Diplomarbeit

Eine schnelle näherungsweise Lösung des Euklidischen Travelling Salesman Problems mittels raumfüllender Kurven

angefertigt am
Institut für Angewandte Mathematik

vorgelegt der
Mathematisch-Naturwissenschaftlichen Fakultät der
Rheinischen Friedrich-Wilhelms-Universität Bonn

Juli 2000

von
Markus Ellerbrake aus Remscheid

Vorwort

Eine kürzeste Rundreise durch eine gewisse Anzahl von Städten zu finden, bleibt eine der interessantesten Aufgaben in der Mathematik. Dieses Euklidische *Travelling Salesman Problem* erfreute sich in den vergangenen Jahrzehnten zahlreicher Anstrengungen, eine optimale Tour für gegebene Probleme zu konstruieren – häufig mit nur mäßigem Erfolg.

Es ist mir wichtig, mein Studium mit einer Diplomarbeit abzuschließen, die einen praktischen Wert hat. Ich hoffe, daß mir dies gelungen ist. Sie liefert einen kleinen Beitrag zu diesem wichtigen Themengebiet und damit auch eine Antwort auf die Frage, ob Mathematik nicht langweilig, altmodisch und ausgereizt sei. Ich habe mir Mühe gegeben, darzustellen, daß dies nicht der Fall ist, und bewußt viel Wert auf die Verständlichkeit der Arbeit gelegt.

Mein Dank gilt besonders Herrn Prof. Dr. M. Griebel, der mit seinen Ideen diese Arbeit entstehen ließ, sie gezielt mitverfolgte und ihr mit neuen Vorschlägen Impulse verlieh. Mit viel Einsatz konnte mir Herr Dr. G. Zumbusch, dem ich ebenso Dank schulde, besonders über die Klippen der Programmierung hinweghelfen. Außerdem stand er mir als kompetenter Ansprechpartner zu allen Themen zur Verfügung. Das gute Klima bei der Zusammenarbeit mit meinen genannten Betreuern sorgte für immer neue Ideen und Motivationsschübe. Ein großes Dankeschön auch Herrn Prof. Dr. M. Rumpf für sein Interesse an dieser Arbeit und an alle Mitarbeiter des „6. Stocks" der Angewandten Mathematik in Bonn, die nie vor meinen Fragen sicher waren.

Was wäre aus diesen Seiten ohne meine zahlreichen Korrekturleser und Ratgeber geworden? Auch wenn ich nur ungern jemanden aus diesem Kreis herausheben möchte, hat A. Böker für seine fachliche Begleitung einen besonderen Dank verdient.

Gerne danke ich auch meinen Eltern, die mein Studium so geduldig ertrugen und mich dabei immer unterstützten. Es scheint etwas genützt zu haben.

Abschließend bedanke ich mich beim Neumann Institute for Computing des Forschungszentrums Jülich für die Möglichkeit, die Cray-Parallelrechner nutzen zu können.

Alle Beteiligten seien sich meines Dankes und meiner Hochachtung sicher!

Bonn, im Juli 2000   Markus Ellerbrake

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Ein Handlungsreisender muß eine gegebene Anzahl von Städten besuchen und wieder zur Ausgangsstadt zurückkehren. Die Reihenfolge, in der er die Städte aufsucht, bestimmt die Zeit, die er unterwegs ist. Die Suche nach der besten Reihenfolge, einer kürzesten Tour, nennt man das Handlungsreisenden-Problem (*Travelling Salesman Problem, TSP*).

Es ist trotz der sehr einfachen Formulierbarkeit nur sehr schwer lösbar. Genauer: Das TSP gehört zu der Klasse der $\mathcal{NP}$-harten Probleme, d. h. es ist bis heute kein Weg der Lösungsfindung in polynomieller Zeit bekannt [25]. Das Ziel muß also lauten, gute Näherungen für eine sogenannte optimale Tour zu finden. Was unter „gut" zu verstehen ist, kann variieren: Dem einen Betrachter kann es auf eine möglichst exakte Lösung, d. h. eine Lösung, die wahrscheinlich nur geringfügig von einer optimalen Lösung abweicht, dem anderen auf eine beweisbar gute Lösung ankommen, d. h. eine Tour, die mit Sicherheit höchstens x Prozent vom Optimum entfernt ist.

Viele Heuristiken sind in den letzten Jahrzehnten nach obigem Gedankenschema entstanden und analysiert worden. Beispielhaft seien hier die wohl bekanntesten genannt, nämlich Nearest Neighbour und Lin-Kernighan. Alle diese Ansätze tragen allerdings ein Problem mit sich: eine vergleichsweise lange Laufzeit.

Nun kann man unter einer „guten" Lösung auch eine verstehen, die besonders schnell gefunden wird, im Extremfall in Echtzeit. Sobald die Anzahl der Städte ein gewisses Maß übersteigt, sind oben genannte Lösungsversuche nicht mehr zu empfehlen, falls die Zeit knapp ist: Für ein Problem mit rund 18 Millionen Städten wird die Laufzeit mancher Algorithmen in Tagen statt in Sekunden gemessen [49].

Zu Beginn der Achtziger Jahre wurde eine Idee entwickelt, die den Bedenken in puncto Laufzeit Rechnung trägt: J.J. Bartholdi und L.K. Platzman stellten 1982 eine Heuristik vor, die in ihrer Einfachheit, sowohl bezüglich der Idee als auch der Implementierung, bestechend ist: Man ordne die Städte in der Reihenfolge, in der eine raumfüllende Kurve sie besucht [5]. Dabei ist der Sortierschritt die zeitkritische Aktion. Bekanntlich läßt sich eine Sortierung parallelisieren, so daß selbst dieser Schritt äußerst schnell durchgeführt werden kann. Somit war ein schneller Algorithmus gefunden, der übrigens in der Praxis seine Bestimmung in der SDI-Forschung der Vereinigten Staaten fand: Er sollte dazu dienen, die Radarerfassung von feindlichen

Raketen aus dem Weltraum in (Fast-)Echtzeit zu ermöglichen [6].

In der Praxis konnte sich diese Idee allerdings nicht durchsetzen. Dafür sind zwei Gründe anzuführen: Erste Rechenbeispiele zeigten eine durchschnittliche Abweichung vom Optimum von fast 45 % [47]. Zusätzlich wurden keine Beispiele großer Datenmengen gefunden, für die die genannte Heuristik ihre Schnelligkeit ausspielen könnte. Für wenige tausend Städte oder Raketen leisten andere Algorithmen in nur wenigen Sekunden oder gar Zehntelsekunden eine viel genauere Arbeit.

1.2 Ziel der Arbeit

In dieser Arbeit wird untersucht, ob und wie die Heuristik von Bartholdi und Platzman verbessert werden kann, so daß ihre Leistung die vergleichbarer Heuristiken erreicht oder überbietet. Dazu wird zwei Ansätzen gefolgt:

- Bartholdi und Platzman benutzen in ihren Arbeiten grundsätzlich nur *eine* raumfüllende Kurve. Folglich wird untersucht, wie sich der Einsatz anderer Kurven auf die Leistung auswirkt. Dazu müssen Eigenschaften von Kurven herausgearbeitet und bewertet werden. Die Frage lautet, ob eine bessere Kurve als die schon benutzte Sierpiński-Kurve gefunden oder konstruiert werden kann.

  Numerische und theoretische Ergebnisse legen nahe, daß dies nicht möglich ist. Zwar wird die Sierpiński-Kurve in den realen Testfällen häufig durch andere Kurven unterboten; im Schnitt findet jedoch keine signifikante Verbesserung der Genauigkeit statt.

- Übliche Verbesserungsschritte für TSP-Heuristiken sind von quadratischer Laufzeit. Der Vorteil der betrachteten Heuristik ist ihre subquadratische Laufzeit. Kann ein Verbesserungsschritt eingesetzt werden, der die Komplexität der Heuristik nicht verschlechtert und dennoch erfolgreich ist?

  Dank einer besonderen Eigenschaft von raumfüllenden Kurven bietet es sich an, die Nähebeziehungen von Städten in einem Verbesserungsschritt linearer Laufzeit auszunutzen.

Bei der Bearbeitung der oben aufgeworfenen Fragen werden zusätzlich wichtige Erkenntnisse gewonnen: So wird die Verwandtschaft völlig unterschiedlich erscheinender Kurven zur Sierpiński-Kurve gezeigt, wodurch theoretische Ergebnisse verbunden werden können, die bisher in keinem Zusammenhang standen. Auch sollten einige neu erstellte raumfüllende Kurven für andere Einsatzgebiete außerhalb der genannten Heuristik interessant sein.

Schließlich liefert diese Arbeit im Anhang eine Reihe von lauffähigen Programm-Codes in der Programmiersprache `C`, die es dem interessierten Anwender ermöglichen, die vorgestellte Heuristik zu nutzen.

1.3 Aufbau der Arbeit

Das folgende Kapitel dient zur Einführung in das Travelling Salesman Problem und faßt den Stand der Forschung zusammen. Das Travelling Salesman Problem wird definiert, bekannte Ergebnisse und Probleme werden vorgestellt. Da das TSP $\mathcal{NP}$-hart ist, verfolgt die Forschung seit Jahrzehnten den Weg, Algorithmen zu entwickeln, die in polynomieller Zeit sehr gute Näherungslösungen finden. Einige dieser Algorithmen werden beispielhaft vorgestellt, um die danach näher betrachtete Heuristik von Bartholdi und Platzman einordnen zu können.

Für das Verständnis der kommenden Kapitel ist es notwendig, raumfüllende Kurven und ihre Eigenschaften zu betrachten. Dies geschieht in Kapitel drei. Ein Konstruktionsprinzip für raumfüllende Kurven, das sehr einfach in einem Algorithmus umzusetzen ist, wird vorgestellt. Alle klassischen Kurven werden sowohl definiert als auch veranschaulicht. Zusätzlich zu bekannten Kurven wie den Hilbert- oder Peano-Kurven werden neue Konstrukte dargestellt, die durch Triangulierungen oder quadratische Unterteilungen entstehen.

Zusammengeführt werden die Kapitel zwei und drei dann in Kapitel vier, in dem die bereits genannte Heuristik von Bartholdi und Platzman vorgestellt wird. Die Ergebnisse verschiedener dazu verfaßter Arbeiten der beiden Autoren werden zusammengetragen und einer kritischen Betrachtung unterworfen. Des weiteren wird festgelegt, für welche Instanzen die Heuristik getestet wird. Es handelt sich hier einerseits um randomisierte Fälle mit bis zu 1.400.000 gleichverteilten Städten und andererseits um Probleme, die G. Reinelt in seiner TSP-Bibliothek [48] zusammengestellt hat. Diese Probleme entstammen der Praxis und umfassen bis zu 85.900 Städte.

Im fünften Kapitel wird die Performance der genannten Heuristik herausgearbeitet. Es wird untersucht, wie gut sich der Algorithmus bei Einsatz verschiedener Kurven für reale Testfälle und randomisierte Instanzen dem Optimum annähert. Für die realen Testfälle liegt in fast jedem Fall das Optimum als Vergleichswert vor, in den randomisierten Fällen dient eine Konstante als Abschätzungsgröße. Wie sich verschiedene Kurveneigenschaften auf die Performance auswirken, wird ausführlich dargestellt. Es ergibt sich, daß die Moore- (geschlossene Hilbert-) und die Sierpiński-Kurve die beste Performance zeigen und daher für die Heuristik verwendet werden sollten. Die Erkenntnisse werden durch den Aufbau eines Bewertungsschemas abgerundet. Ferner behandelt dieses Kapitel auch Kurven, die nicht nach dem Konstruktionsprinzip aus Kapitel drei entstehen, so z. B. die Moscato-Norman-Kurven sowie den H-Index und geht auf die Verwandtschaft dieser Kurven mit der Sierpiński-Kurve ein.

Wie der Algorithmus, dessen Performance untersucht wird, genau implementiert werden kann und welche Implementierungsmöglichkeiten für die einzelnen Kurven bestehen, ist im sechsten Kapitel zusammengefaßt. Ein vollständiges und lauffähiges C-Programm mit der Sierpiński-Kurve ist dem Anhang zu entnehmen. Weiterhin wird in Kapitel sechs die Laufzeit des Algorithmus, insbesondere bei Parallelisierung, analysiert.

Im Kapitel sieben wird zunächst dargestellt, wie die Heuristik durch einen 2-opt-Tauschschritt so verbessert werden kann, daß sie eine Performance erreicht, die mit anderen Heuristiken vergleichbar ist, ohne deren Aufwand zu benötigen. Sie ist somit für die klassischen Anwendungen des TSP geeignet. Weiter werden Einsatzgebiete für die reine Kurven-Heuristik vorgestellt: Sie kann ihre Vorteile dann wirksam ausspielen, wenn entweder eine Echtzeitlösung gefragt ist oder sehr große Datenmengen zu behandeln sind, die von anderen Heuristiken aufgrund ihrer Komplexität nicht bearbeitbar sind.

Das letzte Kapitel faßt die Ergebnisse dieser Arbeit zusammen: Die Heuristik von Bartholdi und Platzman liefert bei Einsatz der Sierpiński-Kurve für randomisierte Instanzen Lösungen, die etwa 35% vom Optimum abweichen. Sie ist damit im Vergleich zu anderen Heuristiken sehr schnell, dabei aber leistungsschwach. Mit einfachen, auf diese Heuristik abgestimmten Verbesserungsschritten kann eine konkurrenzfähige Performance von 20 % erreicht werden, ohne eine quadratische Laufzeit zu benötigen. Die Arbeit endet mit Hinweisen auf noch offene Fragen und Anregungen für die weitere Forschung.

Kapitel 2

Das Travelling Salesman Problem

Dieses Kapitel dient zur Einführung in die Problemstellung und zur Darstellung des Forschungsstandes. Dazu wird zunächst das Travelling Salesman Problem (TSP) in seiner allgemeinen Form definiert und für die spätere Verwendung in dieser Arbeit auf das Euklidische TSP reduziert. Im Anschluß daran werden einige Heuristiken, die zur approximativen Lösung des TSP dienen, beispielhaft vorgestellt.

2.1 Einführung in das TSP

Nachdem das TSP in den vorangegangenen Ausführungen mehrfach erwähnt wurde, wird im folgenden die genaue Definition angegeben. Dabei wird auf Grundzüge der Graphentheorie zurückgegriffen. Für eine ausführliche Einführung in die Graphentheorie wird auf das Buch „Graphentheorie" von Diestel verwiesen [13]. Näheres zum TSP kann z. B. bei Lawler et al. gefunden werden [30].

2.1.1 Zentrale Definitionen

Ein ungerichteter Graph (im folgenden: ***Graph***) G ist ein Paar (V, E), wobei V eine endliche Menge von Knoten und E eine endliche Menge von Kanten sind. Man stelle sich z. B. eine Menge von Städten (=Knoten) vor, die durch Straßen (=Kanten) verbunden sind.

Eine Kante $e \in E$ wird dargestellt durch ihre beiden Endknoten v_1 und v_2, $e = v_1v_2$. In einem ungerichteten Graphen gibt es keinen Unterschied zwischen v_1v_2 und v_2v_1. Eine Kantenmenge $P = \{v_1v_2, v_2v_3, \ldots, v_{k-1}v_k\}$ in G mit $v_i \neq v_j$ für $i \neq j$ $(i, j = 1, \ldots, k)$ heißt $v_1 - v_k$ – ***Weg*** in G. Ist $v_k = v_1$, so spricht man von einem ***Kreis***. Einen Kreis mit n Kanten in einem Graphen mit n Knoten nennt man ***Hamiltonkreis***. Wegen der Bedingung $v_i \neq v_j$ für $i \neq j$ bedeutet dies, daß jeder Knoten genau einmal aufgesucht wird, bevor sich der Kreis schließt.

Auf der Kantenmenge E kann eine ***Gewichtsfunktion*** $w : E \rightarrow \mathbb{R}$ definiert werden, die jeder Kante v_iv_j ein ***Gewicht*** w_{ij} zuordnet. Das Gewicht einer Kantenmenge $P = \{v_1v_2, \ldots, v_{k-1}v_k\}$, $w(P)$, ist definiert als die Summe der Gewichte der einzelnen Kanten: $w(P) = \sum_{i=1}^{k-1} w_{i\,i+1}$.

Ist P ein Weg, so nennt man das Gewicht von P meist die ***Länge*** von P. Vergleicht man Wege, so spricht man von kürzeren bzw. längeren Wegen. Im obigen Beispiel wäre die Gewichtsfunktion w z. B. die Strecke in Kilometern oder die Fahrtdauer in Minuten (was je nach Staulage zu völlig unterschiedlichen Ergebnissen führen kann).

Damit sind alle Vorbereitungen für die zentrale Definition abgeschlossen:

> Unter dem ***Travelling Salesman Problem (TSP)*** verstehen wir die Suche nach einem kürzesten Hamiltonkreis in einem Graphen.

Eine spezielle Formulierung im $\mathbb{R}^2$, die diesem Problem den Namen gegeben hat, lautet: Ein Handlungsreisender, der eine bestimmte Anzahl von Städten besuchen muß, sucht die Route, die für ihn die kürzeste Strecke bedeutet.

An dieser Stelle wird bewußt auf weitere Einschränkungen innerhalb der Definition verzichtet. Solche Einschränkungen, namentlich die Vollständigkeit des Graphen, die Reinelt fordert [47] oder die Dreiecksungleichung für w, die bei Gleichsetzung des TSP mit dem symmetrischen TSP notwendig ist [32], sind für eine allgemeine Definition unnötig restriktiv. Auf beide Eigenschaften wird in den folgenden Abschnitten aber noch näher eingegangen.

2.1.2 Lösbarkeit

Obige Definition des TSP sagt noch nichts über die allgemeine Lösbarkeit der Problemstellung aus. Es ist möglich, daß es gar keinen Hamiltonkreis in G gibt. Dies ist beispielsweise dann evident, wenn es einen isolierten Knoten in G gibt, d. h. einen Knoten, der mit keinem anderen Knoten verbunden ist und somit nicht durch einen Kreis berührt werden kann.

Um diesem Problem aus dem Weg zu gehen, bedient man sich folgender Definition: Ein Graph G heißt ***vollständig***, wenn es zu je zwei verschiedenen Knoten v_1 und v_2 eine Kante v_1v_2 gibt. Dies erscheint restriktiv, ist es bei näherer Betrachtung aber nicht immer: Im Beispiel des deutschen Straßennetzes gibt es keine durchgehende Straße Flensburg–Oberammergau. Also ist das Straßennetz nicht vollständig. Dennoch ist es selbstverständlich möglich, von der einen zur anderen Stadt zu fahren. Man kann diesen Graphen also durch eine zusätzliche, mit geeignetem w gewichtete Kante ergänzen und durch sehr viele solcher Ergänzungen vollständig machen.

Diesen Trick benutzt man in vielen Fällen der Praxis. Ermöglicht es das Problem nicht, eine solche künstliche Kante einzufügen, nimmt man sie dennoch hinzu und belegt sie mit einem sehr hohen Gewicht nahe ∞, so daß sie für einen kürzesten Weg nicht in Frage kommt. Sollte diese Kante gewählt werden müssen, um einen kürzesten Weg zu erzeugen, entspricht dies der Unlösbarkeit des betrachteten TSP.

Ist der betrachtete Graph G vollständig, so ist das TSP lösbar: Jede beliebige Permutation der Städte ist dann eine Lösung. Dabei kann das Problem optimal gelöst werden, indem man alle Möglichkeiten berechnet und den kürzesten Kreis auswählt. Dieser Kreis muß keineswegs eindeutig sein; die Eindeutigkeit ist beim TSP selten von Belang.

Warum wird das TSP nun als schwieriges mathematisches Problem bezeichnet, wenn es doch so einfach lösbar ist wie im vorangegangenen Absatz beschrieben?

Das Problem liegt, sehr oberflächlich gesagt, in der Schnelligkeit der Lösungsfindung: Selbst für sehr kleine Probleme, sagen wir 25 Städte, müßten nach obiger Idee $\frac{24!}{2} \approx 3 \cdot 10^{23}$ Lösungen ausgerechnet und verglichen werden. Bediente man sich eines Computers, der eine Million solcher Lösungen pro Sekunde berechnen kann, bräuchte jener dennoch 10 Milliarden Jahre für die Angabe der besten Lösung. Auf diese Eigenschaft des TSP wird im nächsten Abschnitt kurz näher eingegangen.

2.1.3 Zur Komplexität des TSP

Erst vor 30 Jahren hat sich in der Mathematik eine Einordnung von Problemen in Klassen verschiedener Schwierigkeiten durchgesetzt. Hervorzuheben sind dazu Arbeiten von Cook (vgl. z. B. [11]). Bei deutlicher Vereinfachung geht die erste Stufe der Klassifizierung auf die Frage zurück, ob die Anzahl der elementaren Operationen eines Algorithmus für steigende Problemgröße n durch ein Polynom $p(n)$ nach oben abgeschätzt werden kann. Man sagt dann, der Algorithmus habe ***polynomielle Laufzeit*** oder auch eine Laufzeit in der Größenordnung von $p(n)$, kurz $\mathcal{O}(p(n))$.

Die Klasse der Entscheidungsprobleme, die von Algorithmen in polynomieller Laufzeit gelöst werden können, heißt $\mathcal{P}$. Ein Entscheidungsproblem ist dabei ein Problem, das mit „ja“ oder „nein“ beantwortet wird. Optimierungsprobleme können leicht in Entscheidungsprobleme umformuliert werden, wie unten zu sehen sein wird.

Es gibt aber auch Probleme, für die bis heute noch kein polynomieller Lösungsalgorithmus gefunden wurde. Aus theoretischer Sicht ist für solche Probleme folgendes Vorgehen denkbar (vgl. [47]): Ein Algorithmus „rät“ nicht-deterministisch eine Zeichenfolge (einen Code), die durch ein geeignetes Codierungsschema in polynomieller Zeit erhalten wird. Im zweiten Schritt wird (deterministisch) überprüft, ob diese Zeichenfolge tatsächlich zeigt, daß das betrachtete Entscheidungsproblem lösbar ist; dies ebenfalls in polynomieller Zeit. Ein formales Modell für solche Berechnungen ist eine sogenannte nicht-deterministische *Turing-Maschine*.

Die Klasse der Entscheidungsprobleme, die in polynomieller Zeit durch einen solchen nicht-deterministischen Algorithmus gelöst werden können, trägt den Namen $\mathcal{NP}$.

Ein Problem Π wird ***$\mathcal{NP}$-vollständig*** genannt, wenn $\Pi \in \mathcal{NP}$ ist und jedes Problem aus $\mathcal{NP}$ in polynomieller Zeit auf Π reduziert werden kann. Die Wortwahl „vollständig“ soll andeuten, daß die komplette Klasse $\mathcal{NP}$ in polynomieller Zeit gelöst werden kann, sobald ein Algorithmus gefunden wird, der ein $\mathcal{NP}$-vollständiges-Problem in polynomieller Zeit löst. Die Frage „$\mathcal{P} = \mathcal{NP}$?“, für die eine negative Antwort erwartet wird, wäre mit einem solchen Algorithmus positiv beantwortet.

Das ***Travelling Salesman Decision Problem (TSDP)***, nämlich die Frage, ob zu einem vollständigen Graphen G, einer Gewichtsfunktion w und einem gegebenen $b \in \mathbb{R}$ ein Hamiltonkreis in G existiert, der höchstens die Länge b hat, ist $\mathcal{NP}$-vollständig [25]. Das TSP selbst ist ein Optimierungs-, kein Entscheidungsproblem. Da sein zugehöriges Entscheidungsproblem, das TSDP, $\mathcal{NP}$-vollständig ist, wird das TSP der Klasse der ***$\mathcal{NP}$-harten*** Probleme zugeordnet (für eine ausführlichere Darstellung siehe z. B. Reinelt [47]).

Die Hoffnung, das einfachere Euklidische TSP könnte angenehmere Eigenschaften haben, ist trügerisch. Das ETSP ist ebenfalls $\mathcal{NP}$-hart [47]. Immerhin konnte Arora zeigen, daß für jede ETSP-Instanz in einer Laufzeit von $\mathcal{O}(n \cdot (\log n)^{\mathcal{O}(1/\varepsilon)})$ mit Hilfe eines randomisierten Algorithmus eine Lösung gefunden werden kann, die höchstens um den Faktor $(1+\varepsilon)$ vom Optimum abweicht [3]. Da der Exponent in der Laufzeitangabe aber selbst nur eine Größenordnung angibt, wird dies keine Auswirkungen auf die Praxis einer Lösungsfindung haben.

2.1.4 Das Euklidische TSP

In dieser Arbeit wird schwerpunktmäßig das ***Euklidische Travelling Salesman Problem (ETSP)*** behandelt. Dies bedeutet, daß der Graph, in dem wir den kürzesten Hamiltonkreis suchen, in einer Ebene liegt und die Kantengewichte die euklidischen Abstände der zugehörigen Knoten sind.

In diesem Fall ist die sogenannte Dreiecksungleichung für die Kantengewichte automatisch erfüllt. Die Dreiecksungleichung lautet $w_{ik} \leq w_{ij} + w_{jk}$ für alle $v_i, v_j, v_k \in V$, was per definitionem für euklidische Abstände von Punkten in der Ebene richtig ist. Ist die Dreiecksungleichung erfüllt, so bezeichnet man das TSP als ***symmetrisch***.

Das bisher begleitend angeführte Beispiel eines Straßennetzes als Graph ist nun nicht mehr ganz korrekt, da eine Straße selten den euklidischen Abstand zweier Städte angibt. Statt dessen müßte man sich die Luftlinie als Entfernungsmesser vorstellen. Würde man übrigens die benötigte Zeit für eine Strecke als Gewichtsfunktion für die Kanten benutzen, wäre die Dreiecksungleichung im allgemeinen nicht erfüllt, da ein kleiner Umweg durchaus einen zeitlichen Gewinn bedeuten kann.

Für den Rest dieser Arbeit sollen nun hier einige Vereinbarungen festgehalten werden:

- Auch wenn verkürzt vom TSP gesprochen wird, ist ab Kapitel vier grundsätzlich das ETSP gemeint.
- Alle betrachteten Graphen werden als vollständig angenommen. So ist sichergestellt, daß es in jedem zu untersuchenden Graphen Hamiltonkreise gibt. Ist ein Graph nicht vollständig, kann er zur Vollständigkeit ergänzt werden (siehe 2.1.2).
- Bei der betrachteten Ebene handelt es sich o. B. d. A. um das Einheitsquadrat $[0,1]^2$. Alle anderen Fälle können durch eine geeignete Transformation auf diesen Fall zurückgeführt werden (vgl. dazu den Abschnitt Datenverschiebung in Kapitel sechs).

Nachdem das Wort „Heuristik“ in den vergangenen Abschnitten bereits mehrfach verwendet wurde, wird hier außerdem eine Sprachregelung festgehalten, die die Begriffe „Heuristik“ und „Algorithmus“ trennt:

> Unter einer ***Heuristik*** ist hier eine Anleitung zur (näherungsweisen) Lösung einer Aufgabenstellung zu verstehen.
> Mit ***Algorithmus*** wird ein lauffähiges Programm, z. B. eine spezielle Implementierung einer Heuristik, bezeichnet.

