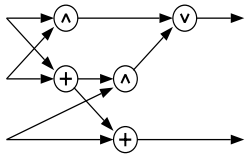
Diplomarbeit

Untersuchungen zu einem polynomiellen Approximationsschema für das Problem des Handelsreisenden im Euklidischen Raum

Axel Thümmler



Diplomarbeit
am Fachbereich Informatik
der Universität Dortmund

3. Februar 1998

Betreuer:

Prof. Dr. H. U. Simon

Inhaltsverzeichnis

Abbildungsverzeichnis **3**

1 Einleitung **5**

1.1 Geschichtliche Entwicklung 6

1.2 Überblick 7

2 Grundlagen **9**

2.1 Grundlegende Definitionen 9

2.2 Die Eingabe für das Euklidische TSP 10

2.3 Verwandte Probleme 13

3 Der Algorithmus **14**

3.1 Das Euklidische TSP in der Ebene 14

3.2 Beschreibung des Algorithmus 16

3.2.1 Normierung der Eingabedaten 19

3.2.2 Konstruktion und Analyse des Quadtrees 21

3.2.3 Das Hauptprogramm des Algorithmus 26

3.3 Beweis des Struktur-Theorems 31

3.3.1 Das Patching-Lemma 31

3.3.2 Das Struktur-Theorem 38

3.4 Wie effizient ist der Algorithmus? 44

4 Der Algorithmus auf einem Parallelrechner **46**

4.1 Einführung in das verwendete Parallelrechnermodell 46

4.2 Parallele Konstruktion des Quadtrees 48

4.3 Beschreibung des parallelen Algorithmus 51

5 Zusammenfassung und Ausblick **59**

A Mathematische Grundlagen **61**

A.1 Summenformeln 61

A.2 Binomialkoeffizienten 62

A.3 Wahrscheinlichkeitsrechnung 63

Literaturverzeichnis **65**

Abbildungsverzeichnis

2.1 Beispielrundreise, bei der es auf die Genauigkeit der Berechnung ankommt, ob sie die Kostenschranke $B = 37$ einhält 12

3.1 Ein Beispiel für eine Dissection 15

3.2 Der zugehörige Quadtree 15

3.3 Die geshiftete Dissection 16

3.4 Der geshiftete Quadtree 16

3.5 Verteilung der Portale auf dem Rand eines Quadrates und seiner vier eingebetteten Unterquadrate für $m = 4$ 17

3.6 Verschiebung der Eingabepunkte auf den nächsten Gitterpunkt im Gitter der Auflösung d 19

3.7 Ungünstigste Möglichkeit, zwei Eingabepunkte im Hinblick auf eine geringe Tourkostenerhöhung zu verschieben 19

3.8 Maximaler Quadtree bei 8 Eingabepunkten, die mindestens Abstand $\sqrt{2}$ haben und deren Bounding-Box Größe 8 hat 24

3.9 Beispiel für einen geschlossenen Pfad, der S fünfmal durchkreuzt 33

3.10 Hilfsgraph zu dem Beispielpfad 33

3.11 Hilfsgraph nach dem ersten Schleifendurchlauf des Patching-Algorithmus . 35

3.12 Endversion des Hilfsgraphen nach dem zweiten Scheifendurchlauf des Patching-Algorithmus 35

3.13 Pfad $\widetilde{\pi}$, der S dreimal durchkreuzt und mit Liniensegmenten der Gesamtlänge von mindestens $2l$ erweitert werden muß, damit er S nur noch höchstens zweimal schneidet 36

3.14 Hilfsgraph zu $\widetilde{\pi}$ ohne die Kanten zwischen der linken und rechten Seite . . 36

3.15 Die drei Möglichkeiten, den Hilfsgraphen zu $\widetilde{\pi}$ um Kanten zu erweitern, so daß ein Eulergraph entsteht, der S nur noch einmal schneidet 36

3.16 Beispiel für eine Kante in der Rundreise, die 3 horizontale und 9 vertikale Gitterlinien schneidet 37

3.17 Gitterlinien mit ihren max-Level Werten 0, 1, 2 und 3 in einer Bounding-Box der Größe 8 39

3.18 Vertikale Gitterlinie l mit max-Level i und einem angrenzenden Level-i-Quadrat zwischen den y-Koordinaten y_p und y_{p+1} 39

4.1 Beispiel für einen Quadtree in der Darstellung als Baum 49

4.2 Das zugehörige Skelett 49

4.3 Die Positionsnummern der vier Unterquadrate in einem Quadrat 49

4.4 Die gemischt sortierte Reihenfolge der Punkte in einem Quadrat der Größe 8; links unten startend und rechts oben endend 50

Kapitel 1

Einleitung

Wenn ein Handelsreisender, startend in seinem Wohnort, Kunden in verschiedenen Städten besuchen möchte, muß er sich eine Reihenfolge überlegen, in der er die Städte anfährt. Er wird dazu zweckmäßigerweise eine Rundreise auswählen wollen, die möglichst kurz ist, um Kosten und Zeit zu sparen. Nehmen wir an, er kennt für alle Städte die geographischen Positionen und für jedes Paar von Städten deren Abstand, dann hat er alle notwendigen Daten, um eine kürzeste Rundreise zu bestimmen, jedoch ist nicht ohne weiteres zu ersehen, wie er diese Daten nutzen kann, um eine kürzeste Rundreise zu finden. Dieses Problem ist als *Traveling Salesman Problem (TSP)* bekannt. In letzter Zeit findet man auch oft die Bezeichnung *Traveling Salesperson Problem* in Anlehnung an die zunehmende Frauenquote in wirtschaftlichen Berufen. Es gibt viele verschiedene Versionen und Spezialfälle des TSP, von denen wir hier drei genauer definieren wollen.

Definition 1.0.1 (Traveling Salesman Problem (TSP))
(1) (*allgemeines TSP*) Gegeben sind n Städte $c_1, ..., c_n$ und eine Kostenfunktion k. Es sind $k(c_i, c_j)$ die Kosten, um von Stadt c_i nach Stadt c_j zu kommen. Gesucht ist eine möglichst kurze Rundreise, die alle Städte besucht, also eine Reihenfolge der Städte $c_{\tau(1)}, ..., c_{\tau(n)}$, so daß die Kosten

$$\sum_{i=1}^{n-1} k(c_{\tau(i)}, c_{\tau(i+1)}) + k(c_{\tau(n)}, c_{\tau(1)})$$

minimal sind.

(2) (*Metrisches TSP*) Die Städte sind hier Punkte in einem metrischen Raum und die Kostenfunktion k erfüllt die Regeln einer Metrik, also Positivität, Symmetrie und Dreiecksungleichung.

(3) (*Euklidisches TSP*) Die Städte sind Punkte im $\mathbb{R}^2$ und die Kosten $k(c_i, c_j)$, die durch die Reise von c_i nach c_j entstehen, sind der euklidische Abstand der Punkte.

Die besondere Bedeutung des TSP rührt nicht daher, daß Handelsreisende versucht haben eine kürzeste Rundreise zu bestimmen, sondern vielmehr daher, daß das TSP stellvertretend für viele andere kombinatorische Optimierungsprobleme steht, so z.B. die Com-

puterverdrahtung oder die Platinenbohrung (Printed Circuit Boards). Es zeichnet sich aus durch die einfache, für jeden verständliche, Problemstellung und die Leichtigkeit „gute“ Lösungen zu finden im Kontrast zur Schwierigkeit eine optimale Lösung anzugeben. Dadurch hat das Problem schon seit jeher nicht nur die Mathematiker und Informatiker begeistert, weshalb bis heute unzählige Veröffentlichungen zum TSP erschienen sind. Eine interessante frühe geschichtliche Entwicklung des TSP ist in [19] zu finden und eine umfangreiche Bibliographie mit aktuellen Arbeiten über das TSP und verwandte Probleme findet man im Internet bei [26].

1.1 Geschichtliche Entwicklung

Seit den 70er Jahren ist bekannt, daß das TSP NP-hart ist, eine optimale Lösung also wahrscheinlich nicht effizient berechnet werden kann [18]. Um so erstaunlicher ist es, daß eine intuitive Wahl einer Rundreise meistens eine sehr gute Näherungslösung ist und es gute Heuristiken gibt, die in nahezu linearer Zeit eine Näherungslösung berechnen, die oft nur um ein paar Prozent vom Optimum abweicht. Heuristiken können aber keine feste Güte garantieren, es gibt also immer Ausnahmefälle, in denen sie eine schlechte Lösung produzieren. Anders ist es bei Approximationsalgorithmen, bei denen eine beweisbare Güte für die gefundene Lösung angegeben werden kann. Sahni und Gonzalez konnten allerdings 1976 zeigen, daß es für das allgemeine TSP unter der P $\neq$ NP-Annahme keinen effizienten Approximationsalgorithmus gibt, der das Optimum bis auf irgendeinen konstanten Faktor approximiert [23].

Die in der Praxis auftretenden Eingabeinstanzen für das TSP sind oft sehr speziell. Die gerade beschriebenen negativen Resultate müssen deshalb dafür nicht auch zwangsläufig gelten. Garey, Graham und Johnson [14] bzw. Papadimitriou [20] zeigten dann allerdings, daß selbst das Euklidische TSP, ein Spezialfall vom Metrischen TSP, NP-hart ist. Es blieb also noch die Hoffnung, daß für diese Spezialfälle gute Approximationsalgorithmen existieren. Christofides [6] hat dann 1976 einen Approximationsalgorithmus entwickelt, der in Polynomialzeit für eine beliebige Eingabe des Metrischen TSP's eine Lösung produziert, die beweisbar nicht größer als $\frac{3}{2}$ mal der optimalen Lösung ist. Für eine gute Beschreibung des anschaulichen Algorithmus siehe z. B. [19].

An die zwei Jahrzehnte der Suche nach einem besseren als Christofides' Algorithmus sind vergangen und viele Forscher glaubten, daß sogar ein PTAS existieren könnte. Ein PTAS ist ein polynomieller Approximationsalgorithmus, der eine Güte beliebig nahe an 1 garantieren kann. Eine genau Definition eines PTAS ist im nächsten Kapitel zu finden. Erst kürzlich gelang es dann Arora, Lund, Motwani, Sudan und Szegedy [2] zu zeigen, daß für das Metrische TSP und viele andere Probleme kein PTAS existiert, falls P $\neq$ NP.

Die Frage, ob ein PTAS für das Euklidische TSP existiert, blieb bis 1996 offen, als Arora [1] zeigte, daß es ein PTAS gibt. Für jedes feste $c > 2$ berechnet eine randomisierte Version des Approximationsalgorithmus eine Rundreise, die nicht länger als $1 + \frac{1}{c}$ mal der Länge der kürzesten Rundreise ist. Die Laufzeit ist dabei $O\left(n(\log n)^{O(c)}\right)$. Der Algorithmus

kann leicht derandomisiert werden, wobei sich die Laufzeit allerdings um einen Faktor $O(n^2)$ verschlechtert. Wir wollen uns in dieser Arbeit mit dem Euklidischen TSP in der Ebene beschäftigen und zuerst Arora's Algorithmus genau analysieren, um schließlich eine mögliche Implementierung auf einem Parallelrechner zu untersuchen.

Das PTAS benutzt eine rekursive Zerlegung der Ebene in Quadrate, die durch einen randomisierten Quadtree dargestellt werden, so daß es eine Rundreise gibt, die die optimale Rundreise bis auf einen Faktor $1 + \frac{1}{c}$ approximiert und die jede Kante jedes Quadrates in der Zerlegung höchstens $O(c)$-mal schneidet. Eine solche Rundreise kann mittels dynamischer Programmierung gefunden werden. Für jedes Quadrat in der Zerlegung „rät" der Algorithmus zuerst, an welchen Stellen die Rundreise die Kanten des Quadrates schneidet und kann dann die Quadrate unabhängig voneinander bearbeiten. Es wird gezeigt, daß der Quadtree nur aus $O(n \log n)$ Quadraten besteht und das „Raten" der Schnittpunkte nicht mehr als $(\log n)^{O(c)}$ Zeit pro Quadrat in Anspruch nimmt. Daraus resultiert die Gesamtlaufzeit von $O\left(n(\log n)^{O(c)}\right)$.

1.2 Überblick

Im zweiten Kapitel definieren wir einige grundlegende Begriffe aus der Komplexitätstheorie zum Thema Approximationsalgorithmen. Dann beschreiben wir im Detail die Eingabe zum Euklidischen TSP und welche Probleme bei der Bestimmung der Länge einer Rundreise entstehen. Am Ende von Kapitel 2 stellen wir noch einige zum Euklidischen TSP verwandte Probleme dar, für die ebenfalls ein PTAS existiert.

Das dritte Kapitel beschreibt sehr ausführlich Arora's PTAS für das Euklidische TSP in der Ebene. Es wird zuerst die Struktur des Quadtrees, auf dem der Approximationsalgorithmus aufbaut, erklärt und dann die Vorgehensweise des Algorithmus beschrieben. Wir werden dabei auch einige grundlegende Einsichten in die Struktur eines Quadtrees erlangen, die über Arora's Ausführungen hinausgehen. Der Approximationsalgorithmus basiert auf dem Struktur-Theorem, das am Ende des Kapitels bewiesen wird. Der Beweis benutzt dabei im wesentlichen das Patching Lemma, das wir in dieser Arbeit auch in einer neuen Version darstellen wollen. Die nächsten drei Abschnitte beschreiben dann die einzelnen Phasen des Algorithmus. Zuerst werden die Eingabedaten normiert, dann wird der Quadtree konstruiert und zuletzt die approximierte Rundreise mit Hilfe des Quadtrees berechnet. Am Ende des Kapitels untersuchen wir schließlich die Effizienz des Algorithmus.

Das vierte Kapitel ist der Parallelisierung des PTAS gewidmet. Wir geben dazu zuerst eine Einführung in das PRAM-Modell. Als nächstes beschäftigen wir uns damit, wie wir einen Quadtree parallel konstruieren können. Wir benutzen dazu im wesentlichen die Resultate von Bern, Eppstein und Teng [5]. Zum Schluß des Kapitels zeigen wir dann, als wesentliches neues Resultat, wie das PTAS parallelisiert werden kann, indem wir verschiedene Versionen mit verschiedenen Graden der Parallelität entwickeln. Die Version mit dem höchsten Parallelisierungsgrad erreicht dabei eine Laufzeit von $O(\log n)$.

Sie fordert dafür aber auch den mächtigsten Prozessortyp.

Im fünften Kapitel fassen wir die grundlegenden Ergebnisse unserer Arbeit nochmal zusammen und geben einen Ausblick auf zukünftige Forschungsinteressen. Die Zusammenfassung ist auch für den Leser gedacht, der nur an den relativ einfach nachzuvollziehenden Resultaten interessiert ist.

Einige allgemeine Grundlagen der Mathematik über Summen, Binomialkoeffizienten und Wahrscheinlichkeitsrechnug sind im Anhang aufgeführt. Auf diese Grundlagen wird in den einzelnen Kapiteln an den entsprechenden Stellen separat verwiesen.

Kapitel 2

Grundlagen

In diesem Kapitel wollen wir einige Grundlagen über Approximationsalgorithmen darstellen, wie sie auch in [13] oder [25] zu finden sind. Wir charakterisieren dazu den Problemtyp, zu dem Approximationsalgorithmen gesucht werden und beschreiben anschließend, wodurch sich ein guter Approximationsalgorithmus auszeichnet. Danach gehen wir darauf ein, wie die Eingabe für unseren Approximationsalgorithmus aussehen soll und welche Probleme daraus entstehen. Zuletzt wollen wir in diesem Kapitel noch einige zum TSP verwandte Probleme darstellen, für die analoge Approximationsalgorithmen existieren.

2.1 Grundlegende Definitionen

Definition 2.1.1 Ein *Optimierungsproblem* Π ist entweder ein Maximierungsproblem oder ein Minimierungsproblem. Es ist gegeben durch:

1. Eine Menge D zulässiger Eingaben für Π.
2. Für jede Eingabe $I \in D$ eine Menge $S(I)$ möglicher Lösungen.
3. Eine Funktion w, die jeder Eingabe $I \in D$ und jeder möglichen zugehörigen Lösung $\sigma \in S(I)$ eine positive rationale Zahl $w(I, \sigma)$, genannt der Wert von σ, zuordnet.

Mit $OPT(I)$ bezeichnen wir eine optimale Lösung σ_{opt} für die Eingabe I und mit opt_I bezeichnen wir den Wert dieser Lösung. Bei einem Minimierungsproblem, wie es das TSP ist, gilt bei Eingabe $I \in D$ für alle Lösungen $\sigma \in S(I)$: $w(I, OPT(I)) = opt_I \leq w(I, \sigma)$. Bei Maximierungsproblemen ist die Beziehung genau umgekehrt.

Betrachten wir das Euklidische TSP. Es ist ein Minimierungsproblem und D besteht aus allen endlichen Mengen von Punkten in der Ebene. Die Menge $S(I)$ der möglichen Lösungen zur Eingabe I besteht aus allen Rundreisen die auf I möglich sind, also allen Permutationen der Städte in I. Der Wert einer Lösung $w(I, \sigma)$ ist die Länge der berechneten Rundreise σ auf der Eingabe I.

Definition 2.1.2 Ein *Approximationsalgorithmus* A für ein Optimierungsproblem ist ein Algorithmus, der bei Eingabe $I \in D$ eine mögliche Lösung $A(I) \in S(I)$ berechnet.

Bezogen auf das TSP bedeutet das, daß der Algorithmus A eine der Permutationen der Städte in I berechnet. $A(I)$ ist dann die aus dieser Permutation abgelesene Rundreise und $w(I, A(I))$ sind die Kosten dieser Rundreise.

Definition 2.1.3 Sei A ein Approximationsalgorithmus für ein Optimierungsproblem. Die *Güte* $R_A(I)$ der Lösung $A(I)$ ist definiert durch:

$$R_A(I) := \frac{w(I, A(I))}{opt_I} \qquad \text{für Minimierungsprobleme und}$$

$$R_A(I) := \frac{opt_I}{w(I, A(I))} \qquad \text{für Maximierungsprobleme}$$

Die Güte eines Approximationsalgorithmus ist so definiert, daß sie immer größer gleich 1 ist. Berechnet der Algorithmus eine optimale Lösung, dann ist die Güte genau 1.

Definition 2.1.4 Ein *Polynomial Time Approximation Scheme*, kurz *PTAS*, für ein gegebenes Optimierungsproblem Π ist ein Algorithmus A, der aus der Eingabe $I \in D$ und einem Genauigkeitswert $c > 0$ eine Lösung $A_c(I) \in S(I)$ produziert, so daß gilt:

$$R_{A_c}(I) \leq 1 + \frac{1}{c}$$

Für die Laufzeit von A muß gelten, daß sie für jedes feste c polynomiell in der Länge von I beschränkt ist.

In der Definition wird der Ausdruck „Scheme“ verwendet, da A eigentlich eine Familie $(A_c)_{c>0}$ von Approximationsalgorithmen ist; einer für jeden festen Wert von c. Zu beachten ist, daß die Laufzeit des Algorithmus nur polynomiell in der Länge der Eingabe beschränkt sein muß, nicht aber in c. Es sind also Laufzeiten mit c im Exponenten möglich. Man spricht bei einem PTAS auch von einem Algorithmus, der eine $\left(1 + \frac{1}{c}\right)$-Approximation erreicht, was bei einem Minimierungsproblem heißen soll, daß der Algorithmus zur Eingabe I eine Lösung σ produziert, deren Wert höchstens $\left(1 + \frac{1}{c}\right) opt_I$ ist.

Im Zusammenhang mit dem Euklidischen TSP erhält man durch das PTAS nur dann eine Verbesserung zu anderen bekannten Approximationsalgorithmen, wenn c größer als 2 gewählt wird, denn für c gleich 2 erhält man nur eine Güte von $\frac{3}{2}$, die auch von Christofides' Algorithmus erreicht wird.

2.2 Die Eingabe für das Euklidische TSP

Die Eingabe für unseren Algorithmus ist eine Menge mit n Punkten aus $\mathbb{R}^2$, die durch ihre Koordinaten gegeben sind. Der Unterschied des Euklidischen TSP zum TSP ohne

Einschränkungen ist, daß die Distanz zweier Punkte über die ℓ_2-Norm bestimmt wird. Die ℓ_2-Norm ist ein Spezialfall der ℓ_p-Norm, $p \geq 1$, in der der Abstand zweier Punkte $x = (x_1, x_2)$ und $y = (y_1, y_2)$ definiert ist durch:

$$d_p(x, y) := \left(|x_1 - y_1|^p + |x_2 - y_2|^p\right)^{\frac{1}{p}}$$

Für $p = 1$ bezeichnet man die ℓ_1-Norm auch als *Rechtecksnorm*. Sie gibt die Länge des kürzesten Pfades zwichen x und y an, der nur aus Kanten besteht, die parallel zur x- bzw. y-Achse verlaufen. Für $p = 2$ bezeichnen wir die ℓ_2-Norm als *euklidische Norm*. Sie mißt die Länge der geradlinigen Verbindung zwischen den beiden Punkten. Für diese Norm werden wir das TSP genauer untersuchen. So wie wir die ℓ_p-Norm eingeführt haben, ist sie eine Metrik über $\mathbb{R}^2$. Es gelten somit für sie auch die Gesetze einer Metrik, insbesondere die Dreiecksungleichung,

$$d_p(x, z) \leq d_p(x, y) + d_p(y, z)$$

die besagt, daß der direkte Weg von x nach z nicht länger ist als der Weg von x nach y und von dort nach z.

Betrachten wir die *Entscheidungsvariante* des TSP. Es ist hier die Frage, ob zu einer gegebenen Eingabe I eine Rundreise σ derart existiert, daß die Kosten der Rundreise nicht größer als eine vorgegebene Schranke B sind. Im allgemeinen können solche Entscheidungsprobleme von einer nichtdeterministischen Turingmaschine gelöst werden, indem sie eine Lösung rät und dann in Polynomialzeit verifiziert, daß die Kostenschranke eingehalten wird. Beim Euklidischen TSP entsteht hier aber ein technisches Problem. Die Distanzen zwischen den Punkten können irrationale Zahlen sein, selbst wenn die Punkte ganzzahlige Koordinaten haben und wir können irrationale Zahlen nicht exakt speichern. Wir könnten das Problem beheben, indem wir die Distanzmatrix implizit angeben. Wir würden dazu immer das Quadrat der Distanzen in die Matrix eintragen und nur mit diesen Quadraten rechnen. Es bleibt aber trotzdem ein Problem bestehen, da bei der Berechnung der gesamten Tourlänge eine Summe von Quadratwurzeln berechnet werden muß, die dann mit der Schranke B verglichen werden soll. Bei dieser Summenberechnung müssen wir uns auf eine Rechnung mit abgeschnittenen Zahlen beschränken und das kann problematisch werden. In Abbildung 2.1 ist eine Beispieleingabe dargestellt, bei der entschieden werden soll, ob die Kosten der Rundreise die Schranke $B = 37$ einhalten. Rechnen wir hier mit einer Genauigkeit von 5 Stellen und schneiden die restlichen Stellen ab so erhalten wir:

$$\sqrt{15^2 + 4^2} + \sqrt{5^2 + 9^2} + \sqrt{10^2 + 5^2} = 15.524 + 10.295 + 11.180 = 36.999$$

Bei einer Rechnung mit 8 Stellen Genauigkeit erhalten wir hingegen:

$$\sqrt{15^2 + 4^2} + \sqrt{5^2 + 9^2} + \sqrt{10^2 + 5^2} = 15.524174 + 10.295630 + 11.180339 = 37.000143$$

Die Entscheidung würde in den beiden Berechnungen verschieden ausfallen. Es ist bis heute noch keine Methode bekannt, um in polynomieller Zeit die Anzahl der benötigten Stellen zu bestimmen, so daß die richtige Entscheidung getroffen werden kann. Es ist also

nicht klar, ob die Entscheidungsvariante des Traveling Salesman Problems überhaupt in NP ist, da nicht effizient verifiziert werden kann, ob eine (geratene) Rundreise die Kostenschranke B einhält.

