2 bis knapp unter 3 verringert werden konnte. Dieser Effekt ist folglich nicht alleine mit der Reduktion der gesamten Knotenzahl zu erklären, die einer Verkleinerung der Baumtiefe um $\frac{1}{2}$ entspricht. Die Optimierung durch geeignete Anordnung der Unterteilungswerte ist also der Teil des hier beschriebenen Verfahrens, der für die mittlere Suchzeit den größten Vorteil bewirkt hat.

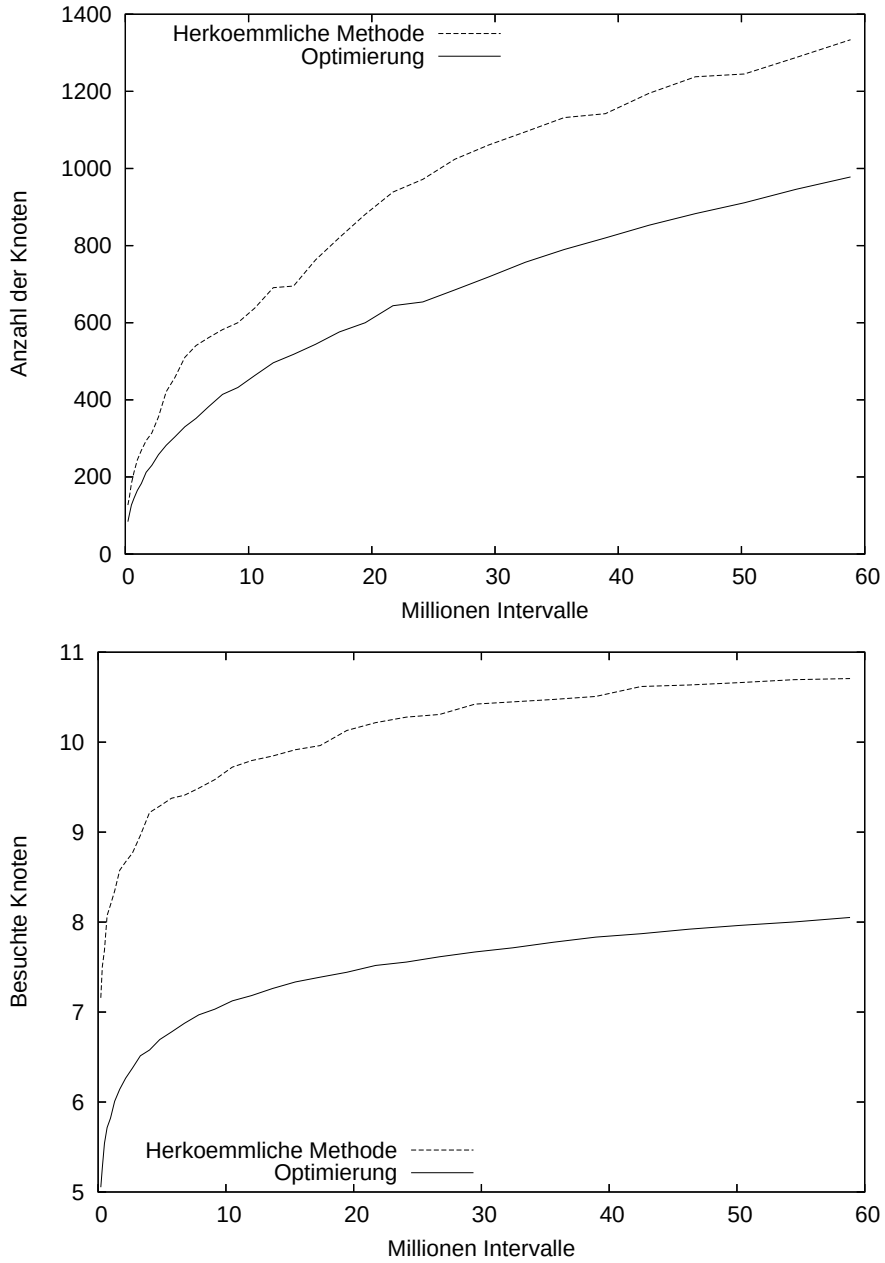


Abbildung 5.9: Vergleich der Knotenzahl (links) und der mittleren Anzahl der besuchten Knoten (rechts) beim optimierten Intervallbaum, im Vergleich zum bereits bekannten Verfahren, als Unterteilungswert jeweils den Mittelwert der Intervallgrenzen zu verwenden.

Kapitel 6

Mathematische Modellierung verschiedener Extraktionsmethoden

Für die Analyse der benötigten Suchzeit eines Extraktionsalgorithmus ist es oft nicht möglich, diese für eine Reihe von Stichproben mit verschiedenen Isowerten zu messen, weil diese Vorgehensweise für eine Berechnung mit entsprechender Genauigkeit zu zeitaufwändig werden kann. Es ist jedoch oft möglich, stattdessen die verwendete Datenstruktur und die Methode zu analysieren und die Anzahl der Rechenschritte verschiedenen Typs zu bestimmen, die für eine Extraktion im Mittel benötigt werden. Wenn der Zeitbedarf eines Rechenschritts bekannt ist, kann die gesamte Suchzeit durch entsprechende Linearkombination ermittelt werden.

In diesem Kapitel werden für verschiedene Methoden der Isoflächen-Extraktion Formeln für den Erwartungswert der Extraktionszeit hergeleitet. Für diesen Erwartungswert wird vorausgesetzt, dass eine Wahrscheinlichkeitsverteilung P gegeben ist, die angibt, mit welcher Wahrscheinlichkeit ein Benutzer einen Isowert γ wählt und die Isofläche dazu anfordert.

Für den jeweils letzten Rechenschritt nehmen wir den einfachen Fall an, dass es sich bei dieser Wahrscheinlichkeitsverteilung um eine Gleichverteilung handelt, es gelte also für jedes Teilintervall $[a, b)$ des Wertebereichs $[\gamma_{\min}, \gamma_{\max})$:

$$P(\gamma \in [a, b)) = \rho(b - a) \quad \text{mit} \quad \rho := \frac{1}{\gamma_{\max} - \gamma_{\min}}$$

Die genaue Bedeutung der Konstanten wird während der nachfolgenden Zeitkosten-Analyse erklärt. Die Linearität der Graphen in den TIME-TIME-Diagrammen, Abbildung 6.1, zeigt, dass mit dieser Wahl der Konstanten hinreichend genaue Ergebnisse erreicht werden. Die unterschiedlichen Skalen der Diagramme kommen daher, dass langsame und schnelle Methoden nebeneinander gestellt werden. Jede Methode für sich ist korrekt modelliert, was an der Übereinstimmung der x- und y-Skala jedes einzelnen Diagramms zu sehen ist.

6.1 Die Suchzeit der Brute-Force-Methode

Um die Suchzeit mit dem Brute-Force-Verfahren zu ermitteln, müssen wir die Anzahl der im entsprechenden Block enthaltenen Zellen bestimmen. Wir gehen dabei von einem Datensatz $(a_{xyz})_{x\in\{0,\ldots,x_{\max}\};y\in\{0,\ldots,y_{\max}\};z\in\{0,\ldots,z_{\max}\}}$ aus und nehmen die naheliegende Vermutung an, dass für

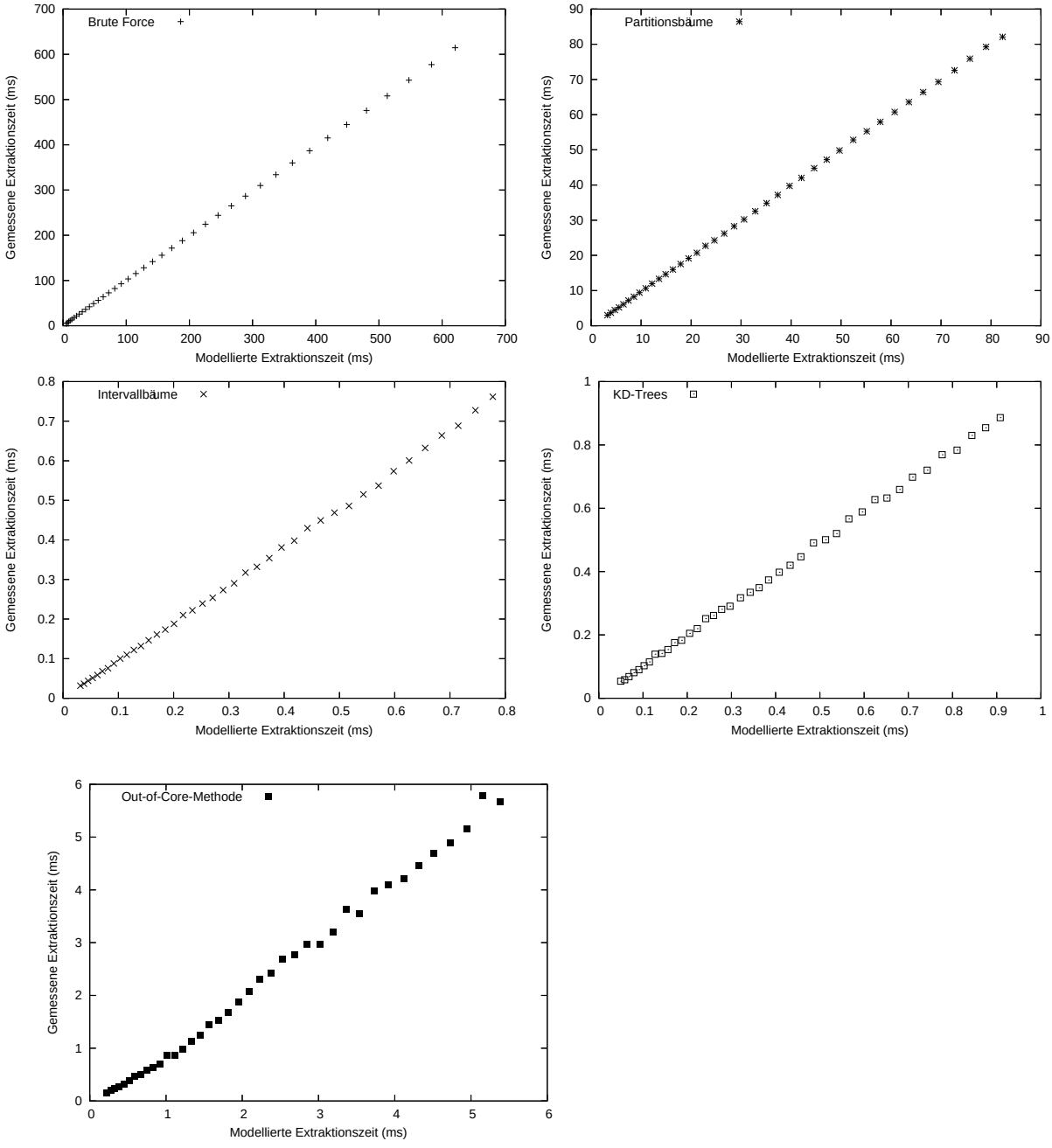


Abbildung 6.1: Die Time-Time-Diagramme zeigen die Abhängigkeit der gemessenen Zeit von der geschätzten Zeit einer Testreihe von Isoflächen-Extraktionen für die verschiedenen Methoden, bei denen die Punkte durch Datensätze verschiedener Größe erzeugt werden.

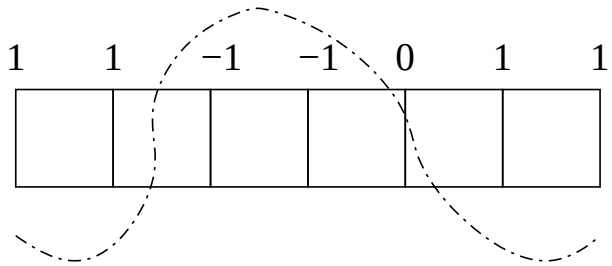


Abbildung 6.2: Eine beschleunigte Version der Brute-Force-Methode.

die Untersuchung einer Zelle stets die gleiche Zeit c_{CELL} benötigt wird. Dann lässt sich die Suchzeit unabhängig vom Isowert γ, und somit auch die mittlere Suchzeit berechnen als:

$$T_{BRU} = x_{\max} y_{\max} z_{\max} \cdot c_{\text{CELL}}$$

Eine erweiterte und etwas schnellere Version der Brute-Force-Methode verwendet jeweils die Informationen der letzten vier Voxel einer Zelle für die nächste Zelle, die ebenfalls von diesen Voxeln begrenzt wird. Wenn dieses Modell benutzt wird, dann lautet die Zeitformel:

$$T_{BRU} = x_{\max} y_{\max} (z_{\max} \cdot c_{\text{QUAD}} + c_{\text{INIT}})$$

Dabei ist c_{INIT} die Zeit, die benötigt wird, um eine Reihe von Zellen für die Suche zu initialisieren, z. B. für den Test der ersten Vierergruppe von Voxeln, die mit γ verglichen werden. c_{QUAD} ist die Zeit, die dann noch für jede zu untersuchende Zelle verbleibt.

Die beschleunigte Version der Brute-Force-Methode wird hier anhand des zweidimensional dargestellten Beispiels von Abbildung 6.2 gezeigt: Wir betrachten eine Reihe von Zellen, die nacheinander untersucht werden und von jeweils zwei Quadraten (in diesem Fall Linien, aber wir stellen uns den dreidimensionalen Fall vor) begrenzt werden. Die Kurve stellt die gesuchte Isofläche dar; die oberhalb davon liegenden Voxel haben Datenwerte, die größer als der Isowert sind; die Datenwerte unterhalb der Kurve seien entsprechend kleiner als der Isowert.

Jeder der Begrenzungen wird ein Zahlenwert zugeordnet. Der Wert 1 bedeutet dabei, dass die Begrenzung vollständig oberhalb der Isofläche liegt. -1 bedeutet, dass sie vollständig unterhalb davon liegt. 0 bedeutet, dass die Begrenzung von der Isofläche geteilt wird.

Die Rechenzeit-Ersparnis besteht nun darin, dass die Bestimmung jedes Zahlenwerts die Betrachtung von vier Datenwerten erfordert und daher ungefähr die Hälfte der Zeit einer naiven Zellenuntersuchung in Anspruch nimmt. Wenn die beiden Werte der Begrenzungen einer Zelle bekannt sind, ist die Untersuchung dieser Zelle so gut wie erledigt: Wenn beide Werte 1 sind, befindet sich die Zelle im Bereich oberhalb der Isofläche. Wenn beide Werte -1 ist, befindet sie sich im Bereich unterhalb davon. In allen anderen Fällen läuft die Isofläche durch die Zelle und ist somit auszugeben. Außer der einmaligen Bestimmung der Zahlenwerte und dieser einfachen Untersuchung braucht in einer Reihe von Zellen nichts getan zu werden; die Methode ist somit schneller als die separate Untersuchung jeder Zelle.

6.2 Die Suchzeit in hierarchischen Zerlegungsbäumen

In einem hierarchischen Zerlegungsbaum lässt sich der Erwartungswert der Extraktionszeit rekursiv über den Aufbau des Baums bestimmen. Jedem Baumknoten p ist durch die rekursive Formel der entsprechende Teil der Suchzeit $T(p)$ zugeordnet; die gesamte Suchzeit ist nach dieser Auswertung in der Wurzel als

$$T_{TREE} = T(\text{root})$$

zu finden.

Wenn der Knoten p ein Blatt ist, dann verwenden wir für die Extraktionszeit im entsprechenden Teilvolumen eine der Formeln aus dem vorangehenden Abschnitt und multiplizieren das Ergebnis mit der Wahrscheinlichkeit, dass dieses Blatt überhaupt angesteuert wird. Es gilt also:

$$T(p) = P(\gamma \in [\min(p), \max(p)))(T_{BRU}(p) + c_{\text{NODE}})$$

Wenn p ein innerer Knoten ist, dann werden die Zeitwerte aller Nachfolger und der Erwartungswert der Untersuchungszeit des Knotens selbst addiert. Unabhängig davon, um welche Art von Zerlegungsbaum es sich handelt (Binärbaum, Octree etc.), lässt sich diese Rekursion formell folgendermaßen ausdrücken:

$$T(p) = P(\gamma \in [\min(p), \max(p))) \cdot c_{\text{NODE}} + \sum_{q \text{ Nachfolger von } p} T(q)$$

In diesen Formeln ist c_{NODE} ein konstanter Wert, der die Zeit für die Untersuchung eines Knotens im Zerlegungsbaum angibt.

Bei der Anwendung dieser Formeln ist es wichtig, zu beachten, dass ein Knoten p genau dann angesteuert wird, wenn $\gamma \in [\min(p), \max(p))$ gilt. Die Grenzen des Intervalls $[\min(p), \max(p))$ müssen also bereits im Vorgänger von p als Information enthalten sein, um dort die Notwendigkeit der Abfrage des Knotens p festzustellen.

6.3 Die Suchzeit der Intervallbaum-Methode

Im wesentlichen ist die Suchzeit in einem Intervallbaum I "output sensitive", d. h., sie ist annähernd proportional zur Länge der Ausgabe (vgl. [13]) Dieser Umstand kann für eine erste Näherungsformel für T_{INT} genutzt werden, indem alle im Baum enthaltenen Zellen betrachtet werden und der Erwartungswert für deren Ausgabezeit bestimmt wird:

$$T_{INT} \approx \sum_{c \text{ Zelle in } I} P(\gamma \in [\min(c), \max(c))) \cdot c_{\text{CELLOUT}} =$$

$$\sum_{c \text{ Zelle in } I} \rho(\max(c) - \min(c)) \cdot c_{\text{CELLOUT}}$$

c_{CELLOUT} ist dabei die Zeit für die Überprüfung und Ausgabe eines Eintrags der Min- oder Maxliste.

Exakter wird die erwartete Rechenzeit berechnet, indem die Zeiten addiert werden, die aus der Suche entlang des von γ abhängigen Baumpfads und der Ausgabe der entsprechenden Stücke der Min- oder Maxlisten jedes Knotens resultieren.

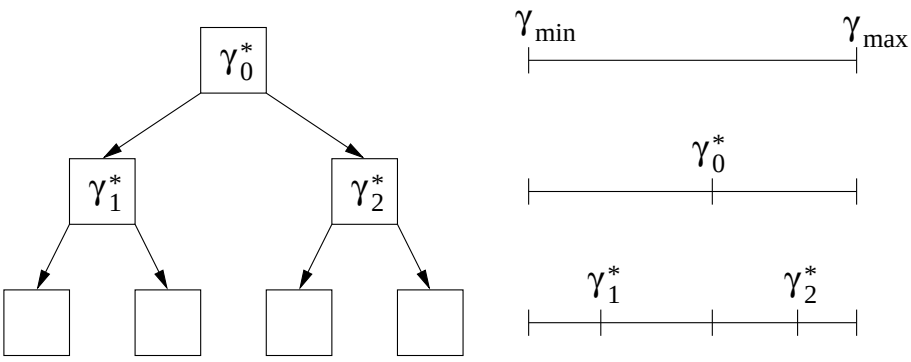


Abbildung 6.3: Die Zuordnung der Intervalle I_η zu ihren Knoten η.

Jeder innere Knoten η des Intervallbaums wird mit einem Wertebereich $I_\eta = [\min(\eta), \max(\eta))$ assoziiert, so dass η genau dann untersucht wird, wenn γ in diesem Bereich liegt. Für die Wurzel des Intervallbaums root ist dies der ganze Wertebereich, für den im Intervallbaum operiert wird:

$$\min(\mathrm{root}) = \gamma_{\min}, \quad \max(\mathrm{root}) = \gamma_{\max}$$

Für die Nachfolger eines bereits bestimmten Intervallbaum-Knotens erhalten wir die Wertebereiche, indem dessen Bereich durch den eingetragenen Grenz-Isowert γ^* zerlegt wird:

$$\min(\mathrm{left}(\eta)) = \min(\eta), \quad \max(\mathrm{left}(\eta)) = \gamma^*(\eta)$$

$$\min(\mathrm{right}(\eta)) = \gamma^*(\eta), \quad \max(\mathrm{right}(\eta)) = \max(\eta)$$

Abbildung 6.3 zeigt anschaulich, wie die Intervalle I_η den Intervallbaum-Knoten zugeordnet werden, indem der Wertebereich $[\gamma_{\min}, \gamma_{\max})$ hierarchisch zerlegt wird.

Um nun daraus T_{INT} zu bestimmen, müssen wir die Erwartungswerte aller Zeiten addieren, die für die Untersuchung der Intervallbaum-Knoten und der zugehörigen Listen benötigt werden:

$$T_{INT} = \sum_{\eta \text{ Knoten in } I} P(\gamma \in [\min(\eta), \max(\eta))) \cdot (c_{\text{INTINIT}} + c_{\text{INTCELL}}) + \sum_{c \text{ Zelle in } I} P(\gamma \in [\min(c), \max(c))) \cdot c_{\text{INTCELL}}$$

c_{INTCELL} ist dabei die Zeit, die für den Test einer Zelle aus der Min- oder Maxliste benötigt wird. Das muss für jede extrahierte Zelle plus eine pro untersuchtem Baumknoten getan werden, bei der der Test negativ ausgeht und die Listenbearbeitung terminiert.

c_{INTINIT} ist die Zeit, die für die Untersuchung eines Intervallbaum-Knotens gebraucht wird, also für den Vergleich des Isowerts γ mit dem eingetragenen Grenz-Isowert γ^*, die Initialisierung der Suche in der entsprechenden Min- oder Maxliste und den Übergang zum linken bzw. rechten Nachfolger.

Anstatt nach dem jeweils letzten auszugebenden Element einer Min- oder Maxliste des Intervallbaums linear von vorne nach hinten zu suchen, kann dieses auch in nur logarithmischer Zeit durch binäre Suche ermittelt werden. Die Routine zur Ausgabe der Isozellen wird dadurch beschleunigt, weil ihre Schleifenabfrage ein einfacher Vergleich des Zählers mit dem so gefundenen Wert ist, anstatt wie bisher ein Vergleich des Isowerts mit dem Min- oder Max-Eintrag im Array. Diese

Verbesserung ist im Programm implementiert und wird auch benutzt. Sie geht aber nicht in das Zeitmodell ein, weil der logarithmische Term verschwindend klein ist; die Verbesserung führt also nur zu einer Verkleinerung der linearen Konstante.

6.4 Die Suchzeit im KD-Tree

Da der KD-Tree im Gegensatz zum Intervallbaum nicht als Baum, sondern als Array gespeichert wird, müssen wir für jedes Element dieses Arrays den Erwartungswert der darin verbrachten Zeit bestimmen. Dieser hängt von dem Teilbereich des Span Space ab, der durch den Knoten bzw. das Array-Element repräsentiert wird. Aus diesem Grund wird dieser Bereich zunächst rekursiv berechnet, wir ordnen also jedem Element η des Arrays den rechteckigen Bereich $[\min_l, \min_h) \times [\max_l, \max_h) =: I_{1\eta} \times I_{2\eta}$ in folgender Weise zu:

$$I_{1\eta_l} \times I_{2\eta_l} := \begin{cases} (I_{1\eta} \cap (-\infty, \min_\eta]) \times I_{2\eta} \\ \text{wenn } \eta \text{ den Min-Wert des Bereichs zerlegt} \\ I_{1\eta} \times (I_{2\eta} \cap (-\infty, \max_\eta]) \quad \text{sonst} \end{cases}$$

$$I_{1\eta_r} \times I_{2\eta_r} := \begin{cases} (I_{1\eta} \cap [\min_\eta, \infty)) \times I_{2\eta} \\ \text{wenn } \eta \text{ den Min-Wert des Bereichs zerlegt} \\ I_{1\eta} \times (I_{2\eta} \cap [\max_\eta, \infty)) \quad \text{sonst} \end{cases}$$

Um diese Rekursion starten zu können, fehlen nur noch die Intervalle $I_{1\eta}$ und $I_{2\eta}$, wenn η die Wurzel des gedachten Baums ist. Diese entsprechen dem vollständigen Intervall der bei Verwendung des KD-Trees möglichen Werte, es gilt also $I_{1\eta} = I_{2\eta} = [\gamma_{\min}, \gamma_{\max}]$.

Bemerkung: die Verwendung von geschlossenen anstelle von rechts offenen Intervallen für die Zerlegung ist in diesem Fall ausnahmsweise korrekt. Der Grund dafür ist, dass es z. B. bei Zerlegung nach dem Maximum außer dem Zerlegungselement noch weitere Elemente mit identischem Maximum geben kann, von denen bei Unterteilung des Arrays in der Mitte einige nach links und einige nach rechts fallen.

Die Suche im KD-Tree führt für einen Knoten η garantiert zum Ergebnis, dass der Eintrag in der Isofläche liegt, wenn $\gamma \in [\min_h, \max_l)$ gilt. In diesem Fall ist ein Schnellverfahren für den entsprechenden Knoten möglich, das diesen und alle darunter liegenden Werte einfach ausgibt. Wir nehmen an, dass diese Ausgabe pro Wert c_{KDININ} Zeit benötigt; für jeden Array-Eintrag, der die Bedingung $\gamma \in [\min_h, \max_l)$ erfüllt, wird diese Zeit nur ein Mal gebraucht.