Die Heuristik stellt also eine Idee dar, die im Algorithmus umgesetzt wird. Der Terminus „Heuristik“ wird im betriebswirtschaftlichen Sinn meist etwas anders verwendet; die hier aufgezeigte Sprachregelung entspricht aber dem Standard der praktischen Mathematik (vgl. [47]).

2.2 Klassische Lösungsheuristiken

2.2.1 Zur Historie des TSP und der Lösungsversuche

Die frühesten Formulierungen des TSP datieren aus dem 19. Jahrhundert: Kirkman schrieb bereits 1855 den wohl ersten Artikel zu diesem Thema (zitiert in [19]). Dieser frühe Zeitpunkt unterstreicht die Vielfältigkeit der Problemstellung – heute liefern ganz andere Bereiche Motivationen für Untersuchungen als vor 150 Jahren. Tatsächlich ist das TSP nicht nur innerhalb der praktischen Mathematik interessant. Reinelt nennt die Biologie, die Physik oder den Bereich der künstlichen Intelligenz als Beispiele weiterer Fachgebiete, in denen das Handlungsreisendenproblem auftaucht [47]. Hinzuzufügen ist insbesondere die Optimierung von Produktionsabläufen, z. B. bei der Herstellung von Leiterplatten. Das TSP ist also auch in der industriellen Praxis von großer Bedeutung. Die heute übliche Bezeichnung und die Formulierung als Optimierungsproblem unter graphentheoretischen Gesichtspunkten dürfte auf die 60er Jahre des 20. Jahrhunderts zurückgehen.

Exakte Lösungsversuche waren ohne Rechnerunterstützung auf sehr kleine Städteanzahlen beschränkt. So ist es erst 1954 gelungen, ein 48-Städte-Problem zu lösen [12]. Trotz der rasanten Entwicklung der Computerleistungen ist das größte derzeit optimal gelöste reale[1] Problem das in Kapitel vier genannte 13.509-Städte-Problem von Applegate [48]. Diese Entwicklung ist zwar imponierend, jedoch zeigt die Aufgabe, das Weltraumteleskop „Hubble“ mit der Radarerfassung von mehreren Millionen Sternen zu programmieren, daß die Praxis immer mehrere Schritte voraus ist und auch bleiben wird. Es ist nicht auszuschließen, daß bereits in wenigen Jahren eine Lösung eines 1-Milliarden-Städte-Problems gesucht wird.

Es bleibt festzustellen, daß Handlungsreisendenprobleme selten optimal gelöst werden. Daher wird a priori der Aufwand zur optimalen Lösung auch selten betrieben. Statt dessen nutzt man Heuristiken, deren Ziel es ist, möglichst gut eine optimale Lösung anzunähern. Unter „gut“ wird dabei häufig verstanden, eine Lösung zu erhalten, deren Länge höchstens um einen gewissen Prozentsatz von der optimalen Länge abweicht. In manchen Anwendungen kann „gut“ aber auch „wahrscheinlich gut“ heißen, wenn die Abweichung nicht zufriedenstellend genau berechnet werden kann, da sie von Wahrscheinlichkeiten abhängt. Schließlich werden in Kapitel sieben dieser Arbeit Fälle betrachtet, in denen „gut“ einfach „schnell“ heißt. Der Anwender sollte wissen, welche dieser genannten Gütebezeichnungen er verfolgt und eine entsprechende Heuristik wählen.

Innerhalb der heuristischen Untersuchungen sind meist zwei Schritte zu unterscheiden: An erster Stelle steht die Suche nach einer möglichst guten Lösung. Da das Optimum aber im allgemeinen nicht bekannt ist, muß für eine qualitative Wertung der heuristisch erhaltenen Lösung zusätzlich, unabhängig vom ersten Schritt, ein Vergleichswert errechnet werden, der unter der optimalen Länge einer Tour liegt, eine sogenannte untere Schranke (***Lower Bound***).

Vergleicht man nun das Ergebnis der gefundenen Lösung mit der unteren Schranke, erhält man die maximal mögliche Abweichung vom Optimum, da das Optimum selbst näher an der gefundenen Lösung liegt als die untere Schranke. Es bedarf kaum

[1] „Real“ bezeichnet hier ein Problem aus der Praxis. Man kann leicht Probleme mit einer beliebigen Städteanzahl *erfinden*, die optimal gelöst werden können (vgl. 5.5.1).

der Anmerkung, daß beide Schritte für eine Einschätzung der Güte einer Heuristik gleich wichtig sind: Sollte die untere Schranke zu weit unterhalb des Optimums liegen, kann der Nutzer nicht einschätzen, wie gut die durch die Heuristik erhaltene Lösung tatsächlich ist.

Das Errechnen einer guten unteren Schranke wird in Abschnitt 2.2.7 vorgestellt. Man bedient sich dabei linearer Optimierungsmodelle. Zuvor werden in den Abschnitten 2.2.2 und 2.2.3 Heuristiken beschrieben, mit deren Hilfe eine Lösung gefunden wird. Im Gegensatz zu diesen Heuristiken, die genau eine Lösung errechnen, stehen sogenannte Verbesserungs-Heuristiken oder auch *Improvement Heuristics*. Diese gehen von einer „Start"-Lösung aus und verändern diese in mehreren Schritten. Durch wiederholtes Anwenden solcher Veränderungsschritte entstehen immer wieder neue Lösungen, von denen der Nutzer nach Abbruch des Programms die letzte oder beste auswählt. Wie solche Veränderungen gelenkt werden, kann a priori vorgegeben werden oder vom Algorithmus im Rahmen von Wahrscheinlichkeiten entschieden werden (z. B. bei *Simulated Annealing*). Verschiedene Improvement Heuristics werden in den Abschnitten 2.2.4 bis 2.2.6 behandelt.

Genauere Informationen über die genannten und noch weitere Heuristiken findet man bei Reinelt in [47]. Soweit nicht ausdrücklich anders vermerkt, entstammen die Performance- und die Laufzeit-Aussagen über die Heuristiken dieser Quelle. Aufgrund der weiten Verbreitung tragen die folgenden Heuristiken auch in der vorliegenden Aufstellung englische Namen.

2.2.2 Nearest Neighbour

Die erste Heuristik, die hier vorgestellt wird, ist eine der einfachsten: Beginnend mit seiner Heimatstadt, wählt der Handlungsreisende in jeder Stadt die nächstgelegene, noch nicht besuchte Stadt als neues Ziel aus. Er reist also jeweils zum „nächsten Nachbarn" (***Nearest Neighbour***). Die „Nähe" der Städte wird durch die Gewichtsfunktion w angegeben; der Begriff darf im allgemeinen nicht mit einer Metrik verwechselt werden.

Dabei wird der Reisende zu Beginn seiner Tour recht kurze und geschickte Fahrten unternehmen. Zum Ende jedoch stellt sich meist heraus, daß einige Städte „vergessen" worden sind, und sein Weg führt ihn zurück in Ballungszentren, die vorher nicht erschöpfend aufgesucht wurden. Außerdem verführt diese Heuristik zu sich kreuzenden Wegen, die nicht optimal sein können, wie man leicht zeigen kann. An diese Feststellung knüpft zum Beispiel der 2-opt-Tausch an (siehe 2.2.4).

Es ist festzustellen, daß diese Heuristik sehr einfach ist und eine in großen Teilen gute Tour erstellt, die erst dadurch verschlechtert wird, daß „vergessene" Städte sehr spät zu sehr hohen Kosten der Tour hinzugefügt werden. Eine solche Heuristik ist dafür prädestiniert, durch Korrekturschritte einer Improvement Heuristic verbessert zu werden. Ohne eine solche Korrektur erreicht die Nearest-Neighbour-Heuristik eine durchschnittliche[2] Abweichung vom Optimum (*Performance*) von etwa 26 % in realen Fällen und 24 % in randomisierten Fällen [26], wenn die Held-Karp-Grenze (siehe 2.2.7) als Vergleich genommen wird. Dabei muß angemerkt werden, daß Reinelt in [47]

[2] Für eine Beschreibung der Testinstanzen sei auf Unterkapitel 4.2 verwiesen. Dort wird auch der Begriff der Performance genau definiert.

nicht die Standard-Form der Heuristik untersucht, sondern nur beschleunigte Versionen, deren Performance unter der Beschleunigung gelitten haben dürfte. Außerdem werden allgemeine Aussagen durch die Abhängigkeit der Performance der Heuristik von der Wahl des Ausgangsknotens erschwert.

Eine Qualitätssicherheit für die Performance der betrachteten Heuristik gibt es nicht: Selbst im euklidischen Fall findet man nach einem Satz von Rosenkrantz, Stearns und Lewis für ein beliebig großes n eine TSP-Instanz mit n Knoten, für die die Nearest-Neighbour-Heuristik eine Lösung erstellt, die um einen Faktor $\mathcal{O}(\log n)$ vom Optimum abweicht [50].

Die Nearest-Neighbour-Heuristik in ihrer Standardform hat eine Laufzeit von $\mathcal{O}(n^2)$. Diese Laufzeit kann durch geschickte Vorprogrammierung von Nachbarn der einzelnen Knoten praktisch auf eine subquadratische gesenkt werden.

2.2.3 Farthest Insertion

Das System vieler Heuristiken ist in wesentlichen Punkten gleich: Ausgehend von einer trivialen Tour mit einem oder zwei Knoten, werden die weiteren Knoten nach einem festgelegten Schema hinzugefügt. Der Unterschied von Einfügeheuristiken zum gerade betrachteten Nearest-Neighbour-Verfahren besteht darin, daß die Knoten nicht an die bestehende Tour *an*gefügt, sondern an irgendeine Stelle *ein*gefügt werden. Reinelt unterscheidet insgesamt neun standardmäßige Einfügeheuristiken [47]. Die erfolgreichste dieser neun ist die ***Farthest Insertion Heuristic***.

Dabei wird wie folgt vorgegangen: In jedem Schritt wird ein bisher noch nicht betrachteter Knoten in die bestehende Tour (anfangs eine Schleife in dem beliebigen Startknoten) eingefügt, und zwar an die Stelle, an der er den geringsten Anstieg der Tourlänge erzeugt. Ausgewählt wird bei der Farthest-Insertion-Heuristik genau der Knoten, dessen minimaler Abstand zu einem bereits betrachteten Knoten maximal unter allen noch nicht betrachteten Knoten ist. So entsteht anfangs ein recht grobes Gebilde mit vielen langen Strecken, in die nach und nach weitere Knoten eingefügt werden.

In der Performance unterscheiden sich die verschiedenen Einfügeheuristiken gewaltig. Die hier beschriebene Farthest Insertion weicht im Schnitt rund 14 % in praktischen und 13,5 % in randomisierten Fällen [26] vom Optimum ab. Zum Vergleich: Fügt man den Knoten ein, dessen maximaler Abstand zu einem betrachteten Knoten minimal unter den noch nicht betrachteten ist, weicht man fast doppelt so stark ab (durchschnittlich über 26 %). Wie die Struktur des Vorgehens bereits zeigt, hängt die Performance der Heuristik auch hier von der Wahl des Startknotens ab.

Interessanterweise kann man für manche eher schlechte Einfügeheuristik eine recht gute *Worst-Case*-Abschätzung vornehmen – z. B. ist bei der sogenannten „Nearest Insertion" die gefundene Lösung höchstens doppelt so lang wie eine optimale [50] –, während für die Farthest Insertion Beispiele konstruiert werden konnten, in denen eine gefundene Lösung 13/2-mal so lang ist wie eine optimale [24].

Die Laufzeit fast aller Einfügeheuristiken beträgt $\mathcal{O}(n^2)$, insbesondere trifft dies auf die Farthest Insertion zu.

2.2.4 2-opt-Tausch

Alle Improvement Heuristics haben ein Ziel: eine bestehende Lösung durch Korrekturschritte zu verbessern. Dabei ist das Vorgehen in den meisten Fällen ähnlich: Zunächst wird eine erste zulässige Lösung erstellt, häufig ohne jeden Bezug auf das Wesen der Verbesserungsheuristik. Dann wird diese Lösung einem Schema unterworfen, das beispielsweise jeden Knoten oder jede Kante auf Verbesserungsmöglichkeiten untersucht. Ist eine Verkürzung der Tour möglich, wird die Lösung verändert. Die Heuristik endet z. B., wenn alle Knoten oder Kanten untersucht wurden, oder (bei iterativem Vorgehen), wenn keine Verkürzung mehr möglich ist. Dies bedeutet nicht, daß eine optimale Tour gefunden ist – lediglich die der Heuristik zugrundeliegende Idee beinhaltet keine Verbesserung der Lösung mehr. Man spricht davon, daß ein ***lokales Optimum*** vorliegt.

Eine der einfachsten Verbesserungsheuristiken wurde im vorletzten Abschnitt bereits angedeutet: Gerade beim Euklidischen TSP ergibt sich eine Verkürzung der Tour sofort dadurch, daß alle sich kreuzenden Kanten entfernt und durch sich nicht kreuzende Kanten ersetzt werden. Allgemeiner formuliert ergibt sich dadurch das Schema des ***2-opt-Tauschs***:

Man entferne zwei Kanten und verbinde die Reststücke der Tour auf eine andere Art als zuvor. Dies ist immer eindeutig möglich. Kann die Gesamtlänge der Tour dadurch verkürzt werden, findet die Kantenersetzung statt, anderenfalls bleibt die Tour unverändert.

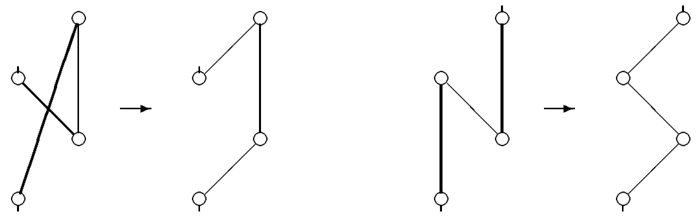

Abbildung 2.1: Verschiedene 2-opt-Tausche

Diese Idee hat nicht nur bei sich kreuzenden Kanten Erfolg; es sind viele andere Fälle denkbar, in denen ein solcher Tausch einen Vorteil bringt (siehe Abbildung 2.1). Die Heuristik kann so formuliert werden: Für jede Kante der vorliegenden Lösung wird überprüft, ob es einen erfolgreichen 2-opt-Tausch mit einer anderen Kante der Lösung gibt. Ist dies der Fall, wird der Tausch durchgeführt, der die Gesamtlänge der Tour am deutlichsten reduziert. Anderenfalls findet keine Änderung statt, und die nächste Kante wird untersucht.

Die Leistungsfähigkeit eines solchen Vorgehens ist nur schwierig in wenigen Zeilen zusammenzufassen, da sie erheblich von der Ausgangslösung abhängt. Hier sei nur erwähnt, daß eine Nearest-Neighbour-Lösung im Schnitt auf unter 10 % Abweichung verbessert werden kann. Für eine ausführliche Analyse wird auf die Ausführungen von Reinelt verwiesen [47].

Aufgrund der offensichtlichen Laufzeit von $\mathcal{O}(n^2)$ wird die Heuristik zumeist nicht in obiger Form durchgeführt. Schon für Probleme mit wenigen tausend Städten ist der Zeitaufwand für viele Anwender nicht mehr akzeptabel. Eine Lösung dieses Problems besteht darin, jede einzelne Kante nicht mit *allen* anderen, sondern nur mit ausgewählten Kanten zu vergleichen. Man kann die Laufzeit so erheblich beschleunigen, ohne allzuviel Leistung einzubüßen. Bei diesen Wertungen muß daran erinnert werden, daß der Anwender vorgeben muß, ob ihm an der Schnelligkeit oder der Genauigkeit der Lösung gelegen ist. In manchen Fällen können genau die Bruchteile an Performance verloren gehen, die gewünscht sind. Allgemein gehaltene Aussagen zu Leistung und Laufzeit sind daher mit Vorsicht zu genießen.

2.2.5 Lin-Kernighan

Eine gereiftere Form einer k-opt-Heuristik veröffentlichten Lin und Kernighan 1973 [32]. Seit dieser Zeit wurde die Heuristik von vielen Anwendern verändert und verbessert. Gelegentlich gingen dabei die Grundideen etwas verloren, wie bei der Version, die bei Reinelt zu finden ist [47]. An dieser Stelle wird die Heuristik in ihrer Grundform vorgestellt, ohne auf die Variationsmöglichkeiten einzugehen.

Die Lin-Kernighan-Heuristik beruht auf k-opt-Tauschschritten. Ihre Hauptidee besteht darin, diese Schritte verschachtelt durchzuführen. Der Begriff Verschachtelung soll andeuten, daß ein k-opt-Tausch vorbereitet, aber nicht durchgeführt wird, solange eine Überprüfung stattfindet, ob genau dieser k-opt-Tausch nicht noch zu einem $k+1$-opt-Tausch vergrößert werden kann. Sollen z. B. in einem 2-opt-Tausch die Kanten e_1 und e_2 durch die Kanten f_1 und f_2 ersetzt werden, überprüft der Algorithmus, ob er bei Verzicht auf die Kante f_2, die die Tour wieder schließen würde, nicht sogar einen 3-opt-Tausch durchführen kann usw. Eine Anmerkung zur Notation: Mit e_i werden Kanten der Ausgangstour, mit f_i Kanten, die dieser nicht angehören, bezeichnet. Im einzelnen lautet die Heuristik in leicht vereinfachter Form wie folgt (vgl. [32]):

Lin-Kernighan

1. Generiere eine Random-Tour.
2. Setze $G^* = 0$ und $i = 1$. Wähle einen beliebigen Knoten v_1 mit einer adjazenten Kante $e_1 = v_1 v_2$.
3. Von v_2 ausgehend, wähle eine Kante $f_1 = v_2 v_3$ mit $g_1 = w_{12} - w_{23} > 0$. Existiert keine solche Kante, gehe zu Schritt (2) und starte mit einem anderen Knoten.
4. Setze $i = i + 1$. Wähle e_i und f_i nach folgenden Bedingungen:
   (a) $e_i = v_{2i-1} v_{2i}$ wird so gewählt, daß
      i. bei Streichen der Kanten $e_1, \ldots, e_i$ und Hinzufügen der Kanten $f_1, \ldots, f_{i-1}$ sowie der Kante $v_{2i} v_1$ eine Tour entsteht. Dies wäre ein i-opt-Tausch.
      ii. $G_{i-1} - w_{2i,1} + w_{2i-1,2i}$ größer als G^* ist. In diesem Fall wird G^* durch obige Summe ersetzt. Diese Bedingung stellt sicher, daß dieser i-opt-Tausch gewinnbringender als der bisherige $(i-1)$-opt-Tausch ist.
   (b) Wähle eine Kante f_i mit Ausgangspunkt v_{2i} nach diesen Kriterien:
      i. $G_i = \sum_{j=1}^{i} g_i > 0$ mit $g_i = w_{2i-1,2i} - w_{2i,2i+1}$
      ii. Die Wahl von f_i muß erlauben, daß eine adjazente Kante e_{i+1} aus der Tour genommen werden darf.

5. Die wiederholte Auswahl von e_i und f_i in Schritt (4) endet, wenn keine solchen Kanten mehr gefunden werden oder wenn $G_i \leq G^*$. Dann ist keine Verbesserung des bisher vorbereiteten Tauschs mehr möglich. Ist $G^* > 0$, wird die Tour T durch einen i-opt-Tausch zu T' mit $w(T') = w(T) - G^*$ verändert. Danach zurück zu Schritt (2) mit T' als Ausgangstour.

6. Die Heuristik wird beendet, wenn in (2) jeder Knoten betrachtet wurde, ohne eine Verbesserung zu erzeugen.

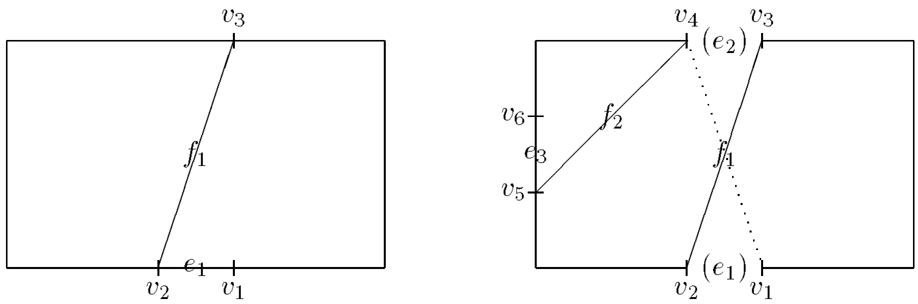

Abbildung 2.2: Zwei Schritte der Lin-Kernighan-Heuristik: Die gepunktete Linie wird nicht als f_2 gewählt, wenn eine weitere Verbesserung möglich ist.

In den vielen verschiedenen Versionen, die heute benutzt werden, erreicht die Lin-Kernighan-Heuristik eine Performance von etwa 0 % bis 4 % Abweichung vom Optimum, je nach Anzahl der Durchläufe und den gewählten Verbesserungsschritten. Gerade bei kleineren Städteanzahlen kann die Heuristik häufig eine optimale Lösung finden. Für größere Städteanzahlen wird die Genauigkeit oft freiwillig eingeschränkt, da die Laufzeit schlechter als $\mathcal{O}(n^2)$ ist[3] und somit schon bei wenigen tausend Städten explodiert. Zum Zeitpunkt der Veröffentlichung merkten Lin und Kernighan an, daß Probleme „bis zu 110 Städten“ untersucht und optimal gelöst wurden. Komplexere Probleme waren 1973 aufgrund der Computerleistungen noch nicht implementierbar. In dieser Diplomarbeit werden Probleme mit mehr als 1.000.000 Städten untersucht – in solchen Fällen sollte vom Einsatz von Algorithmen mit einer Laufzeit von $\mathcal{O}(n^2)$ abgesehen werden.

2.2.6 Simulated Annealing

In diesem Abschnitt wird ein Verfahren vorgestellt, das an einer entscheidenden Stelle auf Wahrscheinlichkeiten zurückgreift und sich dadurch von den bisher betrachteten Heuristiken unterscheidet. Durch das Erlauben von Veränderungen, die eine Tour *nicht* verbessern, soll außerdem erreicht werden, daß genau die lokalen Optima, die die Performance anderer Heuristiken begrenzen, verlassen werden können. Als Literatur für die Darstellung des genauen Hintergrunds, des Metropolis-Filters, können die Arbeiten von Kirkpatrick, Gelatt und Vecchi [28] sowie Metropolis et al. [36] dienen.

Bei der ***Simulated Annealing*** - Heuristik wird im ersten Schritt eine Veränderung der Tour vorbereitet, z. B. ein beliebiger 2-opt-Tausch, unabhängig davon, ob

[3] Lin und Kernighan geben die Laufzeit der Original-Heuristik mit „etwa $\mathcal{O}(n^{2,2})$“ an [32].

diese Veränderung die Gesamtkosten verringert oder nicht. Findet eine Verbesserung statt, wird die Tour verändert – so wie es in jeder bisher betrachteten Heuristik auch der Fall war. Das Neue folgt jetzt: Tritt keine Verbesserung ein, wird die Tour eventuell *trotzdem* verändert, und zwar in Abhängigkeit des Wertes einer Randomzahl x. Dabei ist die Wahrscheinlichkeit einer Änderung desto geringer, je größer die Verschlechterung der Tour oder je weiter fortgeschritten der Algorithmus ist.

Das Wesen der Heuristik ist damit erklärt. Vor einer Ausformulierung des Algorithmus muß jedoch noch auf folgendes eingegangen werden: Ausgangspunkt der Heuristik ist wieder eine beliebige Startlösung. Außerdem wird eine Zeitvariable θ vorgegeben, die die Laufzeit des Algorithmus bestimmt, da sie nach jedem Schritt um einen festen Faktor γ nahe 1 verkleinert wird. Schließlich benötigt man eine Wiederholungszahl r, die angibt, wie lange in einem Schritt verweilt wird. Dieses r erhöht sich im Laufe des Algorithmus, damit die Genauigkeit zum Ende zunimmt. Ein Abbruchkriterium könnte die Güte einer Lösung, die Größe von θ oder das Erreichen einer vorgegebenen Marke von Iterationen ohne Verbesserung sein. Die folgende Ausformulierung sollte nun verständlich sein:

Simulated Annealing

1. Erzeuge eine Startlösung T und wähle eine Zeitvariable $\theta > 0$ sowie einen Wiederholungsfaktor $r > 0$.

2. Solange das Abbruchkriterium nicht erfüllt ist, wiederhole (a) und (b).

   (a) Wiederhole i., ii. und iii. r-mal:

      i. Bereite eine Veränderung der Tour T vor und erhalte T'. Mit $\Delta := w(T') - w(T)$ werde der Kostenunterschied bezeichnet.

      ii. Wähle ein randomisiertes $x \in \mathbb{R}$ mit $0 \leq x \leq 1$.

      iii. Ist $\Delta < 0$ oder $x < \exp(-\Delta/\theta)$, dann setze $T = T'$.

   (b) $\theta = \gamma\theta$, $r = \alpha r$ mit $\gamma = 1 - \varepsilon$ und $1 < \alpha < 2$.

3. Gib Lösung T aus.

Wie man sieht, bleiben viele Parameter offen, die der Anwender für seine Verhältnisse optimieren kann, insbesondere die Art des Veränderungsschrittes und natürlich die Wahl von θ und r. Allgemeine Aussagen zur Performance oder Laufzeit dieses Verfahrens lassen sich daher nicht machen. Bei gut abgestimmten Parametern sind allerdings hervorragende Ergebnisse zu erwarten: Reinelt berichtet von besseren Ergebnissen als bei Anwendung von Lin-Kernighan und sogar von optimalen Lösungen bei sehr langer Laufzeit (siehe [47], vgl. außerdem [26] und [29]).

2.2.7 Lower Bounds

Um die Qualität einer Lösung einschätzen zu können, braucht man eine Vergleichsgröße. In den rechnerischen Untersuchungen des fünften Kapitels fällt dies nicht schwer, da für fast alle untersuchten realen Probleme eine optimale Tour bekannt ist. Dies trifft aber in den meisten Fällen der Praxis nicht zu, sonst wäre die Suche nach einer Lösung ja auch uninteressant. Man kann aber abschätzen, wie nah eine gefundene Lösung am Optimum ist.

Dazu geht man das Problem von einer anderen Richtung aus an. Gesucht ist nun eine Größe, die nahe an der Länge einer optimalen Tour liegt, aber mit Sicherheit *darunter*. Solch eine Grenze, eine Abschätzung nach unten, heißt ***Lower Bound***. Ähnlich wie in den vorangegangenen Abschnitten, kann man auf verschiedene Arten verschieden gute Lower Bounds errechnen. Dabei handelt es sich aber nur um eine Zahl; ein Rückschluß auf eine zugehörige (und damit sehr gute) Tour ist in den meisten Fällen nicht möglich.

Die Idee bei der Suche nach einer unteren Schranke lautet Relaxation. Das TSP ist ein Minimierungsproblem, gesucht wird ein minimierendes Element aus einer bestimmten Menge. Wird diese Menge vergrößert (man sagt, man ***relaxiert*** das Problem), ist es möglich, daß ein Minimierer der größeren Menge schneller gefunden werden kann. Leider ist es aber im allgemeinen der Fall, daß dieser Minimierer nicht der ursprünglich betrachteten Menge angehört, so daß das erhaltene Minimum eine untere Schranke für das Minimum der Ausgangsmenge angibt.