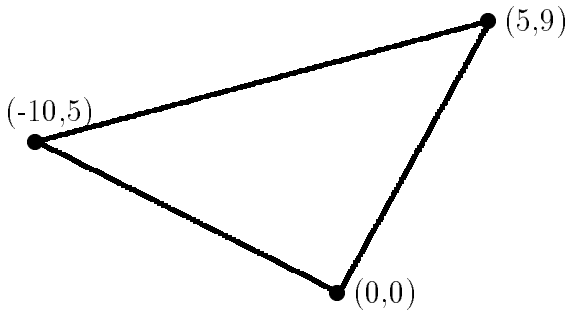


Abbildung 2.1: Beispielrundreise

Bei der Optimierungsvariante des TSP, die im Rahmen dieser Arbeit behandelt wird, ist die Situation etwas anders. Da wir nicht mit unendlicher Genauigkeit rechnen können, wollen wir bei unserem Approximationsalgorithmus den euklidischen Abstand zweier Punkte nur auf $1 + \lceil \log cn \rceil$ Nachkommabits berechnen und den Rest aufrunden. Der Abstand zweier Punkte vergrößert sich dadurch um maximal

$$\frac{1}{2^{1+\lceil \log cn \rceil}} \leq \frac{1}{2^{1+\log cn}} = \frac{1}{2cn}$$

Da eine Rundreise aus n Kanten besteht, erhalten wir eine maximale Verlängerung der exakten Lösung von $\frac{1}{2c}$. Nach dem Struktur-Theorem, das wir in Kapitel 3 präzise formulieren werden, existiert eine Rundreise R mit bestimmten Einschränkungen, die das Optimum bis auf einen Faktor $1 + \frac{1}{c}$ approximiert. Das PTAS berechnet nun eine „kürzeste“ Rundreise $\widetilde{R}$ mit den im Struktur-Theorem beschriebenen Einschränkungen bezüglich der aufgerundeten Distanzen. Wir wollen jetzt untersuchen, wie wir die Genauigkeitsanforderung c anpassen müssen, damit auch $\widetilde{R}$ eine $\left(1 + \frac{1}{c}\right)$-Approximation ist. Dazu bezeichnen wir mit $cost_R$ die mit den exakten euklidischen Distanzen berechneten Kosten der Rundreise R und mit $cost'_R$ die Kosten von R bezüglich der aufgerundeten Distanzen. Nach obiger Überlegung sind dann die Kosten $cost'_R$ um maximal $\frac{1}{2c}$ größer als die Kosten $cost_R$. Wählen wir $c' := 2c$, dann gilt:

$$cost'_{\widetilde{R}} \leq cost'_R \leq cost_R + \frac{1}{2c} \leq \left(1 + \frac{1}{c'}\right) opt + \frac{1}{2c} \leq \left(1 + \frac{1}{c}\right) opt$$

Die letzte Ungleichung fordert, daß $opt \geq 1$ ist. Dies wird durch eine anfängliche Normierungsphase, in der die Koordinaten der Punkte ganzzahlig gemacht werden und der minimale Punkteabstand auf 4 gebracht wird, gewährleistet. Die Vergrößerung der Kosten kann im Zusammenhang mit einem PTAS also durch eine „interne“ größere Wahl von c ausgeglichen werden.

Für die Berechnungen des Algorithmus wollen wir das RAM-Modell mit dem uniformen Kostenmaß verwenden [21]. Wir nehmen dazu an, daß eine arithmetische Operation mit reellen Zahlen in einem Schritt ausgeführt werden kann. Genaugenommen benötigen wir Operationen auf reellen Zahlen nur in der Normierungsphase, in der die reellwertigen Koordinaten ganzzahlig gemacht werden. Wir benötigen dafür eine Division, eine Addition und eine Rundungsoperation auf reellen Zahlen. Danach rechnen wir nur noch mit ganzen Zahlen bzw. den aufgerundeten Distanzwerten. Für die aufgerundeten Distanzen gelten im übrigen immer noch die Regeln einer Metrik. Insbesondere gilt die Dreiecksungleichung.

2.3 Verwandte Probleme

Wir wollen in diesem Abschnitt einige zum TSP verwandte Probleme vorstellen, für die ebenfalls ein PTAS existiert [1]. Alle Probleme bis auf das Euclidean Matching sind NP-hart. Es wird in [1] auch gezeigt, daß für die Probleme ein PTAS existiert, wenn die Eingabepunkte aus $\mathbb{R}^d$ sind, für eine beliebige Dimension d.

1. **Minimum Steiner Tree:** Gegeben sind n Knotenpunkte im $\mathbb{R}^2$. Gesucht ist ein Baum, der die Knoten miteinander verbindet und dessen Kanten zusammen möglichst kurz sind. Um die Verbindungskanten möglichst kurz zu machen, können dabei neue Knotenpunkte, sogenannte *Steiner Punkte*, eingefügt werden.

2. **k-TSP:** Gegeben sind n Punkte im $\mathbb{R}^2$ und eine Zahl $k \in \mathbb{N}$. Gesucht ist die kürzeste Tour, die mindestens k Punkte besucht.

3. **k-Minimum Spanning Tree (k-MST):** Gegeben sind n Knoten im $\mathbb{R}^2$ und eine Zahl $k \geq 2$. Gesucht sind k Knoten mit dem kürzesten minimalen Spannbaum.

4. **Euclidean Min Cost Perfect Matching:** Gegeben sind $2n$ Punkte im $\mathbb{R}^2$. Gesucht ist eine Menge mit nichtadjazenten Kanten, die alle Punkte überdecken und zusammen möglichst kurz sind.

Kapitel 3

Der Algorithmus

Wir wollen in diesem Kapitel ein PTAS für das Euklidische Traveling Salesman Problem in der Ebene beschreiben. Wir erklären dazu, wie die Ebene rekursiv durch einen Quadtree in kleinere Segmente aufgeteilt wird. Im folgenden Abschnitt geben wir dann das Struktur-Theorem an, das die Grundlage für den darauf folgenden Approximationsalgorithmus bildet. Wir beschreiben den Algorithmus in drei Phasen. Zuerst werden die Eingabepunkte normiert, dann der Quadtree konstruiert und anschließend der eigentliche Approximationsalgorithmus gestartet. Wir untersuchen die Laufzeit des Algorithmus und auch, wie sinnvoll es ist, ihn auf einem Rechner zu implementieren. Unsere Beschreibungen sind für das Euklidische TSP in der Ebene gemacht, lassen sich aber auf jede höhere Dimension erweitern, wobei nur geringfügige Einbußen in der Laufzeit zu erwarten sind. Arora hat dies in seiner Arbeit gezeigt [1].

3.1 Das Euklidische TSP in der Ebene

Wie in der Einleitung schon erwähnt, ist die Hauptidee des Algorithmus eine rekursive geometrische Zerlegung der Eingabeinstanzen. Die geometrische Zerlegung ist sehr einfach: es ist eine randomisierte Quadtree-Variante. Als erstes werden einige grundlegende Begriffe definiert, die wir im folgenden gebrauchen.

Ein kleinstes umschließendes, achsenparalleles Quadrat zu einer Punktemenge im $\mathbb{R}^2$ heißt *exakte Bounding-Box* oder kurz *exakte B-Box* (i.a. ist eine exakte B-Box nicht eindeutig bestimmt). Sprechen wir nur von einer *Bounding-Box* bzw. einer *B-Box* so meinen wir eine exakte B-Box, deren Ausmaße gleichmäßig etwas größer sind, so daß keine Punkte auf dem oberen und rechten Rand liegen. Die *Größe der (exakten) Bounding-Box* ist die Kantenlänge des Quadrates. Eine gegebene (exakte) Bounding-Box ist eideutig bestimmt durch Angabe der unteren linken Ecke (x_u, y_u) und der Größe L. Wir nehmen o. B. d. A. an, daß die untere linke Ecke immer an den Koordinaten $(0, 0)$ liegt. (Wir können den Ursprung des Koordinatensystems auf (x_u, y_u) schieben). Die Punkte liegen bei einer B-Box dann alle in $[0, L[^2$ und bei einer exakten B-Box in $[0, L]^2$.

Eine *Dissection* ist eine rekursive Aufteilung einer Bounding-Box in kleinere Quadrate, wobei jedes Quadrat entlang der Mitte in 4 gleichgroße Unterquadrate aufgeteilt wird. Hat die Bounding-Box Größe L, so haben die 4 Unterquadrate jeweils Größe $\frac{L}{2}$. In jedem dieser Quadrate sind dann wieder 4 Unterquadrate der Größe $\frac{L}{4}$ enthalten und so weiter. Die Aufteilung wird gestoppt, sobald die in der B-Box liegenden Punkte voneinander separiert sind, was heißen soll, daß sie in verschiedenen Quadraten liegen (siehe Abbildung 3.1). Bei der Aufteilung kann es vorkommen, daß sich Punkte auf dem Rand eines Quadrates bzw. genau auf einem Eckpunkt befinden. Es gilt dabei, daß Punkte auf dem linken bzw. unteren Rand eines Quadrates zu diesem gehören und Punkte auf dem oberen bzw. rechten Rand außerhalb liegen. Für die Eckpunkte eines Quadrates gilt, daß nur die linke untere Ecke zu diesem gehört und die anderen drei Eckpunkte außerhalb liegen. Da sich keine Punkte auf dem oberen bzw. rechten Rand der B-Box befinden dürfen, sind alle Punkte in Quadraten untergebracht. Dargestellt wird die Dissection durch einen Baum, in dem die Knoten den Quadraten entsprechen. Wir werden im folgenden den Ausdruck „Knoten“ und „Quadrat“ gleichbedeutend gebrauchen. Jeder innere Knoten hat 4 Söhne. Die Wurzel repräsentiert die ursprüngliche B-Box und die Söhne stellen die Unterquadrate dar.

Ein *Quadtree* ist genauso definiert wie die Dissection abgesehen davon, daß wir die rekursive Aufteilung stoppen, sobald ein Quadrat nur noch einen Punkt enthält. Im allgemeinen besteht dadurch der Quadtree aus weniger Quadraten als die Dissection (siehe Abbildung 3.2). In der Darstellung als Baum entspricht wieder die Wurzel der B-Box und die Knoten des Baumes stellen die Quadrate dar. Jeder innere Knoten hat 4 Söhne für die 4 Unterquadrate. Die Anzahl der Quadrate in der graphischen Darstellung des Quadtrees und auch in der Dissection entspricht also der Anzahl der Knoten in der Baumdarstellung.

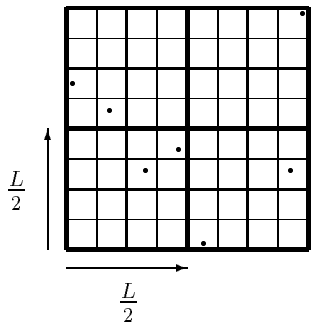


Abbildung 3.1: Die Dissection

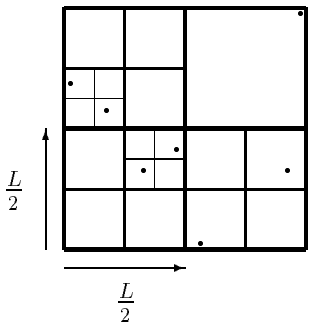


Abbildung 3.2: Der entsprechende Quadtree

Durch die Struktur einer Dissection bzw. eines Quadtrees sind die Quadrate auf *Ebenen* angeordnet. Ebene 0 enthält nur die Wurzel, Ebene 1 die 4 Söhne der Wurzel und Ebene 2 wieder deren Söhne und so weiter. Anstatt Ebene werden wir auch synonym den Ausdruck *Level* verwenden.

Die *Tiefe* einer Dissection bzw. eines Quadtrees ist die größte Ebenennummer, die in der Darstellung vorkommt, und die Tiefe eines Quadrates ist die Nummer der Ebene,

auf der es liegt. Die Wurzel hat z. B. Tiefe 0. Die in Abbildung 3.1 und 3.2 dargestellten Zerlegungen haben Tiefe 3.

Seien $0 \leq a, b < L$ ganzzahlig, dann ist die *Dissection mit Shift* (a, b) dadurch definiert, daß die x- und y-Koordinaten von allen Linien in der Dissection jeweils um a bzw. b verschoben werden und anschließend modulo L reduziert werden. Mit anderen Worten: Die mittlere vertikale Linie wird von der x-Koordinate $\frac{L}{2}$ auf die x-Koordinate $(a + \frac{L}{2})$ mod L und die mittlere horizontale Linie von der y-Koordinate $\frac{L}{2}$ auf die y-Koordinate $(b + \frac{L}{2})$ mod L verschoben. Der Teil der Dissection, der durch die Verschiebung außerhalb der Bounding-Box gerät, wird dann „herumgewickelt" und erscheint wieder im linken bzw. unteren Teil der B-Box. Nach dem Shift befindet sich die linke Kante der Dissection an der x-Koordiante a und die untere Kante an der y-Koordinate b (Siehe Abbildung 3.3). Wir betrachten ein „herumgewickeltes" Quadrat in der geshifteten Dissection als eine Region, wodurch die Notation stark vereinfacht wird. Der Leser kann sich ein „herumgewickeltes" Quadrat auch als eine disjunkte Vereinigung von 2 bzw. 4 Rechtecken vorstellen.

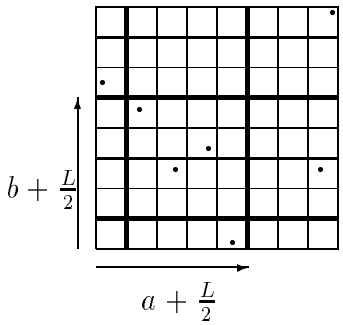


Abbildung 3.3: Die geshiftete Dissection

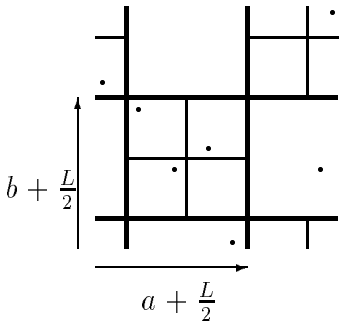


Abbildung 3.4: Der geshiftete Quadtree

Den *Quadtree mit Shift* (a, b) erhält man aus der geshifteten Dissection, indem man die Unterteilungen dort wegläßt, wo ein Quadrat nur noch einen Punkt in sich hat. Wie in Abbildung 3.4 zu sehen ist, hat der geschiftete Quadtree im allgemeinen eine gänzlich andere Struktur als der ursprüngliche. So ist die Tiefe des Quadtrees im Beispiel von 3 auf 2 gesunken. Zur Vereinfachung ist in den Abbildungen 3.3 und 3.4 a gleich b gesetzt worden.

3.2 Beschreibung des Algorithmus

Der Algorithmus zur Bestimmung der approximierten optimalen Rundreise beruht im Kern auf dem unten dargestellten Struktur-Theorem. Es besagt, daß bei zufällig gewähltem Shift (a, b) der Dissection mit einer Wahrscheinlichkeit von mindestens $\frac{1}{2}$ eine $(1 + \frac{1}{c})$-Approximation der optimalen Rundreise existiert, die jede Seite jedes Quadrates in der geshifteten Dissection höchstens $O(c)$-mal durchkreuzt. Genaugenommen enthält das

Struktur-Theorem noch eine stärkere Aussage: Die Durchkreuzungen der Tour mit den Quadratkanten finden an speziellen vorher festgelegten Punkten statt, die *Portale* genannt werden.

Wie kann es aber sein, daß die Schnittpunkte der Rundreise mit den Quadraten schon im Vorfeld der Berechnug bekannt sind ? Der Grund dafür ist, daß wir dem Salesman erlauben, von seiner geradlinigen Tour zwischen den Punkten abzuweichen. Er kann somit scheinbar bedeutungslose Ausflüge zwischen dem Besuch zweier Städte unternehmen. Eine Kante zwischen zwei Punkten ist dann eventuell an mehreren Stellen „geknickt“ bzw. „gebogen“. So eine Rundreise mit gebogenen Kanten wollen wir *Salesman Pfad* nennen; unser Algorithmus berechnet einen solchen Pfad. Sprechen wir im folgenden von einer *Rundreise*, so meinen wir immer eine Rundreise mit direkten Verbindungen. Natürlich können wir am Ende die gebogenen Kanten bei einem Salesman Pfad wieder begradigen, ohne die Kosten des Pfades zu erhöhen. Das liegt an der beim Euklidischen TSP geltenden Dreiecksungleichung. Um das Struktur-Theorem zu formulieren, ist noch eine genaue Begriffsdefinition notwendig:

Definition 3.2.1 Seien $m, r \in \mathbb{N}$. Eine *m-reguläre Menge von Portalen* für eine geshiftete Dissection ist eine Menge von Punkten auf den Kanten der Quadrate in ihr. Jedes Quadrat hat ein Portal in jeder der 4 Ecken und m weitere Portale mit gleichem Abstand auf jeder Seite. Ein Salesman Pfad ist *(m, r)-light* in Bezug auf die geshiftete Dissection, wenn er jede Kante jedes Quadrates höchstens r mal durchkreuzt und jede Durchkreuzung an einem Portal stattfindet.

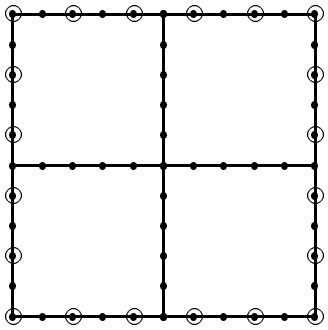


Abbildung 3.5: Verteilung der Portale für $m = 4$

Im folgenden wollen wir die Definition etwas veranschaulichen. Es ist zu beachten, daß bei der Portalvergabe jedes Quadrat jeder Größe für sich betrachtet wird. Nur so kann jede Seite $m + 2$ Portale bekommen. Deckt sich also eine Seite eines kleinen Quadrates mit der eines größeren, so sind nicht alle Portalpunkte der Seite des kleineren Quadrates auch Portalpunkte auf der Seite des größeren. Betrachtet man z.B. die Einbettung eines Quadrates in ein doppelt so großes Quadrat, so kommt nur jeder zweite Portalpunkt zur

Deckung (siehe Abbildung 3.5). Im Beispiel ist die Verteilung der Portalpunkte für $m = 4$ dargestellt. Die Portale des großen Quadrates sind mit $\circ$ gekennzeichnet und die der 4 kleinen mit $\bullet$.

Ein Salesman Pfad der (m, r)-light ist, darf jede Kante jedes Quadrates nur an Portalpunkten kreuzen. Die Portale können dabei eventuell öfters durchlaufen werden. Nach der Definition können bei Kanten, die zu verschieden großen Quadraten gehören, nicht alle Portalpunkte der kleineren Quadrate für den Pfad gebraucht werden, da diese nicht alle auch Portalpunkte auf den Kanten eines großen Quadrates sind. Im Beispiel darf ein Pfad die Kante des äußeren Quadrates nur durch $\circ$-Punkte kreuzen. Es werden also nicht alle $\bullet$-Punkte auf den äußeren Kanten gebraucht. Weiter bedeutet (m, r)-light, daß eine Kante des äußeren Quadrates maximal r-mal geschnitten werden darf. Das heißt dann für die zwei darauf liegenden Kanten der kleineren Quadrate, daß diese *zusammen* nur r-mal geschnitten werden dürfen. Wir kommen darauf beim Beweis des Struktur-Theorems zurück.

Theorem 3.2.2 (Struktur-Theorem) *Sei $c > 2$ eine reelle Zahl. Sei der Abstand zweier Punkte in einer TSP-Eingabe I entweder mindestens 4 oder genau 0 und sei L die Größe der Bounding-Box der Eingabepunkte. Weiter seien $0 \leq a, b < L$ zufällig gewählte ganzzahlige Shifts. Dann gibt es Zahlen $m \in O(c \log L)$ und $r \in O(c)$, so daß mit Wahrscheinlichkeit mindestens $\frac{1}{2}$ in der Dissection mit Shift (a, b) ein zugehöriger (m, r)-light Salesman Pfad existiert, der höchstens Kosten $\left(1 + \frac{1}{c}\right) opt_I$ hat.*

Im folgenden nehmen wir an, daß das Struktur-Theorem, das wir später beweisen, der Wahrheit entspricht. Es bildet die Grundlage für den folgenden Approximationsalgorithmus, der zur Berechnug der approximierten optimalen Rundreise als Eingabe eine TSP-Instanz und einen Genauigkeitswert $c > 0$ erhält. Ist $c \leq 2$, so ruft der Algorithmus einfach Christofides' Algorithmus auf und berechnet eine Rundreise der Güte $\frac{3}{2}$ in polynomieller Zeit. Für $c > 2$ geht der Algorithmus wie folgt vor:

1. Normierung der Eingabe.
2. Konstruktion des Quadtrees mit zufällig gewähltem Shift (a, b).
3. Bestimmung eines optimalen (m, r)-light Salesman Pfades im geshifteten Quadtree.

Der berechnete optimale (m, r)-light Salesman Pfad im geshifteten Quadtree ist mit Sicherheit nicht länger als ein optimaler (m, r)-light Salesman Pfad in der geshifteten Dissection. Dieser wiederum ist nach Aussage des Struktur-Theorems mit Wahrscheinlichkeit mindestens $\frac{1}{2}$ eine $\left(1 + \frac{1}{c}\right)$-Approximation des Optimums, so daß wir im Schnitt bei jeder zweiten Wahl des Shifts (a, b) eine $\left(1 + \frac{1}{c}\right)$-Approximation erhalten. Wählen wir für den Shift (a, b) jeden möglichen Shift und berechnen jeweils den optimalen Salesman Pfad, so können wir am Ende den kürzesten von allen Salesman Pfaden als Lösung ausgeben. Dieser ist dann mit Sicherheit eine $\left(1 + \frac{1}{c}\right)$-Approximation des Optimums. Wir werden in den nächsten Abschnitten die drei Schritte des Algorithmus genauer beschreiben und auch zeigen, daß er wirklich nur polynomielle Laufzeit hat.

3.2.1 Normierung der Eingabedaten

Um unter anderem die Voraussetzungen für die Anwendbarkeit des Struktur-Theorems zu schaffen, werden die Eingabepunkte vom Algorithmus zu Anfang normiert. Dies bedeutet, daß

i) alle Punkte ganzzahlige Koordinaten haben,

ii) der Abstand zweier Punkte entweder genau 0 oder mindestens 4 ist und

iii) der maximale Punkteabstand $O(n)$ ist.

Die Eingabe I wird wie folgt normiert. Sei dazu L_0 die Größe der exakten Bounding-Box der gegebenen Eingabepunkte und opt_I die Kosten einer optimalen Rundreise. Der Ursprung unseres Koordinatensystems liegt auf der unteren linken Ecke der B-Box. Es gilt offensichtlich, daß $opt_I \geq 2L_0$ ist, da die Distanz zwischen den Punkten, die die Größe der Bounding-Box festlegen, mindestens L_0 ist und bei einer Rundreise diese Distanz mindestens 2 mal durchlaufen werden muß. Als nächstes legen wir ein Gitter mit der Auflösung $d := \frac{L_0}{\sqrt{2}nc}$ in die Ebene und bewegen jeden Punkt zu einem am nächsten liegenden Gitterpunkt (siehe Abbildung 3.6). Es kann dabei durchaus vorkommen, daß mehrere Punkte auf den gleichen Gitterpunkt bewegt werden.

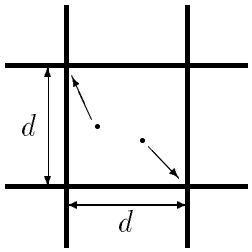


Abbildung 3.6:

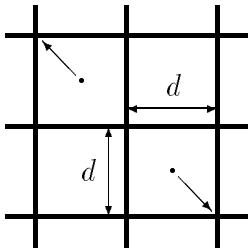


Abbildung 3.7:

Im nächsten Schritt werden die Koordinaten mit dem Faktor $\frac{4}{d}$ multipliziert. Wir erreichen dadurch, daß die Punkte mindestens Abstand 4 haben, falls sie vorher nicht auf dem gleichen Gitterpunkt gelandet sind und somit Abstand 0 besitzen. Die gerade beschriebene Verschiebung der Punkte $p = (x, y)$ auf die Gitterpunkte bekommen wir, indem wir die x und y-Koordinaten der Punkte auf das nächstkleinere bzw. nächstgrößere Vielfache von d verschieben. Das können wir durch die Ersetzung $(x, y) := \left(d \left\lfloor \frac{x}{d} + \frac{1}{2} \right\rfloor, d \left\lfloor \frac{y}{d} + \frac{1}{2} \right\rfloor\right)$ erreichen. Die anschließende Skalierung mit $\frac{4}{d}$ führt dann zu der Transformation t der Punkte:

$$t(p) \;=\; t(x,y) \;=\; \left(4 \cdot \left\lfloor \frac{x}{d} + \frac{1}{2} \right\rfloor, 4 \cdot \left\lfloor \frac{y}{d} + \frac{1}{2} \right\rfloor\right)$$

Wie in obiger Gleichung zu sehen ist, sind nach der Transformation die Koordinaten der Punkte ganzzahlige Vielfache von 4. Die Bounding-Box wird durch die Verschiebung der Punkte auf der rechten und oberen Seite um maximal $\frac{d}{2}$ erweitert. Die Skalierung führt dann zur neuen Bounding-Box Größe L:

$$L \leq \left(L_0 + \frac{d}{2}\right) \cdot \frac{4}{d} = 4\sqrt{2}nc + 2 \in O(nc) = O(n)$$

Damit sind die Bedingungen i), ii) und iii) für die Normierung erfüllt. Es bleibt noch zu untersuchen, wie sich die Kosten einer Rundreise durch die Normierung verändern. Sei dazu A unser Approximationsalgorithmus, der die Punktemenge I und die Genauigkeitsanforderung c als Eingabe bekommt und daraus eine Rundreise R konstruiert, deren Kosten $cost_R$ kleiner gleich $\left(1 + \frac{1}{c}\right) opt_I$ sind. Der Algorithmus A benutzt dabei den eigentlichen Algorithmus A' als Unterprogramm. A geht dabei wie folgt vor:

1. Transformation der Eingabe (I, c) zu der normierten Eingabe (I', c').
2. Starten des eigentlichen Algorithmus auf (I', c') der die Ausgabe R' mit Kosten $cost_{R'}$ produziert.
3. Bestimmung der Ausgabe R aus R'.