Der zweite mögliche Fall ist, dass γ in einem der Intervalle $[\min_l, \min(\min_h, \max_l))$ und $[\max(\min_h, \max_l), \max_h)$ liegt. Im ersten Fall muss γ noch mit dem Minimum, im zweiten Fall mit dem Maximum der eingetragenen Zelle verglichen werden. In beiden Fällen wird also nur ein Vergleich gebraucht; wir bezeichnen die für diesen Vergleich benötigte Rechenzeit als c_{KDINOUT}.

Ein besonderer Fall tritt auf, wenn der Teilbereich des Span Space seine Diagonale schneidet und $\max_l < \min_h$ gilt. In diesem Fall kann es vorkommen, dass der Isowert γ im Intervall $[\max(\min_l, \max_l), \min(\min_h, \max_h))$ liegt und somit gegen beide Intervallgrenzen getestet werden muss. Wir bezeichnen die für diese Vergleiche benötigte Zeit als c_{KDOUTOUT}.

Aus diesen Überlegungen resultiert die Formel:

$$T_\eta = c_{\text{KDININ}} P(\gamma \in [\min_h, \max_l)) + c_{\text{KDINOUT}} P(\gamma \in [\min_l, \min(\min_h, \max_l)) \cup [\max(\min_h, \max_l), \max_h)) +$$

$$c_{\text{KDOUTOUT}} P(\gamma \in [\max(\min_l, \max_l), \min(\min_h, \max_h)))$$

mit $P(\gamma \in [a, b)) = \rho(b - a)$ für $a < b$ und 0 sonst, wie bereits beschrieben.

Zusammengefasst ergibt sich für das ganze KD-Tree-Array also die erwartete Suchzeit

$$T_{KD} = \sum_{\eta \text{ Element des KD-Arrays}} T_\eta$$

mit T_η nach vorstehender Formel.

6.5 Eine schnelle Approximation der Suchzeit im KD-Tree

KD-Trees sind oft zu groß, um in angemessener Zeit vollständig ausgewertet zu können. Bei Intervallbäumen besteht die Möglichkeit, die Kostenfunktion ohne Kenntnis der Min- und Maxlisten nur aus den Knoten des Baums zu bestimmen. Bei KD-Trees ist eine ähnliche Möglichkeit nicht gegeben. Es gibt jedoch die Möglichkeit, den Wert der Kostenfunktion mit einem kleineren KD-Tree abzuschätzen, der nur einen zufällig gewählten Teil der gegebenen Zellen enthält.

Die Idee dieser Näherungsmethode wird hier an zwei exemplarischen Beispielen erklärt, die aus der Fallunterscheidung für die Auswertung einzelner KD-Tree-Knoten stammen.

Wie nehmen also an, das Rechteck $R = [\min_l, \min_h] \times [\max_l, \max_h]$ liege vollständig im Innern des Span Space, schneide also nicht die von der Gleichung $\min = \max$ erzeugten Diagonale; es gelte also $\min_h < \max_l$. Ferner nehmen wir an, das Rechteck R sei einem Knoten des reduzierten KD-Trees zugewiesen, der für die Kostenschätzung des großen KD-Trees verwendet wird. Der große KD-Tree enthalte F Zellen, deren Intervall im Rechteck R liegt, oder F sei alternativ eine Schätzung der Anzahl dieser Zellen, die z. B. durch Division der Anzahl aller Zellen im großen und im kleinen KD-Tree bestimmt wurde. Wir schätzen mit diesen Vorgaben eine Teilkomponente der Kostenfunktion im großen KD-Tree, nämlich die Summe

$$S_1 = c_{\text{KDINOUT}} \sum_{k=1}^{F} [P(\gamma \in [\min_{lk}, \min_{hk}]) + P(\gamma \in [\max_{lk}, \max_{hk}])]$$

Um diese Summe abzuschätzen, nehmen wir an, das Rechteck R aus dem kleinen KD-Tree werde durch die Zerlegung des großen KD-Trees in F kleinere Rechtecke zerlegt, die der Einfachheit halber in einem $\sqrt{F} \times \sqrt{F}$-Schema angeordnet sind. Der mittlere Wert von $P(\gamma \in [\min_{lk}, \min_{hk}]) + P(\gamma \in [\max_{lk}, \max_{hk}])$ ist dann $\frac{P(\gamma \in [\min_l, \min_h])}{\sqrt{F}} + \frac{P(\gamma \in [\max_l, \max_h])}{\sqrt{F}}$ und die Kostenkomponente ergibt sich aus der Summe dieser Werte als

$$S_1 \approx c_{\text{KDINOUT}} \sqrt{F} [P(\gamma \in [\min_l, \min_h]) + P(\gamma \in [\max_l, \max_h])]$$

In entsprechender Weise kann die Teilsumme

$$S_2 = c_{\text{KDININ}} \sum_{k=1}^{F} P(\gamma \in [\min_{hk}, \max_{lk}])$$

approximiert werden, indem wir annehmen, dass die Werte von $\min_{hk}$ und $\max_{lk}$ im Rechteck R gleichmäßig verteilt sind und jeweils ein halbes kleines Intervall über $\frac{\min_l + \min_h}{2}$ bzw. unter

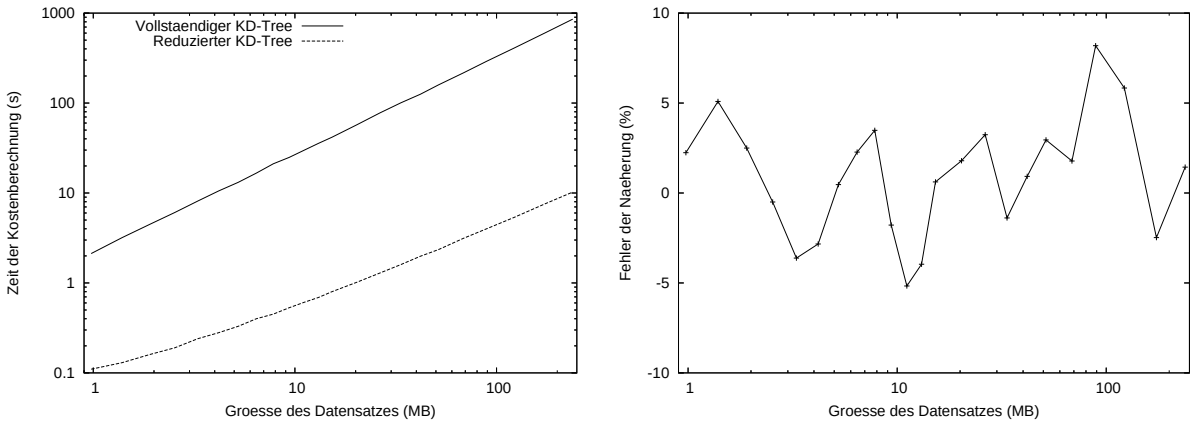


Abbildung 6.4: Berechnung der Kostenfunktion für einen KD-Tree und eine reduzierte Version. Hier wird jeweils der Zeitbedarf und die prozentuale Abweichung der Berechnung vom echten Wert dargestellt.

$\frac{\max_l + \max_h}{2}$ liegen. Die sich aus dieser Annahme ergebende Approximations-Formel für S_2 wäre dann

$$S_2 \approx c_{\text{KDININ}} FP(\gamma \in [\frac{\min_l + \min_h}{2} + \frac{\min_h - \min_l}{2\sqrt{F}}, \frac{\max_l + \max_h}{2} - \frac{\max_h - \max_l}{2\sqrt{F}}])$$

Im Normalfall wird durch diese Formel der Wert von S_2 zu hoch eingeschätzt, weil die Intervalle der Zellen im Rechteck R nicht gleichmäßig verteilt sind, sondern sich nahe bei der Diagonalen des Span Space anhäufen. Aus diesem Grund werden genauere Resultate erreicht, indem anstelle der arithmetischen Mittelwerte die geometrischen Mittelwerte $\sqrt{\min_l \cdot \min_h}$ und $\sqrt{\max_l \cdot \max_h}$ in der Approximations-Formel verwendet werden.

Abbildung 6.4 zeigt anhand einer Serie von Datensätzen steigender Größe, dass bei der Reduktion der Größe von KD-Trees gute Schätzungen der Kostenfunktion erreicht werden können. Verwendet wurden für dieses Experiment Datensätze, deren Größen von 1 bis 250 MB reicht. Berechnet wurden die Original-Kostenfunktion des vollständigen KD-Trees mit c Zellen und eine Schätzung davon aus einem reduzierten KD-Tree mit $f\sqrt[3]{c}$ Zellen, wobei $f = 250$ gewählt wurde. Durch Erhöhung des Werts von f können genauere Ergebnisse erreicht werden, wofür jedoch auch eine größere Rechenzeit benötigt wird.

Im vorliegenden Experiment wird die für die Kostenschätzung benötigte Zeit durch Verwendung eines kleineren KD-Trees um einen Faktor von 20 bis 100 reduziert. Durch diese Reduktion entsteht im Ergebnis ein prozentualer Fehler, dessen Wert nur in wenigen Fällen 5% und in keinem der gemessenen Fälle 10% übersteigt. Das Risiko dieser Vereinfachung ist natürlich, dass mit einer gewissen Wahrscheinlichkeit die zufällige Wahl der für den kleinen KD-Tree verwendeten Stichproben ungünstig verläuft und dadurch ein größerer Fehler entsteht.

Die hier beschriebene Approximation eignet sich nur für Datensätze, in denen die Intervalle im Span Space einigermaßen kontinuierlich verteilt sind. Für Datensätze wie zum Beispiel den Bunny-Datensatz, deren Histogramme größere Anhäufungen von Intervallen in isolierten Punkten enthalten (siehe Abbildung 9.5 weiter hinten), ist dieses Verfahren ungeeignet.

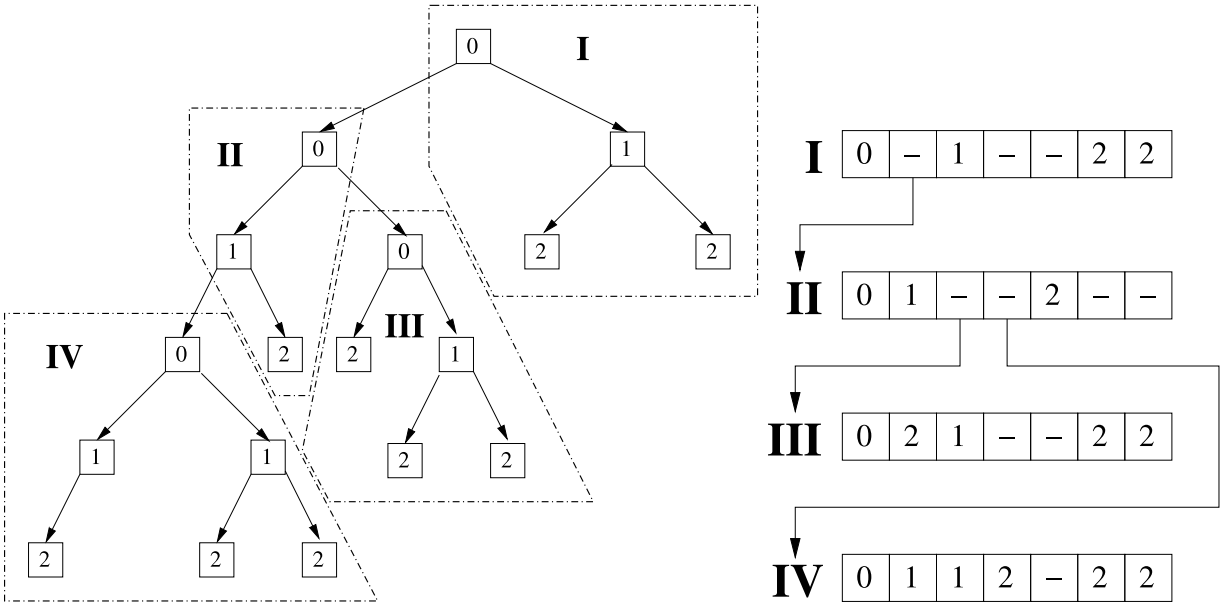


Abbildung 6.5: Die Aufteilung eines extern gespeicherten Intervallbaums in Datenblöcke.

6.6 Die Suchzeit im Out-of-Core-Verfahren

Da bei der Out-of-Core-Extraktion Intervallbäume verwendet werden, sind sich die Formeln zur Bestimmung der Zeitkosten ziemlich ähnlich (siehe Abschnitt 6.3). Bei der Out-of-Core-Extraktion kommt noch ein weiterer Term dazu, der die Anzahl der eingelesenen Datenblöcke schätzt, die sowohl Knotenblöcke als auch Listenblöcke sein können.

Um zusätzlich zur Suchzeit im Intervallbaum die Einlesezeit für die externen Daten zu bestimmen, nehmen wir an, dass ein Datenblock wahlweise $2^{K_1} - 1$ Intervallbaum-Knoten oder K_2 Einträge in die Min- oder Maxlisten enthalten kann.

Die Unterteilung des Knotenbereichs in Datenblöcke wird durchgeführt, indem wir zunächst jedem Knoten η einen Wert $f(\eta) \in \{0, ..., K_1 - 1\}$ zuweisen, und zwar durch die rekursive Formel

$$f(\eta) := \begin{cases} K_1 - 1 & \text{wenn } \eta \text{ ein Blatt ist} \\ 0 & \text{wenn } \eta \text{ die Wurzel ist} \\ \min(\{K_1 - 1\} \cup \{f(\eta') - 1 | \eta' \in \{\eta_l, \eta_r\} \text{ existiert und } f(\eta') \neq 0\}) & \text{sonst} \end{cases}$$

Diese Zuweisung ermöglicht es uns, die Unterteilung in Datenblöcke durchzuführen, deren Teilbäume Wurzeln mit Wert 0 haben und so viele Nachfolger wie möglich enthalten, deren Werte von 0 verschieden sind. Abbildung 6.5 zeigt, wie ein Intervallbaum in hinreichend kleine Teile für externe Speicherblöcke zerlegt wird (links) und wie diese dann aussehen (rechts). In dem gezeigten Beispiel gilt $K_1 = 3$ und die Zahlen geben die Werte von $f(\eta)$ der jeweiligen Knoten an.

Ein extern gespeicherter Knotenblock muss für die Extraktion nur dann gelesen werden, wenn mindestens einer der darin enthaltenen Knoten besucht wird. Das ist genau dann der Fall, wenn die Wurzel des im Block gespeicherten Teilbaums besucht wird, also gilt für den ersten Teil der Einlesezeit:

$$T_1 = c_{\text{BLOCK}} \sum_{f(\eta)=0} P(\gamma \in I_\eta)$$

(c_{BLOCK} ist die Einlesezeit für einen Datenblock)

Ein Block einer Min- oder Maxliste wird genau dann gelesen, wenn γ in seinem charakteristischen Intervall enthalten ist. Das charakteristische Intervall ist dabei nicht das Intervall der ersten im Block enthaltenen Zelle, sondern der letzten Zelle davor, weil das Ende der ausgegebenen Liste lokalisiert werden muss. Es gilt also:

$$I_{BL} := \begin{cases} I_\eta \cap (-\infty, \gamma_\eta^*) & \text{wenn } BL \text{ der erste Block einer Minliste ist} \\ I_\eta \cap [\gamma_\eta^*, \infty) & \text{wenn } BL \text{ der erste Block einer Maxliste ist} \\ I_\eta \cap (-\infty, \gamma_\eta^*) \cap I_{last} & \text{wenn } BL \text{ ein Folgeblock einer Minliste ist} \\ I_\eta \cap [\gamma_\eta^*, \infty) \cap I_{last} & \text{wenn } BL \text{ ein Folgeblock einer Maxliste ist} \end{cases}$$

Dabei ist I_{last} das Intervall der letzten eingetragenen Zelle des jeweils vorherigen Blocks.

Mit diesen Definitionen lässt sich die erwartete Einlesezeit für Blöcke der Min- und Maxlisten insgesamt abschätzen als

$$T_2 = c_{\text{BLOCK}} \sum_{BL \text{ Min- oder Maxblock}} P(\gamma \in I_{BL})$$

Die gesamte Zeit für die Out-of-Core-Extraktion lässt sich nun bestimmen, indem man die Suchzeit für Intervallbäume im Hauptspeicher mit den beiden Einlesezeiten addiert:

$$T_{OOC} = T_{INT} + T_1 + T_2$$

Die Summe zur Berechnung von T_2 ist schwer zu bestimmen, weil die relevanten Intervalle ohne explizite Berechnung der Min- und Maxlisten nicht ohne weiteres ermittelt werden können. Wir können aber annehmen, dass ein normaler Datenblock für K_2 Ausgabe-Intervalle benutzt wird und darüber hinaus durchschnittlich ein halber Datenblock umsonst gelesen wird. Die daraus hervorgehende, schneller auswertbare Näherungsformel lautet

$$T_2 \approx c_{\text{BLOCK}} \left[\sum_{I \text{ eingetragenes Intervall}} \frac{P(\gamma \in I)}{K_2} + \frac{1}{2} \sum_{\eta} P(\gamma \in I_\eta) \right]$$

Kapitel 7

Der Conditioned Tree: Die optimale Kombination mehrerer Datenstrukturen

7.1 Die Struktur des Conditioned Trees

Wie wir gesehen haben, lässt sich Isoflächen-Extraktion mit Partitionsbäumen unter Einsatz von nur wenig Speicher beschleunigen oder mit intervall-basierten Methoden wie dem Intervallbaum- oder dem KD-Tree-Verfahren unter Einsatz von viel Speicher auf eine fast output-sensitive Geschwindigkeit bringen. Es ist jedoch nicht für jede gegebene Speichergröße möglich, eine passende Methode zu finden, die den Speicher zugunsten der Beschleunigung der Extraktion ausschöpft. Hier wird also anstelle einer zwar beliebig erweiterbaren, aber immer endlich langen Liste von vorgeschlagenen Methoden eine vereinheitlichende Methode gebraucht, bei der der Speicherbedarf und die Beschleunigung von einem einstellbaren Parameter abhängen. In diesem Kapitel führe ich das dafür geeignete Conditioned-Tree-Verfahren ein.

Dieses Verfahren hat seinen Namen von der Abfrage der Bedingung (condition), die in jedem Knoten zur Auswahl der entsprechenden Untermethode führt. Im Gegensatz zu den bisherigen Verfahren, die immer die gleiche Methode verwenden, wird der Conditioned Tree durch das in diesem Kapitel beschriebene Optimierungsverfahren auf die jeweils bestmögliche Methode konditioniert.

Das Conditioned-Tree-Verfahren ist eine Methode der Isoflächen-Extraktion, bei der die Datenstruktur von einem Lagrange-Parameter λ abhängt. λ gibt dabei an, wie wichtig die Geschwindigkeit des Suchprozesses im Vergleich zum Speicherbedarf ist. Die Methode kann dann zu jedem vorgegebenen Speicher- oder Zeitwert eine optimale Lösung liefern.

Ein expliziter Speicher- oder Zeitwert lässt sich bei diesem Verfahren nicht direkt vorgeben, aber wenn der Verlauf der Kurve im Parameter-Speicher-Zeit-Diagramm bekannt ist, dann kann zu jeder Speicher- oder Zeitvorgabe der passende Parameter λ gefunden werden.

Formell ausgedrückt ist für ein gegebenes λ die folgende Lagrange-Kostenfunktion zu minimieren, wie es auch im Diagramm (Abbildung 7.1) dargestellt ist:

$$C(\mathcal{T}) := M(\mathcal{T}) + \lambda \cdot T(\mathcal{T})$$

Dabei ist $\mathcal{T}$ die zu optimierende Baumstruktur, $M(\mathcal{T})$ die Größe des von ihr belegten Speichers und $T(\mathcal{T})$ der Erwartungswert der Rechenzeit, die für eine Isoflächen-Extraktion in dieser Struktur gebraucht wird.

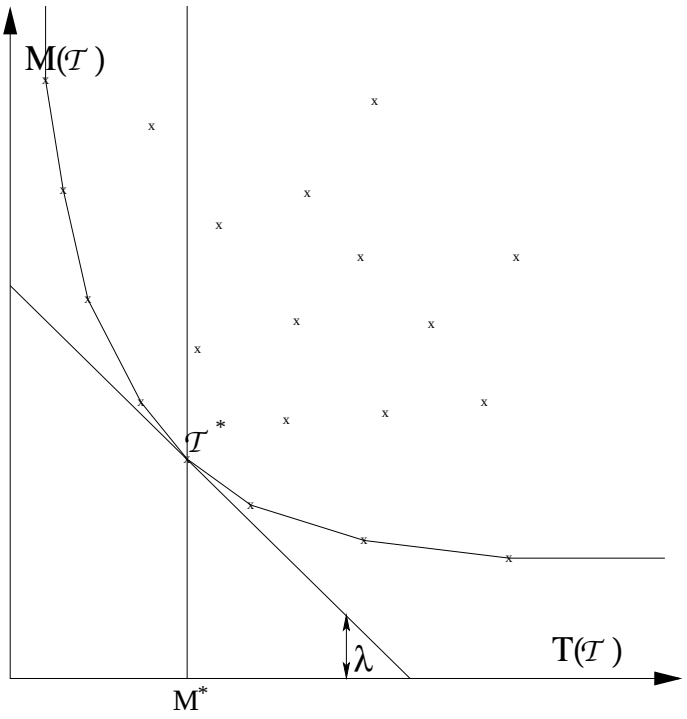


Abbildung 7.1: Die Optimierung der Lagrange-Funktion $C(\mathcal{T}) := M(\mathcal{T}) + \lambda \cdot T(\mathcal{T})$ im Zeit-Speicher-Diagramm. Der optimale Baum zu λ entspricht dem Punkt, an dem die Gerade mit konstantem $C(\mathcal{T})$ die konvexe Hülle der gegebenen Baummenge berührt. Wie am Bild zu sehen ist, ist das Kostenfunktions-Optimum $\mathcal{T}^*$ auch das Suchzeit-Optimum unter allen Bäumen, deren Speicherbedarf M^* nicht überschreitet.

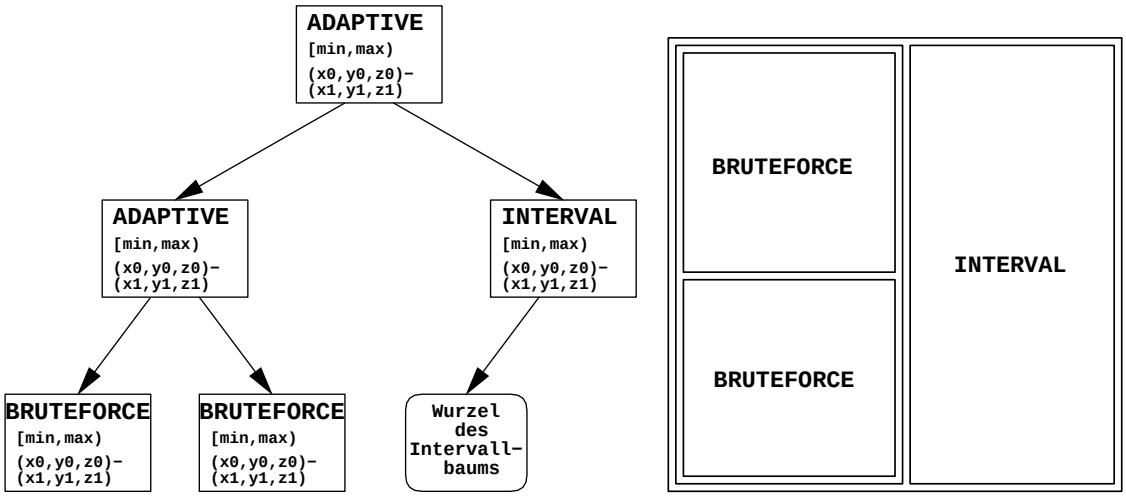


Abbildung 7.2: Die hybride Datenstruktur für die Conditioned-Tree-Methode. Rechts daneben ist die zugehörige Partition der Volumendaten dargestellt.

Der Vorteil der parametrisierten Optimierung ist es, dass sie auch das am Anfang dieses Abschnitts erwähnte Optimierungsproblem bei begrenztem Speicher löst. Im Artikel [16] von Everett wird der Gebrauch von Lagrange-Parametern ausführlich beschrieben und für den allgemeinen Fall gezeigt, dass jedes Objekt $\mathcal{T}$ einer gegebenen Klasse (hier speziell auf Bäume angewendet), das die Kostenfunktion $C(\mathcal{T})$ für ein $\lambda > 0$ minimiert, unter allen möglichen Objekten $\mathcal{T}'$ mit $M(\mathcal{T}') \leq M(\mathcal{T})$ dasjenige mit dem kleinstmöglichen Wert für $T(\mathcal{T}')$ ist.

Bei der Bestimmung des Erwartungswerts wird auf der Menge der möglichen Isowerte eine Verteilung angenommen, abhängig davon, wie häufig diese vom Benutzer angefordert werden. Der Einfachheit halber werden wir im folgenden annehmen, dass die Gleichverteilung auf dem Wertebereich der Volumendaten vorliegt. Wenn eine andere Häufigkeitsverteilung bekannt ist, lässt sich die Methode verallgemeinern, indem eine der folgenden beiden Umformungen vorgenommen wird:

- Die Volumendaten werden so skaliert, dass wir eine Gleichverteilung erhalten. Das ist mathematisch bei allen stetigen Verteilungen möglich, deren Verteilungsfunktion $f(\gamma) = P(-\infty, \gamma)$ bekannt ist; es wird also jeder Eintrag a_{xyz} des Datensatzes durch $P(-\infty, a_{xyz})$ ersetzt.
- Die nachfolgenden Formeln zur Berechnung der erwarteten Suchzeit im Fall der Gleichverteilung lassen sich auch auf den allgemeinen Fall erweitern, wenn $P(-\infty, \cdot)$ bekannt ist. Dafür wird an jeder Stelle, an der eine Wahrscheinlichkeit $P[\min, \max)$ benötigt wird, anstelle von $\max - \min$ die sich aus dieser Funktion ergebende Wahrscheinlichkeit eingesetzt.