Für eine Relaxation muß man daher die Eigenschaften des TSP aufweichen, um mehr Lösungen zuzulassen. Gleichzeitig sollte erreicht werden, daß das neu entstehende Problem in polynomieller Zeit lösbar ist. Im folgenden werden verschiedene Relaxationen expliziter und impliziter Natur beschrieben. Dabei wird auf einige Begriffe und Sätze der Graphentheorie zurückgegriffen, die über die in Kapitel 2.1.1 gegebene Einführung hinausgehen. Ausführliche Informationen können bei Diestel [13] nachgelesen werden.

Minimum Spanning Tree

Eine einfache Umsetzung benutzt folgende Beobachtung: Jede Tour, aus der man eine Kante entfernt, ist ein besonderer spannender Baum in G, nämlich ein Weg. Das Gewicht eines minimal spannenden Baumes in G ist damit, wenn negative Kantengewichte ausgeschlossen werden, eine untere Schranke für die Länge eines Hamiltonkreises. Diese Schranke nennt man ***MST-bound*** für Minimum Spanning Tree-bound.

Man kann sogar noch den Wert einer Kante hinzufügen; es ist ja eine Kante aus der Tour entfernt worden, um einen spannenden Baum zu erhalten. Dazu kann man z. B. für jedes Blatt des Baumes den Abstand zum zweitnächsten Nachbarn (der nächste Nachbar ist bereits im Baum enthalten) bestimmen und den maximalen dieser Werte dem Gewicht des Baumes hinzufügen. Das Maximum ist deshalb erlaubt, da aus der Tour ja eine beliebige Kante herausgenommen werden konnte, um einen spannenden Baum zu erhalten. Viele weitere kleine Verbesserungen dieser Idee sind möglich.

Der Aufwand für die Erstellung eines minimal spannenden Baumes in der euklidischen Ebene beträgt $\mathcal{O}(n \cdot \log n)$, die Verfeinerungsschritte sind meist von linearem Aufwand. Wie zu erwarten ist, erzeugt diese sehr einfache Idee keine besonders gute untere Schranke: In praktischen Fällen liegt sie ungefähr 11 % unter dem Optimum.

1-Baum-Grenze

Für eine substantielle Verbesserung der obigen Idee muß man etwas weiter ausholen. Wir wollen im folgenden den Knoten v_1 als ausgezeichneten Knoten betrachten.

Wird dieser durch Entfernen der beiden adjazenten Kanten von einer Tour getrennt, bleibt im durch die restlichen Knoten induzierten Graphen ein spannender Baum übrig, nämlich ein Weg durch alle anderen $n-1$ Knoten. Eine Möglichkeit, eine untere Schranke zu finden, liegt darin, jeden Knoten als ausgezeichneten Knoten zu betrachten, für die jeweils anderen Knoten einen minimal spannenden Baum zu suchen und die zwei billigsten Kanten, die mit dem ausgezeichneten Knoten adjazent sind, hinzuzufügen. Die beste so erhaltene Grenze heißt ***1-Baum-Grenze***.

Held-Karp-Grenze

Man kann das TSP wie folgt unter 1-Baum-Gesichtspunkten als ganzzahliges lineares Programm formulieren:

$$opt := min \sum_{v_i v_j \in E} w_{ij} x_{ij}, \tag{2.1}$$

$$\sum_{j=1}^{n} x_{ij} = 2, \qquad \forall v_i \in V \backslash \{v_1\}, \tag{2.2}$$

$$\sum_{j=1}^{n} x_{1j} = 2, \tag{2.3}$$

$$\sum_{v_i v_j \in E} x_{ij} = n, \tag{2.4}$$

$$x(C) \leq |C| - 1, \qquad \forall C \subset \{v_2, v_3, \ldots, v_n\}, \tag{2.5}$$

$$x_{ij} \in \{0, 1\}, \qquad \forall v_i v_j \in E. \tag{2.6}$$

Dabei soll x_{ij} den Wert 1 annehmen, wenn die Kante $v_i v_j$ in die Tour aufgenommen wird, entsprechend den Wert 0 bei Nichtberücksichtigung. Die Gleichungen (2.2) und (2.3) sagen aus, daß jeder Knoten in einer Tour zwei adjazente Kanten besitzt; Gleichung (2.4) bewirkt, daß insgesamt n Kanten vorhanden sind. Durch die Bedingung (2.5) wird sichergestellt, daß keine Kreise gebildet werden, außer dem, der durch Einbindung des ausgezeichneten Knotens v_1 entsteht.

Obiges Programm entspricht einer exakten TSP-Formulierung. Da wir aber Relaxationen untersuchen wollen, verzichten wir im folgenden auf die Gleichung (2.2). Nun erst ist Gleichung (2.4) tatsächlich von Bedeutung (zuvor war sie wegen (2.2) und (2.3) redundant). Das dadurch entstehende Programm beschreibt nun genau 1-Bäume.

Zusätzlich soll noch eine weitere Idee Berücksichtigung finden. Verändert man die Kantengewichte w_{ij} derart, daß man feste Werte π_i und π_j hinzuaddiert, bleibt eine optimale Tour unverändert; lediglich ihre Länge vergrößert sich (wie die Länge einer jeden Tour) um $2 \cdot \sum_{i=1}^{n} \pi_i$. Die Idee dieses Vorgehens ist, die Attraktivität mancher Knoten bzw. Kanten zu verändern. Somit kann man folgende Funktion aufstellen, die für $\pi \equiv 0$ genau die Zielfunktion in (2.1) ergibt:

$$L(\pi) := min \sum_{ij \in E} (w_{ij} + \pi_i + \pi_j) x_{ij} \quad - 2 \cdot \sum_{i=1}^{n} \pi_i. \tag{2.7}$$

L steht hierbei für Lagrange, da die π_i als Lagrange-Multiplikatoren bezeichnet werden. Wie man sich schnell klarmacht, liefert L unter den Nebenbedingungen (2.3) bis (2.6), die sich im übrigen noch vereinfachen lassen, für jede Wahl von π eine untere Schranke für unser gesuchtes Optimum.

Das Maximum von L, das man z. B. durch ein Gradientenverfahren bestimmen kann, ist eine sehr gute untere Schranke. Dieses Maximum wird ***Held-Karp-Grenze*** genannt, da Held und Karp den ersten Algorithmus beschrieben haben, der dieses Maximum sucht [22].

Ein Algorithmus dieser Art kann in $\mathcal{O}(n^2)$ implementiert werden, wobei sich hinter dem $\mathcal{O}$ eine sehr große Konstante verbirgt, hervorgerufen durch die vielen Schritte des Gradientenverfahrens. Dafür wird relativ leicht eine Nähe von unter 2 % zum Optimum erreicht. Die Held-Karp-Grenze wird im Trade-off zwischen Laufzeit und Leistung als die beste angesehen.

2.3 Zusammenfassung und Anwendungsgebiete

Es ist einfach, und doch so schwierig: Kaum ein mathematisches Problem läßt sich so einfach beschreiben und auch dem Laien verständlich machen wie das Handlungsreisendenproblem, und dennoch gehört es zu einer Klasse der härtesten Probleme in der Mathematik. Die Suche nach einem kürzesten Weg durch eine gegebene Anzahl von Städten bleibt ohne Tricks in annehmbarer Zeit erfolglos. Auf die Zeit wird aber meist das Hauptaugenmerk gerichtet sein.

Steht viel Zeit zur Verfügung, kann man durch geschickte Benutzung der Lin-Kernighan-Heuristik sehr gute Lösungen erhalten, deren Qualität man durch Vergleich mit der Held-Karp-Grenze ebenfalls sehr gut einschätzen kann. Kombiniert man diese Verfahren mit einer Branch & Cut-Strategie, in der sukzessive offensichtlich unbrauchbare Kanten aus dem Graphen entfernt werden, besteht sogar die Hoffnung, eine optimale Lösung zu finden. Für eine sehr gute oder gar optimale Lösung muß man allerdings schon bei wenigen tausend Städten einige Stunden oder vielleicht auch Tage an Rechenzeit einkalkulieren.

Das andere Extrem, sehr wenig Zeit zur Verfügung zu haben, führt dazu, daß Zugeständnisse an die Qualität der Lösung gemacht werden müssen. Eine der schnellsten Methoden, die für wenige tausend Städte in Hundertstelsekunden zum Erfolg führt, wird in den folgenden Kapiteln genauer untersucht.

Zwischen diesen Polen ist vieles möglich. Für jede Problemstellung können eine geeignete Heuristik und dazugehörende Verbesserungen gefunden werden. Dabei gilt, daß mit steigendem Zeitpotential das Ergebnis besser werden sollte. Investiert man z. B. schon vor der Anwendung einer Improvement Heuristic einen Zeitaufwand von $\mathcal{O}(n^2 \cdot \log n)$, kann man mit einem einfachen Greedy-Algorithmus, der bei Reinelt [47] nachzulesen ist, eine recht gute erste Näherung erzielen. Erfahrungsgemäß sollte aber schon bei einigen tausend Städten jeder Algorithmus gemieden werden, der eine schlechtere als quadratische Laufzeit hat.

Wie schon kurz angedeutet, findet das TSP zahlreiche Anwendungen in der Praxis, von denen hier einige vorgestellt werden:

Eine klassische Aufgabe ist das ***Warenhausproblem***. Angenommen, ein Roboter oder Gabelstapler fährt durch das Lager eines Warenhauses, um bestellte Artikel einzusammeln. Die Minimierung der Fahrtzeit ist ein TSP. Dabei sind die Lagerorte der Artikel die Knoten, die Gänge des Lagers geben die Kanten an (man beachte, daß dieser Graph vervollständigt werden muß), und die Kantengewichte ergeben sich aus der Zeit, die der Roboter braucht, um vom einen zum anderen Lagerort zu fahren.

Ebenfalls schon lange bekannt ist das ***Vehicle Routing Problem***. Die Problemstellung lautet: Wie lange brauchen die m vorhandenen Postautos einer Stadt, um alle Briefkästen der Stadt zu leeren? Da nun m „Handlungsreisende“ zur Verfügung stehen, nennt man dieses Problem ein m-TSP. Es ist als einfaches TSP formulierbar und damit mit den betrachteten Methoden lösbar.

Strukturen von Kristallen können durch die Messung der Reflexion von Röntgenstrahlen erkannt werden. In solch einer ***Röntgen-Kristallographie*** wird der Kristall aus verschiedenen Positionen bestrahlt. Die Bewegung des Meßinstrumentes in diese Positionen kostet Zeit, also möchte man den Bewegungsablauf optimieren. Daraus ergibt sich ein TSP.

Die vielleicht verbreiteste Anwendung des TSP in der Industrie ergibt sich aus dem ***Drilling Problem***. In ein Werkstück, z. B. eine Leiterplatte, müssen an vorgegebenen Stellen Löcher verschiedener Größe gebohrt werden. Das Wechseln der Bohrspitze verbraucht so viel Zeit, daß es notwendig ist, alle Löcher gleicher Größe hintereinander zu bohren und danach die Spitze zu wechseln. Die vorgegebenen Stellen, die durchbohrt werden müssen, werden als Knoten angesehen, der Weg, den der Bohrer zwischen solchen Stellen zurücklegt, als Kanten. Für jede Lochgröße ergibt sich ein TSP. Solche TSP sind häufig von besonderer Form, wie auch die Testfälle ab Kapitel vier zeigen. Dort sind einige solcher ***Printed Circuit Boards*** berücksichtigt. Es ergeben sich Cluster von Punkten entlang horizontaler und vertikaler Linien, manchmal als Ergebnis davon, daß die Bohrer nur horizontal und vertikal bewegt werden (Manhattan-Metrik). Diese Punktmengen erweisen sich für viele Heuristiken als besonders schwierig zu bewältigen und verdienen, gerade wegen ihres Praxis-Bezuges, hohe Aufmerksamkeit.

Kapitel 3

Raumfüllende Kurven

Es ist möglich, eine stetige Abbildung zu konstruieren, die surjektiv das Einheitsintervall auf das Einheitsquadrat abbildet. Oder, unmathematisch ausgedrückt: Wenn man nur lange genug mit einem Stift unsichtbare Linien malt, wird ein weißes Blatt Papier irgendwann vollständig schwarz. Auch wenn der gesunde Menschenverstand der mathematischen Logik widersprechen möchte, existieren solche „topologischen Monster“ [6], die einen Raum ausfüllen.

Dieses Kapitel präsentiert die Definition einer raumfüllende Kurve, einen kurzen Abriß über die Historie, die Generierung der bekanntesten Kurven sowie ihrer Eigenschaften und, im Vorgriff auf folgende Kapitel, Hinweise auf neuere Kurven, die im Zusammenhang mit dieser Arbeit entstanden.

3.1 Grundlegendes

Aufgrund der sehr einfachen Definition ist es angebracht, dieses Unterkapitel mit der Definition einer (raumfüllenden) Kurve und der Darstellung und Beantwortung einiger sich sofort ergebender Fragen zu beginnen, bevor ein kurzer historischer Überblick die Überleitung zu der Entwicklung spezieller Kurven bildet.
Am Anfang steht die Definition einer Kurve:

> Eine ***Kurve*** ist das Bild einer stetigen Abbildung $f : [0,1] \to [0,1]^n$.

Eine für unsere Zwecke angebrachte Definition einer raumfüllenden Kurve lautet:

> Eine ***raumfüllende Kurve*** ist eine Kurve,
> deren zugrundeliegende Abbildung surjektiv ist.

Dabei wird im folgenden grundsätzlich $n = 2$ sein, da sich die Ausgangsproblemstellung in der euklidischen Ebene abspielt. Im Sprachgebrauch kommt es außerdem gelegentlich dazu, daß die Abbildung f selbst als raumfüllende Kurve bezeichnet wird; gemeint ist allerdings immer das Bild von $[0,1]$ unter f. Der Vollständigkeit halber sei hier außerdem erwähnt, daß eine Kurve, die aus einer injektiven Abbildung entsteht, ***Jordan-Kurve*** genannt wird.

Existiert ein solches Gebilde überhaupt, das es erlaubt, ein eindimensionales Konstrukt, wie z. B. ein Intervall, in eine beliebig gewählte Dimension zu versetzen, so daß letztere dabei komplett ausgefüllt wird? Und dies in einer so simplen Form wie einer Abbildung? Die erste Frage beantwortete Peano 1890 positiv, indem er eine solche Kurve konstruierte [43]; die zweite Frage kann ebenfalls bejaht werden, wobei man freilich ein Wort wie „simpel“ auslegen muß: Statt einer eigentlichen Abbildungsvorschrift findet man bei den meisten raumfüllenden Kurven rekursive Konstruktionsprinzipien, die erst im berühmten unendlichen Schritt eine Abbildung ergeben.

Weiter interessiert die Frage, ob eine stetige, surjektive Abbildung zwischen zwei Dimensionen auch noch injektiv, damit also bijektiv, sein kann. Es überrascht nicht, eine negative Antwort zu finden – wenn ein stetiger 1 : 1 - Übergang zwischen beliebigen Dimensionen möglich wäre, so wäre die Welt auch außerhalb der Mathematik wohl nicht die, an die wir uns gewöhnt haben. Überraschend ist allerdings der Zeitpunkt, zu dem dieses Problem bereits gelöst wurde: Netto zeigte 1879, daß eine bijektive Abbildung zwischen unterschiedlich-dimensionalen glatten Mannigfaltigkeiten notwendigerweise unstetig sein muß [40] – ein Ergebnis, das ein Jahr nach Cantors Satz über die Kardinalitätsgleichheit zweier beliebiger, glatter Mannigfaltigkeiten entstand [9]. Eine raumfüllende Kurve kann folglich nicht noch zusätzlich injektiv, eine Jordan-Kurve nicht noch zusätzlich surjektiv sein. Der zeitliche Ablauf dieser Beobachtungen legt nahe, daß die Entdeckung der raumfüllenden Kurven letztlich durch Cantors Satz angestoßen worden sein dürfte.

Peanos Entdeckung einer raumfüllenden Kurve folgte ein Jahr später eine Kurve, die Hilberts Namen trägt [23] und die bis heute bekannteste raumfüllende Kurve sein dürfte. 1904 entdeckte Lebesgue [31] die erste fast überall differenzierbare raumfüllende Kurve, und Sierpiński war es, der 1912 das Konstruktionsprinzip von Quadratunterteilungen in Triangulierungen veränderte [54]. Damit endete die Zeit der „klassischen“ raumfüllenden Kurven. Deutlich später erweckte das Interesse an Fraktalen, die uns aufgrund ihrer gebrochenen Dimension hier nur am Rande interessieren werden, die Untersuchungen zu raumfüllenden Kurven (im folgenden: **SFC** für space-filling curves) zu neuem Leben.

Für eine ausführlichere Zusammenfassung der Historie und Ergänzungen zu diesem Kapitel kann Sagans Buch „Space-Filling Curves“ [51] dienen.

3.2 Generierung einer raumfüllenden Kurve

Wie schon erwähnt, gibt es kaum eine Kurve, deren Funktionsvorschrift explizit angegeben wird. Im folgenden werden wir sehen, warum: Der einfachste Zugang zu einer SFC ist ein geometrischer. Am ausführlichen Beispiel der Hilbert-Kurve können wir das allgemeine Konstruktionsprinzip einer SFC nachvollziehen. Danach ist es möglich, das Wesen fast jeder der nachfolgenden Kurven durch einige wenige Zeichnungen anzugeben, was die Übersichtlichkeit stark erhöht.

3.2.1 Allgemeines Konstruktionsprinzip

Die Idee, eine SFC durch geometrische Anschauung zu beschreiben, entstammt folgender Beobachtung: Angenommen, es ist möglich, das Einheitsintervall stetig und surjektiv auf das Einheitsquadrat abzubilden. Dann ist es mit gleicher Denkweise möglich, z. B. ein Viertel des Einheitsintervalles auf ein Viertel des Einheitsquadrates abzubilden: Man skaliere das Problem lediglich um den Faktor $\frac{1}{4}$. Weiter angenommen, man kann vier Viertel auf beiden Seiten der Abbildung sinnvoll aneinanderfügen (d. h. $[0,1]$ und $[0,1]^2$ erzeugen). Dann hat man die Problemgröße verkleinert, indem man die Objekte in vier Unterobjekte verfeinert hat. So ergibt sich die Idee einer rekursiven Definition, die, wie wir später sehen werden, erfolgreich ist. Der Trick ist dabei die geschickte Verbindung der Unterteilungen, später Anordnung genannt.

Die nächsten Absätze werden dazu dienen, einige Begriffe einzuführen, die zwar zumeist leicht verständlich, aber noch nicht in den allgemeinen Sprachgebrauch übergegangen sind und daher einer Erklärung bedürfen. Zum Verständnis des Konstruktionsprinzips sind folgende Definitionen notwendig:

Mit ***Verfeinerung*** bezeichnen wir eine vollständige, gleichmäßige Unterteilung eines gegebenen geometrischen Objektes gemäß einer angegebenen Verfeinerungsvorschrift[1]. Zum Beispiel soll eine 2×2-Quadrat-Verfeinerung des Einheitsquadrates die Unterteilung des Einheitsquadrates in vier gleich große Unterquadrate bedeuten, wie oben bereits angedeutet. Zur Vereinfachung setzen wir eine „Verfeinerung“ mit einer „Quadrat-Verfeinerung“ gleich. Die Durchführung einer Verfeinerung erhöht die Verfeinerungstiefe oder kurz ***Tiefe*** um 1, beginnend mit Tiefe 1 des Ausgangsobjektes.

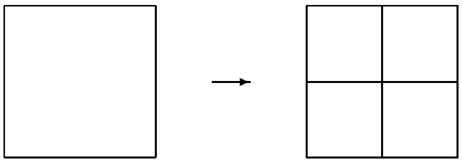

Abbildung 3.1: 2×2-Quadrat-Verfeinerung in Tiefe 1 und 2

Analog dazu verstehen wir unter einer 2×2-Dreieck-Verfeinerung die Unterteilung eines Dreiecks in $2 \cdot 2 = 4$ Unterdreiecke. Im Sprachgebrauch heißt eine allgemeine Dreieck-Unterteilung auch ***Triangulierung***.

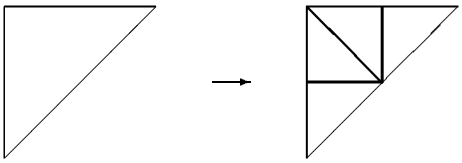

Abbildung 3.2: 2×2-Triangulierung in Tiefe 1 und 2

Von einer ***regelmäßigen Verfeinerung*** sprechen wir, wenn die Verfeinerungsvorschrift eine direkte Rekursion ist. Dies bedeutet explizit, daß die geometrischen Objekte rekursiv nur in gleichartige Objekte überführt werden dürfen, d. h. Quadrate in Unterquadrate oder Dreiecke in Unterdreiecke. Die bisher angegebenen Verfeinerungen sind alle regelmäßig: Durch die Vorschrift „2×2-Quadrat-Verfeinerung“ sind rekursiv beliebig viele Verfeinerungsschritte definiert. Die Vorschrift „2×2-Triangulierung“ bedeutet eine zweimalige *Longest Edge Bisection*; diese Verfeinerung kann ebenfalls beliebig oft durchgeführt werden.

Eine $k \times k$-Verfeinerung mit kleinem k wollen wir als ***grob***, eine mit großem k entsprechend als ***fein*** bezeichnen.

[1]Allgemeiner spricht man bei einer mosaikartigen Unterteilung einer Fläche von ***Tilings***.

Unter einer ***Anordnung*** verstehen wir eine Kennzeichnung geometrischer Objekte, aus der eindeutig hervorgeht, welche Reihenfolge diese Objekte haben. Dies kann z. B. durch Indizierung mit natürlichen Zahlen oder durch Pfeile erfolgen, wie in Abbildung 3.3 angedeutet.

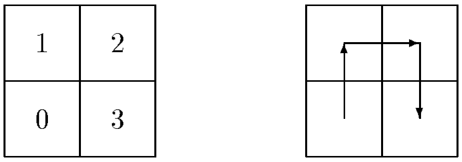

Abbildung 3.3: Anordnung durch Indizes bzw. Pfeile

Das allgemeine ***Konstruktionsprinzip einer raumfüllenden Kurve*** läßt sich damit so erklären: Angegeben werden müssen lediglich eine Verfeinerung des Einheitsquadrates bis in eine beliebige Tiefe und eine Anordnung der Verfeinerungsobjekte. Wir werden im folgenden sehen, daß eine „stetige" Anordnung in einer Tiefe ∞ eine raumfüllende Kurve erzeugt.

Selbstverständlich kann eine SFC auch auf andere Art erzeugt werden, wie zum Ende dieses Kapitels dargestellt wird; das hier vorgestellte Prinzip eignet sich aber am besten für die Umsetzung einer solchen Kurve auf einem Rechner. Praktischerweise nutzt man dazu eine regelmäßige Verfeinerung und versucht, die Anordnung der Unterobjekte ebenfalls rekursiv durchzuführen. Wie man sich dies vorzustellen hat, wird nun am Beispiel der Hilbert-Kurve erklärt.

3.2.2 Die Hilbert-Kurve

Hilberts Anordnung, basierend auf einer 2×2-Verfeinerung, sieht im ersten Schritt so aus: Das Intervall $[0, \frac{1}{4}]$ wird auf das linke untere, das Intervall $[\frac{1}{4}, \frac{1}{2}]$ auf das linke obere, das Intervall $[\frac{1}{2}, \frac{3}{4}]$ auf das rechte obere und schließlich das Intervall $[\frac{3}{4}, 1]$ auf das rechte untere Unterquadrat von $[0, 1]^2$ abgebildet. Diese Anordnung wird in Abbildung 3.4 durch die Indizes 0 bis 3 angedeutet. Man beachte die Gleichmäßigkeit des Vorgehens: Zu jeder der vier Zahlen „gehört" nun noch ein Viertel der Abbildung.

1	2
0	3

Abbildung 3.4: Anordnung der Hilbert-Kurve in Tiefe 2

Im nächsten Verfeinerungsschritt, in Tiefe 3 also, werden die Unterquadrate ihrerseits wieder unterteilt. Aus der Anordnung aus Tiefe 2 folgt logisch, daß die Unterquadrate des mit „1" gekennzeichneten Quadrates erst nach den vier Unterquadraten des „0"-Quadrates angeordnet werden. Somit verteilen sich in Tiefe 3 die Bezeichnungen 0 bis 3 auf das ehemalige „0"-Quadrat, die 4 bis 7 auf das ehemalige „1"-Quadrat usw.

Versucht man, obiges Anordnungsschema rekursiv zu definieren, stellt sich die Frage: Wie ordnet man die einzelnen Indizes genau an? Abbildung 3.5 zeigt, daß

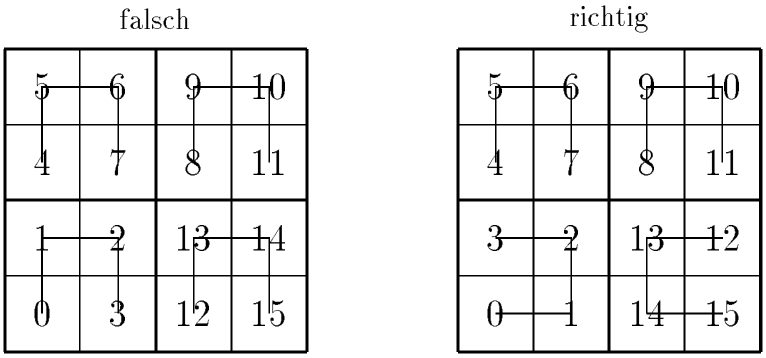

Abbildung 3.5: Eine „sture“ Anordnung links verhindert Stetigkeit, die „richtige“ Änderung der Anordnung rechts ermöglicht Stetigkeit

eine „sture“ Fortsetzung der ersten Idee, d. h. die jeweilige Anordnung links unten, links oben, rechts oben, rechts unten, ein Stetigkeitsproblem hervorruft: Vom neuen Quadrat 3 ist offensichtlich kein stetiger Übergang zu Quadrat 4 möglich, denn die Quadrate berühren sich nicht einmal; die Abbildung macht einen Sprung. Damit die Kurve ihre Stetigkeit behalten kann, müssen Quadrate mit fortlaufenden Nummern einander berühren. Daher der Trick, in jedem Schritt die Anordnung des vorherigen „0“-Quadrates und des „3“-Quadrates (jeweils modulo 4) so zu drehen bzw. zu spiegeln wie in Abbildung 3.5 zu sehen: Nun kann man sich davon überzeugen, daß fortlaufende Nummern genau eine Seite ihrer Quadrate teilen. Der zugehörige Beweis, in der Literatur gerne vernachlässigt, folgt zum Ende dieses Unterkapitels und ist Grundlage für den im nächsten Unterkapitel dargestellten Stetigkeitsbeweis.