Als erstes betrachten wir den Teil der Transformation, der die Verschiebung der Punkte auf die Gitterpunkte beinhaltet. Es wird dabei ein Punkt um maximal $\frac{d}{\sqrt{2}}$ verschoben. Werden zwei verbundene Punkte genau um diese Distanz in entgegengesetzter Richtung verschoben, so erhöht sich ihr Abstand um $\sqrt{2}d$ (siehe Abbildung 3.7). Da eine Rundreise aus n Kanten besteht, ist die durch die Verschiebung hervorgerufene Kostenerhöhung k durch $\sqrt{2}dn$ beschränkt. Es gilt:

$$k < \sqrt{2}dn = \frac{L_0}{c} = \frac{2L_0}{2c} \leq \frac{opt_I}{2c}$$

Für die Kosten der optimalen Rundreise auf I' ergibt sich damit: $opt_{I'} \leq opt_I + \frac{opt_I}{2c}$. Um die Kostenvergrößerung der Verschiebung auszugleichen, muß der Genauigkeitswert c dementsprechend angepaßt werden. Algorithmus A' erhält die Eingabe (I', c') wobei $c' := 2c + 1$ gesetzt werden muß, damit gilt:

$$cost_{R'} \leq \left(1 + \frac{1}{c'}\right) opt_{I'} \leq \left(1 + \frac{1}{2c+1}\right) \cdot \left(opt_I + \frac{opt_I}{2c}\right) = \left(1 + \frac{1}{c}\right) opt_I$$

Betrachten wir nun die Kostenerhöhung, die durch die Skalierung hervorgerufen wird. Durch die Multiplikation der Koordinaten mit dem Faktor $\frac{4}{d}$ werden die Distanzen zwischen den Punkten und somit auch die Kosten einer Rundreise skaliert. Die Reihenfolge,

in der die Punkte in der von A' berechneten Rundreise durchlaufen werden, wird allerdings nicht durch die Skalierung verändert. Algorithmus A kann somit die von A' berechnete Rundreise R' übernehmen.

3.2.2 Konstruktion und Analyse des Quadtrees

Kommen wir jetzt zum zweiten Punkt unseres Algorithmus, der Konstruktion des geshifteten Quadtrees. Wir beschreiben dazu zuerst einen Algorithmus, der uns einen nicht geshifteten Quadtree berechnet. Danach untersuchen wir die Laufzeit und anschließend erklären wir, wie mit dem Algorithmus ein geshifteter Quadtree konstruiert werden kann.

Wir erstellen den Quadtree für die n Eingabepunkte, indem wir die Prozedur `INSERT`, die einem Quadtree einen Punkt hinzufügt, für jeden Punkt anwenden. Ein Knoten im Quadtree enthält dabei ein 5-Tupel $\big((x_u, y_u), l, b, (x_p, y_p), (p_1, p_2, p_3, p_4)\big)$ an Werten, wobei (x_u, y_u) die Koordinaten der linken unteren Ecke des dem Knoten entsprechenden Quadrates sind, l die Ebene ist, in der das Quadrat liegt, b ein „belegt"-Bit ist, das angibt, ob das Quadrat leer ist oder einen Punkt enthält, (x_p, y_p) die Koordinaten eines Punktes sind und (p_1, p_2, p_3, p_4) vier Zeiger auf die Unterquadrate sind. Die Wurzel des Quadtrees wird zu Beginn auf $\big((0,0), 0, FALSE, (0,0), (nil, nil, nil, nil)\big)$ initialisiert.

PROCEDURE `INSERT`(Q, p)

BEGIN
  WHILE Q kein Blatt **DO**
    Q := Unterquadrat von Q, in das p fällt;
  OD
  IF Q belegt **THEN DO**
    q := Punkt, der schon in Q ist;
    IF $q \neq p$ **THEN DO**
      REPEAT
        Erstelle die 4 Unterquadrate zu Q;
        Füge q in das Unterquadrat ein, in das q fällt und markiere es als belegt;
        Q := Unterquadrat von Q, in das p fällt;
      UNTIL p fällt in ein anderes Quadrat als q;
    OD
  OD
  Füge p in Q ein und markiere Q als belegt;
END

Die Prozedur `INSERT` erklärt sich im wesentlichen selbst. Zu begründen ist allein, warum die Größe eines Quadrates in den Knoten nicht gespeichert werden muß zumal sie zur Bestimmung des Unterquadrates, in das p fällt, benötigt wird. Die Größe eines Quadrates ergibt sich nämlich aus der Nummer der Ebene, auf der es liegt. Ist L die Größe der Bounding-Box so hat ein Quadrat auf Ebene l die Größe $\frac{L}{2^l}$.

Welche Zeit benötigt der Algorithmus nun, um den Quadtree zu konstruieren ? Sei t die Tiefe des resultierenden Quadtrees. Ein Aufruf von `INSERT` verursacht maximal Kosten $O(t)$ und da `INSERT` n mal aufgerufen wird, betragen die Gesamtkosten folglich $O(n \cdot t)$. Die Tiefe des Quadtrees hängt offensichtlich davon ab, wie oft die Quadrate aufgeteilt werden müssen, also davon, wie nahe die Punkte beieinander liegen. Das folgende Lemma gibt uns darüber genau Auskunft.

Lemma 3.2.3 *Sei eine Menge mit Punkten aus $\mathbb{R}^2$, die paarweise mindestens Abstand d haben und deren Bounding-Box Größe L hat, gegeben. Dann hat die daraus gebildete Dissection und auch der Quadtree maximal Tiefe $\lceil \log \frac{L}{d} - \frac{1}{2} \rceil$.*

Beweis: Die Frage ist, wie klein ein Quadrat in der Dissection bzw. im Quadtree im ungünstigsten Fall werden kann; denn je kleiner die Quadrate werden, desto tiefer wird auch der Baum. Da die Punkte mindestens Abstand d haben, werden sie spätestens dann voneinander separiert, wenn die Quadrate Größe $\frac{d}{\sqrt{2}}$ haben. Das liegt daran, daß die längste Strecke im Quadrat mit Kantenlänge $\frac{d}{\sqrt{2}}$, nämlich die Hauptdiagonale, Länge $\sqrt{\left(\frac{d}{\sqrt{2}}\right)^2 + \left(\frac{d}{\sqrt{2}}\right)^2} = d$ hat, was gerade der minimale Punkteabstand ist.

Eine weitere Beobachtung ist, daß mit steigender Ebenenzahl die Größe der Quadrate jeweils halbiert wird. Auf den Ebenen $0, 1, 2, 3, ..., i, ...$ hat man also die Quadratgrößen $L, \frac{L}{2}, \frac{L}{4}, \frac{L}{8}, ..., \frac{L}{2^i}, ...$ Gesucht ist die Ebene i, ab der die Größe der Quadrate kleiner oder gleich $\frac{d}{\sqrt{2}}$ wird. Dies führt zu folgender Ungleichung :

$$\frac{L}{2^i} \leq \frac{d}{\sqrt{2}} \iff 2^i \geq \sqrt{2}\frac{L}{d} \iff i \geq \log\left(\sqrt{2}\frac{L}{d}\right) = \frac{1}{2} + \log\frac{L}{d}$$

Es gilt also, daß ab Ebene $\lceil \frac{1}{2} + \log \frac{L}{d} \rceil$ die Größe der Quadrate kleiner oder gleich $\frac{d}{\sqrt{2}}$ ist. Die Tiefe ist die größte Nummer der Ebene, die noch belegt ist, also $\lceil \log \frac{L}{d} - \frac{1}{2} \rceil$. □

Durch die Normierung der TSP-Eingabedaten wird erreicht, daß der minimale Abstand zweier Punkte 4 ist und die Größe der Bounding-Box linear in n beschränkt ist. Es ist also $t \in O(\log n)$. Eine obere Zeitschranke für die Konstruktion des Quadtrees ist somit $O(n \log n)$. Fügen wir nur Punktepaare, die jeweils Abstand d haben, in den Quadtree ein, so muß mindestens jeder zweite `INSERT`-Aufruf zur tiefsten Ebene laufen. Es gibt also Eingabeinstanzen, durch die die obere Laufzeitschranke erreicht wird und damit ist die Laufzeit von `INSERT` in $\Theta(n \log n)$.

Der im nächsten Abschnitt beschriebene Algorithmus zur Bestimmung des optimalen (m, r)-light Salesman Pfades muß, wie wir später sehen werden, die Quadrate im Quadtree ebenenweise bearbeiten können. Wir müssen ihm dazu ermöglichen, auf die Quadrate einer Ebene nacheinander zugreifen zu können. Dazu legen wir für jede Ebene eine Liste an, die die Quadrate der Ebene enthält. Jedesmal, wenn `INSERT` vier neue Quadrate erzeugt, fügt es diese in die Liste der zugehörigen Ebene ein. Wie das nächste Lemma zeigt, befinden sich auf einer Ebene höchstens $2n$ Quadrate. Die Listen können also als Array der Größe

$2n$ realisiert werden und ein Listenelement kann per Direktzugriff in Zeit $O(1)$ eingetragen bzw. ausgelesen werden.

Lemma 3.2.4 *Ein Quadtree, der aus einer Punktemenge mit n Punkten gebildet wird, enthält in jeder Ebene höchstens $2n$ Quadrate.*

Beweis: Eine erste Beobachtung ist, daß die Anzahl der Quadrate auf jeder Ebene (bis auf Ebene 0) ein Vielfaches von 4 ist, da immer 4 Quadrate im Quadtree auf einmal erzeugt werden. Für gerades n gibt es bei der Aussage des Lemmas dabei keine Probleme, da $2n$ dann durch 4 teilbar ist. Damit die Aussage des Lemmas für ungerades n Gültigkeit behält, müssen wir aber dann beweisen, daß maximal $2(n-1)$ Quadrate auf einer Ebene sein können, da das die nächstmögliche, durch 4 teilbare Zahl kleiner als $2n$ ist.

Betrachten wir jetzt die Quadrate einer beliebigen Ebene größer 0. Diese lassen sich zu 4-Tupeln so zusammenfassen, daß immer 4 Quadrate denselben Vater haben. Wir wollen jetzt zeigen, daß in jedem dieser 4-Tupel mindestens zwei Punkte sein müssen. Dann wissen wir, da wir n Punkte gegeben haben, daß es nur $\lfloor \frac{n}{2} \rfloor$ 4-Tupel in jeder Ebene geben kann. Für gerades n sind das maximal $2n$ Quadrate und für ungerades n maximal $4 \cdot \lfloor \frac{n}{2} \rfloor = 2(n-1)$ Quadrate.

Besteht ein 4-Tupel nur aus Blättern, so müssen mindestens zwei davon einen Punkt enthalten, denn sonst wäre das 4-Tupel bei der Erstellung des Quadtrees nicht entstanden. Enthält ein 4-Tupel ein Quadrat, das kein Blatt ist, so muß dieses schon alleine zwei Punkte enthalten. Die anderen drei Quadrate können dann allerdings leere Blätter sein. Insgesamt haben wir aber in jedem 4-Tupel mindestens zwei Punkte. Mit obiger Schlußfolgerung ist die Aussage dann bewiesen. □

Wir wollen jetzt noch ein Resultat darstellen, das zeigt, daß die Prozedur `INSERT` eine bestmögliche Prozedur zur Konstruktion eines Quadtrees ist. Um einen Quadtree zu erstellen, muß jeder Knoten im Quadtree erzeugt werden. Es ist also mindestens für jedes Quadrat im resultierenden Quadtree ein Schritt nötig. Der folgende Satz besagt, daß die Anzahl der Quadrate im Quadtree für $L \in O(n)$ und $d = 4$ zur Menge $\Theta(n \log n)$ gehört und damit ist die Prozedur `INSERT` optimal.

Satz 3.2.5
(1) Die Anzahl der Quadrate eines Quadtrees, der aus einer Punktemenge mit n Punkten gebildet wird, die paarweise mindestens Abstand d haben und deren Bounding-Box Größe L hat, ist nach oben beschränkt durch :

$$2n \log \frac{L}{d} - n \log n + \frac{14}{3} n - \frac{1}{3}$$

(2) Es gibt Punktemengen, so daß die Anzahl der Quadrate im zugehörigen Quadtree die Schranke $\Theta(2n \log L - n \log n + n)$ erreicht.

Beweis: Als erstes ist zu bemerken, daß die Größe der Bounding-Box L und die Anzahl der Punkte n für eine sinnvolle Aussage des Satzes nicht zusammenhangslos sein können.

Es muß $n \leq \left(\frac{L}{d}\right)^2$ sein, da mehr Punkte in die B-Box einfach nicht „hineinpassen“. Den Mindestabstand d zweier Punkte wollen wir im folgenden als konstant gegeben annehmen.

Wir führen jetzt den Begriff der *Kapazität einer Ebene* im Quadtree ein. Die Kapazität einer Ebene soll die maximal mögliche Anzahl der Quadrate auf einer Ebene sein, die durch die Struktur des Quadtrees begrenzt ist. So haben die Ebenen $0, 1, 2, 3, ..., i$ Kapazitäten $1, 4, 16, 64, ..., 4^i$. Es gilt also für die Kapazität $K(i)$ auf Ebene i: $K(i) = 4^i$. Unter einem *maximalen Quadtree* wollen wir einen Quadtree verstehen, dessen Anzahl an Quadraten maximal ist.

Wie sieht nun ein maximaler Quadtree aus ? Anders ausgedrückt lautet die Frage: Wie müssen die n Punkte verteilt sein, damit im Quadtree möglichst viele Quadrate entstehen ? Offensichtlich kann die Anzahl der Quadrate auf einer Ebene nicht die Kapazität der Ebene überschreiten. Weiter wissen wir wegen Lemma 3.2.4, daß die Anzahl der Quadrate pro Ebene durch $2n$ begrenzt ist. Ein Quadtree, auf dessen Ebenen $i = 0, 1, ..., \lceil \log \frac{L}{d} - \frac{1}{2} \rceil$ sich $\min(K(i), 2n)$ Quadrate befänden, wäre somit ein maximaler Quadtree. Ist es möglich, die n Punkte so zu verteilen, daß ein derartiger Quadtree entsteht ? Wollen wir erreichen, daß auf den unteren Ebenen solange wie möglich $2n$ Quadrate pro Ebene erzeugt werden, müssen auf der untersten Ebene immer 2 von 4 Blättern jeweils einen Punkt enthalten, was voraussetzt, daß n gerade ist; auf den darüberliegenden Ebenen müssen immer 3 von 4 Quadraten leere Blätter sein (siehe Lemma 3.2.4).

Als Beispiel siehe Abbildung 3.8, in der $L = 8$, $n = 8$ und $d = \sqrt{2}$ gewählt sind. Wir haben $d = \sqrt{2}$ gewählt, damit das kleinste Quadrat im Quadtree Kantenlänge 1 hat und dadurch die Darstellung vereinfacht wird. Es gibt dort die Ebenen $0, 1, 2, 3$ mit den Kapazitäten $1, 4, 16, 64$. Im Beispiel ist $2n = 16$. Die Ebenen 2 und 3 enthalten 16 Quadrate. Auf den Ebenen 0 und 1 ist mit $2n$ die Kapazität der Ebene überschritten. Sie enthalten $K(0) = 1$ bzw. $K(1) = 4$ Quadrate. Der Quadtree ist also maximal.

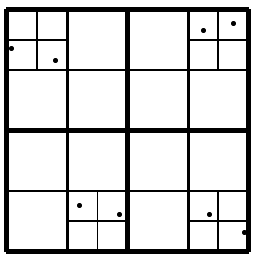


Abbildung 3.8: Maximaler Quadtree

Sei M die maximale Anzahl an Quadraten im Quadtree. Um sie zu bestimmen, müssen wir die maximale Anzahl Quadrate pro Ebene über alle Ebenen aufsummieren. Nach obiger Argumentation steht in der Gleichung genau dann das Gleichheitszeichen, wenn n gerade ist. Es gilt:

$$M \leq \sum_{i=0}^{\lceil \log \frac{L}{d} - \frac{1}{2} \rceil} \min(K(i), 2n)$$

In der entstandenen Summe wird jeweils $K(i) = 4^i$ aufsummiert, solange $4^i \leq 2n$ ist. Danach wird immer nur noch $2n$ hinzugezählt. Um die Summe in zwei Teile zu zerlegen, müssen wir den Index i bestimmen, bis zu dem 4^i aufsummiert wird:

$$4^i \leq 2n \iff 2^{2i} \leq 2n \iff i \leq \frac{1}{2} \log 2n$$

Da der Index i ganzzahlig sein muß, gilt: $i = \lfloor \frac{1}{2} \log 2n \rfloor$. Jetzt können wir die obige Summe in zwei Teile zerlegen und nach oben abschätzen:

$$\begin{aligned}
M &\leq \sum_{i=0}^{\lfloor \frac{1}{2} \log 2n \rfloor} 4^i + 2n \cdot \left(\left\lceil \log \frac{L}{d} - \frac{1}{2} \right\rceil - \left\lfloor \frac{1}{2} \log 2n \right\rfloor \right) \\
&= \frac{4^{\lfloor \frac{1}{2} \log 2n \rfloor + 1} - 1}{3} + 2n \cdot \left(\left\lceil \log \frac{L}{d} - \frac{1}{2} \right\rceil - \left\lfloor \frac{1}{2} \log 2n \right\rfloor \right) \\
&\leq \frac{4 \cdot 4^{\frac{1}{2} \log 2n} - 1}{3} + 2n \cdot \left(\log \frac{L}{d} - \frac{1}{2} + 1 - \frac{1}{2} \log 2n + 1 \right) \\
&= \frac{8}{3} n - \frac{1}{3} + 3n + 2n \log \frac{L}{d} - n \log 2n \\
&= 2n \log \frac{L}{d} - n \log n + \frac{14}{3} n - \frac{1}{3}
\end{aligned}$$

Der zweite Schritt in der Rechnung gebraucht die geometrische Summenformel, die im Anhang in Satz A.1.1 dargestellt ist. Ist L von der Form $L = d2^{a - \frac{1}{2}}$ und n von der Form $n = \frac{1}{2} 4^b$ für $a, b \in \mathbb{N}$, dann fallen die Rundungsklammern in obiger Abschätzung weg und es entstehen nur Gleichheitsbeziehungen. Die Rechnung führt dann zu genau

$$2n \log \frac{L}{d} - n \log n + \frac{8}{3} n - \frac{1}{3} \quad \in \quad \Theta(2n \log L - n \log n + n)$$

Quadraten. Ein Beispiel dafür ist der Quadtree in Abbildung 3.8, in dem $a = 3$ und $b = 2$ gesetzt sind. Die Anzahl der Quadrate ergibt sich dort mit 37. □

Beschäftigen wir uns jetzt mit der Konstruktion des geshifteten Quadtrees. Sei dazu wieder L die Größe der Bounding-Box der Eingabepunkte. Der geshiftete Quadtree entsteht aus der geshifteten Dissection durch Weglassen der Unterteilungen in den Quadraten, in denen die Punkte schon separiert sind. Es ist also nicht gleichbedeutend, den

normal konstruierten Quadtree einfach um (a, b) nach rechts oben zu shiften, da dadurch eventuell mehrere Punkte in einem Quadrat landen. Anstatt den nach rechts oben geshifteten Quadtree zu berechnen, können wir aber den nicht geshifteten Quadtree auf den nach links unten geshifteten Eingabepunkten konstruieren. Wir erhalten so genau die Quadrataufteilung des nach rechts oben geshifteten Quadtrees. Shiften wir den so erhaltenen Quadtree mit den Eingabepunkten wieder nach rechts oben, so erhalten wir den geshifteten Quadtree auf den ursprünglichen Eingabepunkten. Das Shiften der Eingabepunkte benötigt offensichtlich $O(n)$ Zeit und das Shiften des Quadtrees wird beim späteren Bearbeiten eines Quadrates realisiert. Der Quadtree wird also nichtgeshiftet gespeichert. Immer dann, wenn ein Quadrat ausgelesen wird, wird dieses geshiftet und in bis zu vier Rechtecke aufgeteilt.

3.2.3 Das Hauptprogramm des Algorithmus

Kommen wir jetzt zum eigentlichen Algorithmus, der den optimalen (m, r)-light Salesman Pfad im geshifteten Quadtree bestimmt. Der Algorithmus ist deterministisch und benutzt dynamische Programmierung zur Bestimmung des Pfades. Wir werden zeigen, daß seine Laufzeit $O\left(n^3(\log n)^{O(c)}\right)$ ist, unter der Annahme, daß $c \leq \log n$ ist. Diese Annahme stellt im Zusammenhang mit einem PTAS keine Einschränkung dar, da für Eingaben, bei denen $\log n < c$, also $n < 2^c$ ist, alle Permutationen der Eingabepunkte durchprobiert werden können, um eine Lösung zu finden, die eine $(1 + \frac{1}{c})$-Approximation ist. Wir zählen also alle möglichen Rundreisen auf und berechnen die kürzeste bezüglich der aufgerundeten Distanzen. Sei R eine optimale Rundreise und $\widetilde{R}$ die kürzeste Rundreise bezüglich der aufgerundeten Distanzen. Genau wie in Kapitel 2 bei der Beschreibung der Eingabedaten erhalten wir hier:

$$cost'_{\widetilde{R}} \;\leq\; cost'_R \;\leq\; cost_R + \frac{1}{2c} \;=\; opt + \frac{1}{2c} \;\leq\; \left(1 + \frac{1}{2c}\right) opt \;<\; \left(1 + \frac{1}{c}\right) opt$$

Die Zeit, die benötigt wird, alle möglichen Rundreisen aufzuzählen, ist zwar exponentiell in c, aber eine Laufzeit exponentiell in c ist bei einem PTAS erlaubt.

Der Algorithmus beruht auf folgender Beobachtung: Angenommen, Q ist ein Quadrat im geshifteten Quadtree und der gesuchte optimale (m, r)-light Salesman Pfad π durchkreuzt die Umrandung von Q insgesamt $2p \leq 4r$ mal. Sei $a_1, a_2, ..., a_{2p}$ die Sequenz der Portale, an denen diese Durchkreuzungen stattfinden. Die Portale sind in der Reihenfolge durchnumeriert, in der sie von π geschnitten werden. Der Teil des optimalen Salesman Pfades π im Quadrat Q ist dann eine Sequenz von p Pfaden, für die gilt:

(i) Für $i = 1, 2, ...p$ verbindet der i-te Pfad die Portale a_{2i-1} und a_{2i}.

(ii) Alle p Pfade zusammen besuchen jeden Punkt, der in Q liegt.

(iii) Die Zusammenfassung der p Pfade ist (m,r)-light, d. h., daß sie zusammen jede Kante jedes Unterquadrates in Q nur maximal r-mal schneiden und diese Schnitte nur an Portalen stattfinden.

Da der Salesman Pfad π optimal ist, ist die oben beschriebene Sequenz von p Pfaden kostenminimal unter allen Sequenzen mit p Pfaden, die die Bedingungen (i), (ii) und (iii) erfüllen. Diese Beobachtung motiviert uns, ein (m,r)-*Multitour Problem* zu definieren. Eine Eingabeinstanz für dieses Problem ist gegeben durch:

(a) Ein nichtleeres Quadrat im geshifteten Quadtree.

(b) Eine Multimenge mit $\leq r$ Portalen auf jeder der vier Seiten dieses Quadrates, so daß die Summe der Mächtigkeiten der vier Multimengen eine gerade Zahl $2p \leq 4r$ ist.

(c) Eine Paarung $(a_1, a_2), (a_3, a_4), ..., (a_{2p-1}, a_{2p})$ der $2p$ Portale, die in (b) angegeben sind.