Die Struktur des Conditioned Trees ist entsprechend der Abbildung 7.2 folgendermaßen gegeben:

Das Kernstück ist ein binärer adaptiver Partitionsbaum, wie er in dieser Arbeit schon beschrieben wurde, zu einer fest vorgegebenen, zentralen Unterteilung. Die Knoten dieses Baums repräsentieren Teilblöcke der gegebenen Volumendaten und enthalten einen Flag, der aussagt, ob der Baum hier weiter verzweigt, und falls ein Blatt vorliegt, wie der entsprechende Block weiterverarbeitet wird. Dieser Flag kann die Werte $\mathrm{BRUTEFORCE}$ (0), $\mathrm{INTERVAL}$ (1), $\mathrm{ADAPTIVE}$ (2), KDTREE (3) oder $\mathrm{OUTCORE}$ (4) sowie anderen Methoden zugeordnete Werte annehmen, die folgende Bedeutungen haben:

- $\mathrm{BRUTEFORCE}$ bedeutet, dass der Knoten ein Blatt ist und dessen Teilblock gegebenenfalls mit der Brute-Force-Methode durchsucht wird.
- $\mathrm{INTERVAL}$ bedeutet, dass der Knoten ein Blatt ist, von dem aber ein Zeiger auf die Wurzel eines Intervallbaums verweist. Dieser enthält alle Zellen des Teilblocks mit ihren Intervallen und kann gegebenenfalls schnell durchsucht werden.
- $\mathrm{ADAPTIVE}$ bedeutet, dass der Knoten ein innerer Knoten ist und bearbeitet wird, indem man wie im normalen adaptiven Baum einen Intervalltest durchführt und weiter verzweigt.
- KDTREE bedeutet, dass der Knoten ein Blatt ist, von dem ein Zeiger auf die Wurzel eines KD-Trees verweist.
- $\mathrm{OUTCORE}$ bedeutet, dass der Knoten ein Blatt ist, das einen Verweis auf einen auf der Festplatte abgelegten Intervallbaum enthält. Out-of-Core-Methoden benötigen zusätzlich Festplattenspeicher, der in der Kostenformel noch nicht berücksichtigt ist. Später wird beschrieben, wie Out-of-Core-Methoden trotzdem in die Formel eingebaut werden können.

Prinzipiell kann jede Extraktionsmethode, deren Speicher- und Zeitkosten innerhalb angemessener Zeit berechen- oder schätzbar sind, in den Conditioned Tree eingebaut werden. So kann für jeden Teilblock des Datensatzes die bestmögliche aus einer vorgegebenen Liste von Methoden gefunden und verwendet werden.

Die speichersparendste Version des Conditioned Trees enthält gleich in der Wurzel den Flag $\mathrm{BRUTEFORCE}$ und besteht nur aus dieser Wurzel; sie führt nach einer Untersuchung der Wurzel zum klassischen Marching-Cube Verfahren. Die Version, die dagegen die schnellste Extraktion ermöglicht, hat in der Wurzel den Flag der jeweils schnellsten verwendeten Methode (z. B. $\mathrm{INTERVAL}$) und einen Zeiger auf die entsprechende Datenstruktur; in diesem Fall werden alle Daten mit der schnellsten Methode durchsucht. Zwischen den beiden Extremen platzsparend-langsam und speicherintensiv-schnell lassen sich alle möglichen Conditioned Trees im Speicher-Zeit-Diagramm einordnen. Der Fall $\mathrm{OUTCORE}$ nimmt in diesem Schema eine Sonderrolle ein, weil zusätzlich zum Hauptspeicher Festplattenspeicher belegt wird, was zur Bildung eines dritten Extremfalls mit wenig Hauptspeicher, aber viel Festplattenspeicher führt.

Die in der Datenstruktur des adaptiven Baums vorkommenden Knoten sind folgendermaßen aufgebaut:

- Ein Knoten des adaptiven Baums, der das Grundgerüst bildet, enthält die Einträge:

  $(x_0, y_0, z_0) - (x_1, y_1, z_1)$
  flag
  $\min, \max$
  p_1, p_2

  - $x_0, ..., z_1$ sind die Grenzen des repräsentierten Teilblocks.
  - flag ist der Flag, der eine Methode auswählt, also zwischen $\mathrm{BRUTEFORCE}\quad(0)$, $\mathrm{INTERVAL}\quad(1)$ und $\mathrm{ADAPTIVE}\quad(2)$ und den anderen Werten entscheidet.
  - min und max sind der minimale und der maximale Wert der Volumendaten innerhalb des Blocks. Diese werden für den Intervalltest $\gamma \in [min, max)$ verwendet, der entscheidet, ob unterhalb des Knotens weiter untersucht werden muss.
  - p_1 und p_2 sind Zeiger auf andere Datenstrukturen. Im Fall $\mathrm{flag} = \mathrm{BRUTEFORCE}$ werden z. B. beide ignoriert, für $\mathrm{flag} = \mathrm{INTERVAL}$ oder KDTREE zeigt p_1 auf die Wurzel des entsprechenden Baums, für $\mathrm{flag} = OUTCORE$ enthält p_1 die Nummer des Blocks, in dem der extern gespeicherte Intervallbaum beginnt, und für $\mathrm{flag} = \mathrm{ADAPTIVE}$ zeigen p_1 und p_2 auf die Nachfolger des Knotens im adaptiven Baum.

- Ein Knoten des Intervallbaums enthält:

  $\mathrm{flag}, \gamma^*, n$
  $p_{\mathrm{left}}, p_{\mathrm{right}}$
  $p_{\min}, p_{\max}$

  - flag entscheidet, ob der Knoten einen linken Nachfolger (1), einen rechten Nachfolger (2), keinen Nachfolger (0) oder beide Nachfolger (3) hat. Dieser Eintrag ist nicht unbedingt notwendig, weil die Nachfolger auch auf 0 gesetzt werden können; der Vergleich eines Zeigers mit 0 ist jedoch schwerer als die Überprüfung eines einzelnen Flags.

  - γ^* ist der Grenz-Isowert, der in allen Intervallen der Listen $p_{\min}$ und $p_{\max}$ enthalten und für die Aufspaltung in einen linken und einen rechten Teil zuständig ist.
  - n ist die Anzahl der Elemente, die in den Listen $p_{\min}$ und $p_{\max}$ eingetragen sind.
  - p_{left} und p_{right} sind Zeiger auf die Nachfolger des Knotens, wenn diese existieren.
  - $p_{\min}$ und $p_{\max}$ sind Zeiger auf das jeweils erste Element der Min- und Maxliste, die die selben Intervalle nach aufsteigendem $\min$ und nach absteigendem $\max$ enthalten.

- Die Listen, auf die $p_{\min}$ und $p_{\max}$ verweisen, sind Arrays von jeweils n Cellinfo-Strukturen, die folgende Informationen enthalten:

  (x_0, y_0, z_0)
  $a_{\min}$ im Fall $p_{\min}$
  $a_{\max}$ im Fall $p_{\max}$

  - (x_0, y_0, z_0) sind die Koordinaten des Eckpunkts der Zelle, der die kleinsten Koordinaten hat.
  - $a_{\min}$ und $a_{\max}$ sind der minimale und der maximale Datenwert der acht Eckpunkte; diese definieren das Intervall für den Intervallbaum. Abhängig vom Vorkommen in der Min- oder Maxliste wird nur eine der beiden Intervallgrenzen für die Suche benötigt.

- Wenn KD-Trees verwendet werden, dann sind diese als Arrays realisiert und ein Arrayelement enthält die Werte:

  (x_0, y_0, z_0)
  $a_{\min}, a_{max}$

  Diese bezeichnen die Position und den Wertebereich der eingetragenen Zelle. Pointer auf die Nachfolgerknoten werden nicht gebraucht, da sie durch das Prinzip der Array-Halbierung schon vorgegeben sind.

- Die Datenstruktur für Out-of-Core-Extraktion befindet sich anstatt im Hauptspeicher auf der Festplatte und ist dort in Blöcke aufgeteilt. Dabei werden zwei Typen von Blöcken unterschieden:

  - Knotenblöcke repräsentieren einen Ausschnitt des Intervallbaums. Sie enthalten für jeden Knoten dieses Ausschnitts die Informationen über die Position (Nummer des Datenblocks) der Nachfolgerknoten, die Position und Anzahl der Einträge der Min- und Maxliste sowie den eingetragenen Unterteilungswert γ^*.
  - Listenblöcke repräsentieren jeweils eine Min- oder Maxliste oder einen Ausschnitt davon. Sie enthalten für jede darin eingetragene Zelle ihre Koordinaten (x, y, z) und je nach Bedarf den Min- bzw. Maxwert der Zelle.

Zu beachten ist dabei, dass jedem verwendeten Conditioned Tree auf einem gegebenen Datensatz die gleiche Partitionierung zugrunde liegt, die bei der Bestimmung des optimalen Baums stets verwendet wird. Der Eintrag $(x_0, y_0, z_0) - (x_1, y_1, z_1)$ ist daher eigentlich unnötig, wird aber zum Zweck der Zeitersparnis bei der Extraktion dennoch verwendet. Bei der kanonischen Zerlegung wird stets die längste Kante eines Teilblocks genommen und in der Mitte geteilt; die Wahl unter gleich langen Kanten und die Rundung nicht ganzer Zahlen für die 'Mitte' sind dabei einheitlich geregelt, wie es in Abbildung 7.3 zu sehen ist.

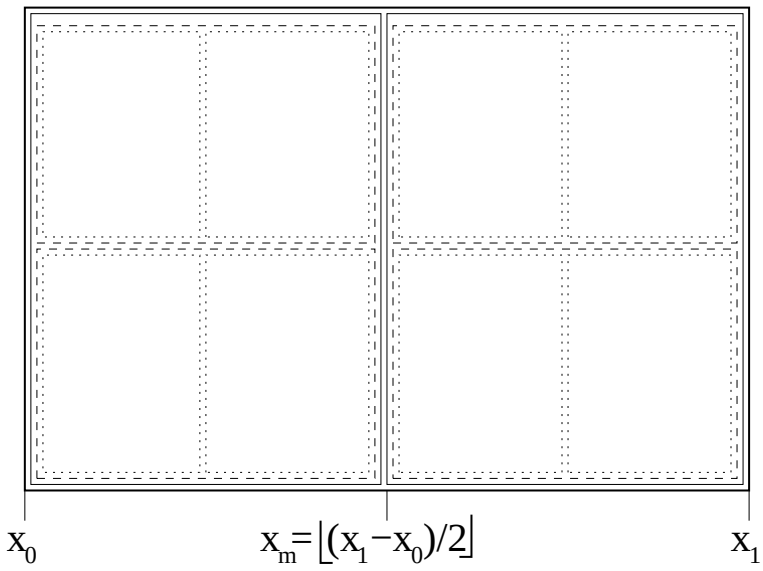


Abbildung 7.3: Die kanonische hierarchische Zerlegung eines Volumen-Datensatzes

Im folgenden Abschnitt wird beschrieben, wie der bezüglich einer vorgegebenen Kostenfunktion optimale Conditioned Tree rekursiv bestimmt wird.

7.2 Lagrange-Optimierung, Minimierung einer Kostenfunktion

Um einen Conditioned Tree mit geringem Speicher und trotzdem kurzen Extraktionszeiten zu finden, benutzen wir rekursive Formeln für $M(p)$, den Speicherbedarf und den geschätzten Erwartungswert der Rechenzeit $T(p)$ pro Extraktion für jeden Knoten s des Baumes $\mathcal{T}$. Dabei sind die auf Baumknoten definierten Funktionen $M(p)$ und $T(p)$ Erweiterungen von den nur für komplette Bäume bestimmten Funktionen $M(\mathcal{T})$ und $T(\mathcal{T})$. $M(p)$ ist definiert als der Speicherbedarf des vollständigen Unterbaums mit der Wurzel p. $T(p)$ ist der Erwartungswert der Suchzeit während einer Extraktion, die im Unterbaum unterhalb des Knotens p verbracht wird.

Das Prinzip der Optimierung ist es nun, für jeden Knoten die beste Methode aus der vorgegebenen Liste, also diejenige mit dem kleinsten Wert für eine Kostenfunktion $C(p) := M(p) + \lambda \cdot T(p)$ zu finden (Abbildung 7.4). Für die Alternative $ADAPTIVE$ (die sich immer unter den Methoden befinden muss, um die Konstruktion des Baums zu ermöglichen) wird dabei rekursiv vorausgesetzt, dass die beiden Unterbäume bereits optimiert worden sind. Da die Kostenfunktion C sich additiv auf der Struktur des Baumes verhält, führt das Prinzip, in jedem Knoten das lokale Optimum zu wählen, stets zum globalen Optimum unter allen möglichen Bäumen. Dieses Prinzip wird bei vielen hierarchischen Optimierungsaufgaben angewendet und oft als dynamisches Programmieren bezeichnet.

Der Einfachheit halber betrachten wir für die Zerlegung von Teilblöcken jeweils nur eine Möglichkeit, obwohl sich ein größerer Block auf viele Arten durch eine achsenparallele Ebene in zwei Teile trennen lässt. Von diesen betrachten wir nur die Möglichkeit, bei der die Teilungsebene orthogonal

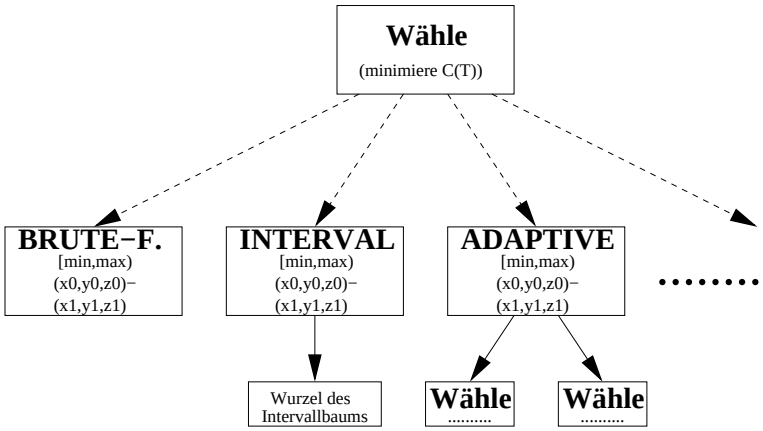


Abbildung 7.4: Der Conditioned Tree wird optimiert, indem nach dem Prinzip des dynamischen Programmierens in jedem Knoten des adaptiven Baums die beste der zur Verfügung stehenden Methoden gewählt wird.

zur längsten Kantenrichtung des Blocks steht und diesen (abgesehen von einer einheitlichen Rundung auf ganze Zahlen) genau in der Mitte teilt. Die unter Berücksichtigung dieser Einschränkung möglichen Conditioned Trees nennen wir zentral unterteilte Conditioned Trees.

Die Berechnung von $M(p)$ ist fast trivial; die Bestimmung von $T(p)$ wird im folgenden beschrieben. Für die Analyse der Einzelmethoden können dabei die Formeln aus Kapitel 6 eingesetzt werden. Die Konstanten dafür, c_{CELL}, c_{QUAD}..., lassen sich durch lineare Approximation und Minimierung des quadratischen Fehlers (Linear Fitting) aus den Ergebnissen bereits ausgewerteter Conditioned Trees bestimmen. Je mehr verschiedenartige Bäume dafür verwendet werden, desto genauer wird natürlich das Ergebnis. Eine genauere Beschreibung der linearen Approximation folgt im Abschnitt 7.3.

Wir nehmen nun bei der rekursiven Optimierung für jeden betrachteten Knoten an, dass der adaptive Baum oder ein Teilbaum bereits vollständig konstruiert ist. Für jeden Knoten p dieses Baums sei $T(p)$ der berechnete Erwartungswert des Zeitbedarfs für die Isowert-Suche in allen Strukturen unterhalb des Knotens. Wenn r die Wurzel des Conditioned Trees ist, dann ist also $T(r)$ der Erwartungswert des Zeitbedarfs für eine Isoflächen-Extraktion abzüglich der Untersuchung der Wurzel r selbst.

Die Wurzel des jeweils untersuchten Teilbaums wird aus rechnerischen Gründen nicht mitgezählt, weil die Auswertung der rekursiven Formel für T dadurch einfacher ist. Dieser Abzug macht für das Optimierungsverfahren nichts aus, weil der Zeitbedarf für die Untersuchung der Wurzel stets der gleiche ist.

Wie bereits im Kapitel 6 erwähnt wurde, nehmen wir an, dass der Isowert γ einer Gleichverteilung unterliegt.

Es ergeben sich abhängig vom Typ des Knotens p für die Berechnung von $T(p)$ mehrere Fälle. In jedem Fall können Formeln aus Kapitel 6 oder ähnliche verwendet werden; diese beziehen sich allerdings nicht auf den vollständigen Datensatz, sondern nur noch auf das Teilvolumen, das dem Knoten p zugewiesen ist.

- **Fall 1:** p **ist ein** BRUTEFORCE**-Knoten.**
  Zur Bestimmung der erwarteten Rechenzeit müssen wir die in Abschnitt 6.1 angegebene Suchzeit mit der Wahrscheinlichkeit multiplizieren, dass der Block wirklich untersucht wird:

  $$T(p) = \rho(\max(p) - \min(p)) \cdot T_{BRU}$$

- **Fall 2:** p **ist ein** INTERVAL**-Knoten.**
  Wenn ein Blattknoten auf einen Intervallbaum verweist, lässt sich die Suchzeit unter Verwendung der Formel aus Abschnitt 6.3 bestimmen als

  $$T(p) = T_{INT}$$

  Bei der Bestimmung von T_{INT} ist zu beachten, dass der Intervallbaum nicht im vollen Werteintervall operiert, sondern nur in dem Bereich, der im Knoten eingetragen ist. Der Wertebereich der Wurzel ist also bestimmt durch

  $$\min(\text{root}) = \min(p), \quad \max(\text{root}) = \max(p) \ .$$

  Die Wertebereiche der anderen Intervallbaum-Knoten werden daraus nach dem im Abschnitt 6.3 beschriebenen Unterteilungsverfahren gebildet.

- **Fall 3:** p **ist ein** ADAPTIVE**-Knoten.**
  Dies ist der Fall, der die Berechnung von $T(p)$ auf eine rekursive Formel führt. Alles unter dem Knoten p zu untersuchen, bedeutet, die Nachfolger von p und alles unterhalb dieser Nachfolger zu untersuchen, also gilt:

  $$T(p) = \rho(\max(p) - \min(p)) \cdot c_{\text{ADAP}} + T(\text{left}(p)) + T(\text{right}(p))$$

  Dabei ist c_{ADAP} ein konstanter Wert, der die Zeit für die Untersuchung der beiden Nachfolger des betrachteten adaptiven Baumknotens angibt.

  Zu beachten ist hier, dass im Gegensatz zu Abschnitt 6.2 $\min(p)$ und $\max(p)$ wirklich im Knoten p selbst eingetragen sind, und nicht in seinem Vorgänger gesucht werden müssen.

- **Fall 4:** p **ist ein** KDTREE**-Knoten.**
  Wenn der Knoten p auf einen KD-Tree verweist, dann lässt sich die erwartete Rechenzeit mit Hilfe der Formeln aus Abschnitt 6.4 oder schneller mit der Näherung aus Abschnitt 6.5 bestimmen als:

  $$T(p) = T_{KD}$$

  Wie beim Intervallbaum ergibt sich auch hier die Einschränkung auf den vom Knoten p definierten Wertebereich. Der Startbereich im Span Space, der der Wurzel des KD-Trees zugeordnet ist, ist also definiert als $I_{1\eta} \times I_{2\eta}$, wobei gilt:

  $$I_{1\eta} = I_{2\eta} = [\min(p), \max(p)) \ .$$

- **Fall 5:** p **ist ein** OUTCORE**-Knoten.**
  Wenn der Knoten auf einen extern gespeicherten Intervallbaum verweist, ist die Kostenformel analog zum Fall eines Intervallbaums im Hauptspeicher nach Abschnitt 6.6 zu bestimmen, wobei die Intervalle I_η der extern gespeicherten Intervallbaum-Knoten wieder ausgehend von der Wurzel als

  $$I_{\text{root}} = [\min(\text{root}), \max(\text{root})) = [\min(p), \max(p))$$

bestimmt werden. Mit dieser Einschränkung des Wertebereichs ist die mittlere Extraktionszeit zu berechnen als

$$T(p) = T_{OOC} \ .$$

Bei der Anwendung des Out-of-Core-Verfahrens eröffnet sich für die Bestimmung der Speicherkosten die Frage, wie Festplattenspeicher im Vergleich zu Hauptspeicher bewertet werden soll. Hierfür wird ein Parameter $\mathcal{H}$ eingeführt, der angibt, wie viel für ein Byte Festplattenspeicher angerechnet wird. Für $\mathcal{H} = 0$ wird angenommen, dass Speicher auf der Festplatte unbeschränkt und kostenlos zur Verfügung steht. Dieser Fall würde bei der Optimierung die Wahl der Out-of-Core-Extraktion begünstigen und ist meiner Ansicht nach nicht immer die sinnvolle Lösung, weil kein System unbegrenzt viel Festplattenspeicher zur Verfügung hat. Selbst in dem Fall, wenn der Festplattenspeicher für die Extraktion ausreicht, wird dieser möglicherweise auch für andere Anwendungen gebraucht; hier steht der Benutzer vor der Entscheidung, den Out-of-Core-Baum entweder bis zum nächsten Aufruf des Programms auf der Festplatte zu lassen und diese damit zu belegen, oder beim nächsten Mal den Baum wieder zu generieren, was eine hohe Preprocessing-Zeit in Anspruch nimmt.

Für $\mathcal{H} = 1$ wird davon ausgegangen, dass ein Byte Festplattenspeicher genau so viel kostet wie ein Byte Hauptspeicher. Dieser Extremfall würde die Verwendung der Out-of-Core-Extraktion von vornherein ausschließen, weil der Grundgedanke dieser Methode gerade darauf basiert, dass mehr Festplattenspeicher als Hauptspeicher zur Verfügung steht. Dieser wird verwendet, um Intervallbäume billiger abzuspeichern; dafür werden große Zeitverluste beim Einlesen der Blöcke in Kauf genommen. Durch Experimente habe ich festgestellt, dass die Out-of-Core-Extraktion für $\mathcal{H} \geq 0.5$ nicht mehr verwendet wird.

Die Berechnungen der Rechenzeit-Formeln haben im Fall der Gleichverteilung der möglichen Isowerte eine Gemeinsamkeit, nämlich den Faktor $\rho = \frac{1}{\gamma_{\max} - \gamma_{\min}}$, der bei jeder Berechnung der Wahrscheinlichkeit eines Intervalls anfällt. Dieser kann also im Optimierungs-Algorithmus weggelassen werden, ohne dass sich dadurch am Verhältnis der verglichenen Zeitwerte etwas ändert.

Ebenfalls eine Vereinfachung, die die Preprocessing-Zeit für mehrere λ-Werte verkürzt, ist es, jeden Wert $M(p)$ und $T(p)$, der sich ohne weitere Verzweigung ergibt und dessen wiederholte Berechnung ansonsten viel Zeit in Anspruch nehmen würde, ganz am Anfang zu berechnen und in einer globalen Kostendatei abzuspeichern. Aus dieser können die von λ unabhängigen Komponenten der Kostenfunktion bei Bedarf entnommen werden.

Die in diesem Abschnitt beschriebene Art der Rechenzeit-Optimierung hat einen großen Vorteil: obwohl die Anzahl der zentral unterteilten Conditioned Trees im Vergleich zur Größe des Datensatzes exponentiell steigt, brauchen bei weitem nicht alle Alternativen zur Bestimmung des besten unter ihnen betrachtet zu werden. Wenn S die Anzahl der Zellen (oder asymptotisch äquivalent die Anzahl der Voxel) des Datensatzes bezeichnet, dann ist nachweisbar, dass der Zeitaufwand für die Optimierung sich durch $O(S \cdot \log^2(S))$ abschätzen lässt. Es werden nämlich für jeden Knoten des adaptiven Baumanteils bei der Optimierung die folgenden Schritte durchgeführt. S' sei dabei die Anzahl der Zellen des betrachteten Teilblocks.