In Tiefe 3 wird nun z. B. das Teilintervall $[\frac{1}{2}, \frac{9}{16}]$ auf das mit „8“ markierte Unterquadrat abgebildet. Durch rekursives Vorgehen in diesem Sinne schrumpfen sowohl die Intervalle als auch die Quadrate auf Punkte zusammen: Wir erhalten eine Abbildung. Die Hilbert-Kurve kann nun durch folgende Beobachtung definiert werden[2]:

> Jedes $t \in [0, 1]$ ist eindeutig bestimmt durch eine Folge von ineinandergefaßten, geschlossenen Intervallen, die durch oben genannte Verfeinerung entstehen und deren Länge auf 0 schrumpft. Mit dieser Folge korreliert eine eindeutige Folge ineinandergefaßter, geschlossener Quadrate, deren Diagonalen zu einem Punkt schrumpfen und die einen eindeutigen Punkt in $[0, 1]^2$ definieren: das Bild $f(t)$ von t. Wir nennen $f([0, 1])$ die ***Hilbert-Kurve***.

Das einfache Aufeinanderfolgen der sogenannten ***Hilbert-Indizes*** (vgl. [1]) ist jedoch nicht gerade besonders übersichtlich. Statt dessen kann man die Anordnung aber auch mit Pfeilen darstellen, die die Mittelpunkte der Quadrate verbinden, was für die Tiefen 2 und 3 der Hilbert-Kurve ein sicherlich bekannteres Bild erzeugt (siehe Abbildung 3.6).

Eine solche Darstellung wurde in dieser Arbeit bisher vermieden (und soll auch im folgenden eher spärlich benutzt werden), um Mißverständnissen vorzubeugen. Die

[2] Übersetzung aus [51], Definition 2.1, Seite 11.

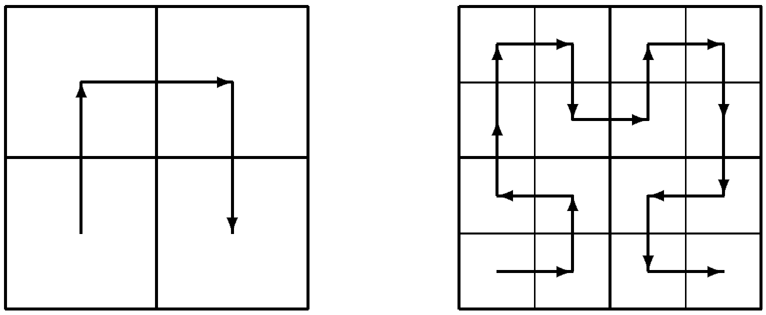

Abbildung 3.6: Pfeil-Visualisierung der Hilbert-Kurve in Tiefe 2 und Tiefe 3

dargestellten Pfeile haben nämlich *nichts* mit dem Verlauf der Hilbert-Kurve zu tun. Die Kurve selbst entsteht, wie wir der Definition entnehmen können, erst in der Verfeinerungsstufe ∞. Es gibt also keine eigentliche Kurve der Tiefe n, sondern nur die Tiefe n des Konstruktionsprinzips, auch wenn dieser feine Unterschied im Sprachgebrauch nicht gemacht wird. Schaut man auf die Tiefe 2 in Abbildung 3.6, so sieht man, daß die Pfeildarstellung nicht den Verlauf der Kurve darstellen kann: Drei Pfeile würden einer Intervallunterteilung in Drittel entsprechen, was definitiv falsch ist. Daher soll diese Darstellung nur aus optischen, nicht aber aus mathematischen Gründen benutzt werden.

Eine andere Darstellungsart, die der Wahrheit ein bißchen näher kommt, ist in Abbildung 3.7 aufgezeigt.

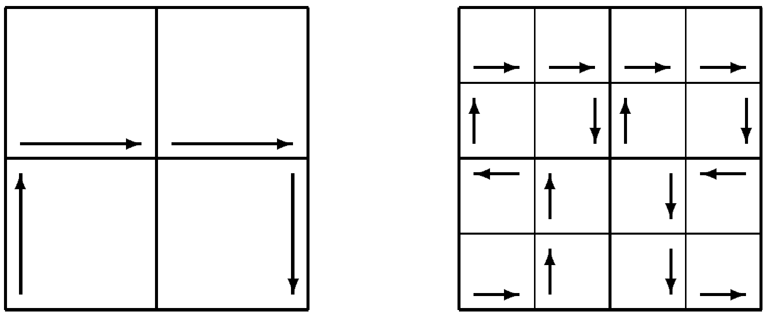

Abbildung 3.7: Verbesserte Pfeil-Visualisierung der Hilbert-Kurve

Diese Kurvenvisualisierung hat entscheidende Vorteile. Erstens bleibt die Gleichzahligkeit von Quadraten und Pfeilen gewahrt, zweitens hat jeder Pfeil einen anderen Nutzen: Er gibt durch die Richtungsangabe einen Hinweis darauf, wie die Anordnung in einem weiteren Schritt auszusehen hat. Die Pfeilspitze zeigt nämlich auf die Stelle, an der in der nächsthöheren Tiefe das dritte Verfeinerungsquadrat plaziert ist; der Pfeilanfang liegt in dem Bereich, in dem das nullte Verfeinerungsquadrat des nächsten Schrittes plaziert ist. Die Anordnung der beiden übrigen Quadrate ist damit trivial. Diese Darstellung gibt also mehr und richtigere Informationen als die weiter verbreitete aus Abbildung 3.6. Dies soll allerdings nicht darüber hinwegtäuschen, daß der Pfeilverlauf auch hier wenig mit dem tatsächlichen Kurvenverlauf zu tun hat.

Die Darstellung aus Abbildung 3.7 hilft ebenfalls bei dem Beweis der bereits angekündigten Aussage:

Proposition 3.1 *Unter Beachtung der vorgestellten Konstruktion der Hilbert-Kurve bezeichnen in jeder Tiefe je zwei aufeinanderfolgende Indizes solche Quadrate, die genau eine Seite miteinander teilen.*

Beweis: Da das Konstruktionsprinzip der Hilbert-Kurve rekursiv aufgebaut ist, bietet sich ein induktiver Beweis an. In Tiefe 2 ist die Behauptung offensichtlich erfüllt. Die Darstellung in dieser Tiefe ist gleichzeitig die erste Rekursionsangabe der Kurve; diese wird auch ***Leitmotiv*** genannt. Wir können also festhalten, daß die Behauptung innerhalb jedes Quadrates gilt, das mit dem Leitmotiv beschriftet ist – dabei ist es offensichtlich unerheblich, ob dieses Quadrat gedreht oder gespiegelt wird.

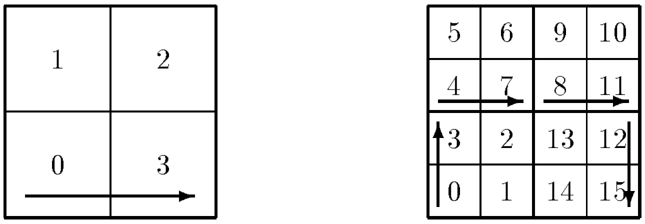

Abbildung 3.8: Leitmotiv der Hilbert-Kurve und Rekursion

Der Induktionsschritt besteht in der Überprüfung, ob die Behauptung nach einem beliebigen Rekursionsschritt noch gilt. Da in der rekursiven Ersetzung des Leitmotivs pro Quadrat wieder vier Quadrate entstehen, die ihrerseits mit dem Leitmotiv beschriftet sind, bleiben lediglich die Übergange zwischen den neuen Quadraten zu untersuchen. Sind diese derart gestaltet, daß jede Pfeilspitze einen Pfeilanfang „berührt“, ist die gewünschte Numerierung sichergestellt, da sich nun die betreffenden End- und Anfangsquadrate berühren. Dies geschieht durch das Umstellen des Leitmotivs in den Quadraten 0 und 3 (modulo 4), wie in Abbildung 3.8 beim Übergang von Tiefe 2 zu Tiefe 3 dargestellt.

3.2.3 Stetigkeit und Differenzierbarkeit

Für die Hilbert-Kurve sind nun folgende Beobachtungen festzuhalten:

Satz 3.2 *Die Hilbert-Kurve ist eine raumfüllende Kurve.*

Beweis[3]: Offensichtlich handelt es sich bei der im letzten Abschnitt erzeugten Abbildung um eine Abbildung des Einheitsintervalls in das Einheitsquadrat. Es bleibt zu zeigen, daß diese Abbildung stetig und surjektiv ist.

Der Stetigkeitsbeweis fällt dank Proposition 3.1 nicht schwer: Betrachten wir die Tiefe $n+1$ der Konstruktion: Das Einheitsintervall ist zu diesem Zeitpunkt in 2^{2n} gleich lange Teilintervalle aufgeteilt; entsprechend gibt es 2^{2n} Unterquadrate des Einheitsquadrates mit Seitenlänge $1/2^n$. Wähle nun t_1 und $t_2 \in [0,1]$, so daß $|t_1 - t_2| < 1/2^{2n}$. Das Intervall $[t_1, t_2]$ kann sich höchstens über zwei der oben bezeichneten Teilintervalle erstrecken. Wir wissen aber, daß dann die zugehörigen Bilder $f(t_1)$ und $f(t_2)$ entweder im gleichen oder schlimmstenfalls in zwei benachbarten Quadraten liegen (Proposition 3.1). Es ergibt sich: $\|f(t_1) - f(t_2)\| \leq \sqrt{5}/2^n$, wobei

[3]Für Satz 3.2 und Proposition 3.3 sei erneut auf Sagan [51] verwiesen.

die rechte Seite die Länge der Diagonale zweier aneinandergesetzter Unterquadrate der Seitenlänge $1/2^n$ ist. Die klassische ε-δ–Definition der Stetigkeit ist damit erfüllt.

Zur Surjektivität: Betrachte einen beliebigen Punkt des Einheitsquadrates. Dieser Punkt liegt in einer Folge von ineinandergefaßten, geschlossenen Quadraten, die durch das Konstruktionsprinzip der Hilbert-Kurve beschrieben sind. Läuft die Tiefe der Kurve gegen ∞, schrumpfen die Quadrate genau zu dem betrachteten Punkt zusammen. Zu jedem genannten Quadrat gehört ein eindeutiges Teilintervall des Einheitsintervalls; die Folge dieser Teilintervalle schrumpft ebenfalls zusammen, nämlich auf das Urbild des gewählten Punktes. Jeder Punkt des Einheitsquadrates hat also ein Urbild im Einheitsintervall; die Abbildung ist surjektiv.

In diesem Vorgehen wird ebenfalls deutlich, daß die Abbildung nicht injektiv ist: Die Folge von Quadraten, in denen ein gewählter Punkt liegt, ist im allgemeinen nicht eindeutig. Liegt der Punkt z. B. am Rand eines Quadrates, kann dieses durch das Nachbarquadrat ersetzt werden. Da aber nicht *jedes* Nachbarquadrat einen fortlaufenden Index hat, kann diese Ersetzung zu einem unterschiedlichen Urbild führen.

Proposition 3.3 *Die der Hilbert-Kurve zugrundeliegende Abbildung ist nirgends differenzierbar.*

Beweis: Sei $n+1$ die Tiefe der Kurve. Wähle dann zu einem beliebigen $t \in [0,1]$ ein $t_n \in [0,1]$, so daß einerseits $|t - t_n| \leq 10/2^{2n}$ und andererseits das Bild von t durch mindestens ein Quadrat der Seitenlänge $1/2^n$ vom Bild von t_n getrennt wird. Dies ist für $n \geq 2$ immer möglich, da ein Quadrat von höchstens acht weiteren Quadraten berührt wird und die Abstandswahl von t und t_n mindestens zehn Intervalle und dazugehörige Quadrate zuläßt. Dann gilt für die Koordinatenfunktionen $\phi(t)$ und $\psi(t)$ des Bildes von t in dieser Tiefe:

$$|\phi(t) - \phi(t_n)|/|t - t_n| \geq 2^n/10 \text{ und}$$
$$|\psi(t) - \psi(t_n)|/|t - t_n| \geq 2^n/10.$$

Die Koordinatenfunktionen und damit die Abbildung selbst sind nicht differenzierbar.

Ähnlich wie bei der Hilbert-Kurve kann man die Gedankengänge der letzten Abschnitte auch für andere Kurven wiederholen. Untersucht man nun genauer, ob die aufgestellten Sätze in diesen Fällen auch noch gelten, kommt man zu folgender wichtiger Feststellung:

Satz 3.4 *Jede Angabe einer regelmäßigen Verfeinerung des Einheitsquadrates und einer Anordnung der Verfeinerungsobjekte, die sicherstellt, daß aufeinanderfolgende Indizes zu einander berührenden geometrischen Objekten gehören, führt zu einer raumfüllenden Kurve.*

Beweis: Der in Satz 3.2 geführte Stetigkeitsbeweis ist nicht von der Verfeinerung abhängig, weder von ihrer Art (Quadrat, Triangulierung oder andere) noch von ihrer Feinheit (2×2, 5×5 oder andere). Der Beweis zu Proposition 3.1 nutzt lediglich die *Regelmäßigkeit* der Verfeinerung aus, nämlich daß aufeinanderfolgende Indizes in benachbarten, zweidimensionalen Objekten liegen. Für eine andere Verfeinerung, sei es in Art oder Feinheit, ändern sich nur die Konstanten. Allerdings ist es selten der Fall, daß die Beobachtung des Verfeinerungsschrittes von Tiefe 2 zu Tiefe 3 bereits ausreicht, wie es in Proposition 3.1 der Fall ist, da die Rekursion im allgemeinen nicht bereits in Tiefe 2 ersichtlich ist.

Des weiteren sei an dieser Stelle darauf hingewiesen, daß auch bei Quadrat-Verfeinerungen *nicht* gefordert werden muß, daß aufeinanderfolgende Quadrate eine Seite teilen. Das Berühren in lediglich einem Punkt ist für obigen Beweis ausreichend, solange durch die Regelmäßigkeit sichergestellt wird, daß in jeder Tiefe diese Berührung stattfindet. Man spricht dann davon, daß die Kurve ***über einen Punkt*** läuft: Der gemeinsame Punkt der beiden benachbarten Teilintervalle wird genau auf den Berührpunkt der Unterquadrate abgebildet.

Die Surjektivität der betrachteten Abbildung ist ebenfalls von Art und Feinheit der Verfeinerung unabhängig. Sichergestellt werden muß lediglich, daß jedes entstehende Unterobjekt einen Index zugeordnet bekommt, was durch das Anordnungsschema gewährleistet ist.

3.3 Weitere klassische Kurven

Im letzten Abschnitt konnten wir feststellen, daß eine raumfüllende Kurve z. B. durch eine regelmäßige Verfeinerung und ein Anordnungsschema ausreichend beschrieben werden kann. Dieser Abschnitt soll nun dazu dienen, die bekanntesten klassischen SFC katalogartig auf diese Art darzustellen. Dabei gelingt es nicht immer, die Rekursion bereits in Tiefe 2 eindeutig anzugeben. Um die Darstellung nicht zu kompliziert werden zu lassen, sind in diesen Fällen nur die Leitmotive und eine Visualisierung angegeben. Die vollständige Konstruktion der Kurve muß der angegebenen Literatur entnommen werden. In vielen Fällen werden, der Übersichtlichkeit zuliebe, beide eingeführten Pfeilvisualisierungen benutzt.

3.3.1 2×2-Verfeinerungen

Name:	**Moore**
Verfeinerung:	2×2
Jahr der Entdeckung:	1900
Anfangspunkt:	(geschlossen)
Endpunkt:	(geschlossen)
Anmerkung:	Moore-Kurve ist geschlossene Hilbert-Kurve
Literatur:	[37]

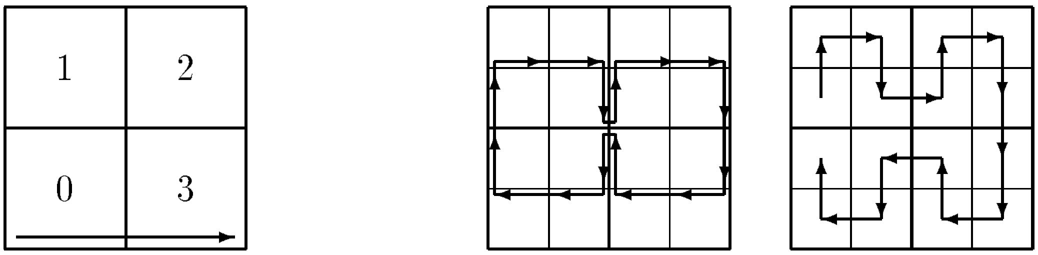

Abbildung 3.9: Rekursion und Visualisierung der Moore-Kurve

Name:	**Lebesgue**
Verfeinerung:	2×2
Jahr der Entdeckung:	1904
Anfangspunkt:	(0, 0)
Endpunkt:	(1, 1)
Anmerkung:	Lebesgue-Kurve ist fast überall differenzierbar
Literatur:	[51]

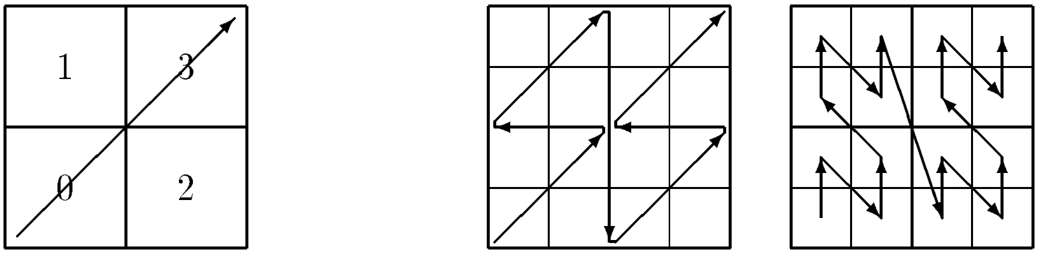

Abbildung 3.10: Rekursion und Visualisierung der Lebesgue-Kurve

Die Stetigkeit der Lebesgue-Kurve läßt sich nicht auf jene Art beweisen, die für die Hilbert-Kurve benutzt wurde. Zurückzuführen ist dies hauptsächlich auf die Konstruktion der Lebesgue-Kurve: Für sie gilt *nicht*, daß z. B. in Tiefe 2 alle vier Unterquadrate durch Viertel des Einheitsintervalls erzeugt werden. Statt dessen werden diese Unterquadrate durch Neuntel des Einheitsintervalls erzeugt; der Rest des Urbildraums wird für lineare Interpolationen zwischen den End- und Anfangspunkten zweier aufeinanderfolgender Quadrate verwendet. Die in Abbildung 3.10 dargestellten Verbindungsstrecken zwischen den Quadraten sind also nicht optischer Natur, sondern tatsächlich Teil der Kurve. So wird die Kurve sowohl stetig als auch fast überall differenzierbar (nämlich dort, wo die lineare Interpolation angewendet wurde).

3.3.2 3×3-Verfeinerungen

Name:	**Peano**
Verfeinerung:	3×3
Jahr der Entdeckung:	1890
Anfangspunkt:	(0, 0)
Endpunkt:	(1, 1)
Anmerkung:	Peano-Kurve war die erste raumfüllende Kurve
Literatur:	[43]

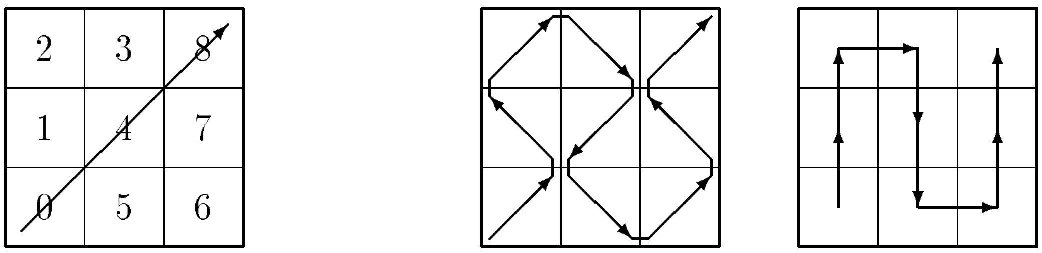

Abbildung 3.11: Rekursion und Visualisierung der Peano-Kurve

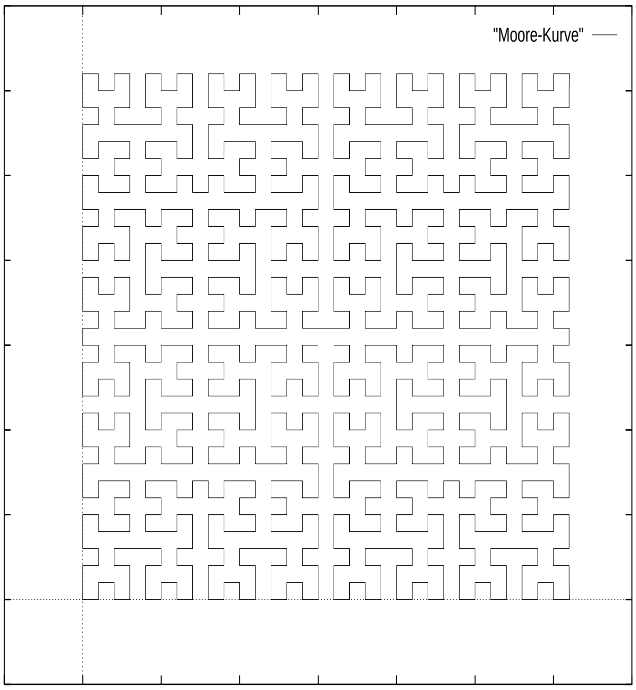

Abbildung 3.12: Darstellung der Moore-Kurve in Tiefe 6

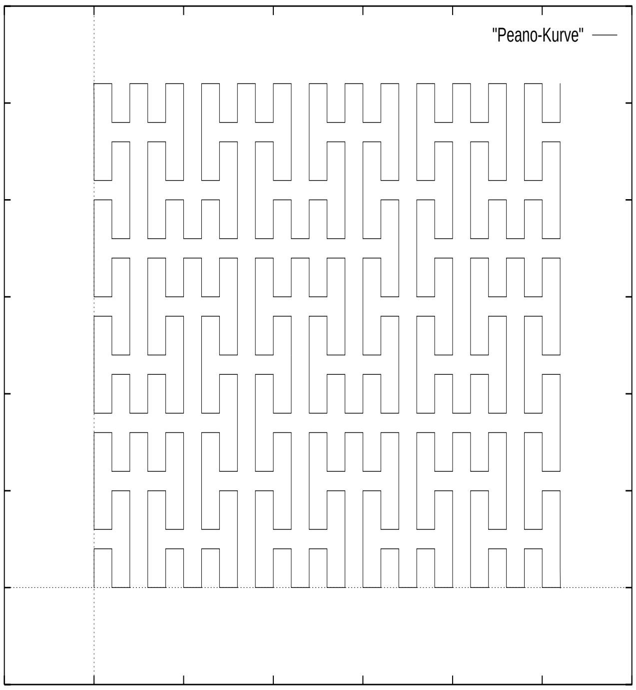

Abbildung 3.13: Darstellung der Peano-Kurve in Tiefe 4

Die folgende Kurve von Wunderlich ist eng mit der Peano-Kurve verwandt. Das Leitmotiv der Peano-Kurve wechselt sich in jedem Schritt mit dem hier dargestellten ab.

Name:	**Wunderlich 1**
Verfeinerung:	3×3
Jahr der Entdeckung:	1973 (Veröffentlichung)
Anfangspunkt:	(0, 0)
Endpunkt:	(1, 1)
Anmerkung:	veränderte Peano-Kurve
Literatur:	[55]

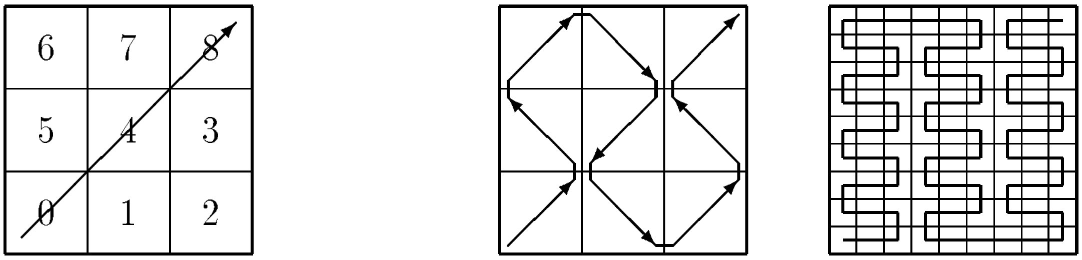

Abbildung 3.14: Rekursion und Visualisierung der Wunderlich 1-Kurve

Es gibt noch weitere Kurven, die auf ähnliche Art wie oben aus 3×3 -Verfeinerungen entstehen; hier ist nur beispielhaft eine solche dargestellt, die einen Peano-artigen Charakter hat. Davon gibt es, nach einer Aufstellung von Sagan, immerhin 272 verschiedene [51]. Wunderlich beschreibt außerdem zwei Kurven eines „Meandertyps“ [55], von denen nun eine folgt:

Name:	**Wunderlich 2**
Verfeinerung:	3×3
Jahr der Entdeckung:	1973 (Veröffentlichung)
Anfangspunkt:	(0, 0)
Endpunkt:	(1, 0)
Anmerkung:	Meander-Typ
Literatur:	[55]

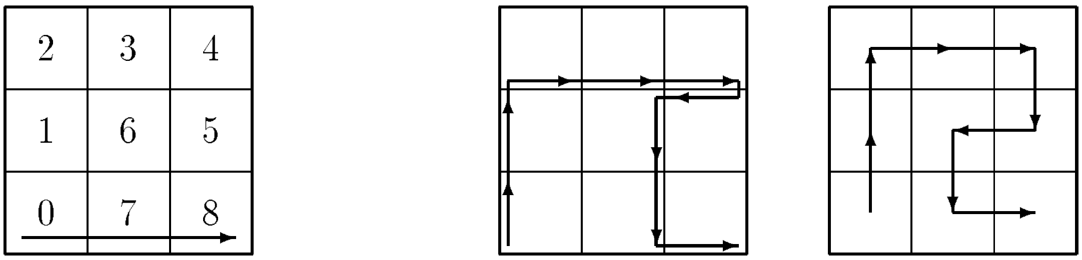

Abbildung 3.15: Rekursion und Visualisierung der Wunderlich 2-Kurve

3.3.3 Triangulierungen

Name:	**Sierpiński**
Verfeinerung:	2×2 - Triangulierung
Jahr der Entdeckung:	1912
Anfangspunkt:	(geschlossen)
Endpunkt:	(geschlossen)
Anmerkung:	gelegentlich auch Sierpiński-Knopp-Kurve genannt
Literatur:	[54]

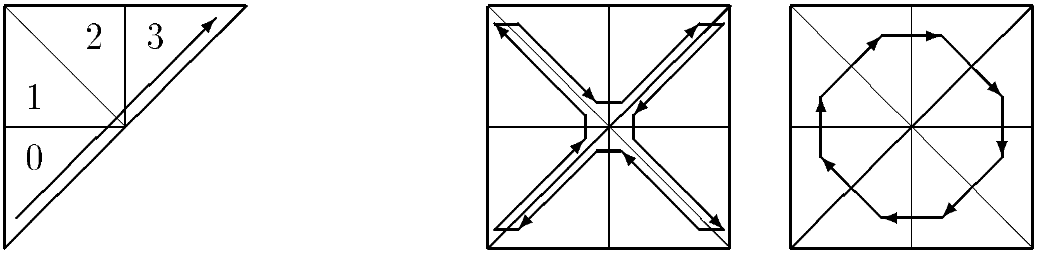

Abbildung 3.16: Rekursion und Visualisierung der Sierpiński-Kurve

Die Rekursion der Sierpiński-Kurve in Form einer 2×2-Triangulierung ist hier beispielhaft gewählt. Führt man in jedem Schritt lediglich eine Longest Edge Bisection durch (und erzeugt so nur 2 Unterdreiecke), entsteht natürlich die gleiche Kurve. Eine Longest Edge Bisection kann auch als eine Form einer hierarchischen Datenstruktur benutzt werden, die ihren Grundbereich rekursiv unterteilt (***Quadtree***, vgl. dazu [52]). Die Indexversion der Sierpiński-Kurve entspricht der einfachsten Datenstruktur, einem Binärbaum. Für eine adaptive Anwendung solcher Strukturen siehe z. B. [16].