Das Ziel des (m,r)-Multitour Problems ist es, p Pfade in dem Quadrate zu finden, die zusammen (m,r)-light und kostenminimal sind. Der i-te Pfad soll dabei die Portale a_{2i-1} und a_{2i}, $1 \leq i \leq p$, verbinden und alle p Pfade zusammen sollen jeden Punkt in dem Quadrat besuchen. Für $p = 0$ ist das Ziel, einen optimalen (m,r)-light Salesman Pfad für die Punkte in dem Quadrat zu finden. Das (m,r)-Multitour Problem kann auch als ein Multiple Traveling Salesman Problem aufgefaßt werden, in dem ein Team von Salesmen eine Menge von Kunden besuchen muß. Jeder Kunde muß dabei von einem Salesman besucht werden und jeder Salesman hat einen festgelegten Start- und Endpunkt.

Der gesuchte optimale (m,r)-light Salesman Pfad ist offensichtlich die Lösung für das (m,r)-Multitour Problem mit der Bounding-Box der Eingabepunkte und $p = 0$ als Eingabe. Wie aber können wir dieses (m,r)-Multitour Problem lösen? Im folgenden ist es unser Ziel, die Lösung für ein (m,r)-Multitour Problem für ein Quadrat auf Ebene i aus den Lösungen der Probleme für Quadrate auf Ebene $i+1$ zu berechnen. Der Algorithmus speichert dazu in jedem Knoten des Quadtrees zusätzlich einen Zeiger auf ein Array, in dem die Kosten der optimalen Lösungen zu *allen* Eingabeinstanzen des (m,r)-Multitour Problems für das zugehörige Quadrat stehen. Die möglichen Eingabeinstanzen werden dazu sortiert aufgezählt und in dem Kostenarray stehen dann an Stelle k die Kosten der optimalen Lösung der k-ten Eingabeinstanz I_k. Die Eingabeinstanz $I_0 := \{\}$ stellt dabei die leere Eingabe dar. Wie wir im letzten Abschnitt gesehen haben, ist es möglich, die Quadrate des Quadtrees ebenenweise auszulesen. Die Reihenfolge, in der die Quadrate auf der Ebene angeordnet sind hängt davon ab, in welcher Reihenfolge die Quadrate bei der Konstruktion des Quadtrees erstellt wurden, also in welcher Reihenfolge die Punkte in den Quadtree eingefügt wurden. Diese Reihenfolge ist beliebig, also auch die der Quadrate auf einer Ebene. Für den Algorithmus spielt das aber keine Rolle. Wir wollen mit $Q(I_k)$ die Kosten der optimalen Lösung des (m,r)-Multitour Problems für das Quadrat Q bei Eingabe I_k bezeichnen.

Bei der Erstellung des Quadtrees wird der Zeiger auf das Kostenfeld zunächst auf *nil* gesetzt. Die Prozedur `INSERT` ändert sich dadurch nur unwesentlich. Die Kostenfelder der Quadrate werden durch den Algorithmus jetzt von der untersten Ebene an aufsteigend beschrieben. Sind wir bei der Wurzel angekommen, dann ist der Algorithmus fertig, denn die Kosten des optimalen (m,r)-light Salesman Pfades können bei $Q(\{\})$ abgelesen werden, wobei Q die Wurzel des Quadtrees ist.

Wir wollen jetzt beschreiben, wie die Kostenfelder „bottom-up" ausgefüllt werden. Die (m,r)-Multitour Probleme für Quadrate auf der tiefsten Ebene im Quadtree lassen sich leicht lösen. Diese Quadrate sind nämlich Blätter und enthalten somit keinen oder nur einen Punkt. Enthält ein Quadrat keinen Punkt, so sind die Kosten des (m,r)-Multitour Problems minimal, falls die p Pfade jeweils von ihrem Eintrittsportal direkt zu ihrem Austrittsportal laufen. Um die Kosten zu bestimmen, müssen also die Distanzen zwischen allen p Portalpaaren aufsummiert werden. Da es maximal $2r$ Portalpaare gibt, sind dafür maximal $2r \in O(c)$ Schritte nötig. Enthält ein Quadrat genau einen Punkt, so muß dieser Punkt probeweise auf jeden Pfad gesetzt werden, um festzustellen, welcher Pfad durch Aufnahme des Punktes die geringsten Kosten verursacht. Es müssen dann genau $2r \cdot (2r+1) \in O(c^2)$ Distanzen berechnet werden. Nehmen wir jetzt an, der Algorithmus hat alle (m,r)-Multitour Probleme für die Quadrate auf den Ebenen größer i gelöst und sei Q ein Quadrat auf Ebene i, das kein Blatt ist. Seien Q_1, Q_2, Q_3, Q_4 die vier Unterquadrate von Q. Für jede Wahl der Eingaben (b) und (c) für Q zählt der Algorithmus nun alle Möglichkeiten auf, in denen eine (m,r)-Multitour die vier inneren Kanten von $Q_1, ..., Q_4$ durchkreuzen kann. Das umfaßt das Aufzählen aller folgenden Möglichkeiten:

(a') Eine Multimenge mit $\leq r$ Portalen auf jeder der vier inneren Seiten der Unterquadrate von Q.

(b') Eine Reihenfolge, in der die Portale, die durch (a') festgelegt sind, von den Pfaden, die durch (b) und (c) festgelegt sind, durchkreuzt werden.

Die Wahlen von (a') und (b') führen zu (m,r)-Multitour Problemen in den vier Unterquadraten. Es führen allerdings nicht alle Wahlen dazu, sondern nur solche die überhaupt gültige Eingaben für die Multitour Probleme darstellen. Addieren wir die minimalen Kosten für die (m,r)-Multitour Probleme in den Unterquadraten $Q_1, ..., Q_4$, so erhalten wir die minimalen Kosten für eine Wahl von (a') und (b'). Das führen wir dann für jede Wahl von (a') und (b') durch und tragen die minimalen gefundenen Kosten in das Quadrat Q an der Stelle ein, die der Eingabeinstanz entspricht, die durch die Wahl von (b) und (c) festgelegt ist.

Der Algorithmus zur Bestimmung des optimalen (m,r)-light Salesman Pfades ist im folgenden dargestellt. Er bekommt als Eingabe eine Liste P mit Punkten und eine Genauigkeitsanforderung c. Als Ausgabe liefert er in der Variable *opt* die Kosten des optimalen Salesman Pfades. Wir haben den Algorithmus in ein Hauptprogramm `ApproxETSP` und eine Unterroutine `CalcCost` zerlegt, damit die Darstellung übersichtlicher wird. Die

Unterroutine löst dabei das Multitour Problem für ein festes Quadrat sowie eine feste Eingabeinstanz.

PROCEDURE `ApproxETSP` (P : Koordinatenliste; c, opt : real);

BEGIN
  `Normierung`(P);
  L := Größe der Bounding-Box um P;
  $opt := \infty$;
  FOR ALL Shifts $0 \leq a, b < L$ **DO**
    `CalcQuadtree`(P, a, b, QT);
    FOR ALL Level $i = depth(QT)$ **DOWNTO** 0 **DO**
      FOR ALL Quadrate j auf Level i **DO**
        Q := Quadrat j auf Ebene i;
        FOR ALL Eingabeinstanzen $I_k = \{(a_1, a_2), (a_3, a_4), ..., (a_{2p-1}, a_{2p})\}$ des (m, r)-Multitour-Problems in Q **DO**
          `CalcCost`(Q, I_k);
        OD
      OD
    OD
    $opt := \min(opt, Q(\{\}))$; {* leere Eingabeinstanz bei der Wurzel *}
  OD
END

Bisher haben wir nicht erörtert, wie der Algorithmus sich bei Quadraten verhält, die durch den Shift (a, b) „herumgewickelt" sind. Das (m, r)-Multitour Problem in diesen Quadraten ist einfacher zu lösen als in „normalen" Quadraten, da ein Pfad der Multitour sich nur in einem der zwei oder vier Teile des Quadrates befinden kann. Es brauchen hier also nicht alle möglichen Wahlen von (b) und (c) aufgezählt werden, sondern nur die, die zu gültigen Eingabeinstanzen führen.

So wie wir den Algorithmus bis jetzt beschrieben haben, erhalten wir am Ende nur die Kosten des optimalen (m, r)-light Salesman Pfades, aber nicht den Pfad selbst. Speichern wir in jedem Knoten des Quadtrees zusätzlich zu den Kosten einer Multitour auch noch die Indizes der vier Eingabeinstanzen, die zu minimalen Lösungen für die Multitour Probleme in den Unterquadraten führten, so können wir am Ende den Salesman Pfad konstruieren, indem wir den Quadtree von der Wurzel startend entlang der gespeicherten Entscheidungsindizes bis zu den Blättern durchlaufen und dort die den Indizes entsprechenden Lösungen in die Quadrate eintragen.

Wie zu Anfang schon erwähnt, können wir aus dem optimalen (m, r)-light Salesman Pfad eine Salesman Tour machen, indem wir die gebogenen Kanten begradigen. Aufgrund der Dreiecksungleichung entsteht dadurch keine Kostenerhöhung. Es ist möglich, daß die resultierende Rundreise sich selbst schneidet. In [19], Seite 163, ist beschrieben, wie diese Schnitte ohne Kostenerhöhung beseitigt werden können.

PROCEDURE `CalcCost` (Q : Quadrat; I_k : Eingabeinstanz);

{* Berechnet die minimalen Kosten für das (m,r)-Multitour Problems in Q *}
{* bei Eingabeinstanz $I_k = \{(a_1,a_2),...,(a_{2p-1},a_{2p})\}$ und speichert sie in $Q(I_k)$. *}
{* $d(x,y)$:= Euklidischer Abstand von Punkt x und Punkt y. *}

BEGIN
IF Q Blatt **THEN** {* Q enthält keinen oder einen Punkt *}
IF Q nicht belegt **THEN** $Q(I_k) := \sum_{i=1}^{p} d(a_{2i-1}, a_{2i})$
ELSE DO
q := Punkt im Quadrat Q;
$$Q(I_k) := \min_{l=1}^{p} \left(d(a_{2l-1},q) + d(q,a_{2l}) + \sum_{\substack{i=1 \\ i\neq l}}^{p} d(a_{2i-1},a_{2i}) \right);$$
OD
ELSE DO {* Q ist kein Blatt *}
Seien Q_1, Q_2, Q_3, Q_4 die Nachfolger von Q im Quadtree;
$Q(I_k) := \infty$;
FOR ALL Eingabeinstanzen $I_{k_1}, I_{k_2}, I_{k_3}, I_{k_4}$ der (m,r)-Multitour-Probleme in $Q_1,...,Q_4$, die sich mit I_k decken **DO**
$$Q(I_k) := \min\left(\sum_{i=1}^{4} Q_i(I_{k_i}), Q(I_k) \right);$$
OD
OD
END

Kommen wir jetzt zur Laufzeitanalyse von `ApproxETSP`. Die erste Schleife läuft über alle Shifts (a,b) und deren Anzahl beträgt $L^2 \in O(n^2)$. Die zweite Schleife läuft über alle $O(\log n)$ Ebenen und die dritte über alle Quadrate auf einer Ebene und das sind maximal $2n$ (siehe Lemma 3.2.4). Es müssen also für insgesamt $O(n^3 \log n)$ Quadrate für jeweils alle Eingabeinstanzen die (m,r)-Multitour Probleme gelöst werden. Wieviele verschiedene Eingabeinstanzen I_k gibt es nun für ein (m,r)-Multitour Problem? Die Anzahl der verschiedenen Eingabeinstanzen ist die Anzahl der möglichen Wahlen von (b) und (c). Betrachten wir zunächst die Anzahl der möglichen Eingabeinstanzen, die durch Wahlen von (b) entstehen. Die Anzahl der Möglichkeiten, auf einer Seite eines Quadrates von den $m+2$ Portalen k auszuwählen ist nach Satz A.2.1 $\binom{m+k+1}{k}$. Wollen wir die Anzahl der Möglichkeiten, $\leq r$ Portale auszuwählen berechnen, so müssen wir alle Möglichkeiten, $k = 0,1,...,r$ Portale auszuwählen, aufsummieren. Nach Lemma A.2.2 gilt:

$$\sum_{k=0}^{r} \binom{m+k+1}{k} = \binom{m+r+2}{r}$$

Machen wir das für jede der vier Seiten so erhalten wir insgesamt $\binom{m+r+2}{r}^4$ mögliche

Wahlen von (b). Um die Rechnung zu vereinfachen, haben wir dabei vernachlässigt, daß die Gesamtzahl der ausgewählten Portale gerade sein muß, um eine gültige Eingabeinstanz zu erhalten. Das Lemma A.2.3 besagt schließlich, daß die Anzahl der Möglichkeiten nicht größer als $(m+3)^{4r}$ ist.

Betrachten wir jetzt die Anzahl der Möglichkeiten für (c). Es gibt maximal $(4r)!$ Möglichkeiten, die in (b) ausgewählten Portale anzuordnen. Da die Reihenfolge der Paare keine Rolle spielt, können wir noch durch die $(2r)!$ Möglichkeiten die Paare anzuordnen dividieren. Insgesamt erhalten wir dann

$$\frac{(4r)!}{(2r)!} \cdot (m+3)^{4r} \in O\left(c^{O(c)} \cdot (\log n)^{O(c)}\right) \in O\left((\log n)^{O(c)}\right) \quad \text{für } c \leq \log n$$

als eine obere Schranke für die Anzahl der möglichen Eingabeinstanzen für ein (m,r)-Multitour Problem.

Zu untersuchen bleibt jetzt noch der Zeitaufwand der Prozedur `CalcCost`. Ist Q ein Blatt, so ist der Zeitaufwand wie oben schon erwähnt $O(c^2)$. Ist Q kein Blatt, dann werden alle Möglichkeiten (a') und (b') aufgezählt. Für (a') gibt es mit der gleichen Begründung wie oben maximal $(m+3)^{4r}$ Möglichkeiten. (b') besteht aus der Anzahl an Möglichkeiten für jedes der maximal $4r$ in (a') gewählten Portale einen der maximal $2r$ Pfade, die durch die Wahl von (b) und (c) festgelegt sind, auszuwählen. Das sind maximal $(2r)^{4r}$ Möglichkeiten. Diese müssen noch multipliziert werden mit der Anzahl an möglichen Reihenfolgen, in der die maximal $4r$ Portale aus (a') auf den Pfaden liegen, also maximal $(4r)!$ Möglichkeiten. Insgesamt haben wir $(m+3)^{4r} \cdot (4r)! \cdot (2r)^{4r} \in O\left((\log n)^{O(c)}\right)$ als eine obere Schranke für die Anzahl der möglichen Wahlen von (a') und (b'). Es führen zwar viele dieser Wahlen nicht zu gültigen Eingaben, aber es sind alle gültigen Eingaben durch diese Wahlen abgedeckt. Damit erreichen wir die zu Anfang erwähnte Gesamtlaufzeit von $O\left(n^3(\log n)^{O(c)}\right)$ Schritten.

3.3 Beweis des Struktur-Theorems

In diesem Abschnitt wollen wir das Struktur-Theorem beweisen. Dazu zeigen wir zuerst zwei Lemmas, die wichtige Bestandteile des Beweises sein werden. Eines davon ist das Patching-Lemma. Es zeigt uns, wie wir die Anzahl der Schnittpunkte eines Pfades mit einem Liniensegment durch umbiegen des Pfades auf maximal zwei reduzieren können. Das zweite Lemma beschreibt, wie wir die Länge einer Rundreise durch die Anzahl der Schnittpunkte der Rundreise mit einem Gitter ausdrücken können.

3.3.1 Das Patching-Lemma

Für den Beweis des Patching-Lemmas benötigen wir einige grundlegende Begriffe aus der Graphentheorie, die wir jetzt kurz erwähnen wollen.

Definition 3.3.1 Existiert in einem Graphen G ein geschlossener Kantenzug Z, der jede Kante genau einmal enthält, so heißt G ein *Eulergraph* und Z eine *Eulertour*.

Offensichtlich ist ein Eulergraph immer ein bis auf eventuell isolierte Knoten zusammenhängender Graph, da es sonst keinen geschlossenen Kantenzug geben könnte. Aus der Graphentheorie ist folgendes Resultat bekannt, welches einen Eulergraphen auf andere Weise charakterisiert. Einen Beweis findet man z. B. in [7], [10] oder [11].

Satz 3.3.2 *Ein Graph G ist Eulergraph genau dann, wenn G bis auf isolierte Knoten zusammenhängend ist und jeder Knoten von G geraden Grad hat.*

Kommen wir jetzt zum ersten Lemma, dem sogenannten Patching-Lemma. Der geschlossene Pfad π, von dem hier die Rede ist, wird in unserem späteren Beweis des Struktur-Theorems ein Salesman Pfad sein. Der daraus konstruierte Pfad π' ist dann ein neuer Salesman Pfad. Zu beachten ist, daß wir im Beweis ausnutzen, daß ein Salesman Pfad gebogene Kanten haben darf, denn sonst wäre es i. a. nicht möglich, den Pfad um Liniensegmente zu „erweitern".

Lemma 3.3.3 (Patching-Lemma)
(1) Sei S ein beliebiges Liniensegment der Länge l und π ein beliebiger, geschlossener Pfad, der S mindestens dreimal durchkreuzt. Dann gibt es auf S Linienstücke, deren Gesamtlänge höchstens $2l$ ist, so daß die Erweiterung von π um diese Linienstücke einen geschlossenen Pfad π' ergibt, der S höchstens zweimal durchkreuzt.

(2) Es gibt einen Pfad $\widetilde{\pi}$ mit obigen Eigenschaften, zu dem die Erweiterung mit Linienstücken auf S mindestens Gesamtlänge $2l$ haben muß.

Beweis: Zuerst beweisen wir die Aussage (1). Sei t die Anzahl der Schnittpunkte von π mit S und seien $M_1, M_2, ..., M_t$ diese Schnittpunkte. Zerteilen wir π an diesen Stellen, so erhalten wir t Pfadsegmente $P_1, P_2, ..., P_t$. In der Abbildung 3.9 ist ein Beispielpfad mit 5 Schnittpunkten dargestellt. Im folgenden wollen wir uns einen Hilfsgraphen G konstruieren, der die beschriebene Szene darstellt. G soll $2t$ Knoten haben, und zwar die Schnittpunkte $M_1, ..., M_t$ in doppelter Ausführung. Wir bezeichnen sie mit $M'_1, ..., M'_t$ und $M''_1, ..., M''_t$. Zwei Knoten werden miteinander durch eine Kante verbunden, wenn es in $P_1, ..., P_t$ ein Pfadsegment gibt, das die zugehörigen Schnittpunkte verbindet. Zusätzlich enthält G die Kanten $(M'_1, M''_1), ..., (M'_t, M''_t)$. Der Hilfsgraph ist damit eine schematische Darstellung des Pfades mit den Schnittpunkten. In Abbildung 3.10 ist der Hilfsgraph zu dem Beispiel aus Abbildung 3.9 dargestellt. Wir stellen uns das Liniensegment S dabei zwischen den beiden Knotenreihen liegend vor. Die Knoten $M'_1, ..., M'_t$ bezeichnen wir als die Knoten der *linken Seite* und die Knoten $M''_1, ..., M''_t$ als die der *rechten Seite*. Der Sinn des Hilfsgraphen ist, daß wir an ihm besser erkennen können, wie wir den Pfad π an den Schnittpunkten umleiten können, so daß er S nur noch höchstens zweimal schneidet.

Da π ein geschlossener Pfad ist, ist der konstruierte Hilfsgraph ein Eulergraph, der das Liniensegment S genau t mal schneidet. Unser Ziel ist es jetzt, durch geschicktes Hinzufügen bzw. Wegnehmen einiger Kanten den Graphen so zu verändern, daß er S nur

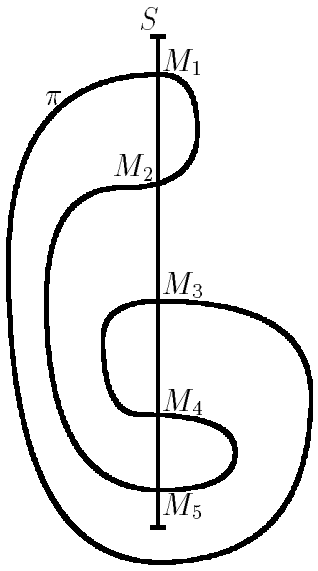


Abbildung 3.9: Beispielpfad

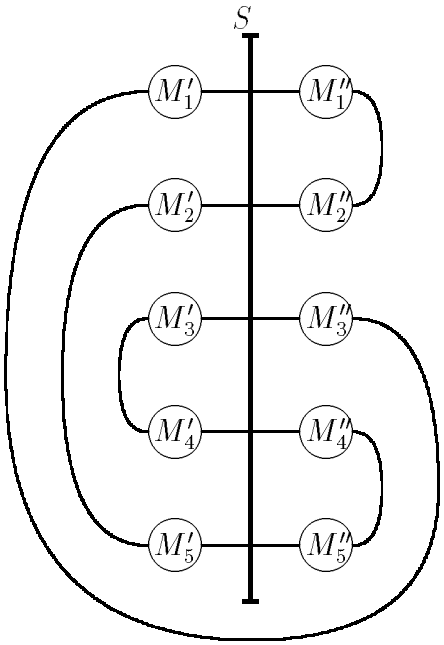


Abbildung 3.10: Hilfsgraph

noch höchstens zweimal schneidet, aber immer noch ein Eulergraph bleibt. Es existiert dann nämlich eine Eulertour und diese entspricht einem geschlossenen Pfad π', der die Pfadsegmente $P_1, ..., P_t$ enthält und der S nur höchstens zweimal durchkreuzt. Wir beschreiben jetzt einen Algorithmus, der diese Umformung des Hilfsgraphen durchführt. Die Anzahl der Schnittpunkte t ist nach Voraussetzung mindestens 3.

1. $i := 1$;
2. Entferne Kante (M'_i, M''_i) aus G;
3. Füge Kante (M'_i, M'_{i+1}) und Kante (M''_i, M''_{i+1}) in G ein;
4. Entferne Kante (M'_{i+1}, M''_{i+1}) aus G;
5. Verbinde die maximal zwei Komponenten durch die Doppelkante (M'_{i+1}, M'_{i+2}) bzw. (M''_{i+1}, M''_{i+2});
6. $i := i + 2$;
7. Falls $i \leq t - 2$ dann gehe zu 2. sonst STOP;

Wir haben jetzt zwei Dinge zu zeigen und zwar erstens, daß der Algorithmus wohldefiniert ist und wirklich einen Eulergraphen, der S nur maximal zweimal schneidet, liefert

und zweitens, daß die hinzugefügten Kanten den Pfad π um maximal $2l$ verlängern. Zum ersten Punkt wollen wir zeigen, daß der Hilfsgraph nach jedem Durchlauf der Schleife ein zusammenhängender Graph ist, in dem jeder Knoten geraden Grad hat und damit nach Satz 3.3.2 ein Eulergraph ist. Da pro Schleifendurchlauf die Kanten zwischen der linken und rechten Seite entfernt werden, bleiben am Ende nur noch maximal die beiden untersten Kanten (M'_{t-1}, M''_{t-1}) und (M'_t, M''_t) (bei geradem t die beiden untersten und bei ungeradem t nur die unterste) bestehen. Am Ende würde die Prozedur dann also einen Eulergraphen liefern, der S nur noch maximal zweimal schneidet.

Bevor wir den Algorithmus starten, ist G ein zusammenhängender Graph, in dem jeder Knoten Grad 2 hat. Da G ein Eulergraph ist, existiert ein geschlossener Kantenzug, der jede Kante genau einmal enthält. Durch Entfernen nur einer Kante bleibt der Graph also zusammenhängend. Nach dem zweiten Schritt haben wir damit immer noch einen zusammenhängenden Graphen. Die Knoten M'_i und M''_i haben dann allerdings Grad 1. In Schritt 3 werden zwei Kanten hinzugefügt. Der Graph bleibt dabei zusammenhängend und die Knoten M'_i und M''_i bekommen wieder Grad 2 und die Knoten M'_{i+1} und M''_{i+1} erhalten Grad 3. Im 4. Schritt wird eine Kante entfernt, was zur Folge hat, daß danach wieder alle Knoten geraden Grad haben. Der Graph ist aber nicht mehr zwingend zusammenhängend, da G vor dem 4. Schritt kein Eulergraph war. Der Hilfsgraph kann aber nur in maximal zwei Komponenten $K(M'_{i+1})$ und $K(M''_{i+1})$ zerfallen sein. Für den Fall, daß der Hilfsgraph nicht zerfällt, also zusammenhängend bleibt, haben wir jetzt wieder einen Eulergraphen. Zerfällt der Hilfsgraph in zwei Komponenten, dann muß eine der beiden Komponenten $K(M'_{i+2})$ oder $K(M''_{i+2})$ verschieden sein von $K(M'_{i+1})$ bzw. $K(M''_{i+1})$, denn wäre $K(M'_{i+1}) = K(M'_{i+2})$ und $K(M''_{i+1}) = K(M''_{i+2})$, dann wäre die Komponente $K(M'_{i+1})$ über die Kante (M'_{i+2}, M''_{i+2}) mit $K(M''_{i+1})$ verbunden und das ist ein Widerspruch dazu, daß $K(M'_{i+1})$ und $K(M''_{i+1})$ verschiedene Komponenten sind. Wir können den Hilfsgraphen also entweder durch die doppelte Kante (M'_{i+1}, M'_{i+2}) oder (M''_{i+1}, M''_{i+2}) wieder zusammenhängend machen. Der Knotengrad bleibt dabei überall gerade und damit ist der Graph in jedem Fall nach Durchlauf der Schleife wieder ein Eulergraph.