- Berechnung der Rechenzeit für die Brute-Force-Suche im Block: $O(1)$
- Addition der Rechenzeiten der beiden adaptiven Bäume mit der Zeit für die Untersuchung des vereinenden Knotens: $O(1)$

- Analyse des Zeitaufwands für die Suche im Intervallbaum. Das für die Liste der Zellen gebrauchte Grundgerüst des Intervallbaums wird gebildet und ausgewertet. Bei geeigneter Vorsortierung ist das in $O(S' \log(S'))$ Rechenzeit möglich.
- Der Zeitaufwand für Out-of-Core-Verfahren und KD-Trees lässt sich analog zu dem für Intervallbäume in $O(S' \log(S'))$ Rechenzeit schätzen.
- Die rekursive Analyse des Intervallbaum-Gerüsts kostet selbst im schlimmsten Fall, dass S' einelementige Min- und Maxlisten auftreten, $O(S')$ Rechenzeit.
- Der Vergleich der drei Kostenwerte kostet pro Lagrange-Wert $O(1)$; überflüssig gewordene adaptive Teilbäume und Intervallbäume können in $O(S')$ Rechenzeit entfernt werden.

Unter der Voraussetzung, dass die Anzahl der untersuchten Lagrange-Werte klein im Vergleich zur Anzahl der Zellen S ist, wird für jede Ebene des Conditioned Trees $O(S \cdot \log(S))$ Rechenzeit gebraucht, also ergibt sich für den ganzen Baum $O(S \cdot \log^2(S))$.

Wenn andere Methoden dazukommen, dann nehmen wir an, dass die Rechenzeit-Analyse jeder dieser Methoden in höchstens $O(S'^n)$ $(n > 2)$ pro Datenstruktur durchgeführt werden kann. Wenn sie sogar in $O(S' \log(S'))$ Zeit möglich ist, ändert sich nichts an der Analyse der gesamten Optimierungszeit. Andernfalls ist die Rechenzeit für die Optimierung des gesamten Baums $O(S^n)$, wie die Zeit für die Analyse der am schwersten untersuchbaren Methode.

Formell lässt sich die Rechenzeit für die Suche nach dem optimalen Conditioned Tree durch folgenden Satz festlegen:

Satz 1:

Es sei $\mathcal{T}$ ein binärer Baum mit S Blättern und einer Tiefe E, für die $2^E = O(S)$ gilt. $\mathcal{A}$ sei ein Algorithmus, der jeden Knoten K von $\mathcal{T}$ in $O(S'(K)^n)$ Rechenzeit bearbeitet, wobei $n \in \mathbb{N}, n > 0$ und $S'(K)$ die Anzahl der unterhalb von K liegenden Blätter ist. Dann lässt sich $\mathcal{A}$ für $n = 1$ in $O(S \cdot \log(S))$, für $n \geq 2$ in $O(S^n)$ Rechenzeit durchführen.

Im von uns betrachteten Fall ist für $\mathcal{T}$ der maximale theoretisch erreichbare Conditioned Tree zu verwenden, dessen Blätter genau den im Datensatz vorkommenden Zellen entsprechen. Der Algorithmus $\mathcal{A}$ ist die Conditioned-Tree-Optimierung, die in jedem Knoten K eine fest vorgegebene Liste von Methoden abarbeitet, von denen die aufwändigste Methode und somit auch die Gesamtuntersuchung $O(S'(K)^n)$ Zeit benötigt. Nun aber zum Beweis:

Beweis:

Der Baum $\mathcal{T}$ hat E Ebenen, die in der Notation des Beweises von den Blättern bis zur Wurzel mit durchnummeriert seien. Aus der oben stehenden Voraussetzung folgt $E = O(\log(S))$. Die Knoten der Ebene k seien mit $K_{k1}, K_{k2}, ..., K_{km_k}$ bezeichnet. Dann gelten für die Anzahl der Blätter unterhalb der Knoten folgende Ungleichungen:

$$S'(K_{kj}) \leq 2^{k-1}$$

$$\sum_{j=1}^{m_k} S'(K_{kj}) \leq S$$

Die erste Ungleichung folgt daraus, dass jeder Knoten der Ebene k in höchstens $k-1$ darunter liegenden Ebenen verzweigen kann, die zweite ist eine obere Abschätzung durch Summierung aller vorhandenen Blätter.

Für $n = 1$ benötigt der Algorithmus $\mathcal{A}$ für die Bearbeitung aller Knoten der Ebene k eine Rechenzeit von $\sum_{j=1}^{m_k} O(S'(K_{kj})) = O(S)$, also für alle Ebenen $O(S \cdot E) = O(S \cdot \log(S)$ Rechenzeit.

Für $n \geq 2$ betrachten wir die Tatsache, dass die Ebene k höchstens 2^{E-k} Knoten enthält, von denen jeder die oben stehende Ungleichung $S'(K_{kj}) \leq 2^{k-1}$ erfüllt. Der Algorithmus benötigt also für die Bearbeitung von Ebene k eine Rechenzeit von:

$$\sum_{j=1}^{m_k} S'(K_{kj})^n = O(2^{E-k} \cdot (2^{k-1})^n) = O(2^{E-k+nk-n}) = O(2^{k(n-1)} \cdot 2^{E-n})$$

Da für die Abschätzung der Rechenzeit implizit die Verwendung eines einheitlichen konstanten Faktors angenommen wird, ergibt die Addition der Ausdrücke für die Ebenen 1 bis E eine Abschätzung der gesamten Rechenzeit des Algorithmus $\mathcal{A}$:

$$\sum_{k=1}^{E} O(2^{k(n-1)} \cdot 2^{E-n}) = O(2^{E(n-1)} \cdot 2^{E-n}) = O(2^{En-n}) = O((2^{E-1})^n) = O(S^n) \qquad \text{(q.e.d.)}$$

Eine beliebige adaptive Zerlegung anstelle einer Zerlegung der Datenblöcke in der Mitte würde hingegen zu zu hohen Rechenzeiten führen, da in einem so gebildeten Zerlegungsbaum jeder Teilblock des Datensatzes vorkommen kann. Schon alleine für das Ansteuern aller möglichen Teilblöcke $(x_0, y_0, z_0) \times (x_1, y_1, z_1)$ zur Bestimmung ihrer Kostenwerte würde somit $O(S^2)$ Rechenzeit benötigt werden, was für Datensätze von mehreren hundert Megabyte inakzeptabel wäre.

Die von mir realisierte Version des Conditioned-Tree-Extraktionsprogramms nutzt die Abhängigkeit vom Parameter λ gezielt aus. Es ist nämlich für jeden neuen Datensatz möglich, gleich am Anfang ein Preprocessing-Programm aufzurufen, das für einen weiten Parameter-Bereich die optimalen Bäume konstruiert und sie zusammen mit einer Tabelle mit den entsprechenden Speicherbedarfs- und Rechenzeit-Werten abspeichert. Das ergibt also eine Liste von Parametern $\lambda_1, \lambda_2, ..., \lambda_n$ mit ihren optimierten Bäumen $\mathcal{T}_k$ ($k = 1, ..., n$), die jeweils M_k Hauptspeicher und durchschnittlich T_k Rechenzeit pro Extraktion benötigen. Das Hauptprogramm erhält dann als Parameter den zur Verfügung stehenden Hauptspeicher M^*, für den in der Tabelle der Baum $\mathcal{T}_k$ mit dem passenden Speicherwert gesucht wird, also sei M_k der größte Wert der Liste mit $M_k \leq M^*$. Dieser Baum wird dann eingelesen und als Datenstruktur für die Isoflächen-Extraktion benutzt. Wenn für die Konstruktion der Tabelle nicht genügend Zeit zur Verfügung steht, dann kann nach dem vorgegebenen Speicherwert M^* auch eine binäre oder anders geartete direkte Suche in den möglichen Werten für λ durchgeführt werden, weil dafür bei gleicher geforderter Genauigkeit viel weniger Optimierungsdurchgänge durchgeführt werden müssen.

Da die Conditioned-Tree-Methode das Hauptthema dieser Arbeit ist, sind die numerischen Ergebnisse dafür im letzten Kapitel ausführlich dargestellt.

7.3 Lineare Approximation der Zeitkonstanten

Wir nehmen nun an, dass wir den programmtechnischen Teil sowohl der Optimierung als auch der Extraktion realisiert haben. Für jede verwendete Untermethode des Conditioned Trees sei ein Modell gegeben, mit dem der Rechenzeit-Bedarf bestimmt werden kann, es fehlen jedoch noch die Konstanten, mit denen die Zeit für die Durchführung eines Arbeitsschritts bestimmt wird.

Der Einfachheit halber nennen wir diese Konstanten nicht $c_{\text{INTINIT}}, c_{\text{INTCELL}}, c_{\text{CELL}}...$ wie im letzten Abschnitt, sondern bezeichnen sie systematisch als $c_1, c_2, ..., c_k$ und fassen sie zum Konstantenvektor C zusammen.

Nun führen wir eine Reihe von Experimenten mit verschiedenen Testbäumen $\mathcal{T}_1, ..., \mathcal{T}_n$ durch, wobei n groß sein sollte, aber auf jeden Fall größer als k. In diesen Bäumen werden die mittleren Extraktionszeiten gemessen, diese seien $t_1, ..., t_n$ und bilden den Zeitvektor T. Ferner werden die Bäume mit dem mathematischen Modell analysiert, das die mittlere Extraktionszeit bestimmen soll. Wir nehmen an, dass der Rechenschritt i im Baum $\mathcal{T}_j$ durchschnittlich a_{ji} Mal durchgeführt werden muss, also ergibt die Rechenzeit-Analyse die Formeln

$$T(\mathcal{T}_j) = \sum_{i=1}^{k} a_{ji} c_i \quad j = 1...n$$

oder einfacher ausgedrückt, wenn die Indizes a_{ji} zur Indexmatrix A zusammengefasst werden:

$$A \cdot C = T \quad (A, T \text{ bekannt, } C \text{ gesucht})$$

Unter den Annahmen, dass $n > k$ gilt und sich bei der Messung Ungenauigkeiten ergeben, hat dieses Gleichungssystem keine genaue Lösung. Wir können die Lösung jedoch approximieren, indem wir sie ersetzen durch das Optimierungsproblem

$$|T - AC| = \text{minimal}$$

Eine altbekannte Methode aus der Statistik zur Bestimmung der Konstanten des Modells besteht darin, die Lösung dieses Näherungsproblems durch das Gleichungssystem

$$A^T A \cdot C = A^T T$$

zu bestimmen, das meistens eine eindeutige Lösung hat, weil die Matrix $A^T A$ quadratisch ist.

Der Grund für die Korrektheit dieser Lösung besteht in folgendem mathematischen Zusammenhang:

$$|T - AC| = \text{minimal} \iff T - AC \perp AV \text{ für alle Vektoren } V \iff$$

$$T - AC \perp AE \text{ für alle Einheitsvektoren } E \iff$$

$$< T - AC, AE >= 0 \text{ für alle Einheitsvektoren } E \iff$$

$$< T - AC, A_i >= 0 \text{ für alle Spaltenvektoren } A_i \text{ der Matrix } A \iff$$

$$A_i^T \cdot (T - AC) = 0 \text{ für alle Spaltenvektoren } A_i \text{ der Matrix } A \iff$$

$$A^T \cdot (T - AC) = 0 \iff A^T A \cdot C = A^T T$$

Die lineare Approximation mit einem Prozessor AMD Athlon(TM) XP 1600+ 1410 MHz hat folgende Näherungswerte ergeben, die für das vorher beschriebene Optimierungsprogramm verwendet werden:

$$c_{\text{INTINIT}} = 0 \ ns \quad c_{\text{INTCELL}} = 4.34 \ ns \quad c_{\text{CELL}} = 42.6 \ ns \quad c_{\text{ADAP}} = 136 \ ns$$

$$c_{\text{KDININ}} = 3.95 \ ns \quad c_{\text{KDINOUT}} = 33.54 \ ns \quad c_{\text{KDOUTOUT}} = 0 \ ns \quad c_{\text{BLOCK}} = 2288 \ ns$$

Die beiden Null-Werte kommen daher, dass die betroffenen Routinen vergleichsweise selten aufgerufen werden und daher durch ein Linear Fitting nicht genau approximiert werden können. Gerade wegen dieses seltenen Aufrufs fallen die Werte dieser Konstanten jedoch auch nicht ins Gewicht. Durch das Entfernen dieser beiden Rechenzeit-Konstanten hat sich auch der Wert der anderen Konstanten geringfügig geändert, da die orthogonale Projektion im Messraum (T wird auf den Suchraum aller noch möglichen AC projiziert) nicht exakt der orthogonalen Projektion im Lösungsraum (C wird auf den Unterraum projiziert, für den $c_{\text{INTINIT}} = c_{\text{KDOUTOUT}} = 0$ gilt, diese werden also einfach auf 0 gesetzt) entspricht.

7.4 Übersicht aller Arbeitsschritte des Optimierungsverfahrens

Damit das Verständnis des Zusammenhangs der beschriebenen Methoden nicht zu schwer wird, ist hier eine Übersicht über die nötigen Rechenschritte vom Datensatz bis zum Bild dargestellt. Das Verfahren teilt sich in zwei große Bereiche auf, nämlich das Preprocessing (Optimierung) und das Hauptprogramm (Extraktion).

Das Preprocessing läuft im Wesentlichen folgendermaßen ab:

- Input: Volumendaten
- Vorbereitung:
  Dies ist der rechenzeit-intensivste Teil des Programms, er muss aber für jeden Datensatz nur ein Mal durchgeführt werden.
  - Volumenpartitionierung
  - An jedem Knoten berechne für die Methoden $1, ..., m$ den Speicherbedarf M und die Suchzeit T
  - Speichere den Baum mit den Kostenwerten
- Optimierung:
  Je nach Anwendung kann eine Tabelle aus vielen verschiedenen Lagrange-Werten λ angelegt oder, bei bereits bekanntem und gleich bleibendem Rechenspeicher, eine binäre Suche nach dem dazu passenden Lagrange-Wert gestartet werden.
  - Für alle λ einer hinreichend großen Liste minimiere die Kostenfunktion $C_\lambda(\mathcal{T}) = M(\mathcal{T}) + \lambda T(\mathcal{T})$
  - Speichere den optimalen Baum als $\mathcal{T}^*_\lambda$

Das interaktive Hauptprogramm, in dem der Benutzer die Isoflächen in Form eines Bildes erhält, wird folgendermaßen aufgeteilt:

- Input: Hauptspeicher M
- Einlesen:
  Zuerst werden der Datensatz und die Informationen für den passenden Conditioned Tree eingelesen und aufgebaut.

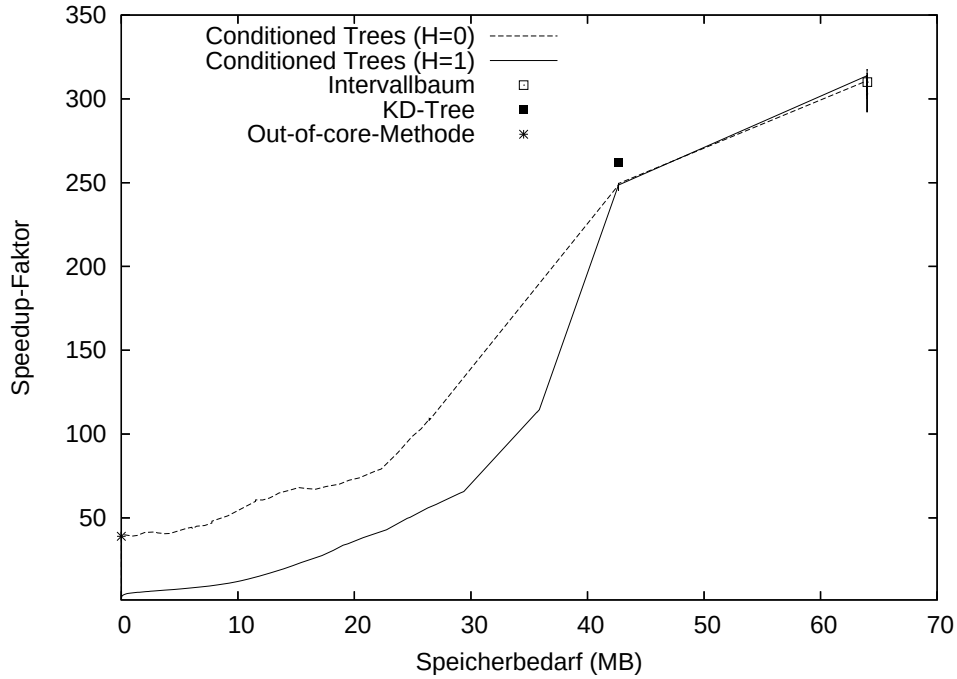


Abbildung 7.5: Das Memory-Speedup-Diagramm für die Isoflächen-Extraktion des $256 \times 256 \times 110$-Engine-Datensatzes.

  - Suche das größte λ, für das $M(\mathcal{T}^*_\lambda) \leq M$ gilt
  - $\mathcal{T}^*_\lambda$ einlesen und aufbauen

- Extraktion:
  Hier folgt der interaktive Teil, in dem sich die Bilder ansehen und steuern lassen.
  Wiederhole

  - Input: Isowert γ
  - Extraktion der Isofläche mit Hilfe von $\mathcal{T}^*_\lambda$
  - Isofläche rendern

  bis der Benutzer das Programm beendet

7.5 Die Gap-Filling-Methode

Wie Everett nachgewiesen hat, erzeugt die Lagrange-Optimierung stets optimale Lösungen für das Problem der Zeitminimierung bei vorgegebenem Speicher. Es ist jedoch nicht garantiert, dass dadurch alle optimalen Lösungen erzeugt werden; es gibt nicht nur theoretisch, sondern auch praktisch Fälle, in denen einige optimale Lösungen nicht erzeugt werden.

Zwei dieser Fälle sind die Datensätze Engine und Bighead, deren Memory-Speedup-Diagramme aus diesem Grund nicht erst in den Ergebnissen erscheinen, sondern in dieses Kapitel vorgezogen

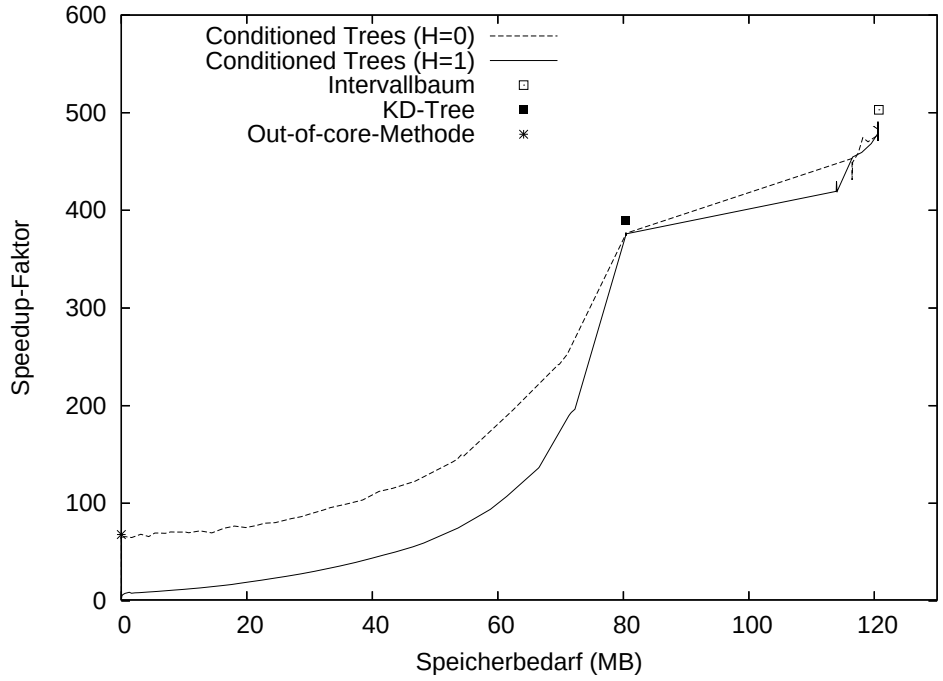


Abbildung 7.6: Das Memory-Speedup-Diagramm für die Isoflächen-Extraktion des $256 \times 256 \times 225$-Bighead-Datensatzes.

wurden (Abbildung 7.5 und 7.6). Hier lässt sich eine Lücke zwischen dem KD-Tree und dem Intervallbaum erkennen, in der sich tatsächlich keine oder beim Datensatz Bighead nur wenige Optima befinden. Eine Verfeinerung der Abstände zwischen den Lagrange-Werten führt dabei auch nicht zu einem besseren Ergebnis.

Es ist aber dennoch möglich, dass sich Lösungen des Optimierungsproblems innerhalb dieser Bereiche befinden. Diese werden durch Lagrange-Optimierung nicht gefunden, weil sie innerhalb der konvexen Hülle liegen. Sie sind aber dennoch optimal, weil sie weniger Speicher als der am rechten Ende liegende Intervallbaum und weniger Zeit als der am linken Ende liegende KD-Tree brauchen (Abbildung 7.7).

In diesem Abschnitt werde ich ein Verfahren beschreiben, mit dem Lösungen des Optimierungsproblems ermittelt werden können, die nicht durch die Lagrange-Optimierung erreicht werden. Der Einfachheit halber beschränke ich mich hier auf die Lücke zwischen dem KD-Tree und dem Intervallbaum, da diese Lücke die einzige von nennenswerter Größe ist, die in den bisherigen Experimenten aufgetreten ist. Der Grund für eine gerade an dieser Stelle vorkommende Lücke ist, dass KD-Trees sich schlecht zerlegen lassen: Während der Speicher- und Zeitbedarf von Intervallbäumen und der Speicherbedarf von KD-Trees sich annähernd additiv verhalten, erhöht sich der Zeitbedarf eines KD-Trees gravierend, wenn er in zwei Teile zerlegt wird.

Das Suchverfahren basiert auf einer Zerlegung der Volumendaten in 2^m Teile (hier wurde $m = 12$ verwendet); für jeden dieser Teile sowie für die in der hierarchischen Zerlegung darüber liegenden Teilvolumen sind aus der Bestimmung des Kostenbaums jeweils der Speicher- und mittlere Zeitbedarf des Intervallbaums und des KD-Trees bekannt. Diese nennen wir hier $M_I(k)$, $T_I(k)$, $M_K(k)$ und $T_K(k)$ ($k = 1, ..., 2^{m+1} - 1$).

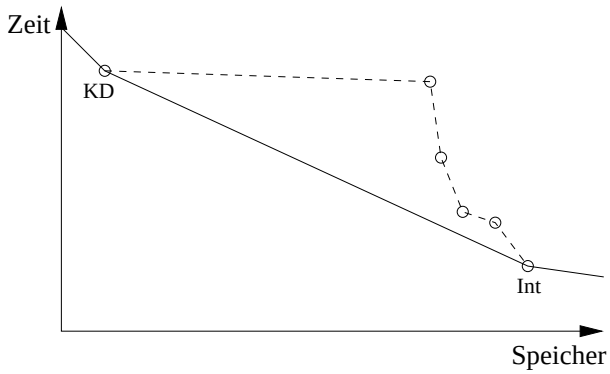


Abbildung 7.7: Eine Optimierungslücke zwischen dem KD-Tree- und dem Intervallbaum-Verfahren. Die durchgezogene Linie ist die durch Lagrange-Optimierung gebildete Grenze der konvexen Hülle, auf der sich die Zwischenlösungen entlang der gestrichelten Linie nicht befinden.

Dabei sind die Indizes k so organisiert, dass sich ihre Zuordnung zur Struktur der Partition leicht rekonstruieren lässt. Die Kostenwerte $M_I(1), ..., T_K(1)$ sind dem vollständigen Datensatz bzw. der Wurzel des Partitionsbaums zugeordnet. Wenn die Werte $M_I(k), ..., T_K(k)$ einem Knoten zugeordnet sind, dann erhält sein linker Nachfolger die Werte $M_I(2k), ..., T_K(2k)$ und sein rechter Nachfolger die Werte $M_I(2k+1), ..., T_K(2k+1)$, sofern diese noch im vorgegebenen Indexbereich liegen.

Ferner definieren wir den Speichergewinn des k. Teilblocks als $\Delta M(k) := M_I(k) - M_K(k)$ sowie seinen Zeitverlust als $\Delta T(k) := T_K(k) - T_I(k)$. Da ein Intervallbaum im allgemeinen schneller ist als der entsprechende KD-Tree, aber dafür mehr Speicher benötigt, sind diese Werte meistens positiv. Der Kostenwert des k. Teilblocks ist definiert als $C_{IK} := \frac{\Delta T(k)}{\Delta M(k)}$ und bezeichnet den Preis in Form von Rechenzeit, der bei einer Änderung des entsprechenden Bereichs von einem Intervallbaum- in einen KD-Tree-Bereich für eine Ersparnis von $\Delta M(k)$ pro ersparte Rechenzeit zu zahlen wäre.

Ausgehend vom vollständigen Intervallbaum stehen also $2^{m+1}-1$ Kandidaten für den Wechsel zum KD-Tree zur Wahl. Von denen werden einige als unbrauchbar gekennzeichnet, z. B. alle Werte, für die $\Delta M(k) \leq 0$ gilt und somit gar keine Speicherersparnis erreicht werden kann. Ebenso unbrauchbar sind alle Teilblöcke, in denen die Umwandlung des Intervallbaums zum KD-Tree bereits zur Überschreitung der Rechenzeit des vollständigen KD-Trees führen würde, also wenn $\Delta T(k) > \Delta T(1)$ wäre.