Neben anderen Auffälligkeiten, die z. T. in den folgenden Kapiteln noch besprochen werden, ist festzuhalten, daß die Sierpiński-Kurve ursprünglich *analytisch* hergeleitet wurde. Um den Vergleich zur sehr einfachen geometrischen Darstellung ziehen zu können, sei die Vorschrift hier angegeben (vgl. [51]): Sei f eine Funktion einer reellen Variablen t, die folgenden Bedingungen genügt:

$$(1) \quad f(t) + f(t + 1/2) = 0 \text{ für alle } t \in \mathbb{R},$$

$$(2) \quad 2f(t/4) + f(t + 1/8) = 1 \text{ für alle } t \in [0, 1].$$

Dann ist für $0 \leq t \leq 1$ der Punkt $(f(t), f(t - 1/4))$ ein Element von $[-1, 1]^2$ (man beachte die notwendige Verschiebung). Läuft t durch das Einheitsintervall, wird das Quadrat vollständig ausgefüllt. Man kann sich leicht vorstellen, daß sowohl die Lösung dieser Funktionsgleichungen als auch die Darstellung des Ergebnisses deutlich schwieriger ist als die geometrische Herleitung.

Natürlich erhebt die Kurven-Aufzählung in diesem Unterkapitel keinen Anspruch auf Vollständigkeit. Insbesondere sind hier (mit Ausnahme der Lebesgue-Kurve) solche Kurven nicht erwähnt, die nicht aus dem vorgestellten Konstruktionsprinzip entstehen, wie z. B. die Schoenberg-Kurve [53]. Es wurden aber alle klassischen raumfüllenden Kurven aufgeführt, die für die Lösung des TSP von Belang sind.

3.4 Neuere Kurven und ihre Eigenschaften

In den weiteren Kapiteln dieser Arbeit geht es darum, *wie* eine raumfüllende Kurve das Einheitsquadrat ausfüllt, denn der Handlungsreisende soll dem Weg dieser Kurve folgen. Ohne bereits jetzt auf die Heuristik vorzugreifen, kann man sich vorstellen, daß der Handlungsreisende auf seiner Städtetour unterschiedliche Wege nehmen wird, wenn er sich unterschiedlichen Kurven anvertraut. Ein Ziel dieser Arbeit ist es daher, zu untersuchen, welche Kurven für einen solchen Zweck besser geeignet sind als andere.

Schon die klassischen Kurven, die im letzten Abschnitt vorgestellt wurden, unterscheiden sich in ihrer Hilfe für den Handlungsreisenden gewaltig. So werden wir sehen, daß die Sierpiński-Kurve den Reisenden schneller ans Ziel bringt als die Lebesgue-Kurve. Wie aber verhält es sich mit der Peano-Kurve und ihren 271 Verwandten? Für allgemeinere Aussagen ist es notwendig, andere als nur die klassischen Kurven zu untersuchen. Solche weiteren SFC werden in diesem Abschnitt in einer Auswahl vorgestellt.

Warum ausgerechnet die nun folgenden Kurven erstellt und untersucht wurden, wird in Kapitel fünf begründet. Außerdem ist es fraglos so, daß einige dieser Kurven nicht unbekannt sind, d. h. keineswegs im Zuge dieser Arbeit „erfunden“ wurden – angesichts des einfachen Konstruktionsprinzips wäre letzteres überraschend. Allerdings war es nicht möglich, für jede Kurve einen Namen und ein Geburtsjahr herauszufinden; daher werden die Kurven in diesem Abschnitt namenlos katalogisiert. Man findet ausschließlich Quadrat- und Dreieck-Verfeinerungen. Die Gründe hierfür werden in späteren Kapiteln behandelt.

Alle Kurven dieses Abschnitts sind tatsächlich Kurven, d. h. *stetige* Abbildungen, da die Bedingungen, die für die Stetigkeit aufgestellt wurden, erfüllt sind. In Kurven, die über einen Punkt laufen, sind diese Punkte graphisch besonders hervorgehoben: Hier berühren sich die zugrundeliegenden geometrischen Objekte nur in einem Punkt. Für die Stetigkeit hat dies keinen Nachteil (vgl. Satz 3.4).

3.4.1 Quadrat-Verfeinerungen

Alle regelmäßigen 2×2-Verfeinerungen, die *nicht* über einen Punkt laufen, sind durch die Hilbert- bzw. Moore-Kurve bereits abgehandelt, was ein kurzer Blick auf die möglichen Anordnungen des einzig möglichen Leitmotivs bestätigt. Die bereits angesprochenen 272 Mitglieder der 3×3-Verfeinerungs-Familie müssen hier nicht gesondert aufgeführt werden. Wie schon im letzten Abschnitt beschrieben, reicht die Drehung oder Spiegelung der Ausgangsanordnung oder des Leitmotivs bereits aus, um eine andere Kurve zu erhalten. Eine *substantielle* Änderung der Kurve bedeutet dies nicht. Statt dessen betrachten wir Beispiele für 4×4- und 5×5-Verfeinerungen.

Name:	**4 × 4 - 1**
Verfeinerung:	4×4
Anfangspunkt:	$(0, 0)$
Endpunkt:	$(\frac{1}{4}, \frac{1}{4})$

3	4	11	12
2	5	10	13
1	6	9	14
0	7	8	15

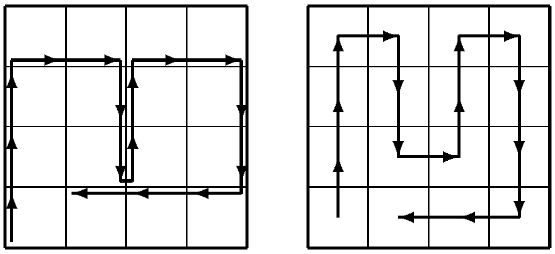

Abbildung 3.17: Rekursion und Visualisierung der 4×4 - 1-Kurve

Name:	**4 × 4 - 2**
Verfeinerung:	4×4
Anfangspunkt:	(0, 0)
Endpunkt:	(1, 0)

3	4	7	8
2	5	6	9
1	12	11	10
0	13	14	15

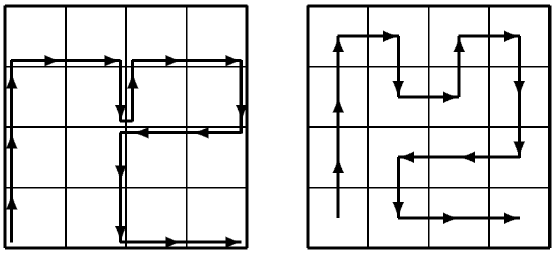

Abbildung 3.18: Rekursion und Visualisierung der 4×4 - 2-Kurve

Je mehr Verfeinerungsobjekte entstehen, desto größer ist die Möglichkeit, diese Objekte anzuordnen. Dabei wird es auch immer schwieriger, die Rekursionen anzugeben, da man eine Vielzahl von Leitmotiven erstellen und auch miteinander verbinden kann. Für die 5×5-Kurven, die in dieser Arbeit untersucht werden, wurden bis zu drei Leitmotive benutzt, die zusammen mit der jeweils gleichen Tiefe 2 in Abbildung 3.19 aufgeführt werden. Der Leser wird um Verständnis gebeten, daß die einzelnen Kurven nicht noch genauer aufgeschlüsselt werden, da dazu eine 25×25-Unterteilung notwendig wäre.

Name:	**5 × 5 - Kurven**
Verfeinerung:	5×5
Anfangspunkt:	(geschlossen)
Endpunkt:	(geschlossen)

10	11	14	15	24
9	12	13	16	23
8	7	6	17	22
1	2	5	18	21
0	3	4	19	20

24	21	20	17	16
23	22	19	18	15
2	3	8	9	14
1	4	7	10	13
0	5	6	11	12

0	1	20	21	24
3	2	19	22	23
4	5	12	18	17
7	6	11	13	16
8	9	10	14	15

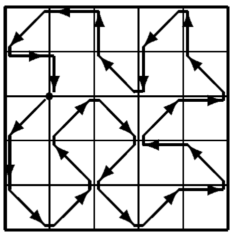

Abbildung 3.19: Rekursion und Leitmotive der 5×5 - Kurven

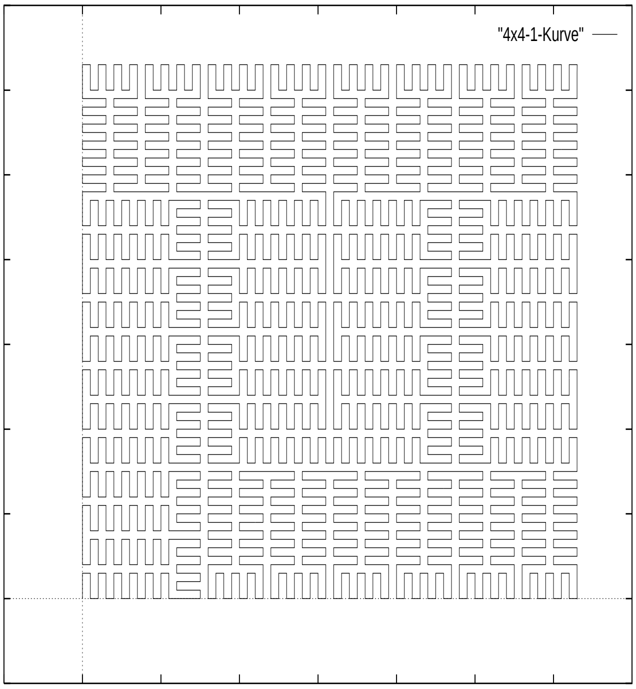

Abbildung 3.20: Darstellung der 4×4 - 1 - Kurve in Tiefe 4

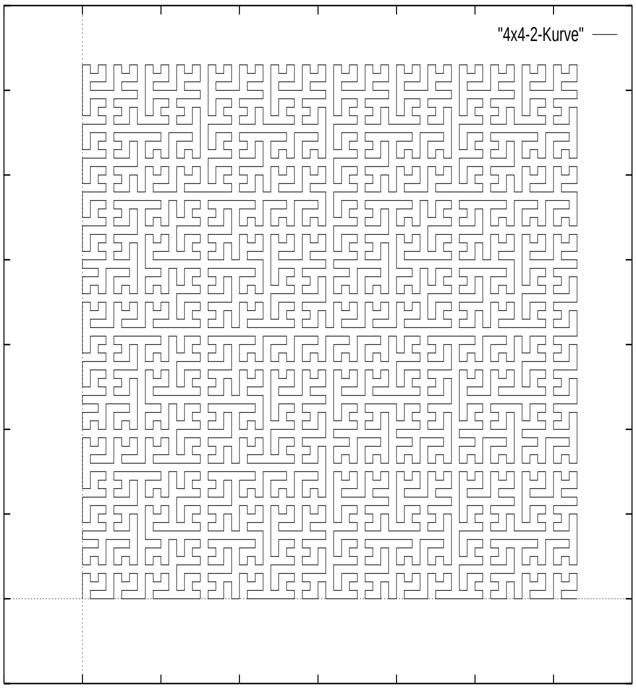

Abbildung 3.21: Darstellung der 4×4 - 2 - Kurve in Tiefe 4

Eine der einfachsten 5×5-Kurven ist die folgende, die der Struktur der Peano-Kurve folgt, allerdings auf eine Fünftelung des Einheitsquadrates zurückgreift. Das Leitmotiv nimmt dabei so wenig Richtungsänderungen wie möglich wahr; in y-Richtung folgen je fünf Unterobjekte nacheinander.

Name:	**5 × 5 - Peano-Kurve**
Verfeinerung:	5×5
Anfangspunkt:	(0, 0)
Endpunkt:	(1, 1)

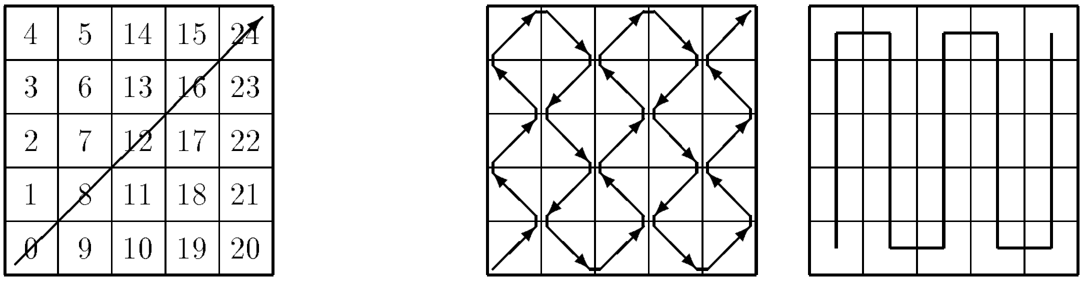

Abbildung 3.22: Rekursion und Visualisierung der 5×5 - Peano-Kurve

3.4.2 Triangulierungen

Wie schon früher angedeutet, fällt jede Triangulierung, die durch konsequente *Longest Edge Bisection* entsteht, auf die Sierpiński-Kurve zurück. Nun ist aber bereits unter den 2×2-Triangulierungen eine andere Möglichkeit der Verfeinerung (mit zwei Möglichkeiten der Anordnung) vorhanden:

Name:	**Dreieck 1**
Verfeinerung:	2×2 - Triangulierung
Anfangspunkt:	(geschlossen)
Endpunkt:	(geschlossen)

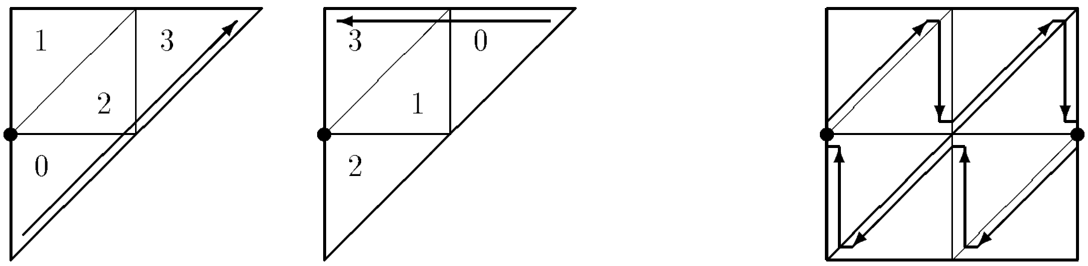

Abbildung 3.23: Leitmotive und Visualisierung der Dreieck 1-Kurve

Name:	**Dreieck 2**
Verfeinerung:	2×2 - Triangulierung
Anfangspunkt:	(geschlossen)
Endpunkt:	(geschlossen)

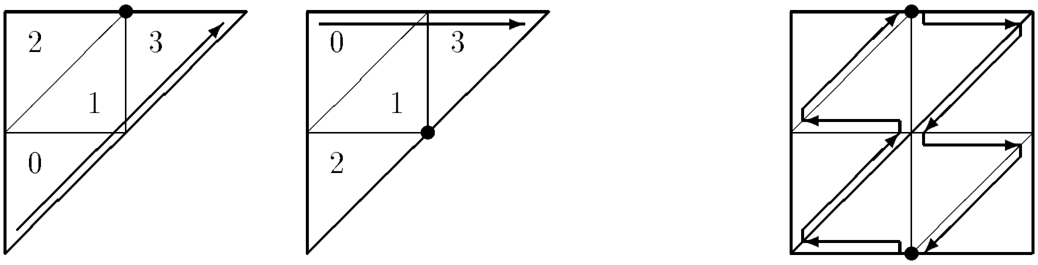

Abbildung 3.24: Leitmotive und Visualisierung der Dreieck 2-Kurve

Unter den 3×3-Triangulierungen ist es grundsätzlich nicht möglich, eine Kurve zu konstruieren, die *nicht* über einen Punkt läuft. Dies ist in Abbildung 3.25 leicht zu erkennen.

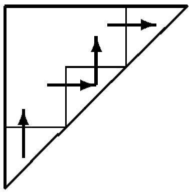

Abbildung 3.25: Beliebige 3×3 - Triangulierung in Tiefe 2

Die eingetragenen Dreiecke können nicht anders angeordnet werden, wenn die Verfeinerung eine Triangulierung sein soll. Die sich dadurch ergebenden Richtungen (o. B. d. A. startet die Kurve in (0, 0) und endet in (1, 1) – offensichtlich müssen Start- und Endpunkt an der Hypotenuse liegen, da sonst über einen Punkt gelaufen werden müßte) zeigen nach kurzer Untersuchung der möglichen Fälle die Behauptung. Der Beweis läßt sich auch über den Adjazenzgraphen der Verfeinerung führen. Zwei Beispiele solcher Kurven, die über einen Punkt laufen, schauen wir uns genauer an.

Name:	**3 × 3 - 1**
Verfeinerung:	3×3 - Triangulierung
Anfangspunkt:	(geschlossen)
Endpunkt:	(geschlossen)

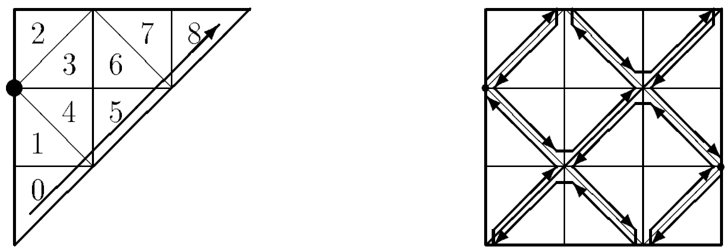

Abbildung 3.26: Rekursion und Visualisierung der 3×3 - 1-Kurve

Name:	**3 × 3 - 2**
Verfeinerung:	3×3 - Triangulierung
Anfangspunkt:	(geschlossen)
Endpunkt:	(geschlossen)

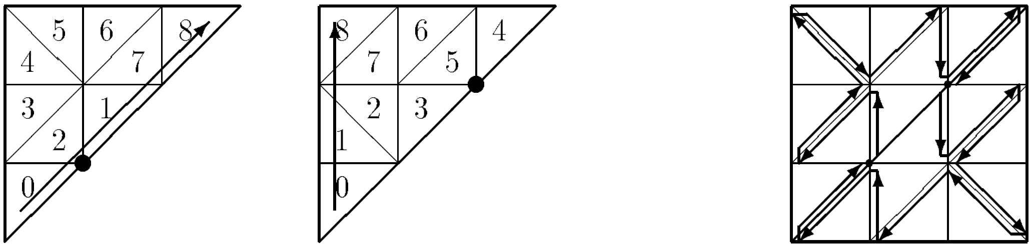

Abbildung 3.27: Leitmotive und Visualisierung der 3×3 - 2-Kurve

Um ein Beispiel für eine noch feinteiligere Verfeinerung zu geben, sei hier schließlich eine 6×6-Triangulierung vorgestellt.

Name:	**6 × 6**
Verfeinerung:	6×6 - Triangulierung
Anfangspunkt:	(geschlossen)
Endpunkt:	(geschlossen)

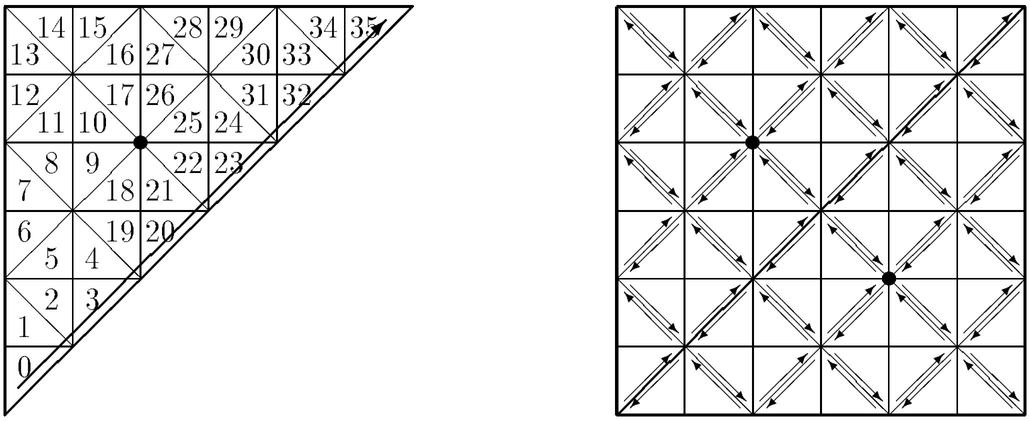

Abbildung 3.28: Rekursion und Visualisierung der 6×6 - Kurve

Neben den in diesem Unterkapitel vorgestellten SFC wurden noch weitere Kurven untersucht, die hier nicht aufgeführt werden. Die Übersichtlichkeit einer solchen Zusammenstellung hätte gelitten, außerdem sollten die dargestellten SFC eine ausreichende Grundlage für die Untersuchungen der nächsten Kapitel sein. Zusätzlich werden dem Leser im folgenden weitere SFC begegnen, deren Konstruktionsprinzip von dem bisher bekannten abweicht und die deshalb auf diesen Seiten noch nicht besprochen werden.

3.5 Zusammenfassung und Bemerkungen

Die Darstellungen in den vorangegangenen Unterkapiteln beleuchten nur einen kleinen Teil der Faszination einer raumfüllenden Kurve. So wurde nur ein einziges Konstruktionsprinzip ausdrücklich vorgestellt, nämlich die Angabe einer regelmäßigen Verfeinerung und einer stetigen Anordnung. Viele andere Prinzipien wurden nur angerissen oder noch gar nicht erwähnt:

- Die Lebesgue- und die Schoenberg-Kurven gehen auf Eigenschaften der Cantormenge zurück - ein vielleicht unabsichtlicher Tribut an einen geistigen Vater der Forschung [51].
- Niedermeier, Reinhardt und Sanders haben in den vergangenen Jahren andere Index-Schemata untersucht, die nicht so leicht rekursiv darstellbar sind. Auf den „Tübinger“ *H-Index* wird im fünften Kapitel noch eingegangen [41]. Dieser *H-Index* ist nicht zu verwechseln mit dem *Hilbert-Index* [1], der lediglich die Indizierung der Hilbert-Kurve wie in Kapitel drei beschreibt.
- Steinhaus zeigte 1936, daß stochastisch unabhängige Funktionen eine raumfüllende Kurve definieren können, solange sie stetig und nicht konstant sind (zitiert in [51]).

Auch mit dieser Aufstellung ist das Gebiet keineswegs abgedeckt. Es ist problemlos möglich, einige der obigen Gedanken auf den n-dimensionalen Raum auszudehnen. Gerade das hier näher betrachtete Konstruktionsprinzip kann man wörtlich übertragen. Es entstehen neue Freiheiten, da Kurven z. B. im $\mathbb{R}^3$ über Seiten, Kanten und Ecken laufen können. Dafür werden die geometrischen Untersuchungen, welche Anordnung eine Stetigkeit erlaubt, immer schwieriger, insbesondere bei komplizierteren Verfeinerungsobjekten. In manchen Anwendungen ist eine höchstdimensionale Berührung der Unterobjekte erforderlich – damit wird die Frage der Anordnung eine Wissenschaft für sich. So kann man im $\mathbb{R}^2$ Eigenschaften der Sierpiński-Kurve für Datenkompressionen benutzen (vgl. [16]), im $\mathbb{R}^3$ funktioniert dies nicht mehr.

Auch nicht-surjektive Kurven können als raumfüllend angesehen werden, wenn man „intuitiv fühlt, daß [sie] die Ebene dichter in Anspruch nehmen“[4] als andere Kurven, d. h. einen Teil des Raumes zumindest optisch ausfüllen. Man sagt dann, solche Kurven haben eine „gebrochene“ Dimension, weshalb sie als *Fraktale* bezeichnet werden. Sie sind meist sehr leicht mit dem Computer zu erstellen. Neben ihrem faszinierenden Aussehen (siehe z. B. [35]) sind sie in der Theorie der Ähnlichkeitstransformationen von Interesse. Auch für unterhaltsame Aspekte sind sie gut: Peitgen, Jürgens und Saupe haben ausgerechnet, daß die britische Küstenlinie eine „gebrochene“ Dimension von 1,31 haben soll (zitiert in [38]).

Ein komplizierter, aber vielversprechender Ansatz, Fraktale zu erzeugen, wird von Drakopoulos und Dalla in [14] vorgestellt: Es ist möglich, Fraktale durch Projektionen des Graphen von fraktalen Interpolations-Funktionen in höher-dimensionalen Räumen zu erhalten. Dieser Ansatz könnte Bedeutung bei der Suche nach noch unbekannten Fraktalen finden.

[4] Übersetzt aus: [51], Seite 156.

Kapitel 4

Die SFC-Heuristik

Wie passen raumfüllende Kurven und das Euklidische Travelling Salesman Problem zusammen? Bartholdi und Platzman entwickelten eine Heuristik zur Lösung des ETSP, die auf raumfüllende Kurven zurückgreift. Die folgenden Ausführungen gehen auf einige ihrer Veröffentlichungen aus den Jahren 1982 bis 1989 zurück ([5], [6], [46]).

Das Vorgehen ist leicht verständlich: Da eine raumfüllende Kurve Punkte im Einheitsquadrat in einer gewissen Reihenfolge abläuft, kann der Handlungsreisende die Städte in der gleichen Reihenfolge besuchen. Das vorliegende Kapitel beschreibt diese Heuristik genauer und ordnet bisherige Ergebnisse ein. Außerdem werden Testfälle vorgestellt, anhand derer die Performance verschiedener SFC untersucht werden kann.

4.1 Die Heuristik von Bartholdi und Platzman

1982 stellten Bartholdi und Platzman jene TSP-Heuristik vor, die in dieser Arbeit genauer betrachtet wird: die SFC-Heuristik [5]. Der zugrundeliegende Gedankengang ist bestechend einfach. Punkte, die im Einheitsintervall nah beieinander liegen, werden auch nach einer Transformation durch eine SFC auf dieser Kurve nah beieinander liegen. Umgekehrt verhält es sich fast genauso: Die Urbilder zu Punkten mit geringem Abstand auf der Kurve haben meist einen geringen Abstand im Einheitsintervall.