Kommen wir jetzt dazu zu zeigen, daß die hinzugefügten Kanten den Pfad π um maximal $2l$ verlängern. Im Algorithmus werden dem Hilfsgraphen nur Kanten im dritten und fünften Schritt hinzugefügt. Im dritten Schritt fügen wir zwei Liniensegmente zwischen M_i und M_{i+1} ein und im fünften Schritt eventuell zwei Liniensegmente zwischen M_{i+1} und M_{i+2}. Jedes Liniensegment zwischen zwei Schnittpunkten wird also höchstens zweimal eingefügt. Also ist die Gesamtlänge aller hinzugefügten Liniensegmente höchstens $2l$. In den Abbildungen 3.11 und 3.12 ist der Hilfsgraph zu dem Beispiel aus Abbildung 3.9 nach dem ersten bzw. zweiten Schleifendurchlauf dargestellt. Nach dem zweiten Schleifendurchlauf ist der Algorithmus fertig und der Hilfsgraph ist der endgültige Eulergraph.

Kommen wir jetzt zum Beweis von Aussage (2). Wir müssen hier zeigen, daß es einen Pfad gibt, der erzwingt, daß die Gesamtlänge der hinzugefügten Linienstücke mindestens $2l$ ist. Wir geben dazu einen Pfad $\widetilde{\pi}$ an, bei dem die Erweiterung um Linienstücke aus S mindestens Länge $2l$ haben muß, damit daraus ein geschlossener Pfad $\widetilde{\pi}'$ entsteht, der S höchstens zweimal schneidet. Es ist offensichtlich so, daß zu jedem geschlossenen Pfad $\widetilde{\pi}'$,

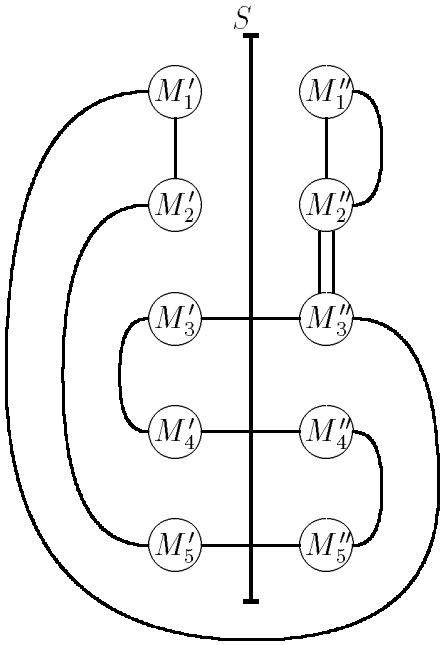


Abbildung 3.11: Nach erster Schleife

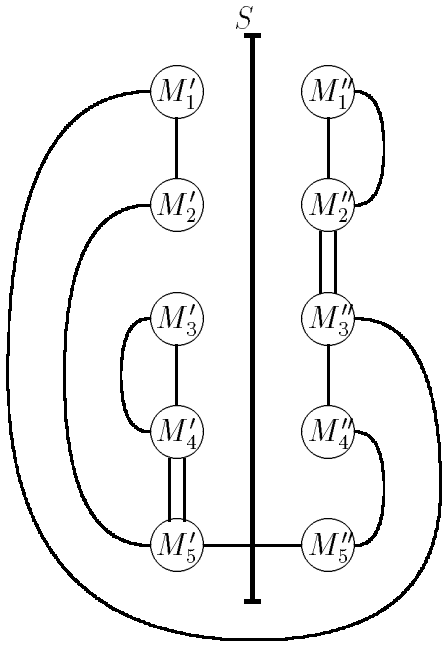


Abbildung 3.12: Nach zweiter Schleife

der S höchstens zweimal schneidet, sich aus dem Hilfsgraphen zu $\widetilde{\pi}$ ein passender Eulergraph konstruieren läßt, der S höchstens zweimal schneidet. Können wir also zeigen, daß zu dem Hilfsgraphen für $\widetilde{\pi}$ Liniensegmente der Gesamtlänge mindestens $2l$ hinzugefügt werden müssen, damit daraus ein Eulergraph entsteht, der S höchstens zweimal schneidet, dann sind wir fertig. Wir betrachten im folgenden immer den Hilfsgraphen ohne die Kanten zwischen der linken und rechten Seite. (Die müssen bei der Umformung sowieso bis auf maximal zwei Kanten entfernt werden.) Der Hilfsgraph enthält dann nur die Pfadsegmente $P_1, ..., P_t$ und die muß der zu konstruierende geschlossene Pfad $\widetilde{\pi}'$ auch enthalten. In Abbildung 3.13 ist der Pfad $\widetilde{\pi}$ dargestellt und in Abbildung 3.14 der zugehörige Hilfsgraph. S wird dabei von $\widetilde{\pi}$ in den Punkten M_1, M_2 und M_3 geschnitten.

Zuerst schauen wir uns an, durch wieviele Kanten die Knoten M'_1, M'_2, M'_3 mit den Knoten M''_1, M''_2, M''_3 wirklich verbunden sein können. Da S von $\widetilde{\pi}'$ nur zweimal geschnitten werden darf, darf es auch nur maximal 2 dieser Verbindungskanten geben. Allgemein gilt aber: Ist die Anzahl der Schnittpunkte $M_1, ..., M_t$ ungerade, so können die Knoten $M'_1, ..., M'_t$ mit den Knoten $M''_1, ..., M''_t$ nur durch genau eine Kante verbunden sein, denn wären sie mit keiner oder zwei Kanten verbunden, dann würde im Hilfsgraphen auf jeder Seite von S eine ungerade Anzahl Knoten mit ungeradem Grad übrigbleiben (im Hilfsgraphen hat jeder Knoten Grad 1). Diese könnten aber - ohne S zu schneiden - nie mehr so verbunden werden, daß jeder Knoten geraden Grad hat, da eine Kante immer bei 2 Knoten den Grad um eins erhöht. Es gibt in unserem Hilfsgraphen nur drei Möglichkeiten eine Kante zwischen der linken und rechten Seite einzufügen und für jeden dieser drei

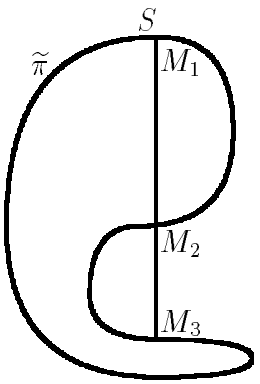


Abbildung 3.13: Pfad $\widetilde{\pi}$

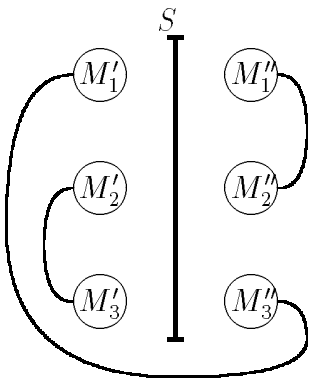


Abbildung 3.14: Hilfsgraph zu $\widetilde{\pi}$

Fälle gibt es nur eine Möglichkeit, daraus einen Eulergraphen zu machen. Die drei möglichen Eulergraphen sind in Abbildung 3.15 zu sehen. Die Gesamtlänge der hinzugefügten Kanten ist in allen drei Graphen $2l$.

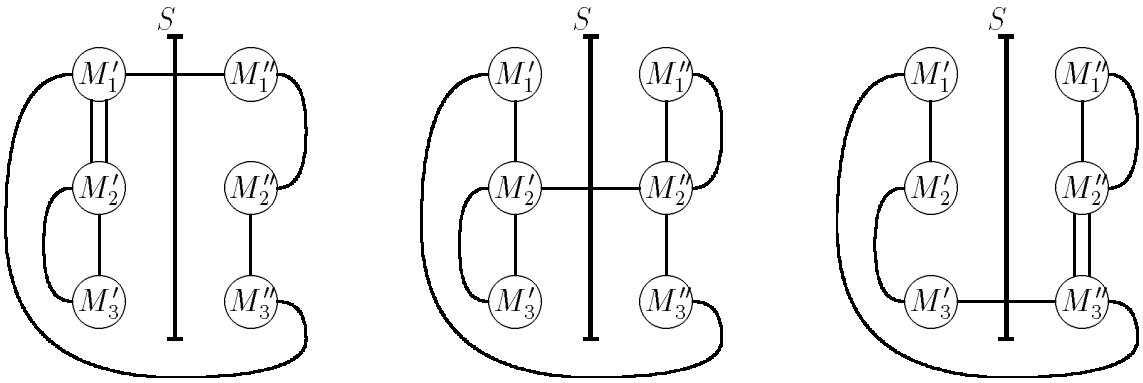


Abbildung 3.15: Die drei möglichen Eulergraphen zu $\widetilde{\pi}$

Auf ähnliche Weise lassen sich übrigens auch für jede beliebige Anzahl von Schnittpunkten Beispiele konstruieren, so daß die hinzugefügten Kanten die Gesamtlänge $2l$ haben müssen. Wären die Schnittpunkte M_1 und M_5 im Pfad aus Abbildung 3.9 am oberen bzw. unteren Ende von S, dann wäre dieser Pfad auch ein Beispiel dafür. Es gibt aber auch Pfade, bei denen es möglich ist, Kanten mit einer Gesamtlänge kleiner als $2l$ hinzuzufügen, so daß der Graph ein Eulergraph wird, der S nur höchstens zweimal schneidet. □

Das nächste Lemma stellt einen Zusammenhang her zwischen der Länge einer Rundreise und der Anzahl an Schnitten der Rundreise mit senkrechten und waagerechten Linien. Dazu legen wir ein Gitter mit vertikalen und horizontalen Linien, die Abstand 1 haben, in die Bounding-Box der Eingabepunkte. Sei π eine Rundreise und l eine der Gitterlinien, dann sei $t(\pi, l)$ die Anzahl der Schnitte von π mit l.

Lemma 3.3.4 *Haben die Eingabepunkte einer TSP-Instanz paarweise mindestens Abstand 4 und sei π eine Rundreise der Länge T, dann gilt:*

$$\sum_{l_{vertikal}} t(\pi, l) \quad + \sum_{l_{horizontal}} t(\pi, l) \quad \leq \quad 2T$$

Beweis: Eine wichtige Beobachtung für den Beweis ist, daß die linke Seite der Ungleichung grob gesprochen die ℓ_1-Länge der Tour ist. Sei e eine Kante aus π und s die Länge dieser Kante. Seien u und v die horizontale und vertikale Projektion der Kante e. Es ist dann $u+v$ die ℓ_1-Länge von e und $\sqrt{u^2+v^2} = s$ die ℓ_2-Länge. Die ℓ_1-Länge einer Kante beträgt maximal das $\sqrt{2}$-fache der ℓ_2-Länge, da

$$\begin{aligned} 0 \leq (u-v)^2 &\iff 2uv \leq u^2+v^2 \\ &\iff (u+v)^2 = u^2+2uv+v^2 \leq 2(u^2+v^2) \\ &\iff |e|_{\ell_1} = u+v \leq \sqrt{2}\cdot\sqrt{u^2+v^2} = \sqrt{2}\cdot|e|_{\ell_2} \end{aligned}$$

Die Kante e erhöht die linke Seite der Ungleichung um maximal $(u+1)+(v+1)$. Siehe dazu Abbilung 3.16, in der $u = 8,5$ und $v = 2,6$ ist. Es werden dort 3 horizontale und 9 vertikale Gitterlinien geschnitten.

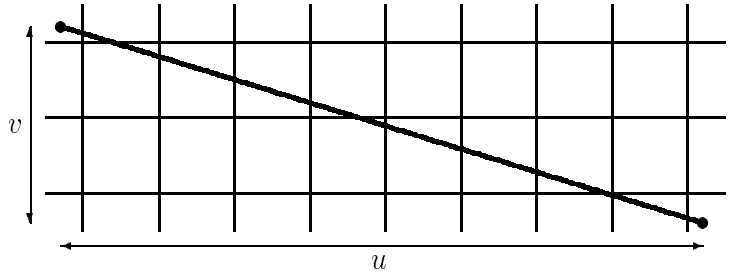


Abbildung 3.16:

Wir können jetzt den maximalen Beitrag einer Kante zur linken Seite der Ungleichung nach oben abschätzen, woraus dann die Behauptung folgt.

$$u+v+2 \leq \sqrt{2}\cdot\sqrt{u^2+v^2}+2 \leq \sqrt{2}\cdot s+2 \leq 2s$$

Die letzte Ungleichung gilt für alle $s \geq 2+\sqrt{2} \approx 3,41$. Dies stellt kein Problem dar, da laut Voraussetzung s mindestens 4 ist. □

3.3.2 Das Struktur-Theorem

Wir sind jetzt bereit, unsere Kernaussage, das Struktur-Theorem, zu beweisen. Es werden dazu einige Grundlagen der Wahrscheinlichkeitsrechnung vorausgesetzt. Die verwendeten Resultate sind zur Erleichterung des Verständnisses im Anhang dieser Arbeit zu finden. Ebenfalls ist dort das „Zufallsexperiment“ im Struktur-Theorem als Beispiel aufgeführt (siehe Beispiel A.3.7).

Beweis (Struktur-Theorem): Sei π eine optimale Rundreise mit Kosten *opt* zu einer beliebigen TSP-Eingabe und seien (a, b) zufällig gewählte Shifts. Wir beweisen das Struktur-Theorem, indem wir π über mehrere Schritte hinweg in einen Salesman Pfad transformieren, der (m, r)-light ist in bezug auf die zufällig geshiftete Dissection. Wir verwenden dazu eine deterministische Prozedur, die die Transformation vornimmt. Es werden die Tourkosten dabei leicht erhöht. Wie wir aber im folgenden sehen werden, läßt sich die Erhöhung nach oben beschränken.

Um den Kostenzuwachs der Transformation besser abrechnen zu können, legen wir ein Gitter mit vertikalen und horizontalen Linien, die Abstand 1 haben, in die Bounding-Box der Eingabepunkte und „belasten“ die Gitterlinien mit jeder Kostenerhöhung der Tour. Sei $t(\pi, l)$ die Anzahl der Schnitte von π mit Gitterlinie l. Wir werden zeigen, daß die erwartete Belastung für jede feste Linie l des Gitters bei Shift (a, b) beschränkt ist:

$$E_{a,b}(\text{Belastung von } l) \;\leq\; \frac{t(\pi, l)}{4c}$$

Wegen der Linearität des Erwartungswertes (Satz A.3.4) ergibt sich die erwartete Erhöhung der gesamten Tourkosten dann durch Aufsummieren der erwarteten Kostenerhöhung für jede Linie l des Gitters:

$$\sum_l \frac{t(\pi, l)}{4c} \;=\; \frac{1}{4c}\sum_l t(\pi, l) \;=\; \frac{1}{4c}\left(\sum_{l_{vertikal}} t(\pi, l) \;+\; \sum_{l_{horizontal}} t(\pi, l)\right) \;\leq\; \frac{opt}{2c}$$

Die letzte Ungleichung gilt wegen Lemma 3.3.4, das wir anwenden können, da der minimale Abstand zweier Punkte 4 ist. Wenden wir nun die Markoffsche Ungleichung (Satz A.3.5) für $\lambda := 2$ an, so erhalten wir:

$$P\left(\text{Kostenerhöhung} \geq \frac{opt}{c}\right) \;\leq\; \frac{1}{2}$$

Diese Ungleichung ist äquivalent zu $P\left(\text{Kostenerhöhung} \leq \frac{opt}{c}\right) \;\geq\; \frac{1}{2}$. Es folgt also, daß mit einer Wahrscheinlichkeit von mindestens $\frac{1}{2}$ der Kostenzuwachs nicht größer als $\frac{opt}{c}$ ist. Die Kosten des kürzesten (m, r)-light Salesman Pfades für die geshiftete Dissection sind also mit einer Wahrscheinlichkeit von mindestens $\frac{1}{2}$ höchstens $\left(1 + \frac{1}{c}\right) opt$.

Um das Theorem zu beweisen, reicht es also aus, zu beschreiben, wie wir die optimale Rundreise transformieren und die Kostenerhöhung auf die Gitterlinien verteilen. Sei die Größe der Bounding-Box L der TSP-Eingabe eine Zweierpotenz. Dies können wir o. B. d. A. annehmen, da wir die Größe der B-Box immer auf die nächste Zweierpotenz erhöhen können. Weiter wollen wir die Quadrate in der Dissection solange in kleinere Quadrate zerlegen, bis die Größe der Quadrate genau 1 ist und nicht dann stoppen, wenn die Punkte separiert sind. Die Dissection entspricht dann vom Aussehen her genau dem Gitter und die Tiefe der Dissection ist genau $\log L$. Die geshiftete Dissection deckt sich ebenfalls mit dem Gitter, da die Gitterlinien Abstand 1 haben und die Shifts a, b ganzzahlig sind.

Wie wir wissen, bildet die Dissection auf natürliche Weise eine Hierarchie, die die Quadrate auf Ebenen anordnet. Wir bezeichnen im folgenden eine Gitterlinie als *Level-i-Linie*, wenn sich auf ihr eine Kante eines Quadrates von Ebene i befindet. Beachte, daß ein Quadrat von Ebene i auf Ebene $i+1$ in vier Quadrate eingeteilt wird. Eine Kante eines Quadrates von Ebene i teilt sich somit auf Ebene $i+1$ in zwei Kanten von je einem Quadrat dieser Ebene. Eine Level-i-Linie ist also auch eine Level-j-Linie für alle $j > i$. Für jedes $i \geq 0$ gibt es 2^i horizontale und 2^i vertikale Level-i-Linien. Die vertikalen Level-i-Linien haben x-Koordinaten $x_p := \left(a + p \cdot \frac{L}{2^i}\right) \mod L$ und die horizontalen Level-i-Linien y-Koordinaten $y_p := \left(b + p \cdot \frac{L}{2^i}\right) \mod L$, wobei $p \in \{0, 1, ..., 2^i - 1\}$ ist.

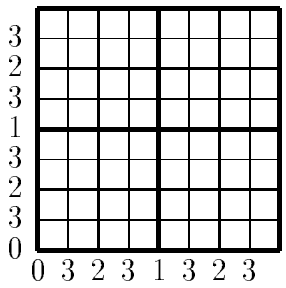


Abbildung 3.17: Gitterlinien mit ihren max-Level Werten

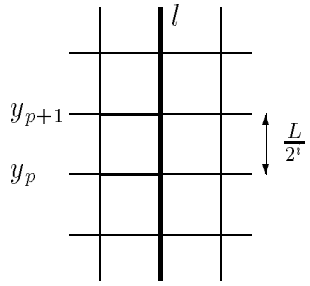


Abbildung 3.18:

Der *max-Level* einer Gitterlinie ist die kleinste Zahl i, für die die Gitterlinie noch eine Level-i-Linie ist, also die höchste Ebene, auf der sie liegt. In Abbildung 3.17 ist ein Gitter der Größe 8 mit den Ebenen 0 bis 3 dargestellt. Die Linien sind mit dem zugehörigen max-Level Wert beschriftet, wobei die letzte Linie keine Beschriftung bekommen hat, da sie der ersten Linie nach der Modulo-Rechnung entspricht.

Wieviele horizontale Shifts a führen nun dazu, daß eine feste vertikale Gitterlinie l max-Level i bekommt ? Wir definieren dazu eine Zufallsvariable $ML_l : \{0, 1, ..., L-1\} \longrightarrow \{0, 1, ..., \log L\}$ für eine feste vertikale Gitterlinie l, die den max-Level von l bei horizontalem Shift a angibt. Die Anzahl der horizontalen Shifts, durch die eine feste vertikale Linie l auf max-Level $0, 1, 2, 3, 4, ..., i, ..., \log L$ landet, ist $1, 1, 2, 4, 8, ..., 2^{i-1}, ..., \frac{L}{2}$ (siehe

Abbildung 3.17). Es folgt daraus, daß die Wahrscheinlichkeit, mit der eine feste vertikale Gitterlinie l max-Level i bekommt, bei horizontalem Shift a

$$P_a(ML_l = i) \;=\; P_a(\{a' | ML_l(a') = i\}) \;=\; \frac{|\{a' | ML_l(a') = i\}|}{L} \;=\; \begin{cases} \frac{2^{i-1}}{L} & i \geq 1, \\ \frac{1}{L} & i = 0 \end{cases}$$

ist. Entsprechendes gilt natürlich auch für horizontale Gitterlinien bei vertikalem Shift b.

Was bedeutet das nun für einen Salesman Pfad, der (m, r)-light sein soll? Der Einfachheit halber beschreiben wir die Situation nur für vertikale Linien; für horizontale Linien gilt eine analoge Aussage. (m, r)-light zu sein bedeutet,

(i) daß für eine vertikale Linie l, die max-Level i hat, das Segment der Linie, das zwischen den y-Koordinaten y_p und y_{p+1} liegt, von dem Salesman Pfad höchstens r mal geschnitten wird und

(ii) alle diese Schnitte an Portalen stattfinden.

Siehe dazu Abbildung 3.18, in der eine vertikale Linie mit max-Level i dargestellt ist. Das Liniensegment zwischen y_p und y_{p+1} ist dann die Kante eines Quadrates auf Ebene i, die nur r-mal geschnitten werden darf. Da i der max-Level von l ist, ist diese Kante die größtmögliche und es ist nicht zu befürchten, daß eine Kante eines Quadrates von Ebene $i-1$ auch auf Linie l liegt. Wir werden jetzt die Rundreise π in zwei Schritten so verändern, daß sie die beiden Bedingungen erfüllt.

Wie können wir für π nun die erste Bedingung erfüllen? Eine naheliegende Idee ist, alle Gitterlinien l zu durchlaufen und falls l den max-Level i hat, jedes der 2^i Segmente der Länge $\frac{L}{2^i}$ mit dem Patching-Lemma zu bearbeiten, falls die Anzahl der Schnitte der optimalen Rundreise mit dem Segment größer als r ist. Offensichtlich wäre dies eine einfache Lösung, aber es könnte die durch das Patching-Lemma bewirkte Kostenerhöhung zu groß ausfallen. Genauer gesagt, wäre die Erhöhung der Kosten von der Form $k \log L$ für eine Konstante k. Die Rundreise könnte also nicht mehr nur um einen konstanten Faktor verlängert werden, wie es bei einem PTAS gefordert wird.

Bei dem gerade beschriebenen Verfahren wird jedes Liniensegment von l, das mehr als r Schnittpunkte mit der Rundreise hat, „gepatcht“. Die Anzahl der Schnittpunkte wird also auf höchstens 2 reduziert, obwohl eigentlich nur gefordert wird, daß die Anzahl der Schnittpunkte höchstens r ist. Es könnte ja sein, daß sich auf einem kleinen Abschnitt des Liniensegmentes die meisten Schnittpunkte „tummeln“ und es somit ausreichen würde diesen Abschnitt zu patchen. Das führt uns zu der besseren Idee, das Patching-Lemma „bottom up“ für alle Ebenen $j \geq i$ auf der Gitterlinie l anzuwenden. Die folgende Prozedur `MODIFY`(l, i, b) tut genau dieses und geht dabei, wie wir gleich sehen werden, etwas sparsamer mit der Kostenerhöhung um. Die Prozedur wird hier nur für vertikale Gitterlinien beschrieben, da sie für horizontale Linien analog aufgebaut ist.