Der erste Schritt des Gap-Filling-Verfahrens nach der Vorbereitungsphase besteht darin, unter den brauchbaren Kandidaten den Block k mit dem kleinsten Kostenwert $C_{IK}(k)$ zu bestimmen. Für diesen wird als erstes Optimum der kleinstmögliche Baum konstruiert, bei dem jeder Pfad in einem Intervallbaum endet, mit Ausnahme des Pfads auf den Teilblock k, der einen KD-Tree erhält. Wir erhalten dadurch eine Datenstruktur, die die Lücke im Speicher-Zeit-Diagramm in zwei kleinere Lücken zerlegt (Abbildung 7.8, links).

Sowohl vom ursprünglichen Intervallbaum, als auch vom so erhaltenen Baum ausgehend lässt sich eine weitere links davon liegende Datenstruktur finden, indem dieser Vorgang wiederholt wird, jedoch mit einer kleineren Liste möglicher Kandidaten. Dafür werden zusätzlich alle Kandidaten als ungeeignet gekennzeichnet, in denen die Umwandlung des Intervallbaums in einen KD-Tree

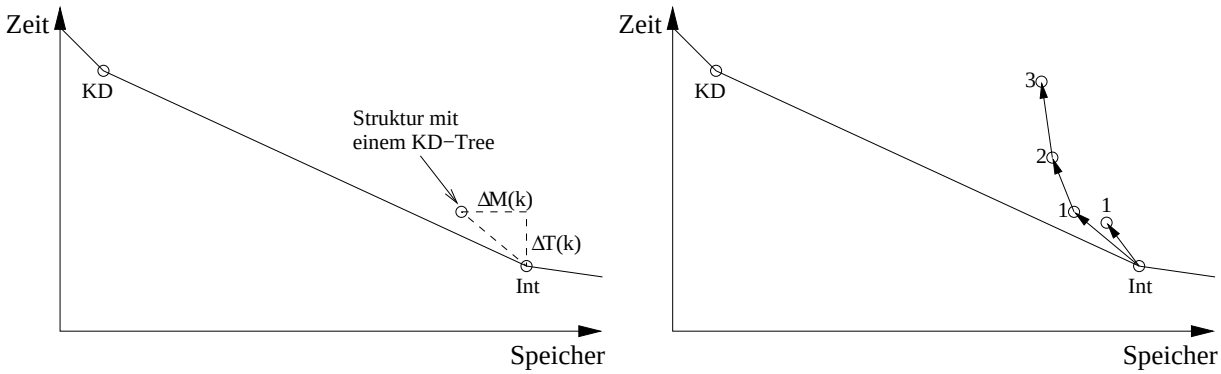


Abbildung 7.8: Die Lücke im Diagramm wird durch weitere Datenstrukturen gefüllt

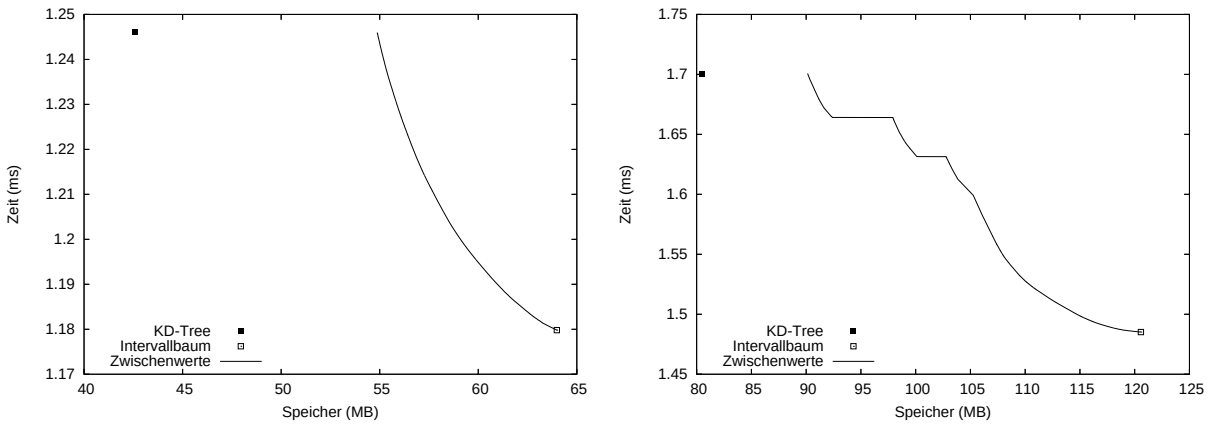


Abbildung 7.9: Die von der Gap-Filling-Methode erzeugten Zwischenwerte beim Engine-Datensatz (links) und beim Bighead-Datensatz (rechts)

dazu führen würde, dass die jeweils betrachtete Lücke im Speicher-Zeit-Diagramm verlassen wird. Ebenso sind für die linke Lücke alle Nachfolger des bereits umgewandelten Knotens ungeeignet. Seine Vorgänger können weiterhin verwendet werden, dafür ist jedoch eine Anpassung der Werte $\Delta M(.)$ und $\Delta T(.)$ und der daraus resultierenden Kostenwerte $C_{IK}(.)$ nötig, da ein KD-Tree in einem Vorgänger des Knotens dazu führt, dass der KD-Tree im Knoten selbst wieder entfernt wird.

Im rechten Teil von Abbildung 7.8 wird gezeigt, wie ausgehend vom Intervallbaum durch Umwandlung von Teilblöcken in KD-Tree-Blöcke weitere Lösungen des Optimierungsproblems gefunden werden. Die Zahlen an den einzelnen Punkten bezeichnen die Anzahl der in der zugehörigen Datenstruktur enthaltenen KD-Trees.

Die Gap-Filling-Methode erzeugt sowohl beim Engine- als auch beim Bighead-Datensatz einige durch die Lagrange-Optimierung nicht gefundene Werte, die in den Speicher-Zeit-Diagrammen (Abbildung 7.9) dargestellt sind. Diese liegen aber alle in der Nähe des Intervallbaums, da wegen der schlechten Teilbarkeit von KD-Trees nur begrenzt viele Bereiche zu KD-Tree-Bereichen gemacht werden können, ohne dass die mittlere Extraktionszeit die Zeit im großen KD-Tree überschreitet. Dadurch bleibt der linke Teil der Lücke zwischen dem KD-Tree und dem Intervallbaum

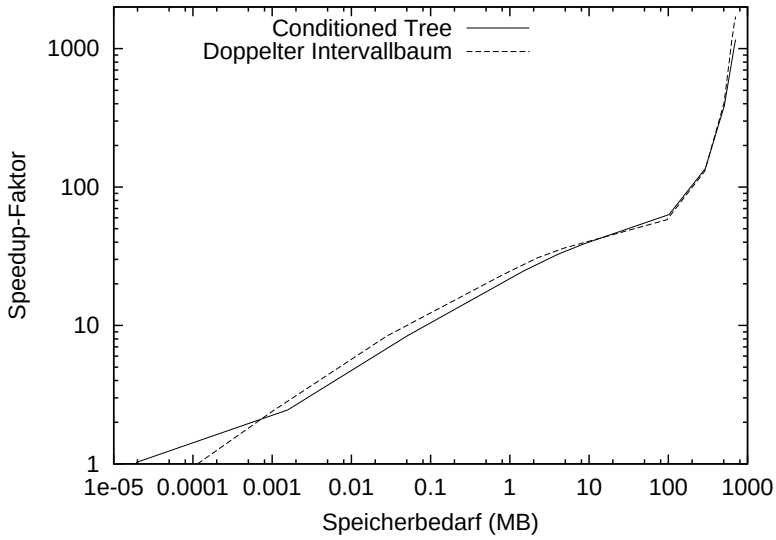


Abbildung 7.10: Das Memory-Speedup-Diagramm der Meta-Intervallbäume für den Datensatz Kummer $400 \times 400 \times 400$, im Vergleich mit dem gleichen Graphen des Conditioned Trees.

unausgefüllt.

7.6 Meta-Intervallbäume

Ein Conditioned Tree mit Intervallbäumen lässt sich auch zu folgenden Datenstrukturen umorganisieren:

- Alle im Conditioned Tree enthaltenen Intervallbäume werden zu einem Intervallbaum vereint, der alle intervallbaum-verwalteten Zellen enthält. Die Idee dieser Methode ist es, einen Baum mit weniger Knoten zu konstruieren, der nur ein Mal durchsucht werden muss.
- Im nächsten Schritt wird der adaptive Teil des Conditioned Trees durch einen weiteren Intervallbaum über den von ihm verwalteten Teilblöcken ersetzt. Die Grundidee davon ist, dass der Intervallbaum schneller sein soll als der bisher verwendete adaptive Baum.

Leider geht die Rechnung nicht auf: Es kostet nämlich mehr Rechenzeit, einen großen Intervallbaum zu durchsuchen als einen Teil der kleinen Bäume. Der im zweiten Schritt zusätzlich eingesetzte Intervallbaum ist zwar schneller, aber auch wesentlich größer als der adaptive Baum, in dem die Suchzeit sowieso schon minimal ist und daher den zusätzlichen Speicherbedarf nicht rechtfertigt. Ein direkter Vergleich der Methoden im Memory-Speedup-Diagramm (Abbildung 7.10) bestätigt dieses Ergebnis und zeigt, dass der Graph des doppelt verschachtelten Intervallbaums den des Conditioned Trees zwar in einem größeren Bereich leicht überschreitet, im entscheidenden Bereich ab ca. 10 MB (zusätzlich zu den vorläufig unabänderlichen 122 MB für den $400 \times 400 \times 400$-Kummer-Datensatz) jedoch schlechter ist.

Kapitel 8

Qualitätstest und Rendering-Programme

8.1 Qualitätstest: Vergleich des Speicher- und Zeitbedarfs

Die beschriebenen Extraktionsmethoden werden für eine Liste von Isowerten getestet, die den Wertebereich in gleichen Abständen durchlaufen, um eine Gleichverteilung zu simulieren. Für jeden dieser Werte wird die Suchzeit der Isoflächen-Extraktion gemessen, und deren Durchschnitt als Bewertungsgröße für das Speicher-Zeit-Diagramm ermittelt. Im gleichen Programm werden der Speicherbedarf für die Datenstrukturen und eventuell auch die Vorverarbeitungszeit gemessen.

Es ist auch die Möglichkeit gegeben, Extraktionsprogramme auf Korrektheit zu überprüfen. Zu diesem Zweck kann man - je nach geforderter Zuverlässigkeit - eine nach der Brute-Force-Methode gebildete Liste verwenden und die Ergebnisliste damit vergleichen, ein Bild erzeugen das die Ergebnisfläche darstellt oder einfach die Anzahl der extrahierten Zellen ausgeben und mit dem durch andere Methoden bestimmten wahren Wert vergleichen. Die Darstellung des Bildes ist vor allem auch für die Veranschaulichung der Resultate vorteilhaft.

Speziell für die Conditioned-Tree-Methode liegt ein Rahmenprogramm vor, das die optimalen Bäume für verschiedene Lagrange-Werte λ bestimmt, abspeichert und ihren Speicherbedarf untersucht. Größere Lücken in der Folge der Speicherbedarfs-Werte werden dann geschlossen, indem weitere Zwischenwerte für λ eingesetzt werden. Es wird dadurch eine steigende Folge von Parametern $\lambda_{\min} = \lambda_1 < \lambda_2 < ... < \lambda_n = \lambda_{\max}$ angestrebt, so dass für die Speicherbedarfs-Werte zweier aufeinander folgender Glieder die Bedingung $\frac{M(\lambda_{i+1})}{M(\lambda_i)} < 1 + \delta$ erfüllt ist, wobei $\delta > 0$ vom Benutzer vorgegeben wird. Wenn für einen Datensatz die Tabelle mit der Abhängigkeit Parameter-Speicherbedarf-Baum einmal bestimmt ist, kann der Benutzer für jeden beliebigen Hauptspeicher den passenden Baum einlesen, der diesen Speicher fast optimal zugunsten der Rechenzeit ausschöpft. Eine schnellere Möglichkeit des Preprocessings, die aber keine Daten für das Speicher-Zeit-Diagramm liefert, ist die schon am Ende des Kapitels über Conditioned Trees erwähnte binäre Suche nach einem vorgegebenen Speicherwert.

8.2 Ein interaktives Isoflächen-Darstellungsprogramm

Zur Darstellung der Isoflächen habe ich ein Programm geschrieben, das es ermöglicht, diese interaktiv zu beeinflussen. Dadurch kann die jeweils ausgegebene Isofläche gedreht und von allen

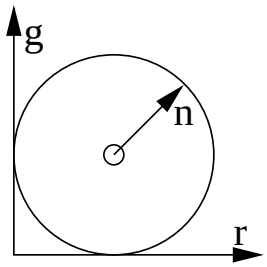


Abbildung 8.1: Die Abhängigkeit der Farbe (r, g, b) einer Zelle vom Normalenvektor n

Seiten betrachtet, sowie auch der Isowert geändert werden. Sämtliche Bilder von Isoflächen in dieser Arbeit wurden von dem hier beschriebenen Programm oder von visuell identischen Versionen des Programms generiert.

Das Programm basiert auf den graphischen und interaktiven Hilfsfunktionen von X-Windows und OpenGL. In diesem Abschnitt werde ich das Programm und seine Möglichkeiten schrittweise beschreiben.

8.2.1 Der interaktive Teil des Programms

Wenn das Programm mit allen nötigen Parametern aufgerufen worden ist, wird ein OpenGL-Fenster geöffnet, in dem die gewünschte Isofläche dargestellt wird. Diese kann nun mit der linken Maustaste interaktiv gedreht, sowie ihr Isowert mit der mittleren Maustaste vergrößert und verkleinert werden. Ein Druck auf die rechte Maustaste beendet das Programm. Natürlich wird die Isofläche nur dann berechnet, wenn sich der Isowert wirklich ändert. Die Zellen, aus denen die Fläche besteht, werden jeweils in einer Liste abgelegt, die deren Positionen (x, y, z) und deren Farben (r, g, b) enthält. Bei Änderungen der Perspektive oder der Fenstergröße wird diese Liste nicht verändert, sondern einfach nur gelesen.

Eine Änderung der Fenstergröße führt dazu, dass die Größe der Isofläche automatisch angepasst wird.

Es besteht auch die Möglichkeit, per Tastendruck zwischen schneller und langsamer interaktiver Änderung der Isowerts zu wählen sowie auch ein paar methodenabhängige Sonderfunktionen.

8.2.2 Die Darstellung der Isofläche in der gegebenen Perspektive

Um eine Isofläche in einer gegebenen Perspektive schnell darzustellen, habe ich für eine einzelne Zelle die einfache Primitive GL_POINT gewählt. Die Größe eines solchen Punkts hängt von der eingestellten Fenstergröße ab; für Fenstergrößen oberhalb der Auflösung erscheint eine Zelle also als kleines Quadrat. Dreiecksnetze sind für die Darstellung ebenfalls möglich, nehmen aber mehr Rechenzeit für einen bei großen Datensätzen nur geringen visuellen Unterschied in Kauf. Die Farbe dieses Quadrats hängt wie in Abbildung 8.1 von der Ausrichtung der Isofläche an dieser Stelle ab, die durch den Normalenvektor n bestimmt wird. Dieser lässt sich aus den Volumen-Datenwerten an den Eckpunkten der Zelle approximieren. Diese nennen wir in den folgenden Formeln b_{xyz} mit $x, y, z \in \{0, 1\}$:

$$n_x = \sum_{y,z\in\{0,1\}} (b_{1yz} - b_{0yz}) \qquad n_y = \sum_{x,z\in\{0,1\}} (b_{x1z} - b_{x0z}) \qquad n_z = \sum_{x,y\in\{0,1\}} (b_{xy1} - b_{xy0})$$

Aus dem auf die Länge 1 normierten Vektor $n' := \frac{n}{|n|}$ erhalten wir die Farbe (r, g, b) durch folgende Formeln:

$$r = \frac{n'_x + 1}{2} \qquad g = \frac{n'_y + 1}{2} \qquad b = \frac{n'_z + 1}{2}$$

Das Bemerkenswerte an dieser Formel ist, dass parallele Flächenstücke mit entgegengesetzter Ausrichtung in Komplementärfarben erscheinen.

Um ein beleuchtetes Grauwertbild aus dem so bestimmten Farbbild zu erhalten, kann einfach eine geeignete Linearkombination der Ebenen r, g und b gebildet werden.

8.3 Das Steuerungsprogramm Isosurf.tcl

Um dem Benutzer der Extraktionsprogramme die Steuerung zu erleichtern, ist dem Programmpaket die Bedienungsoberfläche Isosurf.tcl (englisch) bzw. Isosurf_DEU.tcl (deutsch) beigefügt. Diese ermöglicht es, durch einfache Mausbedienung das Extraktionsprogramm und die Optimierung mit passenden Parametern zu starten.

Zum Start des Extraktionsprogramms ist in der linken Hälfte der Bedienungsoberfläche der gewünschte Datensatz sowie der Lagrange-Faktor oder Speicherbedarf zu wählen und dann der Startknopf zu drücken.

Für eine Optimierung sind die Bedienelemente in der rechten Hälfte zuständig: zunächst ist die Funktion zu wählen und die gewünschte Dateigröße einzustellen. Dann wird durch den Knopf 'Optimierung' ein Prozess gestartet, der den eingestellten Datensatz erzeugt und für eine Folge von Lagrange-Werten die Optimierung durchführt. Abbildung 8.2 zeigt die Bedienungsoberfläche des Programms.

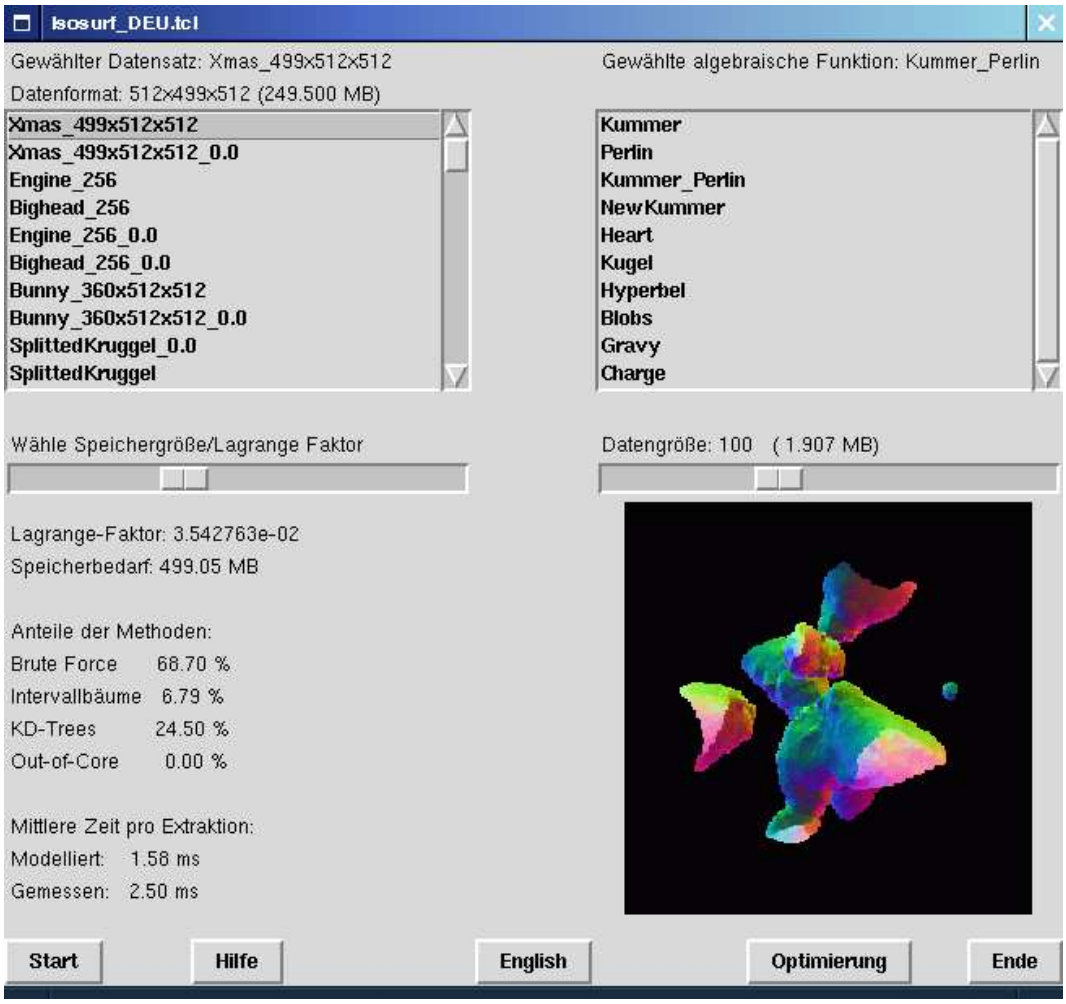


Abbildung 8.2: Die Bedienungsoberfläche der deutschsprachigen Version von Isosurf.tcl

Kapitel 9

Numerische Ergebnisse und Bilder

9.1 Ergebnisse

Dieser Abschnitt enthält Diagramme der Ergebnisse der in dieser Arbeit aufgeführten Methoden. Sämtliche Resultate sind zum Zweck des Vergleichs mit einem Rechner mit dem Prozessor AMD Athlon(TM) XP 1600+ (1410 MHz) bestimmt worden.

Abbildung 9.1 ist eine Übersicht der untersuchten Methoden in einem Memory-Speedup-Diagramm, angewandt auf den $499 \times 512 \times 512$-Xmas-Datensatz. Darin wird die Conditioned-Tree-Methode einmal mit uneingeschränkt zugelassener Out-of-Core-Methode ($\mathcal{H} = 0$) und einmal ganz ohne diese Methode ($\mathcal{H} = 1$) gezeigt. Hier sind die Einzelmethoden aus dieser Messung tabellarisch aufgelistet:

Methode	Suchbäume (MB)	Gesamter Speicher-bedarf (MB)	Suchzeit pro Extraktion (ms)	Speedup-Faktor
Brute Force	0	250	5987	1
Octree	109	359	87.5	68.4
KD-Tree	837	1087	3.92	1530
Intervallbaum	1256	1506	2.00	3000
Out-of-Core	0	250	17.14	350

Abbildung 9.2 zeigt eine Übersicht über die Speedup-Ergebnisse für den $384 \times 400 \times 276$-ScannedBrain-Datensatz. Wie in dieser Grafik zu sehen ist, ist die Intervallbaum-Methode für den ganzen Datensatz nicht in den Ergebnissen des Optimierungsverfahrens enthalten. Das liegt daran, dass wegen der großen Intervalle des Datensatzes lange Min- und Maxlisten untersucht werden müssen, was den größten Teil des Zeitbedarfs der Intervallbaum-Suche ausmacht. Algebraische Flächen ermöglichen einen besseren Speedup als gemessene Datensätze, weil sie glatt verlaufen, kleine Intervalle erzeugen und dadurch für das beinahe output-sensitive Intervallbaum-Verfahren kurze Intervalllisten ergeben.

Abbildung 9.3 zeigt die entsprechenden Ergebnisse für den $360 \times 512 \times 512$-Bunny-Datensatz, der durch CT-Scan des Terracottahasen entstanden ist. Das Flächenmodell entstand durch einen Laserscan dieses Objekts, es handelt sich also hier um die Daten aus einer anderen Messung.

Abbildung 9.4 zeigt die Memory-Speedup-Diagramme zweier sehr großer Datensätze (500/644

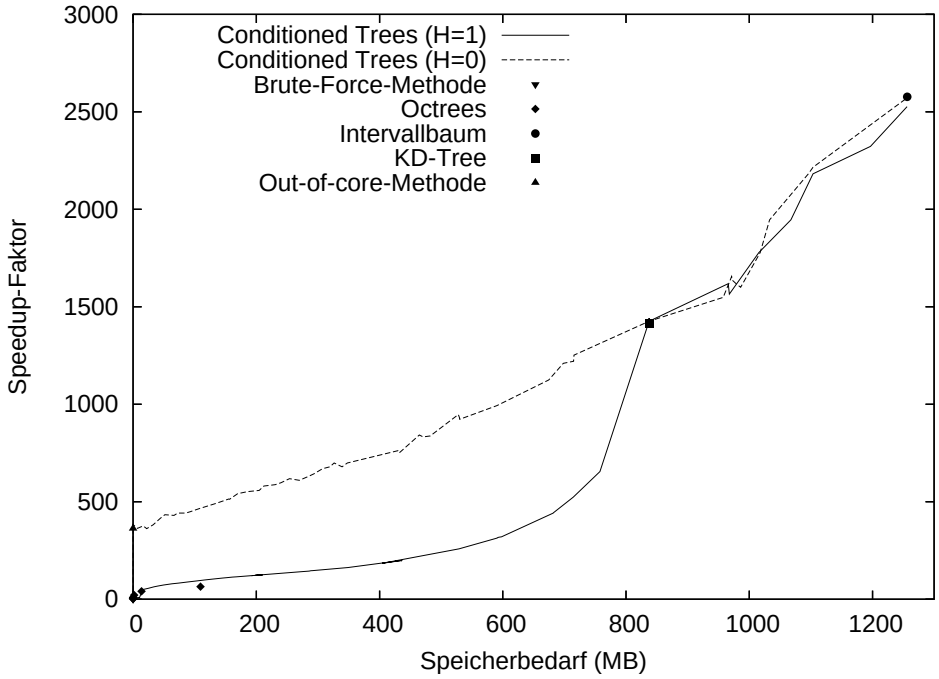


Abbildung 9.1: Das Memory-Speedup-Diagramm für die Isoflächen-Extraktion des $499 \times 512 \times 512$-Xmas-Datensatzes.