Ein Blick auf die Visualisierungen einiger raumfüllender Kurven in Kapitel drei macht dies deutlich – und zeigt auch Gegenbeispiele auf: Betrachtet man die Hilbert-Kurve in Abbildung 3.5, so haben die Urbilder der Punkte $(\frac{3}{10}, \frac{1}{2}\text{-}\varepsilon)$ und $(\frac{3}{10}, \frac{1}{2}+\varepsilon)$ auf dem Einheitsintervall einen Abstand von mindestens 1/4, während der Abstand im Einheitsquadrat 2ε ist. Dennoch ist die Feststellung erlaubt, daß die Transformation durch eine SFC in beiden Richtungen eine Nähebeziehung im allgemeinen erhält. Und die Nähe zwischen Städten ist das, was den Handlungsreisenden interessiert. Wenn jede Stadt mit ihren beiden nächsten Nachbarn verbunden ist und dabei eine Tour ensteht, ist diese Tour nämlich optimal[1].

Wenn man nun zweidimensionale Daten, die es zu ordnen gilt, auf das Einheitsintervall reduzieren und dabei Nähebeziehungen der Daten in gewisser Weise bewahren kann, fällt dem Anwender eine TSP-Lösung praktisch in den Schoß. Und so sieht die Heuristik von Bartholdi und Platzman genau aus:

[1] Dies wird durch die Nearest-Neighbour-Heuristik im allgemeinen *nicht* erreicht.

SFC-Heuristik

1. Verschiebe die Daten in das Einheitsquadrat.
2. Ermittele zu jedem Daten-Punkt ein Urbild unter einer raumfüllenden Kurve im Einheitsintervall.
3. Sortiere die Punkte gemäß der aufsteigenden Reihenfolge ihrer Urbilder im Einheitsintervall.
4. Berechne die Länge der Tour.

Man kann eine raumfüllende Kurve also als die Tour eines Handlungsreisenden ansehen, der jeden Punkt des Einheitsquadrates besucht. Durch die Heuristik wird die Tour auf die benötigten Punkte reduziert. Dabei wird lediglich die Reihenfolge durch die Kurve angegeben; der eigentliche Weg und die Längenberechnung selbst haben mit der Kurve nichts mehr zu tun.

Für die praktische Umsetzung muß man auf die obige Aufstellung näher eingehen, denn es werden nicht wirklich Urbilder für die Punkte berechnet. Dazu müßte die Kurve nämlich bis in die Tiefe ∞ hinein errechnet werden. Statt dessen wählt man eine Tiefe m, bis zu der die Kurve berechnet wird, und legt die indizierte Kurve schablonenartig über die Daten. Jeder zu untersuchende Punkt liegt nun in mindestens einem Dreieck/Quadrat etc., das mit einem Index beschriftet ist. Dieser Index, der seinerseits zu einem Intervall im Einheitsintervall gehört, wird dem Punkt zugeordnet. Es ist also nicht notwendig, das Urbild explizit zu berechnen, da die Sortierung der Indizes den gleichen Effekt wie die Sortierung der Urbilder hat.

Liegt ein Punkt am Rand eines Verfeinerungsobjektes, können mehrere Indizes zugeordnet werden. Der Algorithmus muß in diesem Fall entscheiden, welcher Rand zu welchem Objekt gehört, damit genau ein Index gewählt wird. Je nach gewählter Tiefe ist es außerdem gut möglich, daß mehrere Punkte den gleichen Index tragen. Das Sortieren zwischen diesen Punkten findet dann randomisiert statt. Eine Alternative zu dieser Zufälligkeit besteht darin, die Tiefe m zu erhöhen, denn durch die Verfeinerung sinkt die Wahrscheinlichkeit gleicher Indizierung. Für steigende Städteanzahlen ist es logischerweise notwendig, höhere Tiefen vorzuschreiben. Auswirkungen auf die Geschwindigkeit des Algorithmus und Hinweise zur Handhabung werden in Kapitel sechs noch genauer beschrieben.

In der Praxis kann die SFC-Heuristik also z. B. so umgesetzt werden:

1. Verschiebe die Daten in das Einheitsquadrat.
2. (a) Wähle eine Tiefe m.
   (b) Berechne für jeden Punkt den zugehörigen Index der SFC in Tiefe m.
3. (a) Ordne die Punkte gemäß der zugehörigen Indizes, wobei die Sortierung bei gleichen Indizes zufällig erfolgt.
   (b) Erhöhe eventuell die Tiefe und gehe in diesem Fall zurück zu 2(b).
4. Berechne die Länge der Tour.

Abbildung 4.1 verdeutlicht die Heuristik für ein erfundenes 12-Städte-Problem, wenn die Hilbert-Kurve in Tiefe 3 eingesetzt wird.

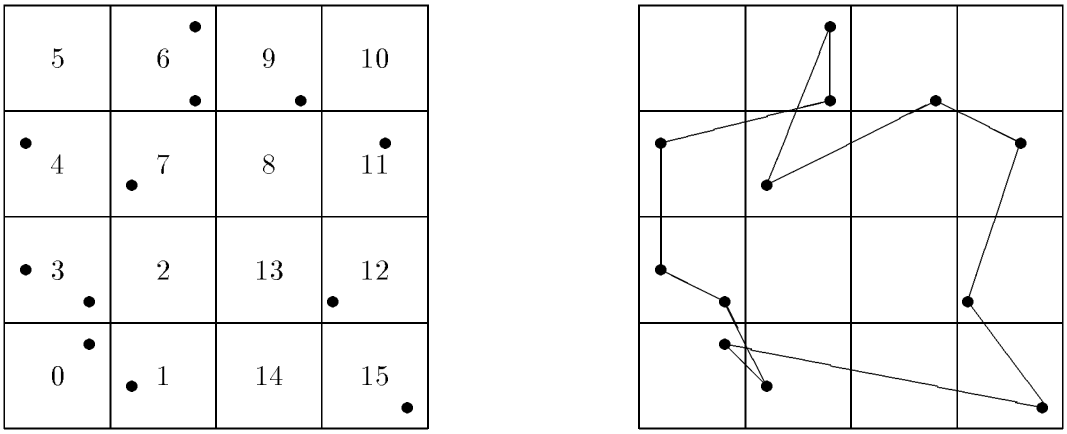

Abbildung 4.1: Die Nutzung der offenen Hilbert-Kurve in Tiefe 3 zur Lösung eines 12-Städte-Problems

Würde man um eine Stufe verfeinern, würde sich in Quadrat 3 nichts an der Ordnung ändern, wohingegen die Punkte in Quadrat 6 ihre Anordnung tauschen und die Länge der Tour verringern würden. Gleichzeitig kann man erkennen, daß die Hilbert-Kurve als nicht-geschlossene SFC Nachteile gegenüber geschlossenen Kurven hat, da das Schließen der Tour eine unnötig schlechte Kante erfordert. Für eine wachsende Problemgröße ist dies aber nicht mehr von Belang.

Zu dem beschriebenen Vorgehen ist anzumerken, daß Bartholdi und Platzman selbst in allen drei zitierten Veröffentlichungen nicht mit Indizes arbeiten, sondern tatsächlich Positionen im Einheitsintervall berechnen. Diesem Verfahren wird hier nicht gefolgt. Durch die Vorgabe einer endlichen Schrittzahl erhalten Bartholdi und Platzman nämlich anstelle einer Position auch nur ein Intervall (genauer: eine 2^{m-1}-stellige Binärzahl, die für ein Intervall steht), welches seinerseits einem Index entspricht. Das Ergebnis beider Verfahrensweisen ist daher gleich; das hier vorgestellte Verfahren sollte aber stringenter sein und geht außerdem auf das Konstruktionsprinzip einer raumfüllenden Kurve ein.

Man sieht sofort, daß die Heuristik problemunabhängig arbeitet. Dies kann als Vor- oder als Nachteil angesehen werden. So ist einerseits ein einmal geschriebener Code für jedes TSP anwendbar. Andererseits besteht kaum eine Möglichkeit des „Feintunings“ von Parametern, welches eine Verbesserung der Lösung ermöglicht.

Die Laufzeit der Heuristik beträgt $\mathcal{O}(n \cdot \log n)$. Ist die benutzte SFC durch das in Kapitel drei betrachtete Konstruktionsprinzip entstanden, so kann der zweite Schritt der Heuristik mit linearem Aufwand durchgeführt werden. Für den Sortierschritt ist dann ein Aufwand von $\mathcal{O}(n \cdot \log n)$ erforderlich. Die Gesamtlaufzeit kann z. B. durch Parallelisierung verbessert werden, wie in Kapitel sechs zu sehen ist.

Es ist ebenfalls sofort ersichtlich, daß neu hinzukommende Punkte leicht berücksichtigt werden können, ohne daß die Heuristik eine komplett neue Tour berechnen müßte: Jeder neue Punkt erhält seinen Index und wird an der richtigen Stelle in die bestehende Tour eingefügt. Dazu ist nur ein Aufwand von $\mathcal{O}(\log n)$ erforderlich.

Von besonderem Interesse ist natürlich die Performance. Diese ist abhängig von der benutzten raumfüllenden Kurve. Die Untersuchung der Auswirkungen der Nutzung verschiedener SFC ist ein Ziel dieser Arbeit und wird in Kapitel fünf ausführlich behandelt. Im nächsten Unterkapitel soll zunächst geklärt werden, was unter dem Begriff der Performance genau zu verstehen ist und wie eine Performance gemessen wird.

4.2 Testfälle und Randominstanzen

Um die Güte einer TSP-Heuristik zu messen, gibt es keine bessere Möglichkeit, als diese auf eine Instanz anzuwenden und die Lösung zu analysieren – theoretische Aussagen sind meist nur als grobe Abschätzungen möglich (siehe z. B. Kapitel 2.2.3). In diesem Unterkapitel werden reale Testfälle sowie Randominstanzen vorgestellt, um in diesem Zusammenhang den Begriff der Performance einer Heuristik zu erklären.

4.2.1 Reale Testfälle aus der TSP-Library

Da eine Heuristik auf einer speziellen Instanz vielleicht nur zufällig schlechter oder besser als andere abschneidet, liegt es nahe, die praktischen Versuche mit einer Heuristik auf mehrere Instanzen auszudehnen. Eine geeignete Menge von Testfällen muß dabei gewisse Kriterien erfüllen:

- Es sollte darauf geachtet werden, daß mehrere Problembereiche ausgeschöpft werden; zehn reine Drilling-Probleme sind sicher nicht aussagekräftig.
- Die Städteanzahl sollte ein breites Spektrum abdecken.
- Optimale oder fast-optimale Lösungen sollten als Vergleichswerte bekannt sein.
- Die Daten sollten leicht zugänglich sein.
- Bei denjenigen Heuristiken, in denen die Wahl des Startknotens oder der Startlösung Einfluß auf das Ergebnis hat, müssen mehrere Durchläufe durchgeführt und gemittelt werden[2].

Mit seiner Library TSPLIB [48] hat Reinelt den Suchenden in dieser Materie einen großen Dienst erwiesen. In dieser im Internet zugänglichen Library ist eine Vielzahl von praktischen Instanzen von 14 bis 85.900 Städten abgelegt, für die optimale Lösungen oder sehr gute Lower Bounds bekannt sind und auf deren Daten leicht zugegriffen werden kann. Für seine eigenen Testläufe der Heuristiken, die in Kapitel zwei erwähnt werden, hat Reinelt einen Teil der o. g. Instanzen benutzt.

In dieser Arbeit wird ebenfalls auf die TSPLIB zurückgegriffen. Da die SFC-Heuristik gerade bei großen Datenmengen ihre Laufzeit-Vorteile ausspielt, wurden als reale Testfälle für die vorliegende Arbeit die 20 größten Euklidischen TSPs aus der TSPLIB gewählt. Diese seien hier kurz aufgeführt, da sich die Library ständig vergrößert und eine Überprüfbarkeit hergestellt werden soll (die Zahlen geben die Städteanzahlen an):

[2]Dieser Punkt trifft auf die SFC-Heuristik nicht zu, wird aber bei den Untersuchungen in Kapitel zwei beachtet.

pla85900	usa13509	fnl4461	u2152
pla33810	rl11849	fl3795	d2103
d18512	pla7397	pcb3038	rl1889
d15112	rl5934	pr2392	u1817
brd14051	rl5915	u2319	vm1748

Tabelle 4.1: Die 20 größten ETSP aus der TSPLIB 95

Einige dieser Instanzen (d18512, d15112, brd14051, usa13509, fnl4461) bilden Städte eines Landes oder die Grenze eines Landes ab, manche der kleineren Instanzen (u2152, u2319, pcb3038) sind Drilling-Probleme, wieder andere (z. B. fl3795) spiegeln Cluster von Punkten wider. Eine Bandbreite der Probleme ist also gesichert. Die mit „pla“ bezeichneten Instanzen sind mit einer besonderen Distanzfunktion belegt: Die euklidischen Abstände werden hier immer nach *oben* gerundet.

Nur für die fünf größten Instanzen ist bisher keine optimale Lösung bekannt. Die als Vergleichswert herangezogenen Lower Bounds weichen aber höchstens um 0,08 % vom Optimum ab. An dieser Stelle sei noch einmal erwähnt, daß die Instanz usa13509 das bisher größte gelöste reale TSP ist.

Die Koordinaten der Punkte in den genannten Instanzen sind meist als vier- bis fünfstellige Zahlen angegeben und liegen in einem Textfile vor. Die euklidische Entfernung zwischen zwei Städten wird auf ganze Zahlen gerundet. Eigene Untersuchungen haben bestätigt, daß die entstehenden Rundungsfehler nur geringen Einfluß auf die tatsächliche Länge der Tour des Handlungsreisenden haben. Die Rundung findet statt, da Reinelt in seinen Ausführungen zur TSPLIB dieses Vorgehen vorschlägt [48].

Die ausgewählten und dargestellten Instanzen erfüllen also alle vier für das ETSP relevanten Bedingungen und werden daher als Testfälle zur Analyse der SFC-Heuristik benutzt.

Unter der ***Performance*** einer Heuristik versteht man nun den relativen Abstand der Lösung, die durch die Heuristik generiert wird, zu einer optimalen Lösung beziehungsweise der besten bekannten unteren Schranke. Beschreiben w_H die Länge der Lösung aus der Heuristik und w_{opt} die Länge einer optimalen Lösung, ergibt sich die Performance als Prozentwert zu $(w_H/w_{opt} - 1) \cdot 100$.

Die ***durchschnittliche Performance*** einer Heuristik, die hier von Interesse ist, beschreibt den Mittelwert der Performances auf den ausgewählten Testfällen. Es muß darauf hingewiesen werden, daß Reinelt in seinen in Kapitel zwei zitierten Analysen eine andere Testmenge untersucht hat. Die Heuristiken aus Kapitel zwei könnten auf der hier benutzten Testmenge leicht unterschiedliche Daten liefern.

4.2.2 Randominstanzen

Praktische Testfälle sind zweifelsohne die interessanteren Fälle; der Städteanzahl ist jedoch eine Grenze gesetzt. Größere Probleme treten seltener auf, und bereits die Datenspeicherung und -präsentation stellt handelsübliche Rechner vor Probleme. Auch wird es für steigende Städteanzahlen immer schwieriger, gute Lower Bounds oder gar optimale Lösungen anzugeben. Will man nun die Performance einer Heuristik für größere Städteanzahlen untersuchen – dies ist z. B. im Hinblick auf Laufzeiten sehr interessant – muß man sich auf künstliche, d. h. erfundene Datensätze beschränken. Die einfachste Art, solche Datensätze zu erzeugen, ist die Randomisierung.

Ein zufälliger Datensatz, der in jedem Lauf des Algorithmus neu entsteht, kann natürlich nicht auf sehr gute Lower Bounds oder gar eine optimale Lösung untersucht werden – schon bei kleinen Datenmengen würde dies Stunden dauern. Für gleichverteilte, voneinander unabhängig gewählte Punkte in $[0,1]^2$ gibt es jedoch Hinweise zur Abschätzung der Länge einer optimalen Tour. Dank der Gleichmäßigkeit und Zufälligkeit der Punktverteilung hat nämlich jede optimale Tour zu *jeder* Instanz von n Städten fast die gleiche Länge - unabhängig von der Instanz, nur abhängig von der Städteanzahl n.

So zeigten Beardwood, Halton und Hammersley, daß eine Konstante C existiert, so daß

$$\lim_{n\to\infty} \frac{w_{opt}}{\sqrt{n}} = C \tag{4.1}$$

fast sicher gilt, wobei w_{opt} die Länge einer optimalen Tour bezeichnet [7]. Stein gab 1977 eine Abschätzung von $C \approx 0{,}765$ an (zitiert in [42]). Diese Abschätzung ist seit einigen Jahren umstritten, wird aber noch in vielen Veröffentlichungen gehalten. Dagegen vermuten Johnson, McGeogh und Rothberg einen Wert von $0{,}7124 \pm 0{,}0002$ [27], der durch ähnliche Resultate von Percus und Martin [44] unterstrichen wird. Auch Arbeiten von Norman und Moscato sowie von Applegate, Cook und Rohe legen einen Wert von unter 0,715 nahe ([2] und [42]). Wichtige und neueste Erkenntnisse zu diesem Thema können auf einer Webseite von Finch eingesehen werden [15].

Ohne einen direkten theoretischen Beitrag leisten zu können, soll auch diese Arbeit die Auffassung unterstützen, daß der Wert von 0,765 unhaltbar ist und deutlich niedriger angesetzt werden muß, wie im nächsten Kapitel aufgezeigt wird.

Wichtig für den Vergleich von Performances ist aber lediglich die Konstanz von C, nicht der Wert. Daher werden in Kapitel fünf solche Punktmengen als Instanzen untersucht, die zufällig, uniform und unabhängig auf $[0,1]^2$ verteilt werden. Dabei werden Fälle von 10.000 bis 1.400.000 Städten betrachtet. Um den Einfluß der Zufälligkeit zu dämpfen, werden für jede Städteanzahl unter 500.000 Städten zehn Durchläufe mit jeweils neuen zufälligen Instanzen gemittelt. Ab 500.000 Städten werden für eine gute Mittelung nur noch fünf solcher Durchläufe benötigt. Für steigende Städteanzahlen ist die Varianz der SFC-Heuristik für verschiedene Instanzen bei gleicher Städteanzahl nur noch sehr gering.

Die Performance einer Heuristik auf randomisierten Instanzen wird hier bewußt nicht definiert, da sich je nach Städteanzahl n die Ergebnisse unterscheiden. Die Fokussierung nur auf die größte Anzahl von 1.400.000 Städten ist nicht gerechtfertigt, wie z.B. bei der Peano-Kurve ersichtlich ist (siehe 5.2.2). Aussagen zur Performance auf randomisierten Fällen gehen auf kurvenabhängige Mittelungen eines *Performance-Indexes* zurück, der im nächsten Kapitel eingeführt wird.

4.3 Bekannte Ergebnisse und Einordnung

Wie gut ist die SFC-Heuristik? Bartholdi und Platzman sind dieser Frage theoretisch nachgegangen [46], in Reinelts TSP-Kompendium findet man praktische Versuche [47]. Im Vergleich zu anderen Heuristiken schneidet sie schlecht ab, was mit der sehr schnellen Laufzeit erklärt werden kann.

Wenden wir uns zunächst den theoretischen Ergebnissen zu. Viele Aussagen zu der betrachteten Heuristik sind abhängig von der benutzten raumfüllenden Kurve. Daher sei angemerkt, daß die beiden Autoren grundsätzlich die Sierpiński-Kurve benutzen, auch wenn sie dies nicht direkt angeben: In ihrer ersten Veröffentlichung zu diesem Thema bemerken sie, daß sie eine Kurve benutzen, die der Sierpiński-Kurve „ähnelt“ [5], in den weiteren Ausführungen fällt der Name gar nicht mehr. Und auch wenn insgesamt drei verschiedene Konstruktionen einer Kurve benutzt werden, ergibt sich in allen Fällen genau die Sierpiński-Kurve. Dabei ist die Konstruktion in [46] als gelungen anzusehen, da hier die Dreiecks-Struktur umgesetzt wird, wohingegen die Kurve in den Veröffentlichungen zuvor in Quadraten dargestellt wird und sich dem Leser schwieriger erschließt.

Eine *Worst-Case*-Abschätzung der Performance der SFC-Heuristik ist möglich. So weicht eine heuristische Lösung höchstens um den Faktor $\mathcal{O}(\log n)$ von einer optimalen Lösung ab [5]. Die Hoffnung, daß diese Grenze unterboten werden kann, wurde durch Bertsimas und Grigni zerschlagen, die ein Beispiel konstruierten, für das die SFC-Heuristik mit der Sierpiński-Kurve tatsächlich um den genannten Faktor abweicht [8]. Dies ist nicht für alle SFC bewiesen; Bertsimas und Grigni gehen aber davon aus, daß ein solcher Beweis für jede raumfüllende Kurve möglich ist. In den praktischen Untersuchungen werden wir sehen, daß die SFC-Heuristik im allgemeinen deutlich besser abschneidet, als es die Abschätzung vermuten läßt.

Für Randominstanzen ist Bartholdi und Platzman der Nachweis gelungen, daß für die SFC-Heuristik eine ähnliche Grenzabschätzung wie in Gleichung (4.1) fast sicher möglich ist [46]. Der Unterschied besteht lediglich darin, daß ein Limes inferior mit dazugehöriger Konstante C^- und ein Limes superior mit einer Konstante C^+ existieren, so daß für $n \to \infty$ gilt

$$C^- \leq \frac{w_H}{\sqrt{n}} \leq C^+, \tag{4.2}$$

wobei w_H die (kurvenabhängige) Länge einer Lösung angibt, die durch die SFC-Heuristik erhalten wird.

Die Bandbreite zwischen C^- und C^+ ist in der Regel sehr gering. Im Falle der Sierpiński-Kurve liegen C^- und C^+ in einem Intervall von $0,956 \pm 0,001$, unabhängig von der Dichtefunktion in der Randomisierung [6]. Diese Abschätzung, die bereits 1982 genannt wurde, wird durch die rechnerischen Ergebnisse dieser Arbeit gestützt. Die anderen von Bartholdi und Platzman genannten, aber leider unbewiesenen Abschätzungen für andere Kurven werden durch die vorliegende Arbeit nicht bestätigt. So erwähnen sie in [6] für die geschlossene Hilbert-Kurve einen C-Wert (ohne Angabe, ob es sich um C^- oder C^+ handelt) von 1,12 und für die Hilbert-Kurve einen C-Wert von $1,10 + 1/\sqrt{n}$ sowie für die Peano-Kurve einen C-Wert von $1,12 + \sqrt{2/n}$ [6]. Die in dieser Arbeit errechneten Werte liegen nicht so weit von der Sierpiński-Kurve entfernt. In Kapitel fünf ergeben sich für die genannten Kurven mittlere Abschätzungen von 0,979, 0,98 und 1,045. Solche Werte, auch wenn sie nur rechnerisch hergeleitet sind, erscheinen realistischer, da sonst ein nicht erklärbarer Widerspruch zur Performance in praktischen Fällen entsteht.

Der Aussage, daß die Sierpiński-Kurve die mit Abstand beste Kurve sein dürfte [6], kann daher mit *dieser* Begründung nicht gefolgt werden. Die ebenfalls in [6] vorgebrachten Argumente, die Sierpiński-Kurve weise eine bessere Symmetrie und

Isotropie als andere Kurven auf, sind ohnehin nicht besonders stichhaltig.

Analysen mit praktischen Testfällen aus der TSPLIB wurden von Reinelt durchgeführt [47]. Er erhält eine durchschnittliche Performance von 43,9 % für die SFC-Heuristik. Dabei gibt er ausdrücklich an, die von Bartholdi und Platzman benutzte Kurve implementiert zu haben. Es ist dem Autor der vorliegenden Arbeit auch nach langer Untersuchung unerklärlich, warum die Werte von Reinelt bei manchen Instanzen so deutlich von den in dieser Arbeit mit der Sierpiński-Kurve erhaltenen Daten abweichen (siehe auch Kapitel fünf). *Im Schnitt* liegen allerdings beide Performance-Angaben nah beieinander und unterscheiden sich hauptsächlich durch die benutzten Testfälle, da in dieser Arbeit mehr Probleme mit größerer Städteanzahl untersucht wurden.

Zusammenfassend läßt sich feststellen, daß die SFC-Heuristik mit der Sierpiński-Kurve in randomisierten Fällen eine Performance von knapp 35 % erreicht, wenn man einen C-Wert von 0,7124 zugrundelegt. In praktischen Fällen liegt eine Performance von rund 42 % vor. Dieser Zusammenhang alleine verdeutlicht, warum in Gleichung (4.1) eine Abschätzung für C von 0,765 nicht haltbar ist. In diesem Fall würde die Sierpiński-Kurve nur um 25 % vom Optimum abweichen. Im Vergleich zu anderen Heuristiken schneidet die SFC-Heuristik damit schlecht ab, was aber nicht verwundern darf, da sie mit Abstand die schnellste Heuristik darstellt.

Kapitel 5

Numerische Analyse der SFC-Heuristik

Das Ergebnis der SFC-Heuristik variiert unter dem Einsatz verschiedener Kurven. Bartholdi und Platzman haben in ihren Veröffentlichungen nur die Sierpiński-Kurve benutzt. Das vorliegende Kapitel geht der Frage nach, was geschieht, wenn in den Rahmenalgorithmus eine andere Kurve eingepaßt wird. Es befaßt sich mit der Untersuchung der Performance verschiedener raumfüllender Kurven. Dazu werden zunächst die bekannten klassischen Kurven eingesetzt. Aus deren Ergebnissen werden Rückschlüsse gezogen, wie weitere Kurven konstruiert werden können, die anschließend analysiert werden. Des weiteren werden Kurven betrachtet, die aus anderen Konstruktionsprinzipien entstehen, wie die Moscato-Norman-Kurven, der H-Index oder Fraktale. Abschließend faßt ein Bewertungsschema die Ergebnisse zusammen.

5.1 Allgemeines

Als Einleitung zu den folgenden Untersuchungen müssen zunächst einige allgemeine Aussagen gemacht werden:

Schon das Einlesen der Daten bedarf eines Kommentars. Denn bereits an dieser Stelle kann das Ergebnis beeinflußt werden. Der hier genutzte Algorithmus ist in dem Sinne starr programmiert, daß zu jeder Kurve ein bestimmtes Quadrat das Startquadrat ist. So ist z. B. für die Hilbert-Kurve immer jenes Quadrat das Startquadrat, welches den Ursprung enthält – willkürlich. Stellt man sich die raumfüllende Kurve als Indexschablone vor, die man auf die Daten legt, kann man aber auf den Gedanken kommen, die gesamte Schablone drehen oder spiegeln zu wollen, um so die Indexzuteilung und damit gegebenenfalls die Reihenfolge zu ändern.

Möchte man nun eine Kurve nicht in der Form benutzen, wie sie in Kapitel drei vorgestellt wird, ist dies umsetzbar, ohne die Kurve umzuprogrammieren. Dazu müssen lediglich die Daten manipuliert werden. So kann man z. B. die x-Koordinaten in den y-Vektor einlesen und umgekehrt oder weitere Drehungen und Spiegelungen vornehmen. Anstatt die Schablone zu drehen, dreht man also die Daten. Die Aufgabenstellung wird dadurch nicht verändert; das Ergebnis ist und bleibt eine zulässige Tour durch die vorgegebenen Daten.