PROCEDURE `MODIFY`(l, i, b)

{* l ist eine vertikale Gitterlinie, b ist der vertikale Shift der Dissection und *}
{* i ist der max-Level von l *}

BEGIN
  FOR $j = \log L$ **DOWNTO** i **DO**
    FOR $p = 0$ **TO** $2^j - 1$ **DO**
      Sei l_p das Liniensegment auf l zwischen den y-Koordinaten $\left(b + p \cdot \frac{L}{2^j}\right) \bmod L$ und $\left(b + (p+1) \cdot \frac{L}{2^j}\right) \bmod L$;
      IF Anzahl Schnitte der Rundreise mit $l_p > r$
      THEN Wende Patching-Lemma auf l_p an;
    OD
  OD
END

Da es nur eine Gitterlinie mit max-Level 0 gibt und diese auf dem Rand der Dissection liegt, also von der Rundreise gar nicht geschnitten werden kann, sind die durch `MODIFY` verursachten Kosten auf dieser Linie 0. Wir können uns daher im folgenden auf die Linien mit max-Level größer als 0 konzentrieren. Durch die Prozedur `MODIFY` wird der Salesman Pfad dynamisch vom tiefsten Level bis zum max-Level einer Linie aktualisiert. Dadurch beeinflußt die Anwendung des Patching-Lemmas auf ein Liniensegment einer tieferen Ebene eine eventuelle Anwendung auf einer höheren Ebene. Anders ausgedrückt wird immer erst versucht, das Patching-Lemma auf den tieferen Ebenen, auf denen die Liniensegmente ja kürzer sind als auf höheren, anzuwenden. Eine Anwendung des Patching-Lemmas reduziert die Anzahl der Schnittpunkte je Segment i. a. auf maximal 2. Eine Ausnahme bilden die Liniensegmente, die durch den vertikalen Shift „herumgewickelt“ sind. Jede Ebene kann so ein Liniensegment enthalten. Um diese Liniensegmente zu bearbeiten, müssen wir das Patching-Lemma auf jeden Teil des Segments anwenden. Dadurch können wir die Anzahl der Schnittpunkte auf diesen Liniensegmenten nur auf maximal 4 reduzieren. Noch zu bemerken ist, daß die Anwendung des Patching-Lemmas auf eine vertikale Gitterlinie die Anzahl der Schnitte der Rundreise mit einer horizontalen Line erhöhen kann. Wir ignorieren diesen Effekt vorerst und kommen am Ende des Beweises darauf zurück.

Beschäftigen wir uns jetzt mit der Kostenerhöhung durch `MODIFY`. Sei dazu l eine vertikale Gitterlinie mit max-Level i, auf die wir `MODIFY` anwenden. Für $j \geq i$ sei $c_{l,j}(b)$ die Anzahl der Level-j-Liniensegmente auf l, auf denen wir das Patching-Lemma anwenden. Für $j < i$ ist $c_{l,j}(b)$ offensichtlich 0. Beachte, daß $c_{l,j}(b)$ unabhängig von i ist, da die zweite Schleife im Algorithmus unabhängig von i ist. Hier liegt der wesentliche Unterschied zu unserem ersten Ansatz, in dem die Anzahl der Anwendungen des Patching-Lemmas von i abhängig war. Bei jeder Anwendung des Patching-Lemmas werden mindestens $r + 1$ Schnittpunkte durch maximal 4 ersetzt. Die Anzahl der Schnittpunkte wird also mindestens um $r - 3$ reduziert. Da die optimale Rundreise π die Gitterlinie l nur $t(\pi, l)$-mal durchkreuzt, haben wir für jede vertikale Gitterlinie l und *jeden* vertikalen Shift b folgen-

den Zusammenhang:

$$\sum_{j=i}^{\log L} c_{l,j}(b) \quad \leq \quad \frac{t(\pi,l)}{r-3}$$

Da das Patching-Lemma zu einem Pfad, der ein Liniensegment der Länge s durchkreuzt, maximal Linienstücke der Gesamtlänge $2s$ hinzufügt, erhalten wir für die Erhöhung der Tourkosten:

$$cost_l(i,b) \;:=\; \text{Kostenerhöhung durch } \texttt{MODIFY}(l,i,b) \quad \leq \quad \sum_{j=i}^{\log L} \left(c_{l,j}(b) \cdot 2 \cdot \frac{L}{2^j} \right)$$

Wir rechnen diese Kostenerhöhung der Gitterlinie l an. Dazu definieren wir eine Zufallsvariable $K_l : \{0,1,...,L-1\} \longrightarrow \mathbb{R}$ für eine feste vertikale Gitterlinie l, die den durch Schritt (i) hervorgerufenen Kostenzuwachs zu l bei horizontalem Shift a angibt. Die Belastung von l ist für alle horizontalen Shifts a, durch die l auf max-Level i landet $cost_l(i,b)$. Summieren wir jetzt über alle horizontalen Shifts auf, so können wir die Belastungen für die Shifts, durch die l auf dem gleichen max-Level landet, gruppieren (siehe zweite Gleichung in der Rechnung unten). Wir erhalten für jeden Level eine Gruppe, und die Wahrscheinlichkeit, mit der die Kosten $cost_l(i,b)$ dieser Gruppe auftreten, ist die Wahrscheinlichkeit mit der Linie l max-Level i hat, also $P_a(ML_l = i)$. Die zu erwartende Belastung einer festen vertikalen Gitterlinie l bei horizontalem Shift a ist für *jeden* vertikalen Shift b:

$$\begin{aligned}
E_a(\text{Belastung von } l \text{ durch Schritt (i)}) \;&=\; \sum_{a=0}^{L-1} \frac{1}{L} \cdot K_l(a) \;=\; \sum_{i=0}^{\log L} P_a(ML_l = i) \cdot cost_l(i,b) \\
&=\; \sum_{i=1}^{\log L} \frac{2^{i-1}}{L} \cdot cost_l(i,b) \;\leq\; \sum_{i=1}^{\log L} 2^{i-1} \cdot \sum_{j=i}^{\log L} \frac{c_{l,j}(b)}{2^{j-1}} \\
&=\; \sum_{j=1}^{\log L} \frac{c_{l,j}(b)}{2^{j-1}} \cdot \sum_{i=1}^{j} 2^{i-1} \;=\; \sum_{j=1}^{\log L} \frac{c_{l,j}(b)}{2^{j-1}} \cdot (2^j - 1) \\
&\leq\; 2 \cdot \sum_{j=1}^{\log L} c_{l,j}(b) \;=\; 2 \cdot \sum_{j=i}^{\log L} c_{l,j}(b) \;\leq\; \frac{2\,t(\pi,l)}{r-3}
\end{aligned}$$

In der Rechnung wurde im 5. Schritt die Summengleichheit aus Satz A.1.2 und im 6. Schritt die geometrische Summenformel (Satz A.1.1) verwendet.

Kommen wir jetzt zur zweiten notwendigen Modifikation der Rundreise, dem Bewegen der Schnittpunkte auf die Portalpunkte. Hat eine Gitterlinie l den max-Level i, so haben die Liniensegmente auf l Länge $\frac{L}{2^i}$. Jedes Liniensegment besitzt $m+2$ Portale, die in gleichem Abstand verteilt sind. Der Abstand zwischen zwei Portalpunkten ist somit $\frac{L}{2^i(m+1)}$. Der Abstand eines Schnittpunktes zum naheliegendsten Portal ist dann maximal $\frac{L}{2^{i+1}(m+1)}$. Ändern wir die Tour nun so ab, daß sie, wenn sie auf das Liniensegment trifft, zum nächsten Portal läuft und dann wieder zurückkehrt zu ihrer ursprünglichen Route, so ist die Kostenerhöhung pro Schnittpunkt nicht größer als $\frac{L}{2^i(m+1)}$. Sei $K'_l : \{0,1,...L-1\} \longrightarrow \mathbb{R}$ wieder eine Zufallsvariable für eine feste vertikale Gitterlinie l, die den durch Schritt (ii) hervorgerufenen Kostenzuwachs zu l bei horizontalem Shift a angibt. Mit derselben Argumentation wie in Schritt (i) ergibt sich daraus eine erwartete Kostenerhöhung von maximal

$$\begin{aligned} E_a(\text{Belastung von } l \text{ durch Schritt (ii)}) &= \sum_{a=0}^{L-1} \frac{1}{L} \cdot K'_l(a) \leq \sum_{i=1}^{\log L} \frac{2^{i-1}}{L} \cdot t(\pi,l) \cdot \frac{L}{2^i(m+1)} \\ &= \frac{t(\pi,l)\log L}{2(m+1)} \leq \frac{t(\pi,l)}{kr} \end{aligned}$$

Die letzte Ungleichung gilt, falls $m \geq \frac{k}{2} r \log L$ ist für eine Konstante $k > 0$. Wir können jetzt die zu erwartenden Gesamtkosten, um den Salesman Pfad auf der Gitterlinie l (m,r)-light zu machen, nach oben abschätzen:

$$\begin{aligned} E_a(\text{Belastung von } l) &\leq \frac{2\,t(\pi,l)}{r-3} + \frac{t(\pi,l)}{kr} = \frac{(2kr+r-3)\,t(\pi,l)}{kr(r-3)} \\ &\overset{(\star)}{\leq} \frac{(19k+8)(r-3)\,t(\pi,l)}{8kr(r-3)} = \frac{(19k+8)\,t(\pi,l)}{8kr} \leq \frac{t(\pi,l)}{4c} \end{aligned}$$

Die Ungleichung $(\star)$ setzt für alle $k > 0$ voraus, daß $r \geq 19$ ist und die letzte Ungleichung gilt für $r \geq \frac{(19k+8)c}{2k}$. Da c mindestens 2 ist, ist $r \geq 19 + \frac{8}{k} > 19$ für alle k. Die Bedingung für die Ungleichung $(\star)$ ist somit für alle k erfüllt. Das ist kein Zufall, denn die Abschätzung $(\star)$ ist genau so gewählt, daß die durch die Ungleichung entstehende Bedingung für r durch die letztendliche Wahl von r genau eingehalten werden kann. Betrachten wir das Verhalten von r für $k \to \infty$, so ist das Ergebnis, daß $r > 9.5c$ sein muß. Damit haben wir eine untere Schranke für r. Je näher r an die untere Schranke herankommt, desto größer muß allerdings m gewählt werden.

Wir haben jetzt die anfängliche Behauptung gezeigt. Um den Beweis zu beenden, müssen wir noch den durch `MODIFY` hervorgerufenen Effekt erklären. Immer dann, wenn wir `MODIFY` auf einer vertikale Gitterlinie l anwenden, benutzen wir eventuell das Patching-Lemma und erweitern den Salesman Pfad um einige Linienstücke, die auf l liegen. Diese

Linienstücke können einige horizontale Gitterlinien l' durchkreuzen, so daß der Salesman Pfad diese Linien nachher öfter als $t(\pi, l')$-mal schneidet. Wir können aber o. B. d. A. annehmen, daß die Erhöhung der Schnitte des Salesman Pfades mit l' durch das Patching auf l nicht größer als 2 ist; denn ist die Erhöhung größer als 2, dann reduzieren wir sie mit dem Patching-Lemma einfach auf 2. Die Anwendung des Patching-Lemmas auf l fügt dem Pfad nur vertikale Linienstücke hinzu, die auf l liegen. Deswegen brauchen wir auf der horizontalen Linie l' auch nur einen Abschnitt patchen, der die Ausdehnung 0 hat, also einen Punkt. Die Kosten des Salesman Pfades werden also durch das Patching auf l' nicht erhöht. Die Prozedur `MODIFY` geht dann so vor, daß sie als erstes alle Liniensegmente von l auf den Ebenen $\log L$ bis zum max-Level von l bearbeitet und dann auf allen horizontalen Linien, die durch das Patching auf l geschnitten werden, das Patching-Lemma anwendet, um die Anzahl der Schnittpunkte auf 2 zu reduzieren. Durch die Struktur der Dissection haben die horizontalen Linien l', die von einem Liniensegment auf l geschnitten werden, immer einen kleineren max-Level als l (siehe Abbildung 3.17). Es ist deswegen nicht möglich, daß ein horizontales Liniensegment an beiden Seiten durch jeweils ein vertikales Liniensegment, welches das horizontale schneidet, begrenzt ist. Die Erhöhung der Anzahl der Schnittpunkte auf l' ist also maximal 2.

Den gleichen Trick, wie gerade beschrieben, müssen wir auch für Schritt (ii) anwenden, da hier die Erweiterungen des Salesman Pfades auch horizontale Linien schneiden können. Wir können die Anzahl der Schnitte aber wieder mit dem Patching-Lemma auf 2 reduzieren.

Insgesamt kann also jede Gitterlinie maximal $(r+2)$-mal geschnitten werden und nicht r-mal, wie wir es eigentlich wollen. Das ist aber kein Problem, wenn wir „r“ einfach durch „$r+2$“ ersetzen. □

3.4 Wie effizient ist der Algorithmus?

Wir wollen hier untersuchen, wie effizient eine direkte Implementierung der Prozedur `ApproxTSP` tatsächlich ist. Im Beweis des Struktur-Theorems wurde gezeigt, daß r für jede beliebige Eingabe größer als 19 sein muß. Die Zahl r gibt die maximale Anzahl an Schnittpunkten des Salesman Pfades mit einer Quadratkante an und muß daher eine ganze Zahl sein. Es sollte also $r \geq 20$ für alle Eingaben sein. Die Anzahl der Portale $m+2$ auf einer Kante sollte damit auch mindestens 20 betragen.

Wir wollen jetzt zeigen, daß die Anzahl der möglichen Eingabeinstanzen für ein (m, r)-Multitour Problem schon so groß ist, daß ein normaler Rechner eine nicht mehr vertretbare Anzahl an Rechenschritten benötigen würde, um sie aufzuzählen. Dazu wollen wir allein die Anzahl der Möglichkeiten betrachten, aus $m+2$ Portalen genau r auszuwählen und das sind nach Satz A.2.1

$$\binom{m+r+1}{r} \geq \binom{39}{20} = 68923264410$$

Bedenken wir, daß wir diese Möglichkeiten kombiniert für jede der vier Seiten eines Quadrates wählen müssen, so erhalten wir $\approx 2.25665 \cdot 10^{43}$ Möglichkeiten. Unter der Annahme, daß ein moderner Rechner 10 Milliarden Rechenschritte pro Sekunde schafft, erhalten wir eine Rechenzeit von rund 10^{25} Jahren. Wir haben dabei noch lange nicht alle Möglichkeiten aufgezählt, die der Algorithmus wirklich durchlaufen muß. Eine direkte Implementierung des Algorithmus erweist sich daher als wenig sinnvoll.

Kapitel 4

Der Algorithmus auf einem Parallelrechner

In diesem Kapitel wollen wir untersuchen, wie sich das sequentielle PTAS aus Kapitel 3 auf einem Parallelrechner implementieren läßt. Wir führen dazu zuerst ein Parallelrechnermodell ein. Im zweiten Abschnitt beschreiben wir, wie wir einen Quadtree in $O(\log n)$ parallelen Schritten konstruieren können und im dritten Abschnitt untersuchen wir verschiedene Versionen des parallelen PTAS wovon die beste eine Laufzeit von $O(\log n)$ parallelen Schritten erreicht. Wir werden sehen, daß wir dazu allerdings den mächtigsten Prozessortyp, die CRCW PRAM, benötigen.

4.1 Einführung in das verwendete Parallelrechnermodell

Es gibt verschiedene Parallelrechnermodelle, so z. B. gerichtete azyklische Graphen, das Shared-Memory-Model oder das Netzwerkmodell. Eine gute Einführung in diese Modelle findet man z. B. in [17]. Wir haben uns hier für die PRAM (*parallel random-access machine*) als Shared-Memory-Model entschieden, da sie eine natürliche Erweiterung des grundlegenden sequentiellen Modells, der RAM, darstellt und auf ihr schon eine Vielzahl an Problemen gelöst ist. Eine PRAM besteht aus mehreren gleichen Prozessoren, von denen jeder seinen eigenen lokalen Speicher hat und sein eigenes lokales Programm ausführen kann. Die Kommunikation der Prozessoren findet dabei über einen gemeinsamen Speicher statt, der auch als globaler Speicher bezeichnet wird. Jeder Prozessor erhält einen Index, der die Nummer des Prozessors angibt und den der Prozessor selber auch kennt. Er kann also in seinem Programm mit seiner Nummer operieren. Die PRAM ist ein synchrones Shared-Memory-Model, d. h., daß alle Prozessoren synchron, durch einen zentralen Takt gesteuert, arbeiten.

Es gibt verschiedene Versionen von PRAM Modellen, die beschreiben, in welcher Weise auf die gleiche Zelle im gemeinsam genutzten Speicher zugegriffen werden kann. Das Mo-

dell mit der größten Einschränkung ist dabei die EREW (*exclusive read exclusive write*) PRAM, die keinen gleichzeitigen Zugriff mehrerer Prozessoren auf eine Zelle des globalen Speichers duldet. Die CREW (*concurrent read exclusive write*) PRAM erlaubt gleichzeitigen Zugriff nur bei einer Leseoperation und die CRCW (*concurrent read concurrent write*) PRAM erlaubt simultanen Zugriff beim Lesen und Schreiben. Gleichzeitiges Lesen einer Speicherzelle durch mehrere Prozessoren ist dabei unproblematisch - im Gegensatz zu einer Situation, in der mehrere Prozessoren versuchen in die gleiche Speicherzelle zu schreiben. Dazu gibt es drei Varianten der CRCW PRAM. Die *common* CRCW PRAM erlaubt gleichzeitiges Schreiben nur dann, wenn alle beteiligten Prozessoren den gleichen Wert schreiben. Bei der *arbitrary* CRCW PRAM gewinnt ein beliebiger Prozessor den Kampf um die Speicherzelle, während bei der *priority* CRCW PRAM angenommen wird, daß die Indizes der Prozessoren linear geordnet sind und nur der Prozessor mit dem kleinsten Index die Speicherzelle beschreiben darf.

Die Ausdrucksmächtigkeit der drei PRAM Modelle bildet eine echte Hierarchie. So gibt es Algorithmen für eine CRCW PRAM, die auf einer CREW PRAM nur mit Zeitverlust laufen und Algorithmen für eine CREW PRAM, die auf einer EREW PRAM ebenfalls nur mit Zeitverlust laufen, unabhängig davon wieviele Prozessoren man auch investiert. Weiter ist von den drei verschiedenen CRCW PRAM Modellen die common CRCW PRAM das Modell mit den größten Einschränkungen und die priority CRCW PRAM das Modell mit den meisten Freiheitsgraden. Andererseits läßt sich aber eine priority CRCW PRAM mit p Prozessoren durch eine EREW PRAM mit p Prozessoren mit einem Zeitverlust von $O(\log p)$ simulieren. Beweise für die gemachten Aussagen sowie Verweise auf die Originalarbeiten findet man z. B. in [17].

Es gibt verschiedene Kriterien, die die Leistungssteigerung eines parallelen Algorithmus in Bezug auf einen sequentiellen Algorithmus beschreiben. Sei P ein gegebenes Berechnungsproblem und n die Größe der Eingabe. Mit $T_1(n)$ bezeichnen wir die Laufzeit des besten bekannten sequentiellen Algorithmus für das Problem P. Sei A ein paralleler Algorithmus, der P in Zeit $T_p(n)$ auf einer PRAM mit p Prozessoren löst. Der von A erreichte *Speedup* ist dann definiert durch

$$S_p(n) \;:=\; \frac{T_1(n)}{T_p(n)}$$

Der Speedup ist maximal p, falls kein besserer sequentieller Algorithmus gefunden wird, denn ein sequentieller Algorithmus kann einen Schritt eines parallelen Algorithmus mit p Prozessoren in p Schritten simulieren. Unser Ziel ist es, parallele Algorithmen zu erstellen, die einen Speedup von nahezu p haben.

Ein weiteres wichtiges Maß für das Verhalten eines parallelen Algorithmus ist die von dem Algorithmus geleistete Arbeit. Genauer gesagt ist die *Arbeit* $W_p(n)$ eines parallelen Algorithmus definiert als das Produkt seiner Laufzeit und der Anzahl der benutzten Prozessoren, also

$$W_p(n) \;:=\; p \cdot T_p(n)$$

Ein drittes Leistungskriterium für einen parallelen Algorithmus A ist die *Effizienz*, die definiert ist durch

$$E_p(n) := \frac{T_1(n)}{p \cdot T_p(n)} = \frac{T_1(n)}{W_p(n)} = \frac{S_p(n)}{p}$$

Die Effizienz zeigt an, wie gewinnbringend die p Prozessoren von dem Algorithmus benutzt werden. Eine Effizienz von nahezu 1 bedeutet, daß der Algorithmus A mit p Prozessoren ungefähr p-mal schneller ist als mit nur einem Prozessor. Es folgt also, daß jeder Prozessor pro Zeiteinheit sinnvolle Arbeit verrichtet. Wünschenswert sind demnach Algorithmen die gleichzeitig eine hohe Geschwindigkeit und eine Effizienz von nahezu 1 erreichen. Wir wollen einen parallelen Algorithmus für ein Berechnungsproblem P als *optimal* bezeichnen, wenn für die Arbeit des Algorithmus gilt $W(n) = \Theta(T(n))$, wobei $T(n)$ die Laufzeit des besten bekannten sequentiellen Algorithmus ist. Ein Beispiel für einen Algorithmus dieser Art ist der Merge-Sort Algorithmus aus [8]. Er hat eine Laufzeit von $O(\log n)$ Schritten mit n Prozessoren. Der beste sequentielle Sortieralgorithmus hat eine Laufzeit von $O(n \log n)$. Es ergibt sich also ein Speedup von $O(n)$ sowie eine Effizienz von näherungsweise 1. Die Arbeit des Algorithmus ist $\Theta(n \log n)$ und damit ist er optimal.

Für die Darstellung der folgenden parallelen Algorithmen verwenden wir das *Work-Time Presentation Framework*, wie es in [17] dargestellt wird. Der Algorithmus wird dazu schrittweise sequentiell beschrieben, wobei in einem Schritt mehrere parallele Operationen ausgeführt werden können. Wir verwenden hierbei die **PARDO** Anweisung, um anzudeuten, daß die darauffolgenden Operationen parallel ausgeführt werden.

4.2 Parallele Konstruktion des Quadtrees

Wir wollen in diesem Abschnitt analog zum sequentiellen Fall die Konstruktion eines Quadtrees beschreiben. Genauer gesagt, erstellen wir hier nur ein „Skelett“ des Quadtrees. Wir werden sehen, daß dieses ausreicht für unseren parallelen TSP-Algorithmus.

Ein *Skelett* ist genauso aufgebaut wie ein Quadtree, abgesehen davon, daß jeder innere Knoten mindestens zwei belegte Unterquadrate hat und es keine leeren Quadrate gibt. Entfernen wir in einem Quadtree jedes Quadrat, das keinen Punkt enthält und schieben wir alle verbleibenden Pfade von Quadraten, die nur noch ein Unterquadrat haben zusammen, so erhalten wir einen Baum, in dem jeder innere Knoten mindestens Grad 2 hat, also ein Skelett. Auf umgekehrte Weise können wir auch aus dem Skelett einen Quadtree konstruieren. Eine Kante in dem Skelett entspricht dann eventuell mehreren Kanten im Quadtree. In Abbildung 4.1 ist der Quadtree aus Abbildung 3.2 als Baum dargestellt und in Abbildung 4.2 das zugehörige Skelett.

Wir wollen den möglichen vier Unterquadraten eines Quadrates jeweils eine *Positionsnummer* 1 bis 4 geben, damit wir auf ein bestimmtes Unterquadrat eindeutig zugreifen können. Die Positionsnummern sollen wie in Abbildung 4.3 gezeigt angeordnet sein.

Wir beschreiben jetzt, wie die Struktur des Skeletts parallel erzeugt werden kann. Dazu

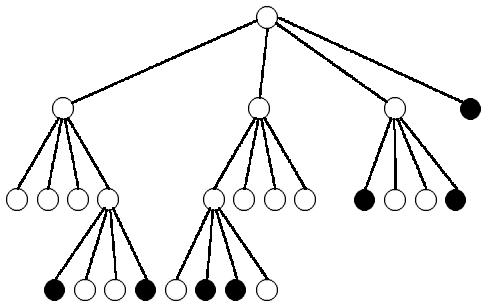


Abbildung 4.1: Der Quadtree

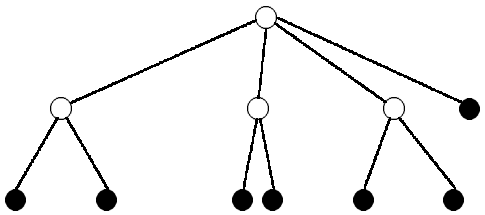


Abbildung 4.2: Das Skelett

nehmen wir an, daß die Koordinaten der Eingabepunkte ganzzahlig sind und die Größe der Bounding-Box linear in n ist. Wir vergrößern die Größe der B-Box auf die nächste Zweierpotenz. Sei L diese Größe. Es gilt dann immer noch $L \in O(n)$, da sich die Größe maximal verdoppelt hat. Nach Transformation des Ursprungs des Koordinatensystems in die linke untere Ecke der B-Box liegen alle Punkte in $[0, L[^2$. Da L eine Zweierpotenz ist, liegen die linken unteren Ecken von allen Quadraten an Koordinaten, die ganzzahlige Vielfache von Zweierpotenzen sind und die Größe aller Quadrate ist ebenfalls eine Zweierpotenz.

2	4
1	3

Abbildung 4.3: Die Positionsnummern

Mit einem *Level-i-Quadrat* wollen wir ein beliebiges Quadrat auf Ebene i im Quadtree bezeichnen. Ein Level-i-Quadrat hat Größe $\frac{L}{2^i}$. Sprechen wir von einem Quadrat, daß maximal ein Level-i-Quadrat ist, so bezieht sich das „maximal“ auf die Größe des Quadrates und nicht auf den Level.