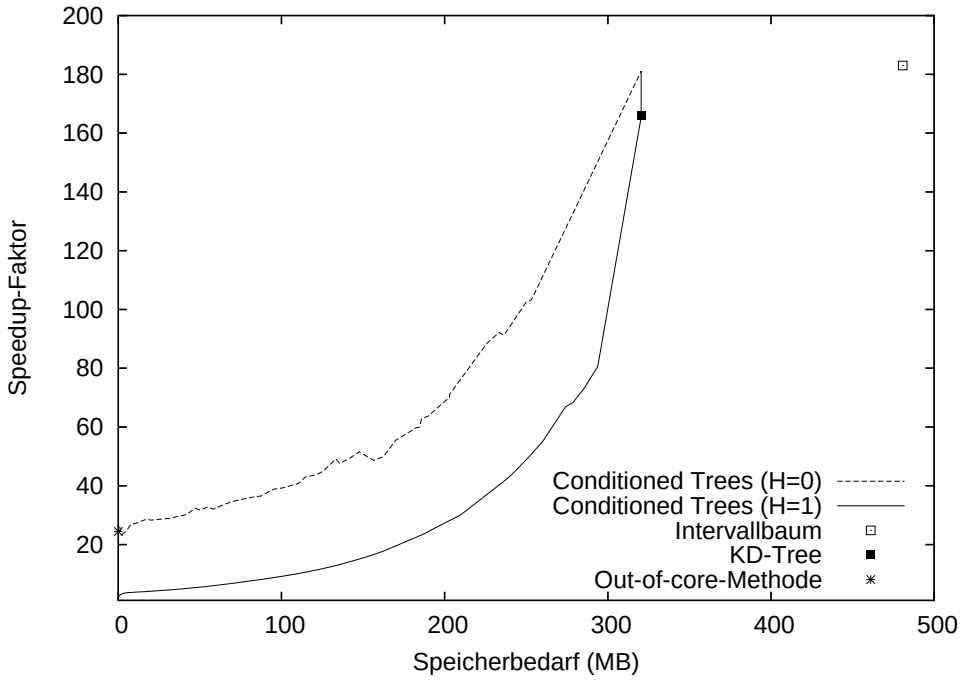


Abbildung 9.2: Das Memory-Speedup-Diagramm für die Isoflächen-Extraktion des $384 \times 400 \times 276$-ScannedBrain-Datensatzes.

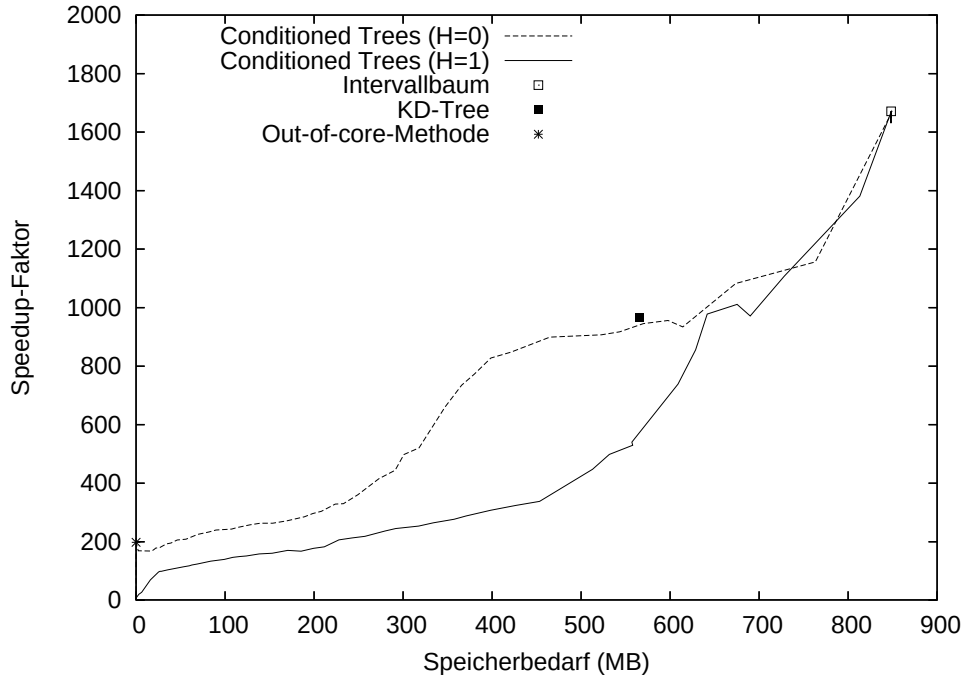


Abbildung 9.3: Das Memory-Speedup-Diagramm aller Methoden für die Isoflächen-Extraktion des $360 \times 512 \times 512$-Terracottahasen.

MB), die nicht mehr zusammen mit ihren jeweiligen Intervallbäumen in den Hauptspeicher passen. In diesem Fall ist eine Rechenzeit-Optimierung dennoch möglich, weil dafür nur die Grundgerüste der Datenstrukturen gebraucht werden. Der Speedup-Faktor wird in diesen Diagrammen aus den modellierten Rechenzeiten bestimmt, weil die wirklichen Rechenzeiten wegen Speichermangel nicht mehr gemessen werden können.

Abbildung 9.5 ist eine Zusammenstellung der Histogramme, die die für die Extraktionszeit maßgeblichen Verteilung der Intervallgrenzen darstellen. In diesen Bildern sind diese Grenzen nach steigender Häufigkeit geordnet in den Farben weiß (keine Intervalle), blau, grün, rot und purpur dargestellt.

Die Datensätze ScannedBrain, Bighead und Engine weisen eine kontinuierliche Verteilung auf, in der kurze Intervalle häufig und lange Intervalle seltener vorkommen. Entsprechendes gilt auch für den Xmas-Tree, dessen Wertebereich aber auf das Intervall $[0, 4000]$ beschränkt ist, Dabei kommen Werte von über 2500 extrem selten vor und wurden deshalb im Histogramm abgeschnitten. Der große Xmas-Tree ($999 \times 512 \times 512$) ist der original eingescannte Datensatz, der mit 500 MB mehr Information enthält wie die kleine Version ($499 \times 512 \times 512$), aber im Gegensatz zu dieser nicht maßstabsgetreu ist. Die Histogramme der beiden Versionen sind sich aus verständlichen Gründen sehr ähnlich.

Der Bunny-Datensatz erzeugt ausschließlich sehr kurze und sehr lange Intervalle. Das kommt daher, dass die Volumendaten ausschließlich in der Nähe von zwei Werten zu finden sind, zwischen denen eine scharfe Grenze gebildet wird, nämlich die Oberfläche des Hasen. Im Histogramm ergibt das drei Bereiche, in denen sich alle Intervalle befinden.

In den Abbildungen 9.6 und 9.7 wird die Verteilung der Zellen auf die verwendeten Methoden in

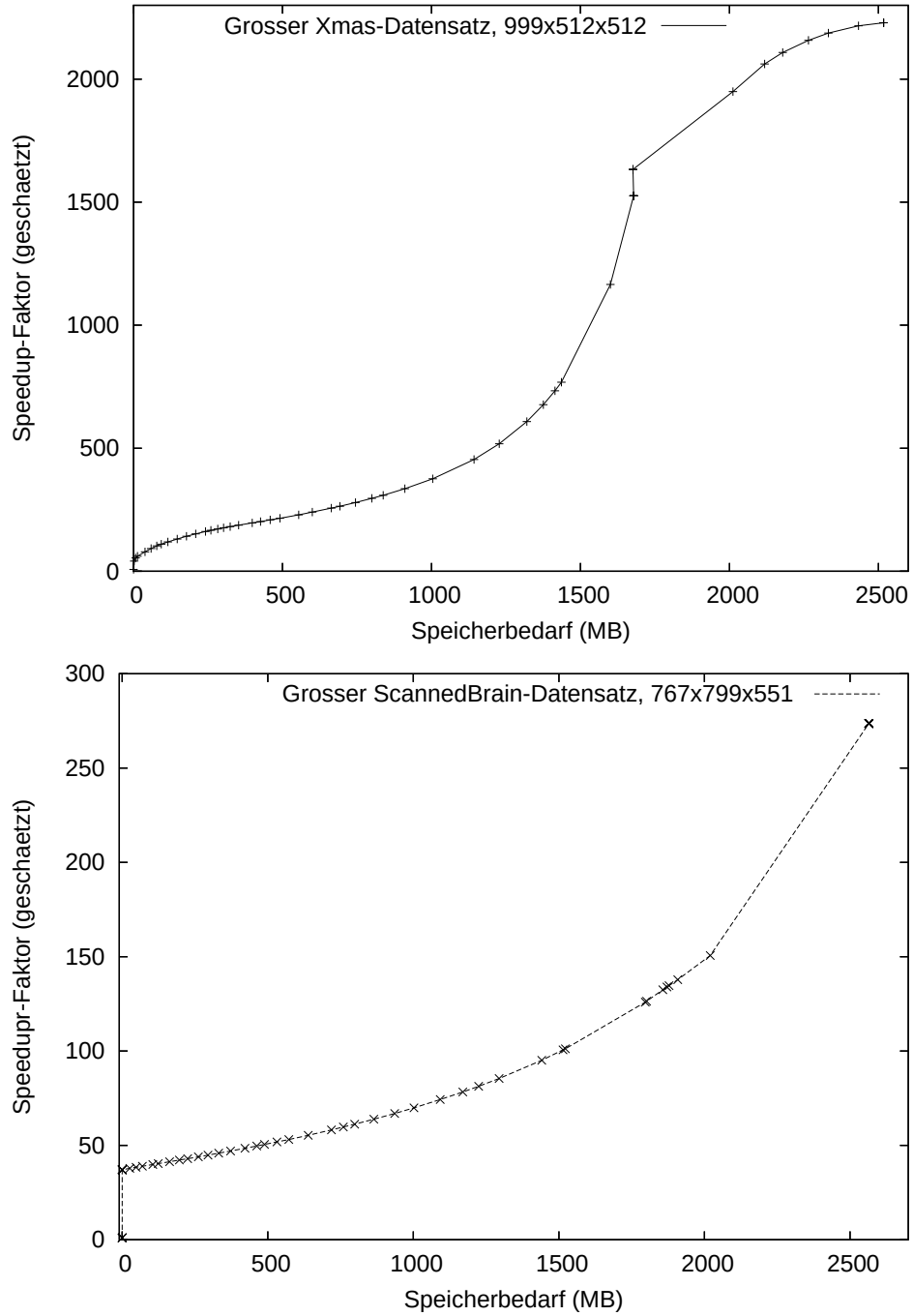


Abbildung 9.4: Das Memory-Speedup-Diagramm für die Isoflächen-Extraktion des großen ($999 \times 512 \times 512$) Xmas-Tree-Datensatzes sowie eines auf $767 \times 799 \times 551$ erweiterten ScannedBrain-Datensatzes. Der Speedup-Faktor wurde hier nicht mit den gemessenen, sondern mit den modellierten Zeitwerten bestimmt.

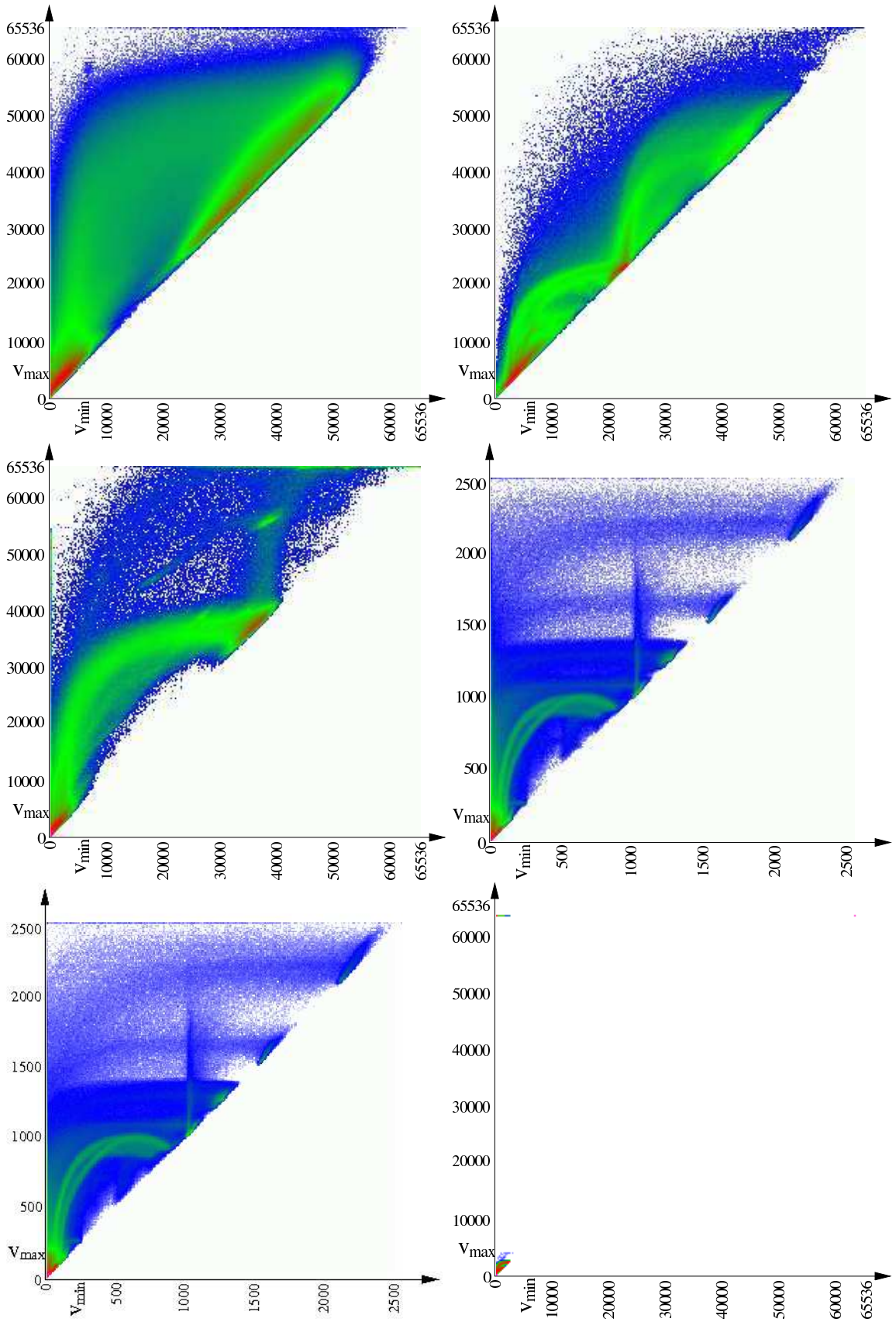


Abbildung 9.5: Die Histogramme der Datensätze in der gegebenen Reihenfolge: ScannedBrain, Bighead, Engine, Xmas, Xmas groß, Bunny. Die Farben geben die Häufigkeit der gegebenen Intervallgrenzen $[v_{\min}, v_{\max}$ an, bei steigender Häufigkeit wird dabei ein kontinuierlicher Verlauf durch folgende Farben angenommen: weiß, blau, grün, rot, magenta (letztere ist extrem selten).

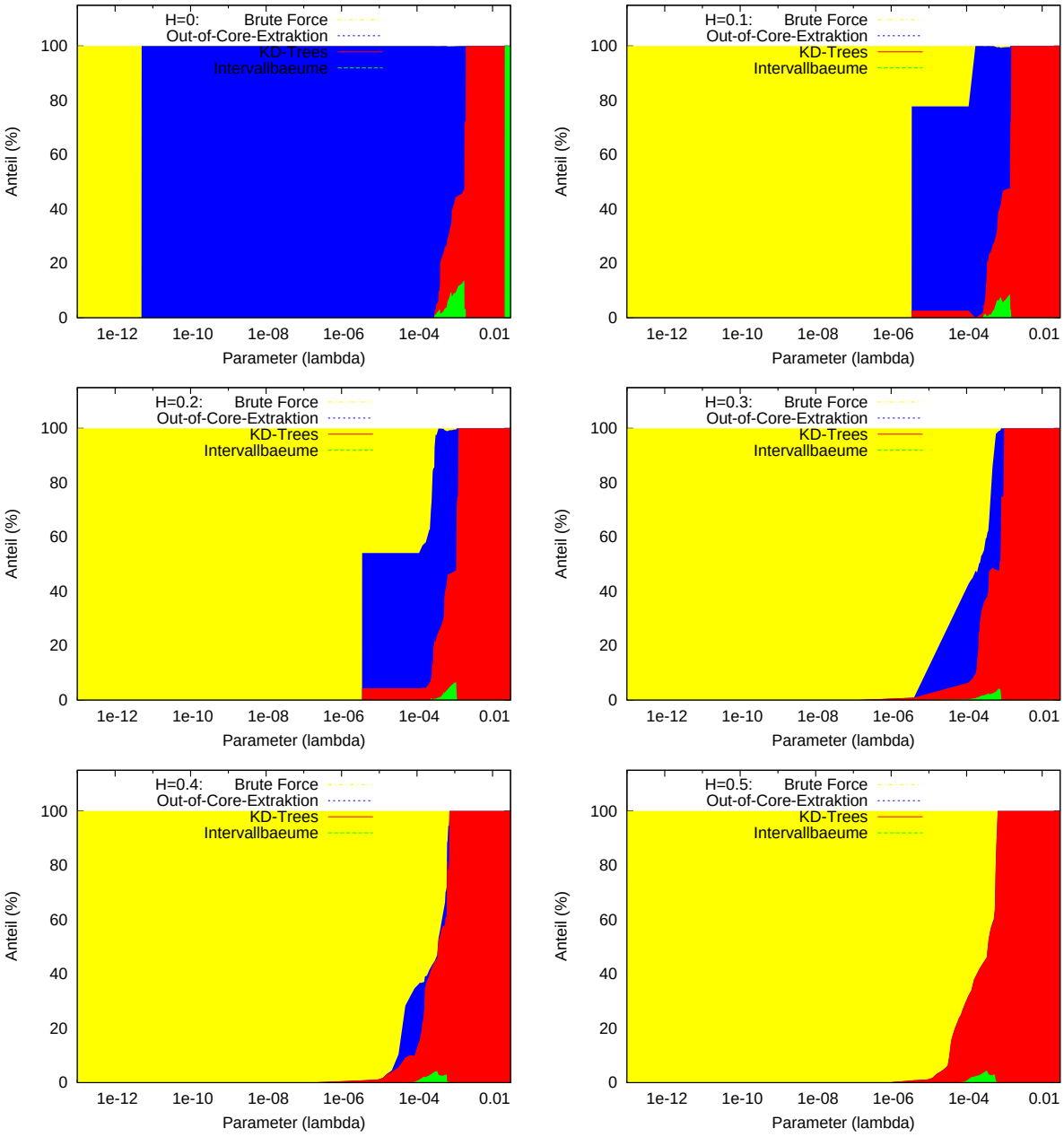


Abbildung 9.6: Die Verteilung des Datensatzes auf die vorhandenen Methoden, in Abhängigkeit vom Lagrange-Faktor λ und von der Festplattenkonstante $\mathcal{H}$. Die Graphen wurden für den Engine-Datensatz und für $\mathcal{H} = 0, 0.1, 0.2, 0.3, 0.4$ und 0.5 erstellt. Wenn $\mathcal{H}$ den Wert 0.5 überschreitet, ändert sich am Verhalten der Verteilung nichts mehr.

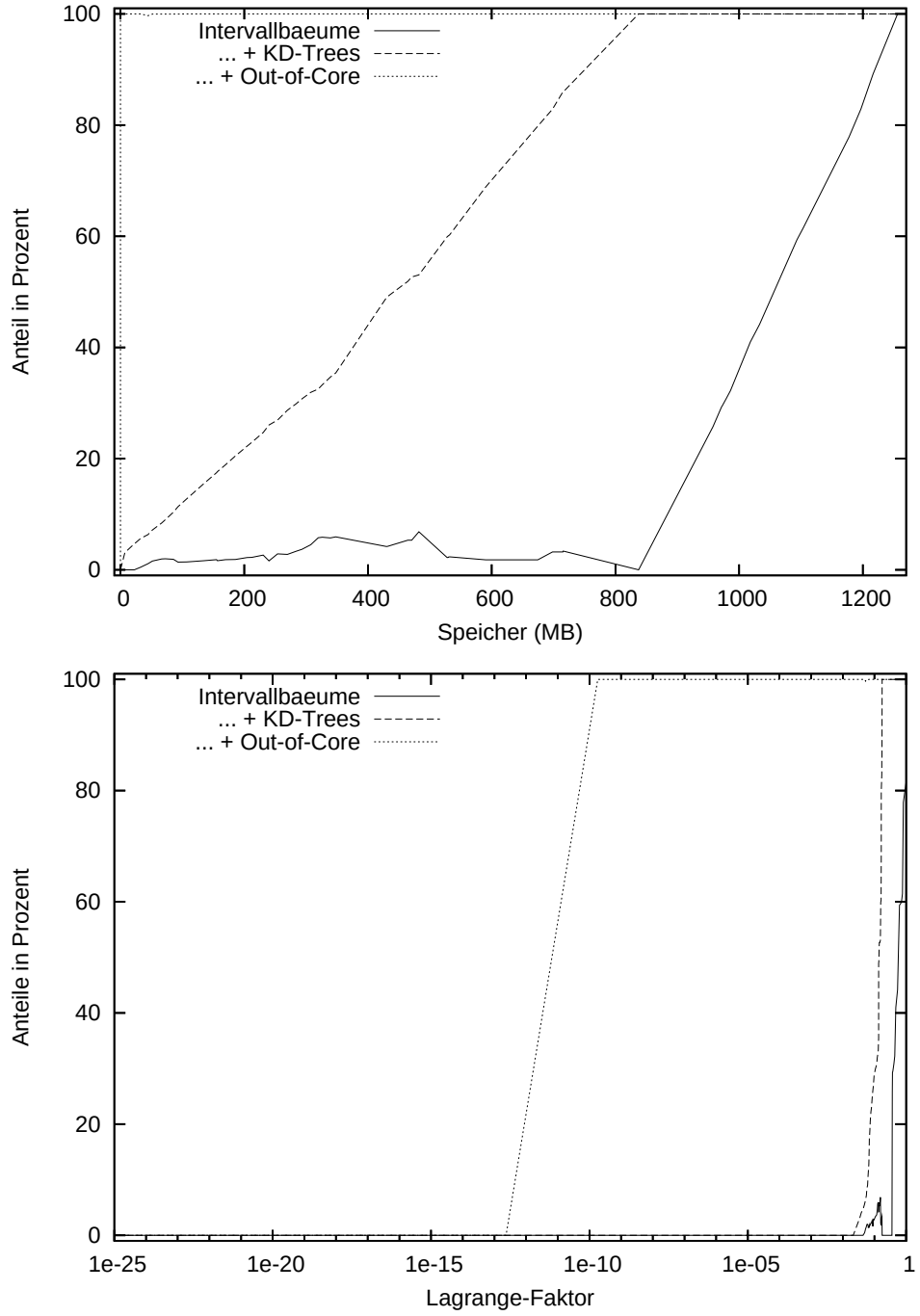


Abbildung 9.7: Die Verteilung der Volumendaten auf die einzelnen Methoden für den Xmas-Datensatz, links abhängig vom belegten Speicher, rechts abhängig vom Lagrange-Faktor λ.

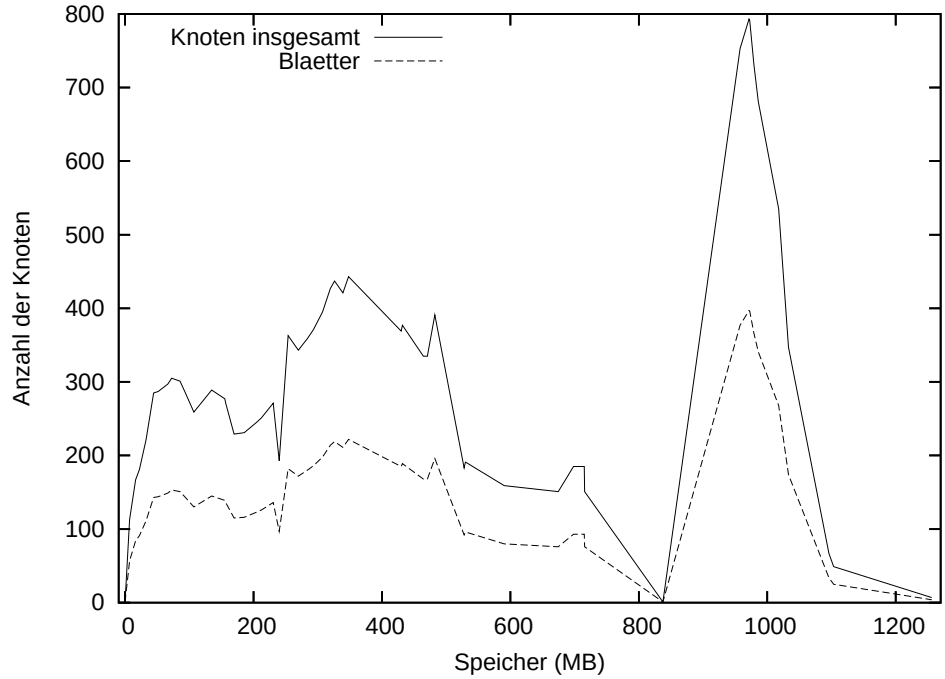


Abbildung 9.8: Die Anzahl der Knoten des Conditioned Trees in Abhängigkeit vom Speicher für den Xmas-Datensatz.