Ist die benutzte Kurve symmetrisch bezüglich vieler möglicher Dreh- und Spiegelachsen, so ändert sich das Ergebnis nicht. Ein Beispiel für eine solche Kurve ist die Sierpiński-Kurve. Dagegen liefern andere SFC deutlich unterschiedliche Ergebnisse, wenn man die Anordnung der Daten ändert. Eine einfache Drehung der Moore-Kurve um 90° ändert die Performance in Einzelfällen um bis zu fünf Prozentpunkte und im Schnitt um immerhin 1,2 Prozentpunkte. Auf randomisierten Instanzen dagegen können Drehungen oder Spiegelungen aufgrund der Gleichverteilung der Punkte keine Auswirkung haben. Diese Instanzen müssen daher für die Bewertung einer Kurve als aussagekräftiger angesehen werden.

Die Performanceangaben der folgenden Unterkapitel beziehen sich auf die Nutzung der Kurven in ihrer Darstellung aus Kapitel drei. Dies entspricht nicht in jedem Fall dem optimalen mit dieser Kurve erreichbaren Wert.

Eine weitere Ergebnisbeeinflussung, auf die noch nicht eingegangen wurde, kommt durch die Verschiebung der Daten in das Einheitsquadrat zustande. Diese Verschiebung kann auf zweierlei Art geschehen: Man kann die Daten in x- und y-Richtung ***gleichmäßig*** (um den gleichen Faktor) stauchen und damit einen Teil des Einheitsquadrates eventuell nicht nutzen, oder man skaliert in den beiden Koordinatenrichtungen unterschiedlich stark, d. h. in jede Richtung ***unabhängig*** von der anderen. In letzterem Fall wird in x- und y-Richtung jeweils das volle Intervall $[0, 1]$ ausgenutzt, im ersten Fall gilt das im allgemeinen nur für eine Richtung. Anschaulich bedeutet dies in einem Beispiel die Überführung einer Datenraute in eine Raute im Einheitsquadrat oder in ein Quadrat im Einheitsquadrat, wie es die Abbildung 5.1 zeigt.

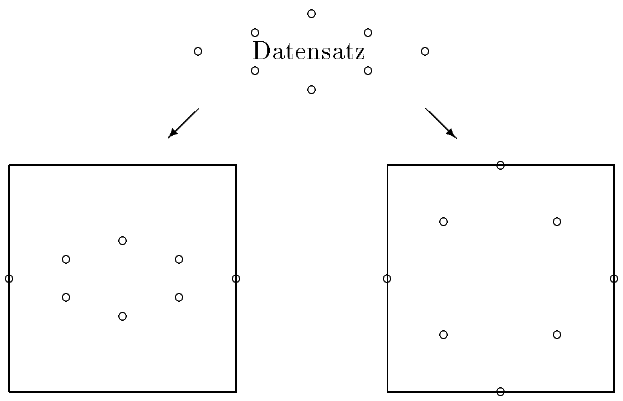

Abbildung 5.1: Zwei Arten der Datenverschiebung: gleichmäßig (links) und unabhängig (rechts)

Beide Vorgehensweisen haben ihre Berechtigung: Im ersten Fall bleibt die geometrische Bedeutung des Problems erhalten, im zweiten Fall wird die Surjektivität der Kurve besser genutzt, da die Daten über das gesamte Einheitsquadrat verteilt sind.

Die Ergebnisunterschiede sind beeindruckend: Verschiebt man die Instanz usa 13509, die 13.509 größten Städte der USA also, gleichmäßig nach $[0, 1]^2$, liefert die Sierpiński-Kurve eine Lösung, die um 40,7 % vom Optimum abweicht. Bei unabhängiger Skalierung in x- und y-Richtung weicht die gleiche Kurve um 55,3 % ab. Die gleiche Instanz führt bei anderen Kurven zu noch größeren Differenzen: Für die Peano-Kurve wird eine Performance von 55 % bzw. 96 % erzielt. Insgesamt kann man feststellen, daß gerade die Instanzen, die Ländergrenzen oder Städte widerspiegeln, besser gleichmäßig behandelt werden, so daß die geometrische Struktur erhalten bleibt. In Fällen, deren Datenstruktur von vornherein rechteckig oder quadratisch ist (Drilling-Probleme, Chips), liefert zumeist die unabhängige Verschiebung bessere Werte.

Prinzipiell kann weder die eine noch die andere Verschiebungsart bevorzugt werden. Ist mehr über die Datenstruktur bekannt, kann man sich nach obiger Richtschnur verhalten. Der in Anhang A beigefügte Code unterstützt beide Möglichkeiten. Randomisierte Instanzen sind von diesen Auffälligkeiten nicht betroffen.

In den Rechnungen zu realen Testfällen wurden bei jeder Kurve beide Arten der Datenverschiebung genutzt. Dabei wird in den Performance-Tabellen der jeweils bessere Wert angegeben. Im Schnitt über alle 20 Testfälle unterscheiden sich die beiden Varianten kaum, wobei die gleichmäßige Variante geringe Performance-Vorteile verbucht. Für jede Kurve gibt es allerdings Instanzen, in denen die eine oder andere Variante deutliche Vorteile erreicht.

Des weiteren nimmt die gewählte Tiefe der Kurve Einfluß auf deren Performance. Grundsätzlich gilt natürlich, daß eine höhere Tiefe zu einer größeren Index-Genauigkeit und damit zu einer Ergebnisverbesserung führt. Denn mit jedem Schritt verringert sich die Anzahl der Punkte, die den gleichen Index tragen und infolgedessen zufällig geordnet werden. Solch eine zufällige Ordnung ist im allgemeinen schlechter als eine durch die Kurve erzwungene. Also werden die Punkte mit höherer Tiefe besser in die Tour eingereiht. Allerdings gibt es auch Fälle, in denen sich die Performance marginal verschlechtert, wenn eine bereits hohe Tiefe noch einmal erhöht wird. Der Grund liegt darin, daß manchmal eine zufällige Ordnung *doch* besser ist als eine vorgegebene. Solche Veränderungen haben aber in keinem der betrachteten Fälle eine Änderung der Performance von mehr als 0,1 % erbracht.

Für die Berechnungen, die in diesem Kapitel vorgestellt werden, wurde grundsätzlich die höchstmögliche Tiefe, die der Rechner zugelassen hat, benutzt. Im Regelfall ergibt dies die beste Performance. Die Einflußnahme der zufälligen Anordnung von Punkten mit gleichen Indizes auf das Ergebnis ist in diesem Fall praktisch gleich null, da jede Kurve mit bis zu vier Milliarden Indizes beschriftet wird. Für die realen Testfälle wurde beobachtet, daß eine viel geringere Tiefe bereits ausreicht, um das beste Ergebnis zu erzielen. Von vornherein darauf zu spekulieren ist riskant, spart aber Rechenzeit. Genauere Aussagen zu diesem Thema wie auch zur Umsetzung der einzelnen Programmteile finden sich in Kapitel sechs. Da in den randomisierten Instanzen z. T. sehr große Städteanzahlen untersucht wurden, muß hier die höchstmögliche Tiefe verwendet werden.

5.2 Peformance der klassischen Kurven

Dieses Unterkapitel dient zur Darstellung der Performancedaten der klassischen Kurven. Eine Bewertung der Ergebnisse und der daraus zu ziehenden Schlüsse wird im nächsten Unterkapitel aufgestellt.

Um einen Eindruck zu vermitteln, wie sich der Einsatz verschiedener Kurven auf ein reales Problem auswirkt, sind in den Abbildungen 5.2 bis 5.5 die Lösungen der SFC-Heuristik für das Problem eil51 aus der TSPLIB [48] graphisch dargestellt, wenn die Sierpiński-, die Moore-, die Hilbert- und die Peano-Kurve benutzt werden. Abbildung 5.6 zeigt eine optimale Lösung für das Problem, dessen Koordinaten aus der TSPLIB [48] übernommen wurden.

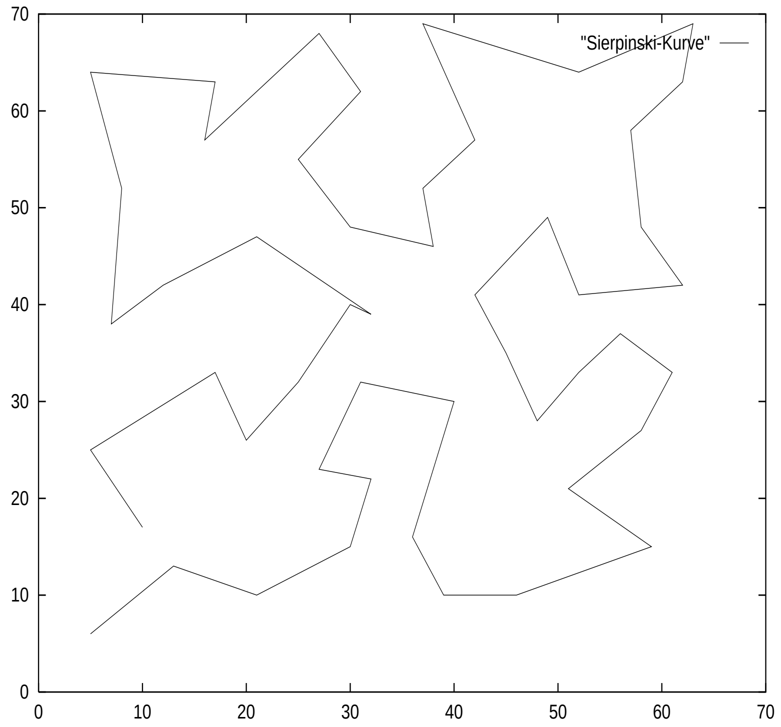

Abbildung 5.2: eil51 mit Sierpiński-Kurve

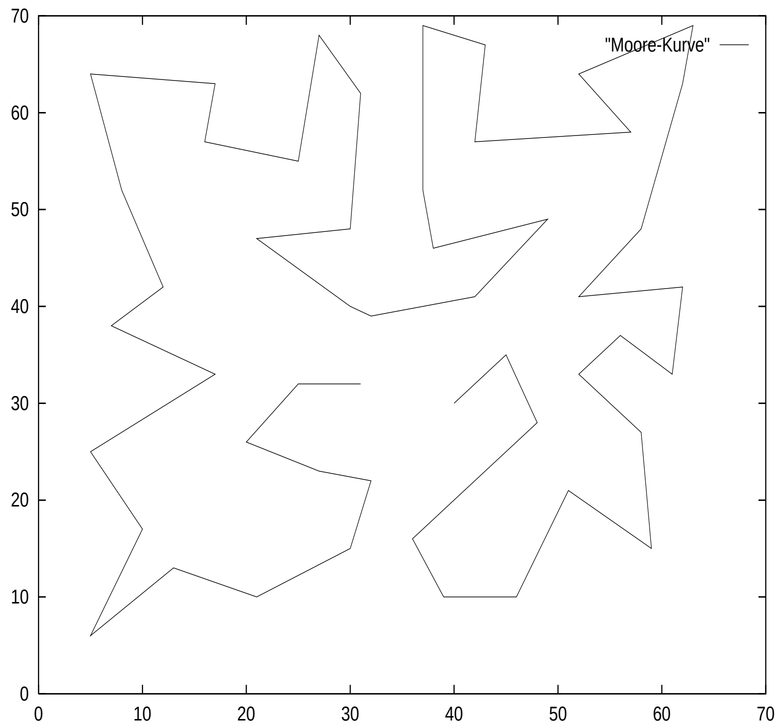

Abbildung 5.3: eil51 mit Moore-Kurve

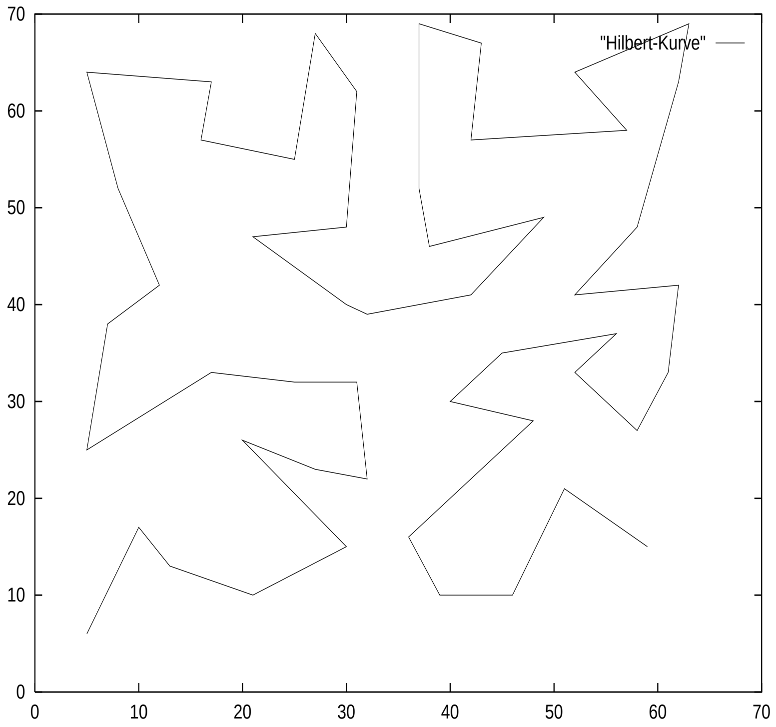

Abbildung 5.4: eil51 mit Hilbert-Kurve

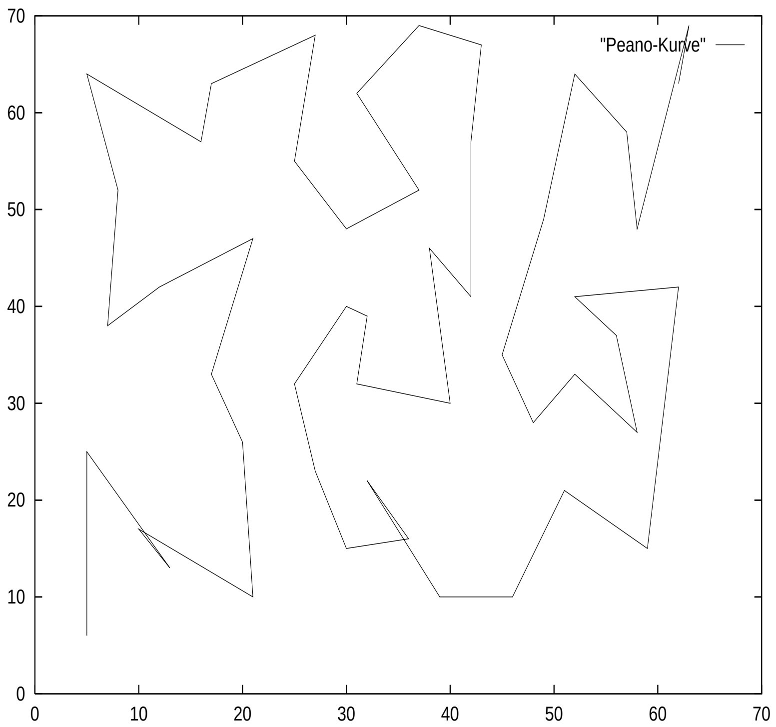

Abbildung 5.5: eil51 mit Peano-Kurve

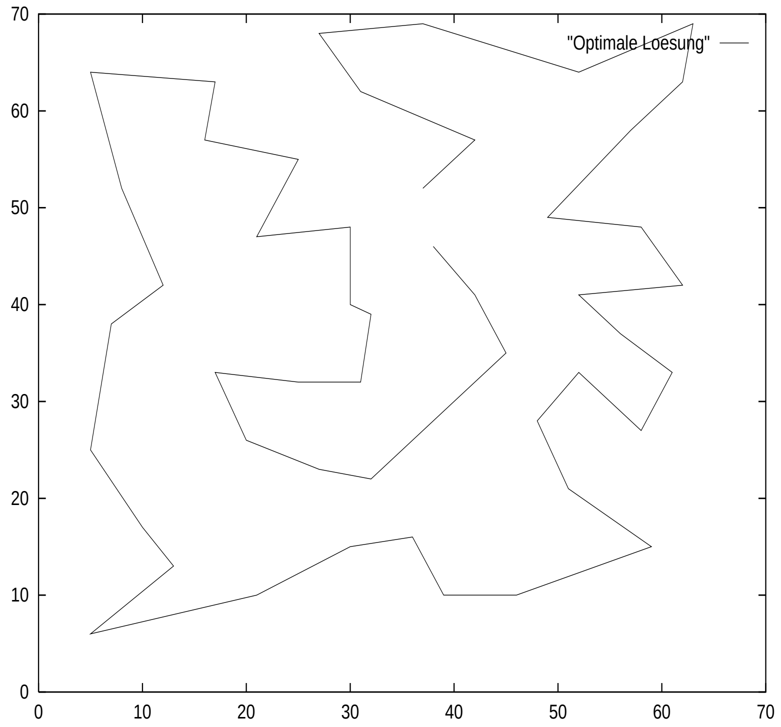

Abbildung 5.6: Optimale Lösung zu eil51

Bei Betrachtung des Weges des Handlungsreisenden kann man den groben Verlauf der jeweiligen Kurve noch gut erkennen. Die Wege werden bewußt nicht geschlossen dargestellt, damit die „Start"- und „End"-Knoten gefunden werden können. Man vergleiche hier insbesondere die Moore- und die Hilbert-Kurve: Letztere muß durch eine lange Kante geschlossen werden, was sich nachteilig auswirkt. Sehr auffällig ist auch der dreigeteilte Verlauf der Peano-Kurve. Die Längen der Lösungen betragen 461 (Sierpiński), 487 (Moore), 538 (Hilbert) und 600 (Peano) bei einer optimalen Lösung von 426.

5.2.1 Reale Testfälle

Für alle betrachteten Kurven gilt: Ihre Performance auf einem realen Problem ist nicht vorauszusehen und kann gut bis sehr schlecht enden. So schwanken selbst die Ergebnisse der besten Kurven zu den 20 betrachteten Testfällen zwischen 10 % und 60 % Abweichung von einer optimalen Lösung. Die Durchschnittsperformance der besten Kurven, der Moore- und der Sierpiński-Kurve, liegt bei 42 %. Eine Auflistung der einzelnen Ergebnisse kann in Anhang B nachgelesen werden; in der Tabelle 5.1 sind die Durchschnittsperformances aufgeführt.

Die Moore- oder die Sierpiński-Kurve sind es auch, die die meisten Fälle am besten lösen: In vier Fünfteln der betrachteten Fälle sind sie besser als alle anderen Kurven. Dennoch lohnt sich ein Blick auf weitere Kurven; z. B. im Fall rl5934 schneidet die Wunderlich 1-Kurve als einzige mit einem respektablen Ergebnis ab.

Kurve	Durchschnittsperf. in %
Moore	41,7
Sierpiński	42,4
Hilbert	44,1
Wunderlich 1	44,7
Wunderlich 2	48,6
Peano	50,4
Lebesgue	98,2

Tabelle 5.1: Reale Durchschnittsperformance der klassischen Kurven

Leider ist es nicht einfach, gewisse Probleme als gut oder weniger gut geeignet zu klassifizieren. Länderprobleme wie d18512, d15112, brd14051, usa13509 und fnl4461 werden besser gelöst als der Schnitt, was aber eher auf die Knotenanzahl zurückzuführen ist als auf die Probleme selbst. Mit steigender Knotenanzahl wird nämlich die Chance größer, daß die Punkte gleichmäßiger über das Einheitsquadrat verteilt werden. In solchen Fällen hat die SFC-Heuristik einen Vorteil. Insbesondere ist anzumerken, daß auf allen genannten Länderproblemen die Sierpiński-Kurve am besten abschneidet. Andererseits liegen in der Klasse der Drilling- oder Cluster-Probleme beide Extrema: sehr schlechte und gute Lösungen, fast kurvenunabhängig.

Es sei an dieser Stelle erneut darauf hingewiesen, daß beide Arten der Verschiebung von Daten in das Einheitsquadrat benutzt wurden (siehe 5.1). In den Tabellen ist der jeweils bessere erreichte Wert verzeichnet.

Trotz der Verschiebevarianten und möglicher Abweichungen durch die Behandlung von Indexrändern können die Unterschiede zwischen den von Reinelt in [47] aufgeführten und den hier errechneten Ergebnissen nicht erklärt werden. Reinelt nutzt in seiner Zusammenstellung zwölf Fälle, die auch für diese Arbeit verwendet wurden und zitiert die Arbeiten von Bartholdi und Platzman; d. h. er hat die Sierpiński-Kurve benutzt. Aber nur in zwei der zwölf Fälle liegen die Ergebnisse „seiner“ Sierpiński-Kurve und der hier implementierten Sierpiński-Kurve so nah beieinander, daß es durch o. g. Varianten erklärlich ist. Ansonsten unterscheiden sich die Daten deutlich, allerdings nicht zum Nachteil der hier dargestellten Ergebnisse. In Tabelle 5.2 könnte Reinelt nur im Fall pr2392 mit 36,8 % ein um 0,8 Prozentpunkte besseres Ergebnis aufweisen – in allen anderen Instanzen werden seine Resultate unterboten.

5.2.2 Randominstanzen

Es sei in Erinnerung gerufen, daß für jeden randomisierten Fall von n Städten eine optimale Tour (fast) die gleiche Länge hat. Insbesondere ist der Quotient aus der Länge dieser Tour und $\sqrt{n}$ für wachsendes n konstant (siehe Gleichung 4.1). Die Ergebnisse der SFC-Heuristik weisen ein ganz ähnliches Bild auf: Betrachtet man eine fest gewählte Kurve, ist für jeden Fall von n Städten eine Ergebnis-Tour fast gleich lang (siehe Gleichung 4.2). Bartholdi und Platzman haben für die Sierpiński-Kurve gezeigt, daß das Verhältnis von Tourlänge zu $\sqrt{n}$ für $n \to \infty$ in dem Intervall [0,955, 0,957] liegt [46].

Problem	beste Kurve	Abweichung in %
pla85900	Moore	32
pla33810	Moore	40
d18512	Sierpiński	30,5
d15112	Sierpiński	31,6
brd14051	Sierpiński	31,0
usa13509	Sierpiński	40,7
rl11849	Moore	47,6
pla7397	Sierpiński	39,2
rl5934	Wunderlich 1	49,9
rl5915	Peano	57,2
fnl4461	Sierpiński	32,1
fl3795	Moore	63,9
pcb3038	Sierpiński	32,7
pr2392	Sierpiński	37,6
u2319	Moore	9,2
u2152	Peano	42,9
d2103	Hilbert	46,1
rl1889	Moore	55,5
u1817	Moore	42,5
vm1748	Moore	43,4

Tabelle 5.2: Performance der jeweils besten SFC auf den Testfällen

Man kann durch Mittelungen einiger Durchläufe des Algorithmus für jedes n ein Verhältnis von Tourlänge zu $\sqrt{n}$ erhalten. Dieser Quotient soll als ***Performance-Index (P-Index) zu n*** bezeichnet werden. Für jede Kurve wurden in dieser Arbeit die P-Indizes zu mindestens 15 Werten von n errechnet. Dabei wurden zehn Durchläufe (ab 500.000 Städten nur fünf Durchläufe) gemittelt. Die Schwankungen der Ergebnisse für ein festes n sind gering, zumindest für ein großes n.

P-Indizes zu verschiedenen n dagegen unterliegen z. T. erheblichen Schwankungen, wie später im Detail zu sehen sein wird. Wenn man nun die Qualität einer Kurve einschätzen will, sind solche Schwankungen hinderlich, zumal es die Numerik allein nicht erlaubt, Rückschlüsse auf ein Verhalten der P-Indizes für $n \to \infty$ zu ziehen. Um eine Vergleichbarkeit herzustellen, wurde folgendes Verfahren gewählt: Der ***P-Index zu einer Kurve*** ergibt sich als Mittelwert des maximalen und minimalen P-Indexes, der für $n \geq 100.000$ und $n \leq 1.400.000$ Städte errechnet wurde.

Legt man den Wert von 0,7124 für den Quotient aus der Länge einer optimalen Tour und $\sqrt{n}$ zugrunde, kann man anhand der so ermittelten Performance-Indizes die durchschnittliche Performance einer Kurve auf randomisierten Fällen schätzen. Diese Daten sind in Tabelle 5.3 zusammengefaßt.

Mit Ausnahme der (hier) uninteressanten Lebesgue-Kurve schneiden alle Kurven auf randomisierten Fällen besser ab als auf den realen Testfällen, und zwar um vier bis acht Prozentpunkte. Dabei ergibt sich für die Sierpiński-Kurve ein Performance-Index von 0,957, wie von Bartholdi und Platzman erwartet. Für *alle* anderen Kurven

Kurve	Performance-Index	Performance in %
Sierpiński	0,957	34,3
Moore	0,979	37,4
Hilbert	0,98	37,6
Wunderlich 1	0,985	38,3
Wunderlich 2	1,006	41,2
Peano	1,045	46,7
Lebesgue	1,454	104,1

Tabelle 5.3: Performance der klassischen SFC auf Randominstanzen

liegen die von Bartholdi und Platzman angegebenen Werte jedoch deutlich entfernt von den hier erreichten. Im Vergleich zur Performance auf realen Fällen haben obige Werte allerdings ein großes Gewicht; die Qualitätsreihenfolge der Kurven hat sich kaum geändert, und auch die Abstände zwischen den einzelnen Kurven stimmen in etwa überein. Für die Moore-Kurve erwähnen dagegen Bartholdi und Platzman einen P-Index von 1,12 [46] – dies würde einer Performance von 57,2 % entsprechen, 23 Prozentpunkte schlechter als bei der Sierpiński-Kurve, obwohl die Moore-Kurve auf realen Fällen besser ist. Die unbewiesenen Aussagen von Bartholdi und Platzman werden daher hier zurückgewiesen.

Zur Bestimmung des P-Indexes für eine Kurve müssen weitere Aussagen gemacht werden: Im Fall der Sierpiński- oder der Moore-Kurve fällt die Berechnung des P-Indexes der Kurve sehr leicht, da die Indizes für die einzelnen Städteanzahlen eine verschwindend geringe Schwankung aufweisen, die eher eine statistische Ungenauigkeit als eine Eigenschaft der Kurve widerspiegelt. Andere Kurven verhalten sich unangenehmer: Für unterschiedliche Werte von $n \geq 40.000$ schwanken die P-Indizes der Peano-Kurve um 18 Tausendstel und die der Wunderlich 1-Kurve sogar um 25 Tausendstel. Damit kann die Angabe eines P-Indexes für eine solche Kurve nur als Abschätzung angesehen werden.

Kurve	P-Index bei n Städten	P-Index bei n Städten	Differenz
Sierpiński	0,956 (80.000)	0,958 (200.000)	0,002
Moore	0,978 (40.000)	0,979 (1.000.000)	0,001
Hilbert	0,983 (50.000)	0,98 (1.000.000)	0,003[1]
Wunderlich 1	0,999 (48.000)	0,974 (1.400.000)	0,025
Wunderlich 2	1,016 (40.000)	1,001 (1.200.000)	0,015
Peano	1,056 (45.000)	1,038 (1.100.000)	0,018
Lebesgue	1,452 (50.000)	1,455 (1.200.000)	0,003

Tabelle 5.4: Aufschlüsselung der Performance auf Randominstanzen

[1]Diese Differenz wird im wesentlichen dadurch verursacht, daß die Kurve nicht geschlossen ist. Dies fällt für steigendes n weniger ins Gewicht.