Das *abgeleitete Quadrat* zu zwei gegebenen Punkten ist das kleinste Quadrat im Quadtree, das diese Punkte enthält. Die Größe und die linke untere Ecke des Quadrates lassen sich leicht bestimmen. Seien dazu (x_1, y_1) und (x_2, y_2) die Koordinaten von zwei Punkten. Sei i die Position der höchstwertigsten 1 in der Binärdarstellung von $(x_1 \oplus x_2) \vee (y_1 \oplus y_2)$, dann hat das abgeleitetet Quadrat Größe 2^{i+1} und die linke untere Ecke erhält man, indem man die unteren i Bits der Koordinaten von einem der beiden Punkte ausmaskiert.

Wir wollen den Eingabepunkten jetzt eine Reihenfolge geben, in der sie angeordnet sind. Sei (x, y) ein Punkt, dann ist der *gemischte Wert* des Punktes eine Mischung der x- und y-Koordinate des Punktes, die durch abwechselnde Wahl eines Bits von x und eines Bits von y entsteht. Begonnen wird die Mischung beim höchstwertigsten Bit der x-Koordinate. Die Reihenfolge der Punkte für ein Quadrat der Größe 8 ist in Abbildung 4.4 dargestellt. Der erste Punkt ist links unten und die folgenden Punkte befinden sich entlang der Ecken des Pfades bis zum letzten Punkt rechts oben.

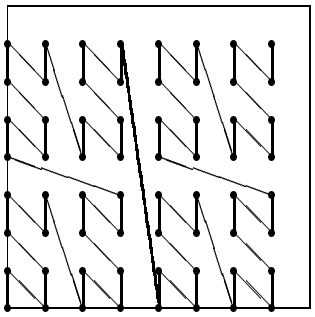


Abbildung 4.4: Die Reihenfolge der Punkte

Im ersten Schritt der Konstruktion des Skeletts werden die Eingabepunkte nach ihren gemischten Werten sortiert. Das kann in $O(\log n)$ Zeit mit n EREW Prozessoren geschehen [8]. Als nächstes werden wir zwei Lemmas beweisen, die Aussagen über die Punkte in der gemischt sortierten Reihenfolge machen.

Lemma 4.2.1 *Die Punkte, die in einem Quadrat im Quadtree liegen, bilden ein durchgehendes Intervall in der sortierten Reihenfolge.*

Beweis: Die Koordinaten der Punkte in einem Quadrat der Größe $\frac{L}{2^i}$, $i = 0, 1, ..., \log L$, haben in ihrer Binärdarstellung die gleichen höchstwertigen i Bits und es liegen alle Punkte mit den gleichen höchstwertigen i Bits in einem Quadrat der Größe $\frac{L}{2^i}$. Falls ein Punkt (x, y) außerhalb eines gegebenen Quadrates liegt, so muß eines seiner höchstwertigen i Bits in x oder y von denen der Punkte in diesem Quadrat abweichen. Ist dieses Bit 0, dann wird (x, y) vor allen Punkten des Quadrates in der sortierten Reihenfolge liegen und ist dieses Bit 1, dann wird sich (x, y) hinter allen Punkten des Quadrates befinden. Es ist also unmöglich für (x, y) *zwischen* den Punkten des Quadrates in der sortierten Reihenfolge zu liegen. Die Punkte in einem Quadrat liegen also in der gemischt sortierten Reihenfolge nebeneinander. □

Wie wir dem Lemma entnehmen können, liegen die Punkte aus einem Quadrat im Quadtree in der gemischt sortierten Reihenfolge nebeneinander. Wir wollen die Punkte in einem Level-i-Quadrat eine *Level-i-Gruppe* nennen und den ersten Punkt *Anführer der Gruppe*. Eine Level-i-Gruppe enthält maximal vier Level-$(i + 1)$-Gruppen, die den maximal vier Unterquadraten entsprechen. Die Level-$(i + 1)$-Gruppen treten dabei in der

Reihenfolge der Positionsnummeren der vier Unterquadrate auf. Die erste Level-$(i+1)$-Gruppe gehört also zu dem Unterquadrat an Position 1, falls dieses belegt ist, die nächste Gruppe zu dem an Position 2 und so weiter.

Lemma 4.2.2 *Falls mehr als ein Unterquadrat von Quadrat Q einen Punkt enthält, dann gibt es in der sortierten Reihenfolge zwei benachbarte Punkte, für die Q das abgeleitete Quadrat ist.*

Beweis: Nach Lemma 4.2.1 bilden die Punkte aus dem Quadrat Q ein durchgehendes Intervall in der sortierten Reihenfolge. Dieses Intervall kann in zwei, drei oder vier kleinere Intervalle eingeteilt werden, in denen sich dann jeweils die Punkte der vier Unterquadrate befinden. Q ist dann das abgeleitete Quadrat für jedes beliebige Punktepaar, das aus zwei verschiedenen der kleineren Intervallen gewählt wird. □

Nach Lemma 4.2.2 sind die abgeleiteten Quadrate für jedes Paar benachbarter Punkte in der sortierten Reihenfolge genau die Quadrate des Skeletts und die Struktur des Skeletts spiegelt sich wieder in der Einbettung der Intervalle, die durch Lemma 4.2.1 beschrieben ist. Um das Skelett zu erhalten, berechnen wir jetzt zu jedem benachbarten Punktepaar das abgeleitete Quadrat und tragen die Größe in eine Liste D ein. Wir erhalten die Quadratgrößen $w_1, ..., w_{n-1}$. Es können dabei bis zu dreimal die gleichen abgeleiteten Quadrate entstehen, denn falls ein Quadrat Q vier Unterquadrate hat, die Punkte aus Q also in vier Intervalle innerhalb der Gruppe von Q zerfallen, dann stoßen die Intervalle an drei Stellen aneinander. Es gibt dann also drei Punktepaare, die Q als abgeleitetes Quadrat haben. Wir müssen jetzt noch die Einbettung der Intervalle ineinander bestimmen. Dazu bestimmen wir zu jedem abgeleiteten Quadrat Q das nächstgrößere zu seiner linken Seite und zu seiner rechten Seite in der Liste D. Das kleinere der beiden ist dann das nächstgrößere Quadrat, in das Q eingebettet ist. Diese Berechnung ist eine all-nearest-larger-values Berechnung, die mit $n-1$ EREW Prozessoren $O(\log n)$ Zeit kostet [4]. Durch die Liste D und die all-nearest-larger-values ist das Skelett ausreichend beschrieben. Insgesamt benötigen wir n EREW Prozessoren und $O(\log n)$ parallele Zeit.

4.3 Beschreibung des parallelen Algorithmus

Wir wollen in einer ersten Version für den parallelen Algorithmus n EREW Prozessoren verwenden. Damit läßt sich die Vorgehensweise des Algorithmus am besten beschreiben. Später werden wir dann die Prozessorenzahl erhöhen und untersuchen, inwiefern sich dadurch die Laufzeit bzw. der Prozessortyp verändern.

Im Gegensatz zum sequentiellen Algorithmus soll der Quadtree hier nicht vorab berechnet werden, vielmehr wird immer nur das Quadrat konstruiert, das gerade benötigt wird. Der sequentielle Algorithmus durchläuft den Quadtree von der tiefsten Ebene startend nach oben und löst für alle Quadrate einer Ebene die (m,r)-Multitour Probleme. Für die Berechnung der Lösung zu einem Multitour Problem eines Quadrates auf Ebene i

benötigt er die Lösungen der Probleme auf Ebene $i+1$. Wir können also nicht alle Ebenen parallel, wohl aber die Quadrate auf einer Ebene parallel bearbeiten, da die zugehörigen Multitour Probleme unabhängig voneinander sind. Die Idee der ersten Version des parallelen Algorithmus ist es also, sequentiell über alle Ebenen zu laufen und die Multitour Probleme auf einer Ebene parallel zu lösen. Gestartet wird bei Ebene $\log L$. Es ist dabei immer ein Prozessor für einen Punkt zuständig und dieser löst das (m,r)-Multitour Problem für das Level-i-Quadrat, in dem der Punkt liegt. Auf höheren Ebenen würden so mehrere Prozessoren das gleiche Multitour Problem bearbeiten. Es genügt also, wenn nur die Anführer einer Ebene die Multitour Probleme für ihre Quadrate lösen.

Wir beschreiben den Algorithmus jetzt etwas genauer. Zuerst wird die Eingabe normiert wie bei der sequentiellen Version. Die Koordinaten der Punkte werden also ganzzahlig gemacht, der minimale Punkteabstand auf 4 gebracht und die Größe der Bounding-Box auf eine Zweierpotenz $L \in O(n)$ gebracht. Es ist dabei immer ein Prozessor für einen Punkt zuständig und damit ist die Laufzeit für die Normierung $O(1)$. Im zweiten Schritt werden die Punkte nach ihren gemischten Koordinaten sortiert. Seien $q_1, ..., q_n$ die Eingabepunkte in der sortierten Reihenfolge. Im folgenden kümmert sich Prozessor P_j um Punkt q_j. Ist Punkt q_j Anführer einer Gruppe, so bezeichnen wir den Prozessor P_j auch als Anführer dieser Gruppe.

Als nächstes wird die Liste $D = \{w_1, ..., w_{n-1}\}$ der abgeleiteten Quadrate in $O(1)$ Zeit berechnet. w_j ist dabei das abgeleitete Quadrat von q_j und q_{j+1}. Damit der Algorithmus in Randfällen keine Fehler macht, definieren wir $w_0 := 2L$. Auf D werden dann die all-nearst-larger-values in $O(\log n)$ Zeit bestimmt. Genaugenommen werden später nur die linken nächstgrößeren Werte benötigt. Sei dazu $ind(w_i)$ der größte Index kleiner i, so daß $w_{ind(w_i)} > w_i$ ist. $ind(w_i)$ ist also der Index des nächstgrößeren links von w_i liegenden Wertes. Spätestens w_0 ist größer als w_i und deshalb ist $ind(w_i) \geq 0$ für alle i. Es soll gelten $ind(w_0) = 0$. Die Prozedur `Init` übernimmt die Aufgabe der Sortierung, der Erstellung der Liste D der abgeleiteten Quadrate und der Bestimmung der left-nearest-larger-values. Zusätzlich wird noch eine Transfertabelle T erstellt, auf deren Bedeutung wir später eingehen werden.

Der Algorithmus läuft jetzt sequentiell über alle Ebenen $i = \log L, ..., 0$ und nur die Prozessoren, die Anführer einer Level-i-Gruppe sind, bearbeiten die Multitour Probleme. Die anderen Prozessoren „legen sich schlafen“. Auf den untersten Ebenen ist jeder Prozessor solange Anführer, bis die Quadrate so groß werden, daß mindestes zwei Punkte in einem Quadrat liegen. Dann ist nur noch einer der beiden Prozessoren der Anführer dieses Quadrates. Ist ein Prozessor Anführer einer Level-i-Gruppe, so ist er auch Anführer einer Level-j-Gruppe für alle $j \geq i$. Ein Prozessor kann leicht testen ob, er Anführer einer Level-i-Gruppe ist. Der einzige Anführer der Level-0-Gruppe, also der Anführer aller Punkte, ist q_1. Für die restlichen Punkte gilt folgendes Lemma:

Lemma 4.3.1 *Sei q_j, $2 \leq j \leq n$, ein Punkt einer Level-i-Gruppe. Es ist q_j genau dann Anführer dieser Gruppe, wenn $w_{j-1} \geq \frac{L}{2^{i-1}}$ ist.*

Beweis: Sei Q ein Level-i-Quadrat. Das abgeleitete Quadrat zweier beliebiger Punkte

aus Q ist dann maximal ein Level-i-Quadrat, denn kommen die Punkte aus verschiedenen Unterquadraten von Q, so erhalten wir genau Q als das abgeleitete Quadrat, befinden sie sich dagegen in demselben Unterquadrat, so erhalten wir ein kleineres Quadrat als Q als abgeleitetes Quadrat. Ein Anführer von Q ist dadurch charakterisiert, daß das abgeleitete Quadrat von ihm und seinem linken Nachbarn mindestens ein Level-$(i-1)$-Quadrat ist. Es ist q_j also genau dann Anführer, wenn das abgeleitete Quadrat von q_{j-1} und q_j mindestens ein Level-$(i-1)$-Quadrat ist, also $w_{j-1} \geq \frac{L}{2^{i-1}}$ ist. □

PROCEDURE `ParApproxETSP` (K : Koordinatenliste; c, opt : real);

BEGIN
`Normierung`(K);
L := Größe der Bounding-Box um K;
$opt := \infty$;
FOR ALL Shifts $0 \leq a, b < L$ **DO**
Shifte Punkte in K um (a, b) nach links unten;
`Init`(K, D, T);
FOR ALL Level $i = \log L$ **DOWNTO** 1 **DO**
FOR ALL Anführer q_j einer Level-i-Gruppe **PARDO**
Q_{ij} := Level-i-Quadrat um Punkt q_j;
FOR ALL Eingabeinstanzen I_k des (m, r)-Multitour Problems in Q_{ij} **DO**
IF $i = \log L$ **THEN** $C[i, j, k] := cost(i, \{q_j\}, I_k)$ **ELSE** `CalcCost`(Q_{ij}, I_k);
OD
pos := Positionsnummer von Q_{ij} im Level-$(i-1)$-Quadrat um Punkt q_j;
$T[i, anf(i-1, j), pos] := j$;
OD
OD
`CalcCost`(Q_{01}, I_0); {* Q_{01} = B-Box, I_0 = leere Eingabeinstanz *}
$opt := \min(opt, C[0, 1, 0])$;
OD
END

Sind die Anführer auf Ebene i bestimmt, so muß jeder Anführer das (m, r)-Multitour Problem für das Level-i-Quadrat um seinen Punkt lösen. Genau wie im sequentiellen Fall müssen wir wieder die Lösungen zu allen möglichen Eingabeinstanzen I_k des Multitour Problems berechnen. Die Kosten der Lösungen werden diesmal in eine Lookup-Table C eingetragen. Ein Eintrag an Position $C[i, j, k]$ enthält die Kosten der optimalen Lösung des Multitour Problems für das Level-i-Quadrat um Punkt q_j bei Eingabeinstanz I_k. Die Prozedur `ParApproxETSP` beschreibt diesen Vorgang. Sie gebraucht dazu wieder eine Unterroutine `CalcCost`, die die optimalen Kosten für ein Quadrat und eine Eingabeinstanz berechnet.

Bei der Lösung des Multitour Problems für ein Quadrat und eine Eingabeinstanz tritt ein Problem auf, da die Lösungen der Teilprobleme benötigt werden. Wie kommt der aktuelle Prozessor aber an diese Lösungen heran? Auf Ebene $\log n$ entsteht kein Problem,

PROCEDURE `CalcCost` (Q_{ij} : Quadrat; I_k : Eingabeinstanz);

{* Berechnet die minimalen Kosten für eine Eingabeinstanz I_k des *}
{* (m,r)-Multitour Problems in Q_{ij} und speichert sie in $C[i,j,k]$. *}

BEGIN
  $C[i,j,k] := \infty$;
  FOR ALL Eingabeinstanzen $I_{k_1}, I_{k_2}, I_{k_3}, I_{k_4}$ der Multitour Probleme in den vier Unterquadraten von Q_{ij}, die sich mit I_k decken **DO**
    $M := 0$;
    FOR $l := 1$ **TO** 4 **DO**
      $next := T[i+1,j,l]$; {* Index des l-ten Unterquadrates *}
      IF $next$ nicht definiert **THEN** inc$(M, cost(i+1,\{\},I_{k_l}))$ {* $Q_{i+1,next}$ ist leer *}
      ELSE inc$(M, C[i+1,next,k_l])$;
    OD;
    $C[i,j,k] := \min(M, C[i,j,k])$;
  OD
END

da das Level-$(\log n)$-Quadrat um einen Punkt nur diesen Punkt enthält und die optimale Lösung des Multitour Problems direkt bestimmt werden kann. Für höhere Ebenen werden aber die optimalen Lösungen der vier Unterquadrate benötigt. Sei P_j ein Anführer eines Level-i-Quadrates Q und seien $P_{j_1}, ..., P_{j_l}$, $l \leq 4$, die maximal vier Anführer der vier Level-$(i+1)$-Unterquadrate von Q. Die Anzahl der Anführer muß nicht genau vier sein, da nur die Unterquadrate, die mindestens einen Punkt enthalten, einen Anführer haben. Einer der Anführer ist natürlich P_j selber, und zwar der Anführer des Unterquadrates mit der kleinsten Positionsnummer. Die optimalen Lösungen der Multitour Probleme für die belegten Unterquadrate sind in der Lookup-Table an den Positionen $C[i+1,j_1,.], ..., C[i+1,j_l,.]$ zu finden. Die optimalen Lösungen der leeren Unterquadrate kann P_j selbst berechnen. Damit Prozessor P_j an die Indizes $j_1, ..., j_l$ der Anführer der Unterquadrate herankommt, führen wir eine Transfertabelle T ein. Ein Eintrag $T[i,j,pos]$, $pos = 1,2,3,4$, enthält den Index des Prozessors, der auf Ebene i Anführer eines Level-i-Quadrates war, das im Level-$(i-1)$-Quadrat Positionsnummer pos hat und dessen neuer Anführer P_j ist. Zu Anfang wird jeder Eintrag der Transfertabelle durch die Prozedur `Init` auf „nicht definiert“ gesetzt. Damit die Tabelle auch beschrieben wird, muß jeder Anführer einer Level-i-Gruppe seinen Index in der Tabelle an die Position des Anführers der übergeordneten Level-$(i-1)$-Gruppe schreiben. Ist er dort auch Anführer, so muß er seinen eigenen Index eintragen. Das nächste Lemma zeigt, wie wir sonst den Index des übergeordneten Anführers finden.

Lemma 4.3.2 *Sei q_j Anführer einer Level-i-Gruppe, $i \geq 1$, der auf Ebene $i-1$ kein Anführer mehr ist, dann ist $ind(w_{j-1})+1$ der Index des Anführers der übergeordneten Level-$(i-1)$-Gruppe.*

Beweis: Wenn q_j Anführer einer Level-i-Gruppe, $i \geq 1$, ist, dann gilt nach Lemma 4.3.1 : $w_{j-1} \geq \frac{L}{2^{i-1}}$. Sei q_l der Anführer der übergeordneten Level-$(i-1)$-Gruppe. Das abgeleitete Quadrat zweier Punkte einer Level-$(i-1)$-Gruppe ist maximal ein Level-$(i-1)$-Quadrat. Es gilt also $w_l, ..., w_{j-1} \leq \frac{L}{2^{i-1}}$ und daraus folgt $w_{j-1} = \frac{L}{2^{i-1}}$. Da $w_{l-1} \geq \frac{L}{2^{i-2}} > \frac{L}{2^{i-1}} = w_{j-1}$ und die Quadratgrößen w_l bis w_{j-1} nicht größer als w_{j-1} sind, ist das nächstgrößere Quadrat links von w_{j-1} gerade w_{l-1}. Es ist also $ind(w_{j-1}) = l-1$ und damit $q_{ind(w_{j-1})+1} = q_l$. □

Wir wissen jetzt, wie wir den Anführer der übergeordneten Gruppe finden können. Ist q_j ein Anführer auf Ebene i, dann wollen wir mit $anf(i-1, j)$ den Index des Anführers der übergeordneten Level-$(i-1)$-Gruppe bezeichnen. Es gilt:

$$anf(i-1,j) := \begin{cases} \text{n.d.} & \text{falls } q_j \text{ kein Anführer auf Ebene } i, \\ j & \text{falls } q_j \text{ Anführer auf Ebene } i \text{ und } i-1, \\ ind(w_{j-1})+1 & \text{sonst} \end{cases}$$

Ein Prozessor P_j, der Anführer auf Ebene i ist, kann über die Tabelle seinen Index dem Anführer auf Ebene $i-1$ mitteilen. Er setzt dazu $T[i, anf(i-1,j), pos] := j$, wobei pos die Positionsnummer des Level-i-Quadrates im übergeordneten Level-$(i-1)$-Quadrat ist. Die Positionsnummer kann P_j dabei durch eine AND-Verknüpfung der Größe des Level-i-Quadrates mit der linken unteren Ecke des Quadrates herausfinden.

In der Beschreibung der Prozedur `ParApproxETSP` und `CalcCost` benutzen wir der Übersichtlichkeit wegen die Funktion $cost(i, \{q\}, I_k)$, die die minimalen Kosten der Lösung des Multitour Problems in dem Level-i-Quadrat um Punkt q bei Eingabeinstanz I_k angibt, bzw. die Funktion $cost(i, \{\}, I_k)$, die die Kosten der Lösung für ein leeres Level-i-Quadrat bei Eingabeinstanz I_k angibt. Die Funktion $cost$ kann wie im sequentiellen Fall in konstanter Zeit berechnet werden.

Wir haben bis jetzt noch nichts dazu gesagt, wie der Algorithmus den geshifteten Quadtree behandelt. Genau wie im sequentiellen Fall shiften wir die Eingabepunkte nach links unten anstatt den Quadtree nach rechts oben zu shiften. Immer dann, wenn auf einem Quadrat ein Multitour Problem gelöst werden soll, müssen wir dieses Quadrat wieder nach rechts oben shiften und das Problem auf dem geshifteten Quadrat lösen.

Wir wollen jetzt den Zeitaufwand unserer ersten Version des parallelen Algorithmus betrachten. Die Prozedur `Init` wird auf die n Prozessoren verteilt. Sie benötigt dann $O(\log n)$ Zeit. Die erste Schleife läuft sequentiell über alle $L^2 \in O(n^2)$ Shifts (a, b) und die zweite Schleife über alle $O(\log n)$ Ebenen. In der nächsten Schleife läuft der Algorithmus parallel über alle Quadrate einer Ebene im Quadtree und löst dabei jeweils, genau wie im sequentiellen Algorithmus, alle möglichen Multitour Probleme. Wie wir bei der sequentiellen Analyse gesehen haben, müssen dafür $(\log n)^{O(c)}$ Schritte aufgewendet werden. Der gesamte Zeitbedarf ergibt sich also zu $O\left(n^2 (\log n)^{O(c)}\right)$ Schritten.

Da immer nur die Anführer einer Gruppe die Multitour Probleme lösen, ist sichergestellt, daß jeder Prozessor mit seinen eigenen Daten beschäftigt ist. Es reichen also EREW

Prozessoren aus. Der Exponent $O(c)$ ist im sequentiellen Algorithmus der gleiche wie im parallelen Algorithmus, da die Multitour Probleme durch die gleichen Schleifen aufgezählt werden. Wir erhalten somit einen Speedup von $O(n)$ und eine Effizienz von näherungsweise 1. Der Algorithmus ist also optimal bezüglich unserer sequentiellen Version. Wie verhält sich der Algorithmus aber, wenn wir den Grad der Parallelisierung steigern, also mehr Prozessoren investieren, um damit die Laufzeit zu veringern ?

Wir wollen $O(L^2) = O(n^2)$ weitere Prozessoren hinzuziehen, und zwar einen für jeden Shift (a, b). Die Berechnungen im Quadtree sind für jeden Shift unabhängig voneinander, womit wir weiterhin EREW Prozessoren verwenden können. Die Tabellen C und T müssen um die Indizes a und b erweitert werden, da sie nicht für jeden geshifteten Quadtree überschrieben werden können. Die einzige Kommunikation der neuen Prozessoren besteht darin, daß am Ende aller Berechnungen die kürzeste von allen Lösungen ausgegeben werden muß. Die Länge des kürzesten (m, r)-light Salesman Pfades zu einem Shift (a, b) ist in der Lookup-Table an der Stelle $C[a, b, 0, 1, 0]$ zu finden. Es muß das Minimum zu allen $O(n^2)$ Shifts (a, b) dieser Tabelleneinträge gefunden werden. Das gelingt mit $O(n^2)$ EREW Prozessoren in $O(\log n)$ Zeit [17]. Wir kommen damit letztlich auf einen Gesamtzeitbedarf von $(\log n)^{O(c)}$ parallelen Schritten mit $O(n^3)$ EREW Prozessoren und erhalten wieder einen optimalen Algorithmus.