Abhängigkeit vom Festplatten-Parameter $\mathcal{H}$ gezeigt. Für $\mathcal{H} = 0$ wird bei steigendem λ zunächst die Brute-Force-Suche verwendet, dann die Out-of-Core-Extraktion, darüber nach einer kleinen Zwischenstufe, die aus allen Methoden besteht, die KD-Tree-Extraktion und schließlich, wenn dafür genügend Speicher zur Verfügung steht, ein Intervallbaum mit allen Zellen. Wenn $\mathcal{H}$ steigt, dann wird der Bereich mit Out-of-Core-Extraktion kontinuierlich durch einen Bereich überdeckt, in dem für einen Teil der Daten Brute-Force-Suche und für den anderen Intervallbaum-Suche verwendet wird. Für hinreichend große $\mathcal{H}$ wird schließlich die Situation erreicht, die bereits vor der Einführung der Out-of-Core-Extraktion gegeben war.

Bemerkenswert ist dabei, dass bei steigendem λ oft kein kontinuierlicher, sondern ein abrupter Übergang stattfindet. Zum Beispiel bei Übergang vom KD-Tree zum Intervallbaum würde man auf den ersten Blick erwarten, dass es Zwischenwerte gibt, für die der optimale Conditioned Tree zum Teil auf KD-Trees und zum anderen Teil auf Intervallbäume verweist, um den gegebenen Speicher angemessen auszunutzen. Das ist aber nicht immer der Fall ! Die Erklärung dafür ist, dass ein KD-Tree bzw. ein Intervallbaum für den halben Datensatz (oder einen anderen Bruchteil) nicht nur die Hälfte des Kostenaufwandes benötigt. Ein Baum wird nämlich durch eine Halbierung seiner Daten nur um eine Ebene reduziert. Daraus folgt, dass der Suchaufwand innerhalb eines KD-Trees (ohne die für die Ausgabe benötigte Zeit) nur geringfügig von seiner Größe abhängt. Wenn also nur die Hälfte eines Datensatzes mit einem KD-Tree durchsucht wird, dann fällt dafür fast genau so viel zusätzliche Suchzeit gegenüber dem Intervallbaum an, wie wenn der ganze Datensatz mit einem KD-Tree durchsucht worden wäre. Es lohnt sich also nicht unbedingt, nur Teile der Volumendaten der KD-Tree-Methode zuzuweisen.

Abbildung 9.8 zeigt die Anzahl der Knoten des Conditioned Trees in Abhängigkeit vom dafür zur Verfügung stehenden Speicher für den Xmas-Tree. Darin sieht man, wie für steigenden Speicher

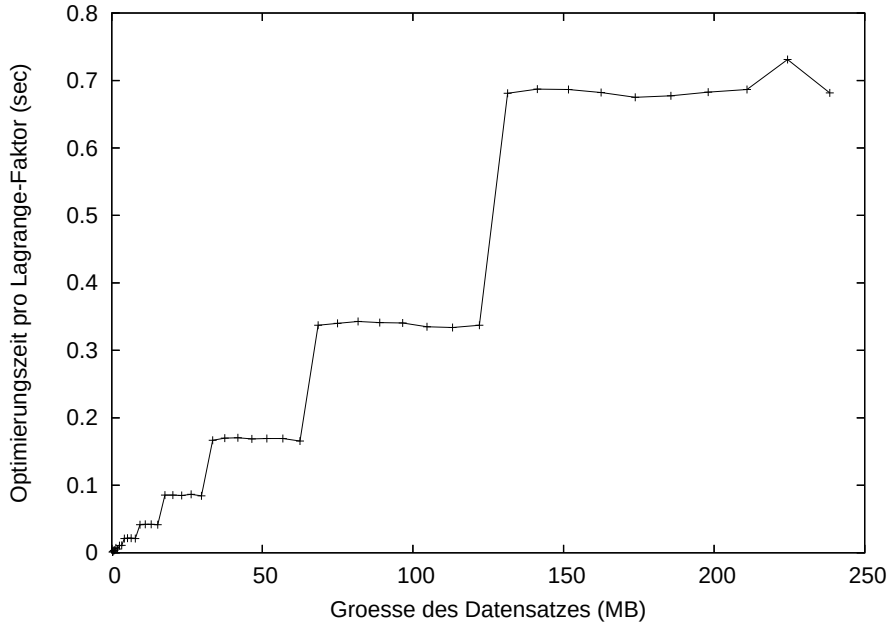


Abbildung 9.9: Zeitbedarf der Optimierung pro Lagrange-Wert λ für eine Serie von Datensätzen.

zunächst ein hierarchischer Zerlegungsbaum wächst und dann wieder auf einen Knoten zusammenschrumpft, der auf eine einzige große Datenstruktur, den Intervallbaum, verweist. Bei einem Wert von etwas mehr als 800 MB wird ebenfalls nur ein Knoten gebraucht, der auf einen KD-Tree zeigt. KD-Trees sind durch Optimierung nur schwer in Einzelteile zerlegbar, da die Suchzeit für eine Struktur aus mehreren kleinen KD-Trees wesentlich höher ist als für einen einzigen großen KD-Tree.

Abbildung 9.9 stellt die Zeit dar, die für die Ermittlung eines optimalen Conditioned Trees benötigt wird, abhängig von der Größe des Datensatzes in Megabyte. Für diese Messung wurde eine Serie von Kummer-Datensätzen mit Perlin-Rauschen verwendet (Beschreibung siehe Anhang), die trotz ihrer verschiedenen Größen statistisch miteinander vergleichbar sind.

In diesem Diagramm lässt sich ein stufenförmiges Muster erkennen. Das kommt daher, dass die Optimierungszeit bei bereits berechneten Kostenwerten nicht mehr von der Größe des Datensatzes selbst abhängt, sondern nur noch von der Anzahl der verglichenen Kostenwerte. Diese wiederum bilden einen analog zum maximalen verwendeten Zerlegungsbaum gebildeten Binärbaum, dessen Gesamtgröße sich bei einer Vergrößerung der Volumendaten nicht kontinuierlich ändert, sondern an bestimmten Stellen einen Sprung auf die doppelte Größe macht. Die Tiefe des maximalen Zerlegungsbaums ist dabei so gewählt, daß die Anzahl der Zellen auf der untersten Ebene zwischen 64 und 128 liegt. Dadurch befinden sich die Stufen im Diagramm gerade an den Stellen, an denen die Anzahl der Zellen des Datensatzes eine Zweierpotenz ist.

In Abbildung 9.10 wird die Zeit für die Kostenberechnung dargestellt, zerlegt in die Einzelmethoden, die für die Optimierung verwendet werden.

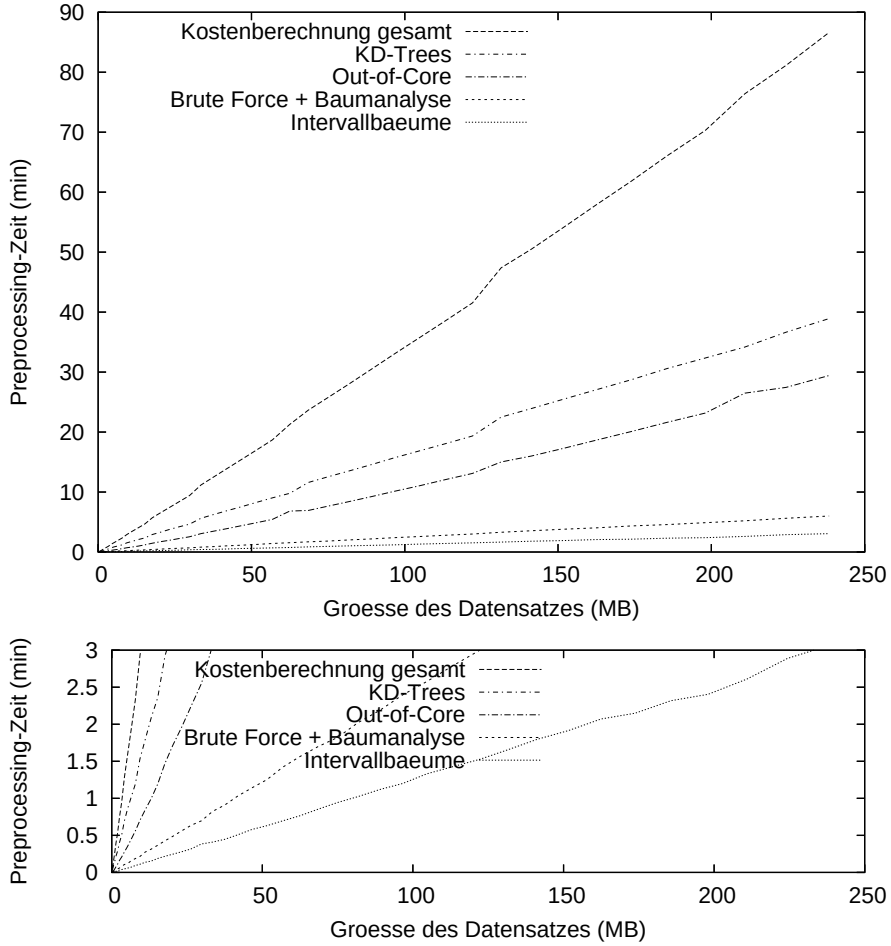


Abbildung 9.10: Die Preprocessing-Zeit für die Kostenberechnung abhängig von der Größe des Datensatzes, aufgeteilt in die Zeiten für die Kostenberechnung der einzelnen Methoden.

9.2 Bilder

Abbildung 9.11 zeigt eine Serie von Isoflächen mit steigendem Isowert für den Engine-Datensatz. Weitere Bilder von Isoflächen sind im Anhang A dargestellt.

Auf Abbildung 9.12 wird an einem Beispiel gezeigt, wie die Teile eines gegebenen Datensatzes durch die Optimierung verschiedenen Methoden zugeordnet werden können. In dieser Abbildung werden, falls ein farbiges Exemplar der Arbeit vorliegt, Brute-Force-Blöcke grün, Intervallbaum-Blöcke blau und Out-of-Core-Blöcke gelb dargestellt. An den verschiedenen Farbtönen kann man außerdem die Zerlegung des Partitionsbaums erkennen.

Abbildung 9.13 stellt jeweils eine Isofläche des kleinen ($499 \times 512 \times 512$) und großen ($999 \times 512 \times 512$) Xmas-Datensatzes dar. Die große Version erscheint verzerrt, weil der Abstand der eingescannten Scheiben kleiner als der Voxelabstand ist. Im kleinen Datensatz wurde dieser Fehler rechnerisch korrigiert. Abbildung 9.14 zeigt eine Isofläche des $360 \times 512 \times 512$-Terracottahasen.

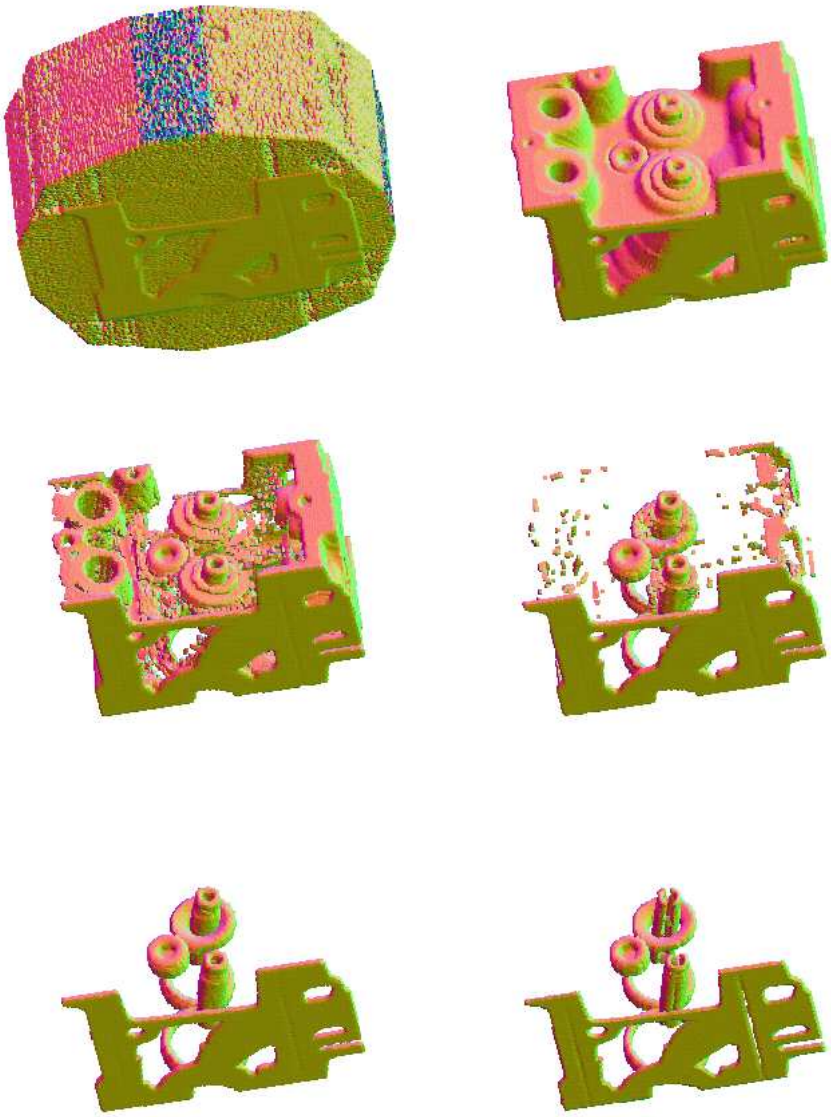


Abbildung 9.11: Isoflächen des Engine-Datensatzes zu den Isowerten 0, 10000, 37000, 40000, 45000 und 60000 in gleicher Perspektive

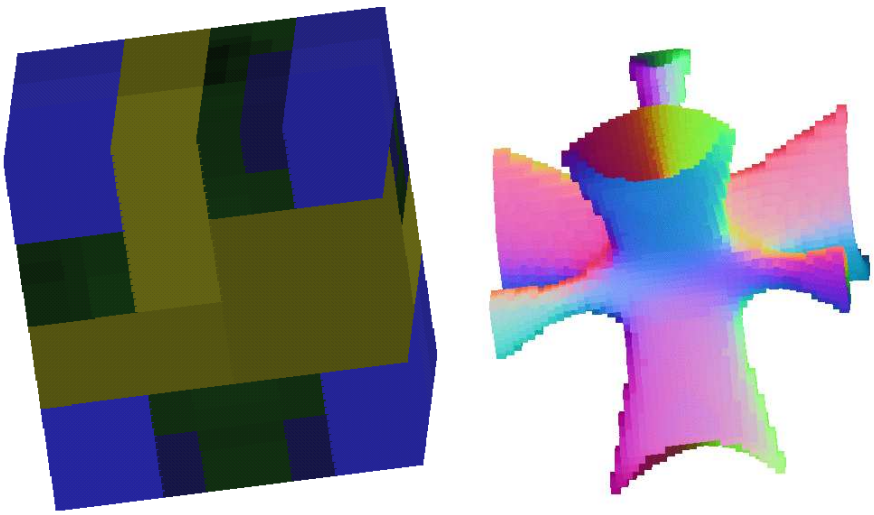


Abbildung 9.12: Die Zerlegung und Aufteilung eines Datensatzes in verschiedene Methoden. Rechts ist die zugehörige Isofläche in der gleichen Perspektive wie der Würfel dargestellt.

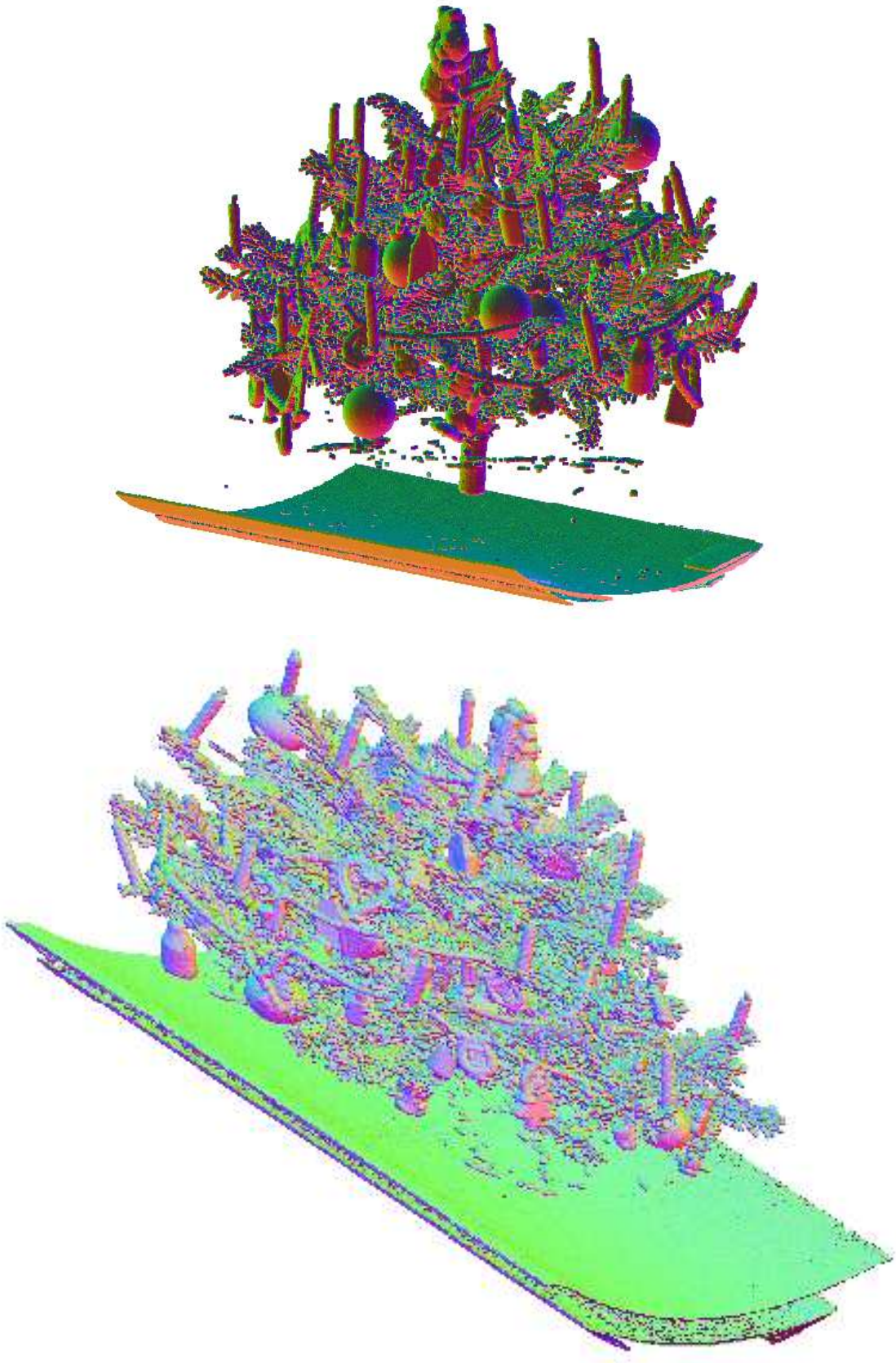


Abbildung 9.13: Eine Isofläche des Xmas-Datensatzes im Format $499 \times 512 \times 512$ (oben) und $999 \times 512 \times 512$ (unten).

Abbildung 9.14: Eine Isoflächen-Darstellung des $360 \times 512 \times 512$-Terracottahasen.

Kapitel 10

Schlussfolgerung

Wir haben viele Methoden der Isoflächen-Extraktion für verschiedene Anforderungen an den Speicher, die Geschwindigkeit und die Genauigkeit entwickelt.

Natürlich ist die trivial-optimale Methode die schnellste, benötigt aber so viel Speicher, dass sie meistens nicht realisierbar ist. Das andere Extrem ist die Brute-Force-Methode, die außer für die Volumendaten keinen weiteren Speicher benötigt, aber sehr langsam ist.

Mit Hilfe von wenig zusätzlichem Speicher kann auf den Volumendaten ein Partitionsbaum errichtet werden, der bereits zu großen Suchzeit-Einsparungen führt. Bei Verwendung von noch mehr Speicher kann bereits eine intervall-basierte Methode wie die KD-Tree- oder Intervallbaum-Methode verwendet werden, bei denen die Suchzeit nicht viel größer als die reine Ausgabezeit des Resultats ist.

Die in [10],[11] und Abschnitt 3.1 dieser Arbeit beschriebenen Out-of-Core-Verfahren speichern die Volumendaten und eventuell weitere Datenstrukturen auf der Festplatte oder einem vergleichbaren Datenträger, und lesen deren Teile nur nach Bedarf ein. Sie haben den Vorteil, Hauptspeicher zu sparen, nehmen dafür aber die Benutzung von Festplattenspeicher mit niedrigerer Zugriffsgeschwindigkeit in Kauf.

Die Seed-Set-Methode verwendet die Tatsache, dass die aktiven Zellen nur innerhalb einer kleinen Menge von Repräsentanten gesucht werden müssen, und alle nicht in dieser Menge enthaltenen Zellen durch eine Suche im Nachbarschaftsgraphen des Datensatzes gefunden werden können.

Ferner gibt es die von Shen und Johnson in [28] sowie im Abschnitt 3.3 dieser Arbeit beschriebene Time-Continuation-Methode, die anstatt einer unabhängig identisch verteilten Folge von Isowerten eine Folge nahe beieinander liegender Isowerte voraussetzt und die Isofläche jeweils durch nur wenige Änderungen anpasst.

Der hier neu erforschte Bereich beginnt mit einer näheren Untersuchung von Partitionsbäumen. Diese können durch Optimierungsverfahren bei der Zerlegung verbessert werden. Ein bereits erzeugter Partitionsbaum kann durch Tree-Growing und -Pruning-Algorithmen auf einen kleineren Speicherbedarf zurecht geschnitten werden, wobei heuristische Methoden zur Wahl der passenden Stellen im Baum verwendet werden, die dabei entfernt werden. Das mathematisch nachweisbar optimale Resultat dieser Art liefert ein durch den BFOS-Algorithmus gewonnener Unterbaum.

In dieser Arbeit nehmen wir an, dass für die Isoflächen-Extraktion die Volumendaten eingelesen und die Hilfsstruktur ein Mal aufgebaut wird. Die darauf folgenden angefragten Isowerte sind nach

einer gegebenen Wahrscheinlichkeit unabhängig identisch verteilt und es ist während des Extraktionsvorgangs kein Lesezugriff auf die Festplatte zugelassen. Für die Out-of-Core-Methoden wird von der letzten Forderung abgesehen, obwohl ein Zugriff auf externe Speichermedien keinen allgemeingültigen Vergleich des Speicher- und Zeitbedarfs ermöglicht. Aus diesen Voraussetzungen ergibt sich die Conditioned-Tree-Methode, die für jeden vorgegebenen Speicher eine Datenstruktur mit dem kleinsten noch möglichen Zeitbedarf bestimmt und für die Isoflächen-Extraktion einsetzt.

Der Intervallbaum zu einer gegebene Liste von Intervallen ist nicht eindeutig bestimmt. Hier wird eine Methode untersucht, mit der ein bezüglich Speicher- und Zeitbedarf optimierter Intervallbaum konstruiert werden kann.

Es ist nicht nur möglich, sondern sogar sehr wahrscheinlich, dass in Zukunft weitere Methoden der Isoflächen-Extraktion entwickelt werden oder sogar heute schon existieren. Die Conditioned-Tree-Methode wird jedoch immer den Vorteil haben, dass sie um diese Methoden erweiterbar ist und ihre Qualitäten analysieren und diese, wenn sie lokal oder global eine Verbesserung der bisherigen Methoden darstellen, übernehmen kann.

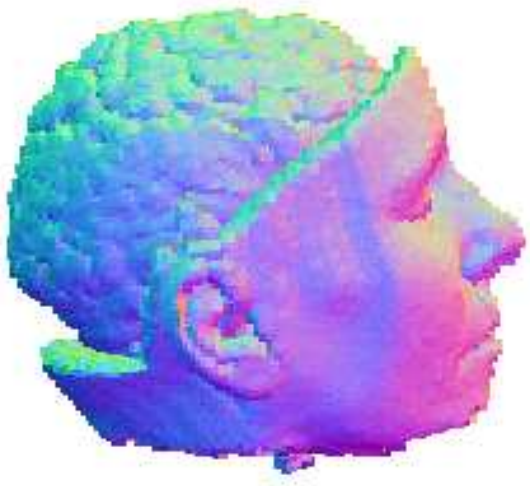
Anhang A

Einige Beispiel-Datensätze

In diesem Anhang werden einige verwendete Datensätze beschrieben. Die gemeinsame Auflösung aller Datensätze beträgt 2 Byte/Voxel oder wurde der Einfachheit halber auf diese Auflösung normiert.

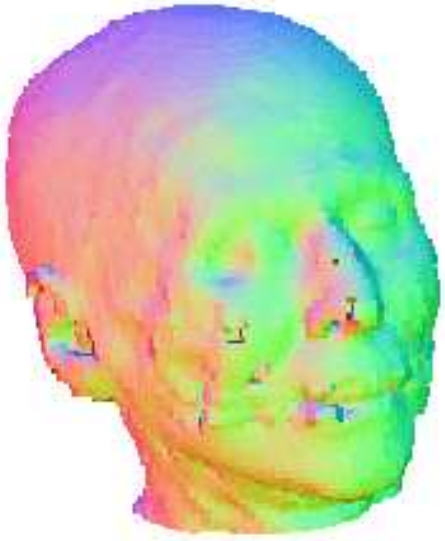
A.1 Medizinische Daten

Brain, Bigbrain, Halfbrain



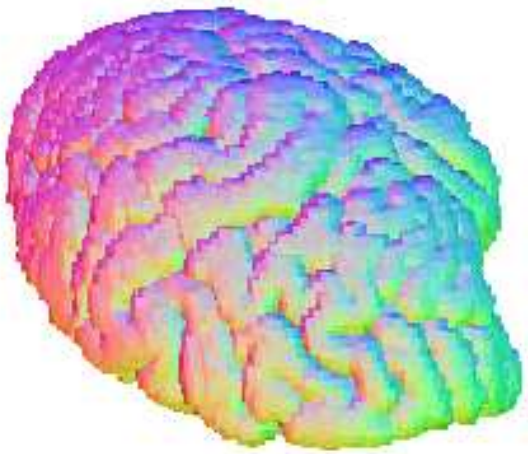
Ursprüngliche Größe: $128 \times 128 \times 84$ (2.6 MB)
MR-Scan
Chapel Hill Volume Rendering Test Dataset, Volume I
Brain wurde auf $64 \times 64 \times 64$ (0.5 MB) verkleinert.
Bigbrain durch Spiegelung auf $128 \times 128 \times 128$ (4 MB) vergrößert.
Halfbrain mit der Größe $128 \times 128 \times 42$ (1.3 MB) entsteht aus dem ursprünglichen Datensatz durch Abschneiden.
file://www-graphics.stanford.edu/pub/volpack/data/brain/

Bighead



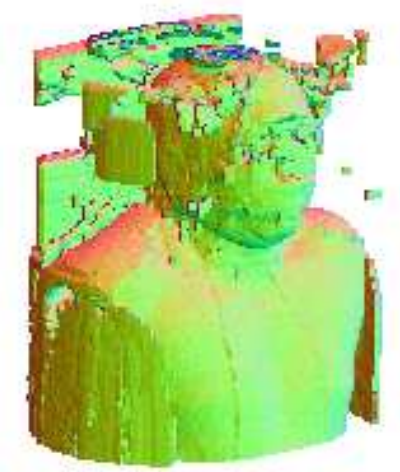
Größe: $256 \times 256 \times 225$ (28 MB)
CT-Scan
Chapel Hill Volume Rendering Test Dataset, Volume II
file://www-graphics.stanford.edu/pub/volpack/data/head/

ScannedBrain



Größe: $384 \times 608 \times 276$ (123 MB)
Messung von Herrn Frithjof Kruggel im Max-Planck-Institut für neuropsychologische Forschung
http://www.cns.mpg.de
Gestreckte Version:
Größe: $767 \times 799 \times 551$ (644 MB)
(Zwischenwerte künstlich eingefügt zur Erzeugung eines übergroßen Datensatzes)

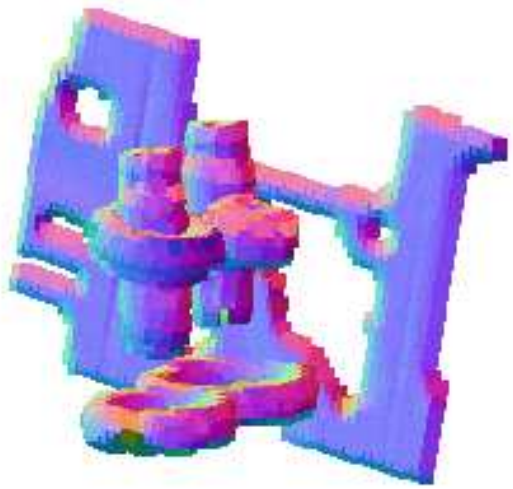
VisiMan



Größe: $512 \times 512 \times 512$ (256 MB)
Datensatz aus dem Visible-Man-Projekt

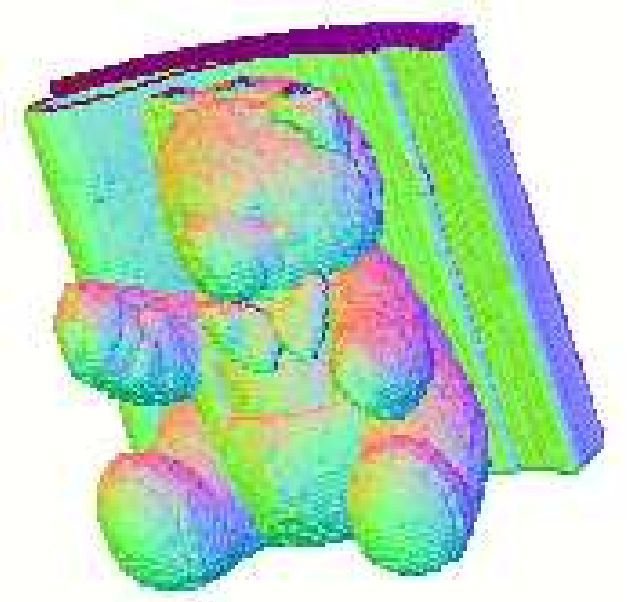
A.2 Sonstige Datensätze

Engine



Größe: $256 \times 256 \times 110$ (13.8 MB)
CT-Scan
Chapel Hill Volume Rendering Test Dataset, Volume II
file://www-graphics.stanford.edu/pub/volpack/data/engine/

Teddybear

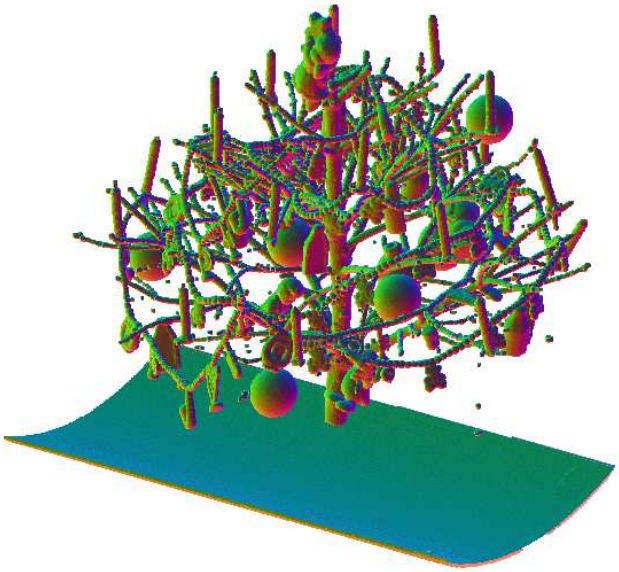


Größe: $128 \times 128 \times 124$ (3.9 MB)
CT-Scan
Computer Graphics Group, Universität Erlangen-Nürnberg, ehemals unter
http://www9.informatik.uni-erlangen.de/∼cfrezksa/VolRen/

Bunny

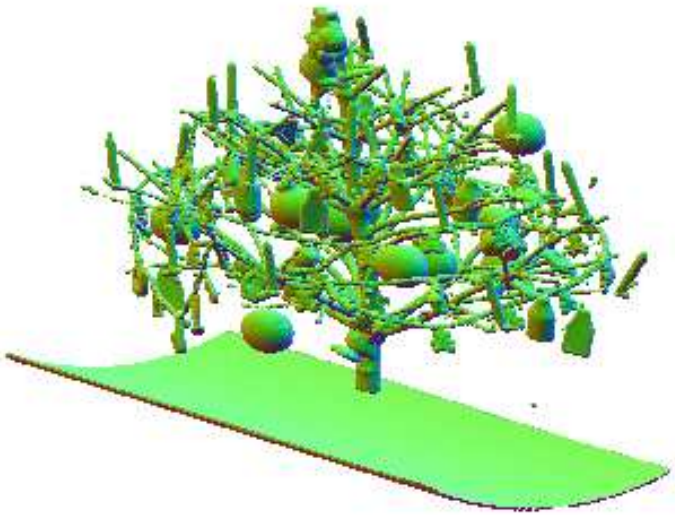
Größe: $360 \times 512 \times 512$ (180 MB)
CT-Scan des Stanford Terracottahasen
http://graphics.stanford.edu/data/voldata

Xmas-Tree



Größe: $499 \times 512 \times 512$ (250 MB)
CT-Scan eines Weihnachtsbaums
http://www.cg.tuwien.ac.at/xmas/

Huge Xmas-Tree

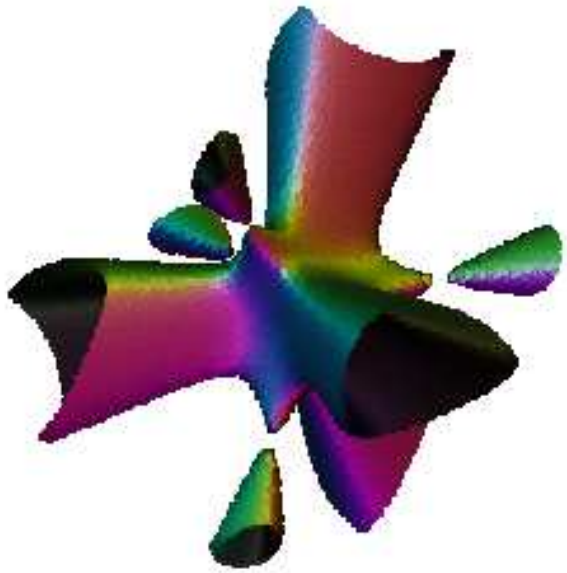


Größe: $999 \times 512 \times 512$ (500 MB)
Große Version des Xmas-Tree-Scans
http://www.cg.tuwien.ac.at/xmas/

A.3 Mathematische Funktionen

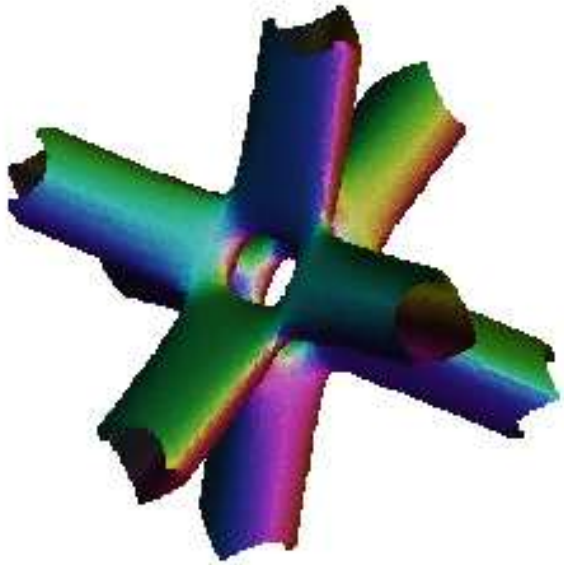
Datensätze von algebraischen Funktionen verlaufen gleichmäßig und haben den Vorteil, dass man ihre Größe beliebig regulieren kann.

Kummer



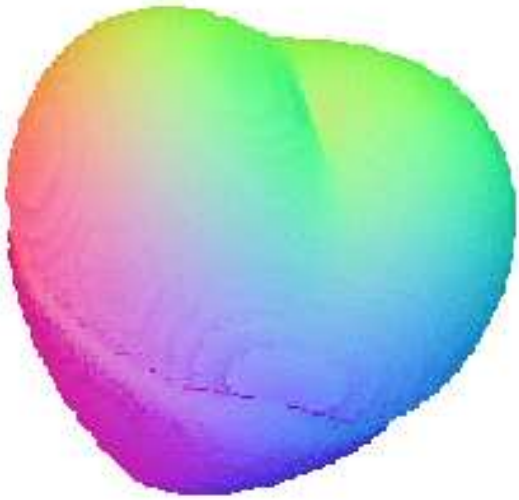
$f(x,y,z) = (x^2+y^2+z^2-2)^2 + 5(1-z-x\sqrt{2})(1-z+x\sqrt{2})(1+z+y\sqrt{2})(1+z-y\sqrt{2})$
Empfohlener Definitionsbereich: $[-10,10]^3$

NewKummer



$f(x,y,z) = (x^4+y^4+z^4) - (x^2y^2+x^2z^2+y^2z^2) - (x^2+y^2+z^2)$
Empfohlener Definitionsbereich: $[-3,3]^3$

Heart

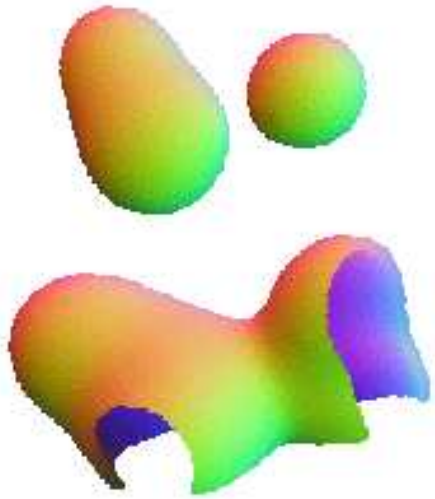


$f(x, y, z) = (x^2 + y^2 + z^2 - 1)^3 - y^3(x^2 + 0.1z^2)$
Empfohlener Definitionsbereich: $[-2, 2]^3$

Blobs

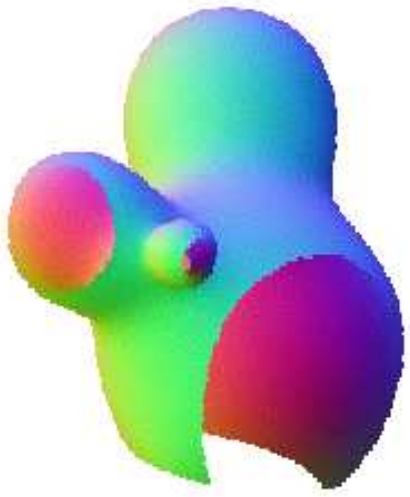
$f(x, y, z) = g(h(x, y, z))$ mit
$h(x, y, z) = \sum_{k=1}^{n} [(x - x_k)^2 + (y - y_k)^2 + (z - z_k)^2]^{-1.5}$
und $g(w) = w^{-1} - sign(w)(w^2 + 1)^{-0.5}$
(Die Funktion g dient der Normierung zur Vermeidung von Polstellen)
Empfohlener Definitionsbereich: Sollte alle Punkte (x_k, y_k, z_k) enthalten

Gravy



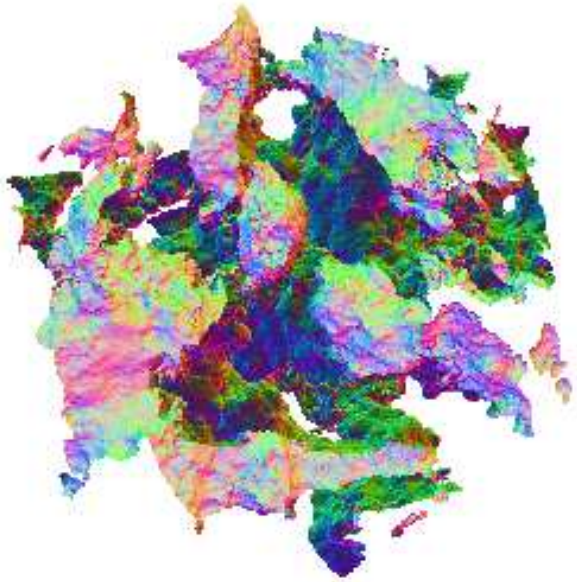
Berechnung des Potentials in einem System mit n Punktmassen der Größe 1
$f(x,y,z) = g(h(x,y,z))$ mit
$h(x,y,z) = \sum_{k=1}^{n}[(x-x_k)^2 + (y-y_k)^2 + (z-z_k)^2]^{-0.5}$
und $g(w) = w^{-1} - sign(w)(w^2+1)^{-0.5}$
(Die Funktion g dient der Normierung zur Vermeidung von Polstellen)
Empfohlener Definitionsbereich: Sollte alle Punkte (x_k, y_k, z_k) enthalten

Charge



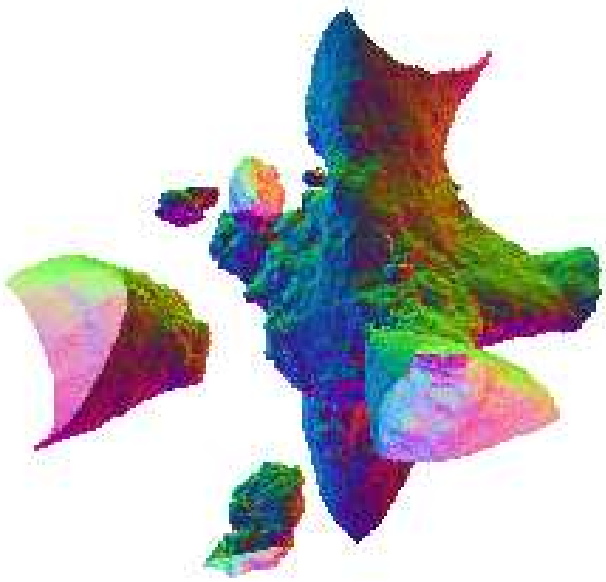
Verallgemeinerung von **Gravy** auf Massen ($g_k > 0$) oder Ladungen (g_k beliebig) beliebiger Größe
$f(x,y,z) = g(h(x,y,z))$ mit
$h(x,y,z) = \sum_{k=1}^{n} g_k[(x-x_k)^2 + (y-y_k)^2 + (z-z_k)^2]^{-0.5}$
und $g(w) = w^{-1} - sign(w)(w^2+1)^{-0.5}$
(Die Funktion g dient der Normierung zur Vermeidung von Polstellen)
Empfohlener Definitionsbereich: Sollte alle Punkte (x_k, y_k, z_k) enthalten

Perlin



Perlin noise, generiert mit einem Programm zur Erzeugung von zufälligem Rauschen
$f(x, y, z) = PN(x, y, z)$
Das Programm zur Berechnung von PN ist zu finden auf der Seite von Ken Perlin:
http://mrl.nyu.edu/∼perlin/doc/oscar.html

Kummer_Perlin



Addition des Rauschens auf die Volumendaten der Kummer-Oberfläche
$f(x, y, z) = f_{Kummer}(x, y, z) + \frac{1}{15} PN(x, y, z)$

Literaturverzeichnis

[1] Volumendaten im Internet:
Xmas tree: Technische Universität Wien, *http://www.cg.tuwien.ac.at/xmas/*
Bunny: Stanford volume data archive, *http://graphics.stanford.edu/data/voldata*
Bighead, Engine: Chapel hill volume rendering test dataset, Volume I,
ftp://www-graphics.stanford.edu/pub/volpack/data
Brain: eingescannt im Max-Planck-Institut, Leipzig, *http://www.cns.mpg.de* .

[2] Chandrajit L. Bajaj, *Accelerated isocontouring of scalar fields*, Data Visualization Techniques, Kap. 3.

[3] Chandrajit L. Bajaj, Valerio Pascucci, Daniel R. Schikore, *Fast isocontouring for improved interactivity*, 1996, ACM Siggraph / IEEE Symposium on Volume Visualization, pp. 39–46.

[4] Chandrajit L. Bajaj, Valerio Pascucci, Daniel R. Schikore, *Fast isocontouring of irregular grids*, Purdue CS Tech. Raft, März 97.

[5] C. Bajaj, V. Pascucci, D. Schikore, *Seed sets and search structures for optimal isocontour extraction*, Technical Report 99-35, Austin (Texas), 1999.

[6] J.L. Bentley, *Multidimensional binary search trees used for associative searching.*, Communications of the ACM, 1975, pp. 509–517.

[7] Udeepta D. Bordoloi, Han-Wei Shen, *Space efficient fast isosurface extraction for large datasets*, Proceedings of IEEE Visualization, Oct 2003.

[8] Yi-Jen Chiang, *Out-of-core isosurface extraction of time-varying fields over irregular grids*, Proc. IEEE Visualization 2003, pp. 217–224.

[9] Yi-Jen Chiang, Ricardo Farias, Cláudio T. Silva, Bin Wei, *A unified infrastructure for parallel out-of-core isosurface extraction and volume rendering of unstructured grids*, Proc. IEEE Symposium on Parallel and Large-Data Visualization and Graphics 2001 , pp. 59–66.

[10] Yi-Jen Chiang, Cláudio T. Silva, *I/O optimal isosurface extraction*, Proceedings IEEE Visualization 1997, pp. 293–300.

[11] Yi-Jen Chiang, Cláudio T. Silva, William J. Schroeder, *Interactive out-of-core isosurface extraction*, Proceedings IEEE Visualization 1998, pp. 167–174.

[12] Philip A. Chou, Tom Lookabaugh, Robert M. Gray, *Optimal pruning with applications to tree-structured source coding and modeling*, IEEE Transactions on Information Theory, vol.35, no.2, pp. 299–315, March 1989.

[13] Paolo Cignoni, Paola Marino, Claudio Montani, Enrico Puppo, Roberto Scopigno, *Speeding up isosurface extraction using interval trees,* IEEE Transactions on Visualization and Computer Graphics 1997, vol.3, no.2, pp. 158–170.

[14] Paolo Cignoni, Claudio Montani, Enrico Puppo, Roberto Scopigno, *Optimal isosurface extraction from irregular volume data,* IEEE/ACM 1996 Symposium on Volume Visualization, pp.31–38.

[15] H. Edelsbrunner, *Dynamic rectangle intersection searching,* Institute for Information Processing Report 47, 1980, TU Graz.

[16] Hugh Everett III, *Generalized Lagrange multiplier method for solving problems of optimum allocation of resources,* Operations Research 11 (1963) 399–417.

[17] Issei Fujishiro, Yuji Maeda, Hiroshi Sato, Yuriko Takeshima, *Volumetric data exploration using interval volume,* IEEE Transactions on Visualization and Computer Graphics, vol. 2, no. 2, pp. 144–155, June 1996.

[18] Takayuki Itoh, Yasushi Yamaguchi, Koji Koyamada, *Fast isosurface generation using the volume thinning algorithm,* IEEE Transactions on Visualization and Computer Graphics 2001, vol.7, no. 1, pp. 32–46.

[19] Marc van Kreveld, René van Oostrum, Chandrajit Bajaj, Valerio Pascucci, Dan Schikore, *Contour trees and small seed sets for isosurface traversal,* Proc. ACM Symp. on Comp. Geom. ‘97, Nice (France).

[20] Philippe Lacroute, Marc Levoy, *The VolPack volume rendering library,* http://www-graphics.stanford.edu/software/volpack .

[21] Yarden Livnat, Charles Hansen, *View dependent isosurface extraction,* IEEE Visualization 1998, pp. 175–180.

[22] Yarden Livnat, Charles Hansen, Steven G. Parker, Christopher R. Johnson, *Isosurface extraction for large-scale data sets,* Data Visualization: The State of the Art 2003 (Dagstuhl), pp. 77–94.

[23] Yarden Livnat, Hai-Wei Shen, Christopher R. Johnson, *A near optimal isosurface extraction algorithm using the span space,* IEEE Transactions on Visualization and Computer Graphics, 1996, vol.2, no.1, pp. 73–84.

[24] William E. Lorensen, Harvey E. Cline, *Marching Cubes: a high resolution 3D surface construction algorithm,* ACM Computer Graphics, vol. 21, no. 4 (7/1987), pp. 163–196.

[25] D. Saupe, J. Toelke: *Optimal Memory Constrained Isosurface Extraction,* Vision Modeling and Visualization 2001, pp. 351–358, http://www.vis.uni-stuttgart.de/vmv01/dl/ .

[26] Han-Wei Shen, *Isosurface Extraction in time-varying fields using a temporal hierarchical index tree,* Proc. Visualization, Okt. 1998, pp. 159–164.

[27] Han-Wei Shen, Charles D. Hansen, Yarden Livnat, Christopher R. Johnson, *Isosurfacing in span space with utmost efficiency (ISSUE),* Proceedings IEEE Visualization 1996, pp. 287–294, 1996.

[28] Han-Wei Shen, Christopher R. Johnson, *Sweeping Simplices: A fast iso-surface extraction algorithm for unstructured grids*, Proceedings IEEE Visualization 1995, pp. 143–150.

[29] Philip M. Sutton, Charles D. Hansen, *Accelerated Isosurface extraction in time-varying fields*, IEEE Transactions on Visualization and Computer Graphics, Vol. 6, No. 2.

[30] Chris Weigle, David C. Banks, *Complex-valued contour meshing*, Proceedings IEEE Visualization 1996, pp. 173–180.

[31] David M. Weinstein, Christopher R. Johnson, *Hierarchical data structures for interactive volume visualization*, University of Utah Technical Report, UUCS-95-012, 1995.

[32] Jane Wilhelms, Allen Van Gelder, *Octrees for faster isosurface generation*, ACM Transactions on Graphics, 1992, vol.11, no.3, pp. 201–227.

[33] Jane Wilhelms, Allen Van Gelder, *Topological considerations in isosurface generation*, ACM Transactions on Graphics, 1994, vol.13, no.4, pp.337–375.