PROCEDURE `ParApproxETSP2` (K : Koordinatenliste; c, opt : real);

BEGIN
  `Normierung`(K);
  L := Größe der Bounding-Box um K;
  FOR ALL Shifts $0 \leq a, b < L$ **PARDO**
    Shifte Punkte in K um (a, b) nach links unten;
    `Init`(K, D, T);
    FOR ALL Level $i = \log L$ **DOWNTO** 1 **DO**
      FOR ALL Anführer q_j einer Level-i-Gruppe **PARDO**
        Q_{ij} := Level-i-Quadrat um Punkt q_j;
        FOR ALL Eingabeinstanzen I_k des Multitour Problems in Q_{ij} **PARDO**
          IF $i = \log L$ **THEN** $C[a, b, i, j, k] := cost(i, \{q_j\}, I_k)$
          ELSE `ParCalcCost`(a, b, Q_{ij}, I_k);
        OD
        pos := Positionsnummer von Q_{ij} im Level-$(i-1)$-Quadrat um Punkt q_j;
        $T[i, anf(i-1, j), pos] := j$;
      OD
    OD
    `ParCalcCost`(a, b, Q_{01}, I_0); {* I_0 leere Eingabeinstanz *}
  OD
  opt := `CalcMinimum` $(C[a, b, 0, 1, 0])$; {* Minimum für alle Shifts (a, b) *}
END

PROCEDURE `ParCalcCost` (a, b : Shifts; Q_{ij} : Quadrat; I_k : Eingabeinstanz);

BEGIN
  FOR ALL Eingabeinstanzen $I_{k'} = (I_{k_1}, I_{k_2}, I_{k_3}, I_{k_4})$ der Multitour Probleme in den vier Unterquadraten von Q_{ij}, die sich mit I_k decken **PARDO**
    $M[k'] := 0$;
    FOR $l := 1$ **TO** 4 **DO**
      $next := T[i+1, j, l]$; {* Index des l-ten Unterquadrates *}
      IF $next$ nicht definiert **THEN** inc$(M[k'], cost(i+1, \{\}, I_{k_l}))$
      ELSE inc$(M[k'], C[a, b, i+1, next, k_l])$;
    OD;
  OD
  $C[a, b, i, j, k] :=$ `CalcMinimum` (M); {* Minimum von $M[k']$ für alle k' *}
END

Die derzeitige Version des parallelen Algorithmus bearbeitet ein Quadrat im Quadtree immer noch sequentiell. Es werden alle möglichen Eingabeinstanzen für ein (m, r)-Multitour Problem in einem Quadrat der Reihe nach aufgezählt und die Probleme mit Hilfe der Unterquadrate gelöst. Wir wollen nun alle möglichen Eingabeinstanzen für ein Multitour Problem in Q parallel bearbeiten. Dabei tritt das Problem auf, daß jeder Prozessor, der eine Eingabeinstanz für das Multitour Problem in Q bearbeitet, die Nummern der Unterquadrate aus der Tabelle T und die Lösungen der Probleme in den Unterquadraten auslesen muß. Es kann hierbei vorkommen, daß die Prozessoren aus gleichen Speicherzellen gleichzeitig lesen. Die berechneten Lösungen für Q werden dann allerdings stets an verschiedenen Stellen in die Lookup-Table C geschrieben - und zwar immer an die Stellen, die den Nummern der Eingabeinstanzen entsprechen. Wir werden für eine erfolgreiche Parallelisierung daher CREW Prozessoren investieren müssen.

Zählen wir die möglichen Eingabeinstanzen für die Multitour Probleme in den vier Unterquadrate von Q auch noch parallel auf, so müssen wir uns zu jeder Möglichkeit $I_{k'}$, die vier Eingabeinstanzen zu wählen, die zugehörigen Kosten merken und am Ende das Minimum in die Lookup-Table eintragen. Wir speicheren die Kosten dazu jeweils in $M[k']$ und berechnen am Schluß das Minimum von allen $M[k']$. Wie beim sequentiellen Algorithmus schon gezeigt, ist die Anzahl der Möglichkeiten, die vier Eingabeinstanzen zu wählen $(\log n)^{O(c)}$. Es muß also das Minimum von $(\log n)^{O(c)}$ Werten bestimmt werden. Das Minimum parallel zu berechnen, benötigt dann Zeit $O\left(\log(\log n)^{O(c)}\right) = O(\log \log n)$ mit $(\log n)^{O(c)}$ EREW Prozessoren. Insgesamt ereichen wir eine parallele Laufzeit von $O(\log n \log \log n)$ mit $O\left(n^3 (\log n)^{O(c)}\right)$ CREW Prozessoren. Der Algorithmus ist jetzt allerdings nicht mehr optimal. Der Exponent $O(c)$ ist im Prozessor-Zeit-Produkt zwar der gleiche wie bei der sequentiellen Laufzeit (wir investieren ja genauso viele Prozessoren, wie der sequentielle Algorithmus Schleifendurchläufe hat), aber der Faktor $\log \log n$ in der parallelen Laufzeit verhindert die Optimalität. Die Prozeduren `ParApproxETSP2` und `ParCalcCost` stellen die beschriebene endgültige Version des parallelen Approximations-

algorithmus dar. Dabei steht die Unterroutine `CalcMinimum` für die beschriebene Minimumsberechnung.

Die Frage, die sich jetzt noch stellt, ist, ob wir mit CRCW Prozessoren noch eine Verbesserung erzielen können. Es ist bekannt, daß das Minimum von n Zahlen mit $O(n^2)$ common CRCW Prozessoren in $O(1)$ Zeit bestimmt werden kann, mit beliebig vielen CREW Prozessoren die Laufzeit allerdings nicht besser als $O(\log n)$ sein kann [17]. Nutzen wir diese Tatsachen bei der Minimumsberechnung in `ParCalcCost` aus, so erreichen wir eine Gesamtlaufzeit von $O(\log n)$ mit $O\left(n^3(\log n)^{O(c)}\right)$ CRCW Prozessoren. Die Minimumsberechnung in `ParApproxETSP2` belassen wir aber bei der alten Methode, da dort eine Minimumsberechnung in $O(1)$ Zeit keine Verbesserung in der Gesamtlaufzeit hervorrufen würde, sondern nur die Prozessorenzahl unnötig steigern würde. Optimal ist der Algorithmus allerdings nicht, da bei der Minimumsberechnung zusätzliche Prozessoren benötigt werden, die bei der Prozessorenzahl im Exponenten $O(c)$ „versteckt“ sind.

Kapitel 5

Zusammenfassung und Ausblick

Zu Beginn haben wir Arora's PTAS für das Euklidische TSP in Anlehnung an seine Arbeit [1] genau beschrieben und untersucht. Wir haben dabei zuerst festgestellt, daß bei dem Euklidischen TSP allgemein technische Schwierigkeiten entstehen, da eine exakte Berechnung der Länge einer Rundreise eine Rechnung mit ungerundeten reellen Zahlen erfordert. Wir haben dieses Problem gelöst, indem wir mit aufgerundeten Distanzen rechnen und die dadurch entstehende Abweichung von der exakten Länge bestimmt und die Genauigkeitsanforderung c dementsprechend angepaßt haben.

Die grundlegende Datenstruktur auf der Arora's PTAS arbeitet ist der Quadtree. Wir haben diese Datenstruktur genau analysiert und einen effizienten Algorithmus angegeben, der einen Quadtree aus einer Menge von Punkten erstellt. Ist die Größe der Bounding-Box um die Punkte linear in n und haben die Punkte paarweise mindestens Abstand d für eine Konstante d, dann hat der Algorithmus zur Konstruktion des Quadtrees eine worst-case Laufzeit von $O(n \log n)$ und es ist nicht möglich einen schnelleren Algorithmus für dieses Problem anzugeben.

Das PTAS benutzt dynamische Programmierung, um die Multitour Probleme in den Quadraten des Quadtrees zu lösen. Wir zeigen, daß die Anzahl der möglichen Multitour Probleme in einem Quadrat durch $O\left((\log n)^{O(c)}\right)$ beschränkt ist; multipliziert mit der Anzahl an Quadraten und der Anzahl an möglichen Shifts (a, b) erhalten wir dann eine Gesamtlaufzeit des PTAS von $O\left(n^3 (\log n)^{O(c)}\right)$.

Die Grundlage für das PTAS ist das Struktur-Theorem, daß uns die Existenz eines (m, r)-light Salesman Pfades bescheinigt, dessen Kosten die optimalen Kosten bis auf einen Faktor von $1 + \frac{1}{c}$ annähern. Um den Salesman Pfad (m, r)-light zu machen, müssen wir verhindern, daß er einige Kanten in der Dissection „zu oft“ schneidet. Wir benutzen dazu das Patching-Lemma, das wir in dieser Arbeit in einer neuen, schärferen Version dargestellt haben, um die Anzahl der Schnitte auf zwei zu reduzieren, ohne dabei den Pfad wesentlich zu verlängern.

Im zweiten und neuen Teil der Arbeit haben wir untersucht, wie sich Arora's PTAS auf einem Parallelrechner implementieren läßt und welche Laufzeitvorteile dadurch zu erwarten sind. Als Parallelrechnermodell haben wir hier die PRAM gewählt, da auf ihr

schon eine Fülle an Algorithmen existieren.

Zuerst haben wir uns hier in Anlehnung an die Arbeit von Bern, Eppstein und Teng [5] mit der parallelen Konstruktion eines Quadtrees beschäftigt. Wir haben gezeigt, daß das Skelett des Quadtrees durch die Liste der abgeleiteten Quadrate mit den zugehörigen all-nearest-larger-values ausreichend bestimmt ist und diese Berechnung mit n EREW Prozessoren in $O(\log n)$ paralleler Zeit möglich ist.

Bei der Parallelisierung des PTAS sind wir schrittweise vorgegangen. So haben wir in der ersten Version nur die Quadrate einer Ebene im Quadtree parallel bearbeitet. Das Hauptproblem dabei war, daß für die Lösung der Multitour Probleme in einem Quadrat die Lösungen der Multitour Probleme in den vier Unterquadraten benötigt werden und es für den aktuellen Prozessor nicht so ohne weiteres möglich ist, an diese Lösungen heranzukommen. Wir haben dafür die Transfertabelle eingeführt, durch die der aktuelle Prozessor die Nummern der Anführer der vier Unterquadrate erhält.

Die erste Version kommt mit n EREW Prozessoren aus und benötigt dabei eine parallele Zeit von $O\left(n^2(\log n)^{O(c)}\right)$ Schritten. Ohne Investition eines mächtigeren Prozessortyps können wir den Parallelisierungsgrad auf $O(n^3)$ EREW Prozessoren, bei einer parallelen Laufzeit von $O\left((\log n)^{O(c)}\right)$ Schritten, steigern. Investieren wir CREW Prozessoren, so genügen $O\left(n^3(\log n)^{O(c)}\right)$ Prozessoren, um eine Laufzeit von $O(\log n \log\log n)$ Schritten zu erzielen und mit common CRCW Prozessoren reichen sogar $O(\log n)$ Schritte aus bei asymptotisch gleicher Prozessorenzahl.

Bei der Untersuchung der Effizienz des PTAS hat sich eine schlechte Praktikabilität gezeigt, da der Exponent $O(c)$ für kleine c schon zu groß wird. Bei der parallelen Implementierung ist es nicht anders. Wir erreichen zwar eine sehr gute Laufzeit von $O(\log n)$ Schritten, müssen dafür aber eine Anzahl an Prozessoren investieren, die die Anzahl der Atome im Universum überschreitet. Es wäre demnach interessant, einen schnelleren PTAS für das Euklidische TSP zu suchen bzw. eine Kombinierbarkeit des PTAS mit bekannten Heuristiken zu untersuchen.

Ein anderer interessanter zu untersucheneder Aspekt ist die Übertragbarkeit des PTAS auf andere euklidische Optimierungsprobleme, wie zum Beispiel *Minimum Weight Euclidean Triangulation* oder *Vehicle Routing*.

Anhang A

Mathematische Grundlagen

A.1 Summenformeln

Satz A.1.1 (geometrische Summenformel) *Ist $q \neq 1$ eine reelle Zahl, dann gilt:*

$$\sum_{i=0}^{n} q^i \;=\; \frac{q^{n+1}-1}{q-1}$$

Beweis: $(q-1)\cdot\sum_{i=0}^{n} q^i \;=\; \sum_{i=1}^{n+1} q^i - \sum_{i=0}^{n} q^i \;=\; \sum_{i=1}^{n} q^i + q^{n+1} - \left(q^0 + \sum_{i=1}^{n} q^i\right) \;=\; q^{n+1}-1$ □

Satz A.1.2 *Für alle Zahlen a_{ij} gilt:*

$$\sum_{i=1}^{n}\sum_{j=i}^{n} a_{ij} \;=\; \sum_{j=1}^{n}\sum_{i=1}^{j} a_{ij}$$

Beweis: Beide Seiten der Gleichung stellen die Summe der Elemente a_{ij} einer oberen Dreiecksmatrix A dar.

$$A := \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ & a_{22} & \cdots & a_{2n} \\ & & \ddots & \vdots \\ & & & a_{nn} \end{pmatrix}$$

Der einzige Unterschied zwischen den beiden Seiten ist, daß in der linken Summe die Elemente zeilenweise aufsummiert werden und in der rechten Summe spaltenweise. □

A.2 Binomialkoeffizienten

Satz A.2.1 *Die Anzahl der k-elementigen Multiteilmengen einer m-elementigen Menge ist* $\binom{m+k-1}{k}$.

Beweis: Das Problem ist in der Literatur als Kombinationen mit Wiederholung zu finden. Für einen Beweis siehe [3]. □

Lemma A.2.2 $\sum_{k=0}^{r} \binom{m+k-1}{k} = \binom{m+r}{r}$.

Beweis: Das Lemma läßt sich mit vollständiger Induktion über r beweisen. Für $r = 0$ gilt:

$$\sum_{k=0}^{0} \binom{m+k-1}{k} = \binom{m-1}{0} = 1 = \binom{m}{0}$$

Als Induktionsvoraussetzung gilt jetzt das Lemma für r. Damit zeigen wir die Behauptung für $r+1$:

$$\sum_{k=0}^{r+1} \binom{m+k-1}{k} = \sum_{k=0}^{r} \binom{m+k-1}{k} + \binom{m+r}{r+1} \overset{\text{IV}}{=} \binom{m+r}{r} + \binom{m+r}{r+1} = \binom{m+r+1}{r+1}$$

Das letzte Gleichheitszeichen gilt aufgrund des Additionstheorems für Binomialkoeffizienten, das allgemein bekannt sein dürfte. □

Lemma A.2.3 $\binom{m+r}{r} \leq (m+1)^r$

Beweis: Das Lemma wird durch vollständige Induktion über r bewiesen. Für $r = 0$ gilt:

$$\binom{m}{0} = 1 = (m+1)^0$$

Als Induktionsvoraussetzung gilt jetzt die Behauptung des Lemmas für r. Damit zeigen wir die Behauptung für $r+1$:

$$\begin{aligned}\binom{m+r+1}{r+1} &= \frac{(m+r+1)!}{m!(r+1)!} = \frac{m+r+1}{r+1} \cdot \frac{(m+r)!}{m!r!} = \frac{m+r+1}{r+1} \cdot \binom{m+r}{r} \\ &\overset{\text{IV}}{\leq} \frac{m+r+1}{r+1}(m+1)^r \leq (m+1)(m+1)^r = (m+1)^{r+1}\end{aligned}$$

□

A.3 Wahrscheinlichkeitsrechnung

Definition A.3.1 Ein *endlicher Wahrscheinlichkeitsraum* ist ein Paar (Ω, P), das aus einer endlichen Menge Ω und einer Abbildung P der Potenzmenge $\mathfrak{P}(\Omega)$ in das Einheitsintervall $[0,1]$ besteht mit den folgenden Eigenschaften:

(i) $P(\Omega) = 1$,

(ii) Für disjunkte Mengen $A, B \in \mathfrak{P}(\Omega)$ gilt: $P(A \cup B) = P(A) + P(B)$

Die Menge Ω heißt *Ergebnisraum*, die Funktion P die *Wahrscheinlichkeitsverteilung* über Ω und jede Teilmenge E von Ω ein *Ereignis*. Ist $P(\{\omega\}) = \frac{1}{|\Omega|}$ für alle $\omega \in \Omega$, dann spricht man von einer *Gleichverteilung*.

Im folgenden beschränken wir uns auf endliche Wahrscheinlichkeitsräume.

Bemerkung: Ist P eine Gleichverteilung, dann gilt für alle $E \in \mathfrak{P}(\Omega)$: $P(E) = \frac{|E|}{|\Omega|}$

Definition A.3.2 Eine Abbildung $X : \Omega \longrightarrow \mathbb{R}$ heißt *Zufallsvariable*. Wir schreiben $P(X \perp a)$ für $P(\{\omega | X(\omega) \perp a\})$, wobei $\perp \in \{=, \neq, <, >, \leq, \geq\}$.

Definition A.3.3 Der *Erwartungswert* einer Zufallsvariable X ist

$$E(X) := \sum_{\omega \in \Omega} X(\omega) \cdot P(\{\omega\})$$

Satz A.3.4 *Seien X und Y Zufallsvariablen, dann gilt:* $E(X+Y) = E(X) + E(Y)$

Beweis:

$$E(X+Y) = \sum_{\omega \in \Omega} (X+Y)(\omega) \cdot P(\{\omega\}) = \sum_{\omega \in \Omega} (X(\omega) + Y(\omega)) \cdot P(\{\omega\}) = E(X) + E(Y)$$

□

Satz A.3.5 (Markoffsche Ungleichung) *Nimmt die Zufallsvariable X nur nichtnegative Werte an und ist $E(X)$ ihr Erwartungswert, dann gilt für jede positive Zahl λ*

$$P(X \geq \lambda \cdot E(X)) \leq \frac{1}{\lambda}$$

Beweis: Für einen Beweis des Satzes siehe z. B. [22] oder [12] □

Beispiel A.3.6 Betrachten wir das Ergebnis eines Wurfs mit zwei Würfeln. Für den Ergebnisraum gilt: $\Omega = \{(i,k)|1 \leq i,k \leq 6\} = \{1,2,3,4,5,6\}^2$. Die Wahrscheinlichkeitsverteilung P ergibt sich hier zu $P(\omega) = \frac{1}{36}$ für alle $\omega \in \Omega$, da wir annehmen wollen, daß die Würfel nicht gezinkt sind. Betrachten wir jetzt die Zufallsvariable X, die der Augensumme der beiden Würfel entsprechen soll, also $X(i,k) := i+k$ für $(i,k) \in \Omega$. Fragen wir nun z. B. nach der Wahrscheinlichkeit mit der die Augensumme gleich 6 ist, so müssen wir $P(X=6)$ berechnen:

$$P(X=6) \;=\; P(\{(i,k)|X(i,k)=6\}) \;=\; P(\{(1,5),(2,4),(3,3),(4,2),(5,1)\}) \;=\; \frac{5}{36}$$

Der Erwartungswert $E(X)$ beschreibt, welche Augensumme wir im Durchschnitt erwarten können:

$$E(X) \;=\; \sum_{(i,k)\in\Omega} X(i,k)\cdot P(\{(i,k)\}) \;=\; \sum_{(i,k)\in\Omega} (i+k)\cdot\frac{1}{36} \;=\; \frac{1}{36}\cdot\sum_{i=1}^{6}\sum_{k=1}^{6}(i+k) \;=\; 7$$

Wir können also im Schnitt erwarten, eine Augensumme von 7 zu würfeln. Nehmen wir an, wir haben 2 weitere Würfel, mit denen wir zwei weitere Zahlen würfeln. Die Zufallsvariable Y soll deren Summe beschreiben. Dann beschreibt die Zufallsvariable $Z := X+Y$ die Gesamtsumme. Wie sieht nun der Erwartungswert $E(Z)$ aus ? Nach Satz A.3.4 gilt folgender Zusammenhang:

$$E(Z) = E(X+Y) = E(X) + E(Y) = 7 + 7 = 14$$

Beispiel A.3.7 Betrachten wir das Zufallsexperiment im Struktur-Theorem als Beispiel. Der Shift (a,b) wird zufällig gewählt. Im Ergebnisraum sind somit die möglichen Wahlen von (a,b) zu finden: $\Omega = \{0,1,...,L-1\}^2$. Die Wahrscheinlichkeit, mit der der Shift (a,b) gewählt wird, ist $P(a,b) = \frac{1}{L^2}$, also ist P eine Gleichverteilung. Wir haben für jede horizontale und vertikale Gitterlinie l Zufallsvariablen $K_l : \{0,1,...,L-1\}^2 \longrightarrow \mathbb{R}$ und $K_l' : \{0,1,...,L-1\}^2 \longrightarrow \mathbb{R}$, die den Kostenzuwachs zu l bei Shift (a,b) für Schritt (i) und (ii) angeben. (Im Beweis haben K_l und K_l' nur den horizontalen Shift a als Argument, da o. B. d. A. nur vertikale Gitterlinien betrachtet werden und die dafür gemachten Aussagen für alle vertikalen Shifts b gelten.) Die zu erwartende Kostenbelastung des gesamten Gitters ergibt sich damit zu:

$$E_{a,b}(\text{Kostenbelastung}) \;=\; \sum_{l}\sum_{a=0}^{L-1}\sum_{b=0}^{L-1}\frac{1}{L^2}\cdot(K_l(a,b)+K_l'(a,b))$$

Literaturverzeichnis

[1] S. Arora. „Polynomial Time Approximation Schemes for Euclidean TSP and other Geometric Problems“. Journal Version, 1997.
Erhältlich bei `http://www.cs.princeton.edu/~arora`.

[2] S. Arora, C. Lund, R. Motwani, M. Sudan, M. Szegedy. „Proof verification and intractability of approximation problems“. Proceedings of the 33rd IEEE Symposium on Foundations of Computer Science, 1992, Seite 13-22.

[3] H. Bartel, E. Anthes. „Kombinatorik und Wahrscheinlichkeit“. Vogel-Verlag, 1978.

[4] O. Berkman, D. Breslauer, Z. Galil, B. Schieber, U. Vishkin. „Highly Parallelizable Problems“. Proceedings of the 21st ACM Symposium on Theory of Computing, Mai 1989, Seite 309-319.

[5] M. Bern, D. Eppstein, S.-H. Teng. „Parallel Construction of Quadtrees and Quality Triangulations“. 3rd Workshop on Algorithms and Data Structures (WADS), Aug. 1993, Seite 188-199.

[6] N. Christofides. „Worst-case analysis of a new heuristic for the traveling salesman problem“. Symposium on new directions and recent results in algorithms and complexity. Academic Press, 1976, Seite 441.

[7] J. Clark, D. A. Holton. „Graphentheorie“. Spektrum Verlag, 1994.

[8] R. Cole. „Parallel Merge Sort“. SIAM Journal on Computing 17, 1988, Seite 770-785.

[9] G. Das, S. Kapoor, M. Smid. „On the Complexity of Approximating Euclidean Traveling Salesman Tours and Minimum Spanning Trees“. Algorithmica Vol. 19 Nr. 4, Dez. 1997, Seite 447-460.

[10] R. Diestel. „Graphentheorie“. Springer-Verlag, 1996.

[11] W. Dörfler, J. Mühlbacher. „Graphentheorie für Informatiker“. Walter de Gruyter, 1973.

[12] M. Fisz. „Wahrscheinlichkeitsrechnung und mathematische Statistik“. VEB Deutscher Verlag der Wissenschaften, 1976.

[13] M.R. Garey, D.S. Johnson. „Computers and Intractability - A Guide to the Theory of NP-Completeness". Bell Telephone Laboratories, 1979.

[14] M.R. Garey, R.L. Graham, D.S. Johnson. „Some NP-complete Geometric Problems". 8th ACM Symposium on Theory of Computing, Mai 1976, Seite 10-22.

[15] R.L. Graham, D.E. Knuth, O. Patashnik. „Concrete Mathematics". Addison-Wesley, 1989.

[16] F. Heigl, J. Feuerpfeil. „Stochastik". Bayerischer Schulbuch-Verlag, 1978.

[17] J. JáJá. „An Introduction to Parallel Algorithms". Addison-Wesley, 1992.

[18] R.M. Karp. „Reducibility among combinatorial problems". In Complexity of Computer Computations, Plenum Press, 1972, Seite 85-103.

[19] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, D.B. Shmoys. „The Traveling Salesman Problem". John Wiley & Sons Ltd., 1985.

[20] C.H. Papadimitriou. „The Euclidean Traveling Salesman Problem is NP-complete". Theoretical Computer Science 4, 1977, Seite 237-244.

[21] K.R. Reischuk. „Einführung in die Komplexitätstheorie". B.G. Teubner, 1990.

[22] A. Rényi. „Wahrscheinlichkeitsrechnung". VEB Deutscher Verlag der Wissenschaften, 1971.

[23] S. Sahni, T. Gonzalez. „P-complete approximation problems". Journal ACM 23, 1976, Seite 555-565.

[24] H. Samet. „The Quadtree and Related Hierarchical Data Structures". ACM Computing Surveys Vol. 16, Juni 1984, Seite 187-260.

[25] I. Wegener. „Theoretische Informatik". B.G. Teubner, 1993.

[26] TSPBIB-Home Page.
`http://www.ing.unlp.edu.ar/cetad/mos/TSPBIB_home.html`