Solche Abweichungen dürfen nicht überraschen: Für die SFC-Heuristik kann nur ein Intervall angegeben werden, in dem der P-Index für $n \to \infty$ liegt (vgl. 4.3). Dieses Intervall ist bei der Sierpiński-Kurve nachweislich sehr klein, bei anderen Kurven scheint es viel größere Ausmaße anzunehmen. Die theoretische Untersuchung dieser Kurven dürfte also sehr schwierig und vermutlich nicht besonders aussagekräftig sein.

Was sind die Gründe für solche Abweichungen? Eine fehlerhafte Verteilung der Zufallszahlen scheidet aus, da in diesem Fall alle Kurven gleichermaßen betroffen sein müßten. Das unterschiedliche Verhalten jeder einzelnen Kurve weist darauf hin, daß auch kein Software-Fehler vorliegt. Bei näherer Betrachtung fällt aber folgende Besonderheit auf: So hoch die Schwankungen auch sind, sie verlaufen regelmäßig, ähnlich einer stetigen Abbildung, wenn man von der Diskretheit absieht. In Abbildung 5.7 ist dies für die Peano-Kurve dargestellt.

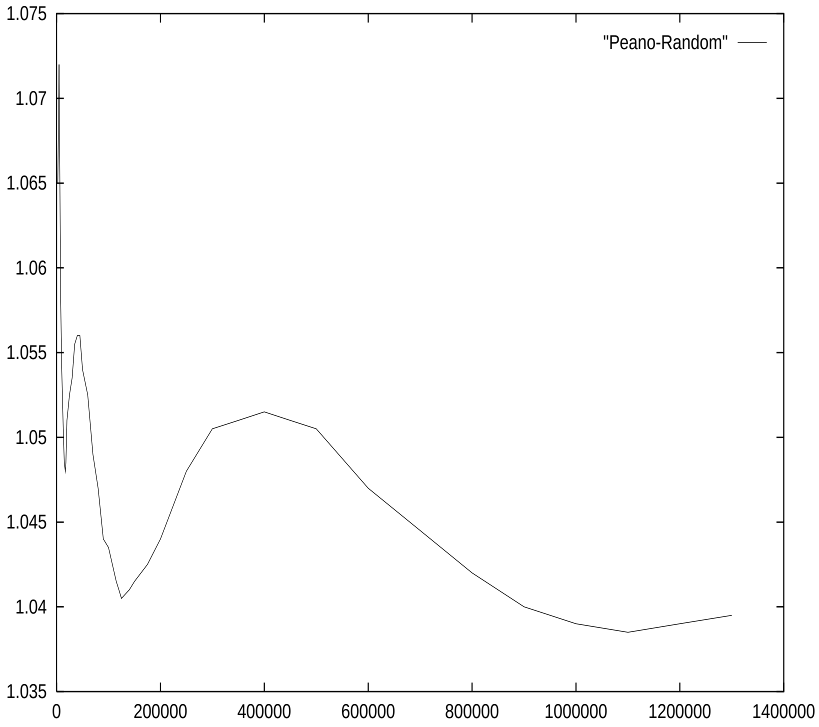

Abbildung 5.7: P-Indizes der Peano-Kurve, aufgetragen gegen die Städteanzahl

Außerdem ergeben sich die Extrema der P-Indizes der Peano-Kurve jeweils bei Verdreifachung der Städtezahl. Dies ist besonders bemerkenswert, da der Peano-Kurve eine 3×3-Verfeinerung zugrundeliegt. Die gleiche Auffälligkeit trifft auf die beiden Wunderlich-Kurven zu, die ebenfalls aus einer 3×3-Verfeinerung entstehen. Allerdings treten hier die Extrema zu anderen Werten von n auf. Die genauen Daten und die Auflistung aller Random-Ergebnisse können dem Anhang B entnommen werden. Die Schwankungsbreite der P-Indizes wird außerdem in Kapitel 5.4 noch einmal aufgegriffen.

Es stellt sich weiterhin die Frage, wie gut der Vergleichswert für C von 0,7124 ist. Die Daten legen eine positive Antwort nahe, da die Performance der Kurven realistisch erscheint. Randomisierte Fälle unterscheiden sich von den realen Instanzen neben der größeren Städteanzahl dadurch, daß die Punkte im Einheitsquadrat gleichverteilt sind. Dies kommt der SFC-Heuristik zugute, da diese jeden Teilabschnitt absolut gleichwertig behandelt. Sind nun also auch die Punkte gleichmäßig verteilt, dürfte das Ergebnis besser sein als in realen Fällen. Zudem ist schon aus Tabelle 5.2 zu ersehen, daß die Heuristik für wachsende Städtezahl besser abschneidet als auf den „kleineren“ Testfällen (und dies hängt wahrscheinlich mit der Gleichverteilung der Punkte zusammen). Ein besseres Abschneiden um einige Prozentpunkte ist also zu erwarten und wird durch Tabelle 5.3 bestätigt.

Natürlich dürfen numerische Daten keinen Anspruch darauf erheben, sich mit theoretischen Ergebnissen messen zu können, insbesondere wenn es bei dem umstrittenen C-Wert um Tausendstel oder sogar Zehntausendstel geht. Dennoch machen die erreichten P-Indizes deutlich, daß die ursprüngliche C-Abschätzung von 0,765 nicht unterstützt werden kann. In diesem Fall würden sich die Kurvenergebnisse zwischen realen und zufälligen Instanzen um fast 20 Prozentpunkte unterscheiden – bei z. T. gleicher Städteanzahl wäre das unerklärlich. Ein kleines, etwas willkürlich gewähltes Rechenexempel legt einen deutlich niedrigeren Wert für C nahe: Betrachtet man nämlich die realen Testfälle mit mehr als 10.000 Städten, so stellt man fest, daß die jeweils besten Kurven, die Moore- und die Sierpiński-Kurve also, eine durchschnittliche Performance von 36,2 % aufweisen. Der gemittelte Performance-Index der beiden Kurven beträgt 0,968. Diese beiden Daten würden auf einen C-Wert von 0,7107 hinweisen, der sehr nahe an der aktuellsten Abschätzung liegt.

5.3 Zur Erstellung neuer Kurven

Betrachtet man die Ergebnisse des vorangegangenen Unterkapitels, drängt sich eine Vielzahl von Schlußfolgerungen auf. Der Sinn der folgenden Seiten besteht darin, diesen Schlußfolgerungen eine Struktur zu geben und daraus Ideen für die Erstellung neuer Kurven zu ziehen. Die Zielrichtung soll dabei sein, raumfüllende Kurven zu finden, die in der SFC-Heuristik noch besser abschneiden als die Sierpiński- oder die Moore-Kurve.

5.3.1 Vermutungen

Die wichtigsten Erkenntnisse, die aus den bisherigen Untersuchungen gewonnen werden können, sind in den folgenden sechs Vermutungen zusammengefaßt, in denen sich eine Wertung wie „schlecht“ nur auf den Einsatz in der SFC-Heuristik bezieht:

Vermutung 1 *Kurven, die über erzwungene Strecken laufen, schneiden schlecht ab.*

Vermutung 2 *Je feinteiliger die Verfeinerung, desto schlechter die Kurve.*

Vermutung 3 *Kurven, deren Leitmotive wenig Richtungsänderungen beinhalten, schneiden schlecht ab.*

Vermutung 4 *Geschlossene Kurven sind vorzuziehen.*

Vermutung 5 *Je komplizierter die Art der Verfeinerung, desto schlechter die Kurve.*

Vermutung 6 *Die Performance der Sierpiński- und der Moore-Kurve wird nicht deutlich unterboten werden können.*

Leider kann man von keiner dieser Vermutungen erwarten, daß sie beweisbar ist. Außerdem müssen solche Aussagen durch die Untersuchung weiterer Kurven unterstrichen werden, da eine Bewertung der bisher betrachteten sieben Kurven nicht ausreicht. Wichtig ist aber, daß obige Vermutungen Hinweise darauf geben, wie neue Kurven zu gestalten sind, um einerseits die dargestellten Ergebnisse zu bestärken und andererseits bessere Kurven zu erstellen.

Zur **Vermutung 1**: Von den bisher betrachteten Kurven schneidet die Lebesgue-Kurve mit deutlichem Abstand am schlechtesten ab. Dabei erfüllt sie einige Bedingungen, die für eine gute Kurve durch die weiteren Vermutungen nahegelegt werden: Sie hat eine einfache Art der Verfeinerung, eine grobe Unterteilung und genauso viele Richtungsänderungen im Leitmotiv, wie z. B. die Moore-Kurve.

Ihr Nachteil liegt darin, daß sie jeden Handlungsreisenden zwingt, über gewisse Strecken zu reisen, die nichts mit der Verteilung der Städte zu tun haben. Dieses Reisen darf nicht so verstanden werden, daß tatsächlich zusätzliche Städte besucht werden. Da aber sehr viele in der Tour aufeinanderfolgende Unterquadrate nicht benachbart sind, ist ein Umweg im Vergleich zu anderen Kurven vorgeschrieben. So führt die Reise von dem Unterquadrat, das den Punkt ($\frac{1}{2}$-ε, 1-ε) enthält, direkt in das Unterquadrat, das den Punkt ($\frac{1}{2}$+ε, 0+ε) enthält ($\varepsilon > 0$, ε sehr klein). In der Abbildung 3.10 ist dies durch eine durchgezogene Strecke zwischen den Punkten ($\frac{1}{2}$, 1) und ($\frac{1}{2}$, 0) angedeutet.

Gerade wenn viele Städte besucht werden, ist die Chance groß, daß Punkte in den o. g. Unterquadraten liegen und daß damit diese unnötige Strecke gereist werden muß. Die gleiche Argumentation gilt für *jede* Begrenzungslinie von Unterquadraten innerhalb des Einheitsquadrates. Für $n \to \infty$ wird jede solche Begrenzungslinie abgefahren, was die Tour erheblich verlängert. Dieses Argument erklärt auch, warum die Lebesgue-Kurve in großen randomisierten Fällen noch schlechter abschneidet als auf kleinen realen Testfällen.

Die Lebesgue-Kurve ist die einzige in dieser Arbeit untersuchte raumfüllende Kurve, deren aufsteigend indizierte Unterquadrate teilweise nicht benachbart sind. Sie erfüllt damit nicht die Bedingung aus Proposition 3.1. Ihre Performance zeigt, daß keine weiteren solcher Kurven betrachtet werden müssen. Die hier geäußerte Vermutung kann sogar so weit ausgedehnt werden, daß auch die Nachbarschaftsbeziehung in nur einem Punkt (statt einer Seite) erheblichen negativen Einfluß auf die Performance hat. Dies wird mit den im folgenden benutzten Kurven leicht nachgewiesen.

Für **Vermutung 2** reicht es festzustellen, daß die besten Kurven auf randomisierten und realen Fällen aus $2{\times}2$-Verfeinerungen entstammen, während alle $3{\times}3$-Verfeinerungen schlechter abschneiden.

Auch die Besonderheit des Verhaltens der Peano- und der Wunderlich-Kurven auf randomisierten Fällen könnte mit der Feinteiligkeit der Verfeinerung zusammenhängen: Während die $2{\times}2$-Verfeinerungen eine auch bei unterschiedlichem n sehr stabile Peformance aufweisen, schlägt die Performance der $3{\times}3$-Verfeinerungen in regelmäßigen Abständen nach oben und nach unten aus. Eine solche Amplitude in den Ergebnissen ist nicht erstrebenswert, da sie in einem praktischen Fall auch zu Lasten der Performance gehen kann.

Die Peano-Kurve liefert den Grund für **Vermutung 3**. Ihre Performance ist schlechter als die der beiden Wunderlich-Kurven, obwohl alle drei Kurven viele Ähnlichkeiten aufweisen; so entstammen sie alle einer 3×3-Quadrat-Verfeinerung.

Der Unterschied zwischen diesen Kurven besteht darin, daß in der Peano-Kurve lange Strecken ohne Richtungswechsel zurückgelegt werden. Schon das Bild zur Lösung des Problems eil51 zu Beginn dieses Kapitels zeigt, daß dies im allgemeinen nicht für die Verwendung einer solchen Kurve spricht. Das Abarbeiten aller Punkte in Form einer Drittelung des Einheitsquadrates ähnelt mehr einer sogenannten *Strip-Heuristik* [47], die nur in wenigen Fällen eine gute Performance aufweist. Das Umstellen des Leitmotivs in jedem zweiten Schritt, wie es die Wunderlich 1-Kurve vorschreibt, führt zu viel besseren Ergebnissen. Daraus resultiert die Vermutung, daß Richtungswechsel für die Performance einer Kurve positiv sind.

Vermutung 4 ist fast eine Selbstverständlichkeit. Geschlossene Kurven, so sie denn verfügbar sind, vermeiden eine sehr schlechte Kante, die zum Schließen der Tour benutzt werden muß. Dies fällt für steigendes n kaum ins Gewicht; die Unterschiede zwischen der Hilbert- und der Moore-Kurve, die ja nichts anderes ist als eine geschlossene Hilbert-Kurve, sprechen auf den realen Testfällen jedoch für sich. Auf randomisierten Fällen gleichen sich die Performances beider Kurven für steigendes n an.

Diese Vermutung sollte aber nicht dazu führen, nur noch geschlossene Kurven zu betrachten. Denn dies würde den Kreis aller Kurven erheblich einschränken. Bei der Konstruktion neuer Kurven kann allerdings darauf geachtet werden, Anfangs- und Endpunkt der Kurve nah beieinander zu halten.

Die Unterteilung eines Objektes in Unterobjekte ist Thema von **Vermutung 5**. Bislang sind nur Triangulierungen und Quadrat-Verfeinerungen dargestellt worden. Die Triangulierung erweist sich dabei auf randomisierten Fällen in doppelter Hinsicht überlegen: Erstens ist die Performance der Sierpiński-Kurve die beste, zweitens erscheint diese Kurve sehr stabil im Sinne von Schwankungen für verschiedene n.

Wie sich andere Arten der Unterteilung auswirken, ist für diese Arbeit ansatzweise untersucht worden. Eine Unterteilung in n-Ecke mit $n > 4$ birgt mehrere Probleme: Erstens wird die Implementierung erheblich schwieriger. Die Abfrage nach einem Index dauert in jedem Fall deutlich länger als bei Quadraten oder Dreiecken. Selbst wenn eine solche Kurve einen Performance-Vorteil von ein oder zwei Prozentpunkten erreichen würde, wäre mit einer deutlich höheren Laufzeit zu rechnen. Da ist es erstrebenswerter, eine vielleicht schlechtere, aber schnellere Kurve zu wählen und diese z. B. mit einem 2-opt-Tausch zu verbessern. Zweitens ist eine rekursive Struktur des Programms praktisch nur bei regelmäßiger Unterteilung in Polygone mit 3, 4 oder 6 Ecken möglich. In den weiteren Fällen müßte man auf nicht-regelmäßige Vielecke zurückgreifen, um die Fläche überhaupt vollständig parkettieren zu können; eine Rekursion scheidet damit aus (vgl. zu diesem Komplex [52] und [10]). Das Laufzeitargument und der beobachtete Vorteil der Triangulierung gegenüber der Quadratunterteilung sind Anlaß genug, andere Arten der Verfeinerung abzulehnen.

Zusammengefaßt ergibt sich **Vermutung 6**. Sowohl die Sierpiński- als auch die Moore-Kurve erfüllen die positiven Eigenschaften, die bisher formuliert werden können: Sie sind geschlossen, von grober Verfeinerungsstruktur, greifen auf einfache Unterteilungsobjekte zurück, vermeiden Nachbarschaftsbeziehungen in einem Punkt und weisen im Leitmotiv das Maximum möglicher Richtungsänderungen auf. Dies bedeutet nicht, daß es keine besseren Kurven geben kann; die bisher untersuchten Kurven geben allerdings keinen Hinweis darauf, wie eine bessere Kurve aussehen könnte.

5.3.2 Konsequenzen

Bei der Darstellung der als „neu“ bezeichneten Kurven in Kapitel drei wurde nicht begründet, warum ausgerechnet diese Kurven für die Analysen in der vorliegenden Arbeit ausgewählt und erstellt wurden. Dies ist nach der Untersuchung der Ergebnisse im vorangegangenen Abschnitt nun so zu erklären:

- Es werden nur Triangulierungen und Quadrat-Verfeinerungen untersucht.
- Falls es möglich ist, werden geschlossene Kurven untersucht.
- Um den Einfluß der Nachbarschaftsbeziehungen zwischen den Unterteilungsobjekten zu untersuchen, müssen einige Kurven eingesetzt werden, die den Handlungsreisenden zwingen, über einen Punkt zu laufen. Solche Punktläufe treten bei Triangulierungen sehr schnell auf. Alle in Kapitel 3.4 dargestellten 2×2- und 3×3-Triangulierungen haben diese Eigenschaft.
- In den klassischen Kurven liegen nur 2×2- und 3×3-Unterteilungen vor; zur Unterstützung von Vermutung 2 müssen feinere Unterteilungen untersucht werden.
- Wie sehr häufige Richtungsänderungen im Leitmotiv die Performance verbessern können, wird durch die 5×5-Kurven unterstrichen. Die 5×5-Peano-Kurve führt von allen untersuchten Kurven die *wenigsten* Richtungsänderungen durch und steht damit im Gegensatz zu den anderen dargestellten 5×5-Kurven, die bewußt häufig ihre Richtung ändern.

5.4 Performance der neueren Kurven

Die numerischen Untersuchungen der nicht-klassischen Kurven bestätigen im wesentlichen die im letzten Unterkapitel geäußerten Vermutungen. Tabelle 5.5 faßt die durchschnittliche Performance von neun weiteren Kurven zusammen.

Kurve	Reale Perf. in %	P-Index	Random-Perf. in %
4×4 - 1	42,7	0,994	39,5
5×5	45,2	1,019	43
6×6	47,2	0,99	39
4×4 - 2	48,5	1,018	42,9
3×3 - 1	53,9	1,034	45,1
5×5-Peano	54,4	1,103	54,8
3×3 - 2	55	1,061	48,9
Dreieck 1	67	1,121	57,4
Dreieck 2	69,2	1,132	58,9

Tabelle 5.5: Durchschnittsperformance der neuen Kurven

Außer den hier dargestellten Kurven wurde ein gutes Dutzend weiterer Kurven implementiert und teilweise analysiert. Da die Ergebnisse aber entweder schlechter waren oder lediglich bekannte Ergebnisse bestärkten (wie bei weiteren 3×3-Quadrat-Verfeinerungen nach Wunderlich), wird hier der Übersichtlichkeit halber auf deren Darstellung verzichtet.

Die wichtigste Erkenntnis nach obigen Daten ist die, daß keine Kurve gefunden wurde, die die Sierpiński- oder die Moore-Kurve übertrifft. Vermutung 6 ist damit bestärkt. Allerdings kommt die 4×4-1-Kurve in den realen Testfällen sehr nah an die Performance der beiden besten Kurven heran. Außerdem gelingt es ihr, in sieben der 20 Instanzen die beste Tour zu erzeugen. Doch leidet ihre Random-Performance unter den sehr starken Schwankungen der P-Indizes. Sie erreicht z. T. ähnlich gute Indizes wie die Sierpiński-Kurve, fällt aber bei anderen Städteanzahlen weit zurück. Die gewählte Mittelung der Extremwerte ergibt folglich einen Gesamtindex, der sogar über dem der stabileren 6×6-Kurve liegt.

Weiter zeigt sich, daß Kurven, die über einen Punkt laufen, Nachteile gegenüber anderen Kurven haben, und zwar um so stärker, je häufiger solche Punktläufe stattfinden. Die beiden „Dreieck"-Kurven gehen auf eine 2×2-Triangulierung zurück und beinhalten in den vier Unterdreiecken einen Punktlauf. Ihre Performance ist sehr schlecht. Die beiden 3×3-Triangulierungen führen den Weg der Kurve pro Unterteilung ebenfalls einmal über einen Punkt, dies allerdings bei neun Unterdreiecken. Somit ist die Performance zumindest besser als bei den Dreieck-Kurven. Schließlich enthält das Leitmotiv der 6×6-Kurve zwei ausgezeichnete Punkte bei einer Unterteilung in 36 Unterdreiecke. Auswirkungen auf die Performance sind kaum zu erkennen. Vermutung 1 kann also noch schärfer formuliert werden: Je häufiger eine Kurve über einen Punkt läuft, um so schlechter ist ihre Performance.

Vorteile von Triangulierungen gegenüber Quadrat-Verfeinerungen sind der Tabelle 5.5 nicht unmittelbar zu entnehmen. Allerdings muß jede Kurve, die durch eine $k \times k$-Triangulierung mit $k = 2$ oder $k = 3$ entsteht – mit Ausnahme der Sierpiński-Kurve – an mindestens einer Stelle im Leitmotiv über einen Punkt laufen (vgl. Kapitel drei). Solche Kurven sind, wie wir gesehen haben, nicht konkurrenzfähig. Die Tatsache, daß die sehr feinteilige und Punktläufe beinhaltende 6×6-Kurve *dennoch* so gut abschneidet, kann als Indiz für die Überlegenheit von Triangulierungen angesehen werden. Vermutung 5 kann daher als bedingt gestärkt betrachtet werden.

Die 5×5-Peano-Kurve ist in die Untersuchungen aufgenommen worden, um die Auswirkungen sehr starrer Leitmotive zu testen. In der Tat schneidet diese Kurve sehr schlecht ab und bestätigt damit Vermutung 3. Dabei ist das verbesserte Abschneiden der Kurve auf realen Testfällen darauf zurückzuführen, daß sie von der Auswahl des besseren Ergebnisses der beiden Datenverschiebungen profitiert. Während dies bei den anderen Kurven im Schnitt eine Verbesserung von ein bis zwei Prozentpunkten gegenüber einer fest gewählten Verschiebungsart ergibt, steigert die 5×5-Peano-Kurve ihre Performance im Schnitt um 6,5 (gegenüber gleichmäßiger Verschiebung) bzw. 9,3 (gegenüber unabhängiger Verschiebung) Prozentpunkte. Pro Instanz sind Unterschiede von mehr als 20 Prozentpunkten häufig; auch 50 Prozentpunkte und mehr werden erreicht.

Insbesondere zur Überprüfung von Vermutung 2 wurden die 4×4- und 5×5-Kurven betrachtet. Von letzteren ist nur die beste von insgesamt vier untersuchten Kurven dargestellt. Tatsächlich ergibt sich ein eindeutiges Bild, wie sich Kurven mit steigender Feinteiligkeit verhalten (man beachte die auf erheblichen Punktläufen beruhende Sonderstellung der 3×3-1-Kurve):

k	P-Index	Kurve
2	0,979	Moore
3	0,985	Wunderlich 1
4	0,994	4×4 - 1
5	1,019	5×5

Tabelle 5.6: Beste Kurven für $k \times k$-Quadrat-Unterteilung

k	P-Index	Kurve
2	0,957	Sierpiński
3	1,034	3×3 - 1
6	0,99	6×6

Tabelle 5.7: Beste Kurven für $k \times k$-Triangulierung

Je mehr Struktur der Kurve also von vornherein mitgegeben wird, d. h. je genauer das Leitmotiv und je feinteiliger die Kurve ist, um so schwerfälliger und schlechter ist das Ergebnis. Insbesondere erkennt man auch, daß es erst für $k = 5$ gelingt, mit einer $(k+1) \times (k+1)$-Triangulierung eine $k \times k$-Quadratunterteilung zu schlagen. Für $k = 2$ funktioniert dies nicht, so daß die Moore-Kurve nicht durch eine feinteiligere Triangulierung unterboten wird.

Eine steigende Feinteiligkeit der Kurven hat noch einen anderen Nachteil: Die Schwankungen der P-Indizes für unterschiedliche n nehmen zu. Dabei sind Quadrat-Unterteilungen noch anfälliger als Triangulierungen. Die Schwankungsbreite umfaßt bis zu 78 Tausendstel bei der 5×5-Peano-Kurve, wie Tabelle 5.8 zeigt. Eine solche Schwankung entspricht einer zusätzlichen Abweichung vom Optimum von 11 %.

Kurve	P-Index bei n Städten	P-Index bei n Städten	Differenz
4×4 - 1	0,964 (180.000)	1,024 (700.000)	0,06
5×5	1,001 (40.000)	1,035 (180.000)	0,034
6×6	1,001 (40.000)	0,977 (250.000)	0,024
3×3 - 1	1,029 (45.000)	1,039 (130.000)	0,01
4×4 - 2	1,034 (30.000)	1,008 (130.000)	0,026
3×3 - 2	1,065 (35.000)	1,058 (110.000)	0,007
Dreieck 1	1,119 (50.000)	1,121 (500.000)	0,002
Dreieck 2	1,131 (100.000)	1,133 (400.000)	0,002
5×5-Peano	1,143 (180.000)	1,065 (800.000)	0,078

Tabelle 5.8: Aufschlüsselung der Performance auf Randominstanzen

Dabei fällt bei allen Kurven mit nennenswerter Amplitude der P-Indizes eine regelmäßige Schwankung auf. In allen Fällen wiederholen sich die Extrema der P-Indizes einer $k \times k$-Unterteilung derart, daß der Quotient zwischen den zugehörigen Städteanzahlen ungefähr ein Vielfaches von k ist. Zum Beispiel liegt bei der 3×3-1-Kurve bei 15.000 Städten ein Maximum vor, dann bei 45.000 ein Minimum, bei 130.000 wieder ein Maximum und bei 400.000 ein Minimum usw. Bei der 6×6-Kurve liegen diese Extrema entsprechend weiter auseinander.

Für wachsendes n wird der Unterschied zwischen einem Extremwert und dem vorangegangenen meist geringer; außerdem nimmt die Größe der Extremwerte ab. Dennoch kann man anhand der gewonnenen Daten nicht davon ausgehen, daß sich für $n \to \infty$ die Schwankungen auspendeln, d. h. daß ein eindeutiger P-Index vorliegt. Die theoretische Beobachtung von Bartholdi und Platzman [46], daß nur ein Intervall für den P-Index für $n \to \infty$ angegeben werden kann, unterstreicht diese These.

Ein möglicher Grund für dieses Phänomen ist die Verteilung der Städte. Bei wachsender Städteanzahl wird wohl in gleichmäßigen Abständen eine Sättigung der Verteilung erreicht, d. h. zu manchem n ist die Anzahl der Städte pro Unterquadrat einer gewissen Tiefe für diese Heuristik besser verteilt als zu anderem n. Ein zwingendes Argument für ein solches Verhalten der Kurven kann hier nicht vorgebracht werden.

5.5 Weitere raumfüllende Kurven

Das in Kapitel drei vorgestellte Konstruktionsprinzip einer raumfüllenden Kurve ist nicht die einzige Möglichkeit, solche Kurven zu erstellen. In den vergangenen Jahren sind einige SFC gefunden worden, die auf andere Entstehungsprinzipien zurückgreifen. Solche Kurven können z. T. gar nicht oder nur sehr schwer auf die hier dargestellte Art erzeugt werden (Fraktale, Schoenberg-Kurve).

Die Fokussierung auf das in Kapitel drei eingeführte Prinzip ist darauf zurückzuführen, daß es unmittelbar eine leichte Programmierung solcher Kurven sicherstellt und es außerdem ermöglicht, alle klassischen Kurven darzustellen. Die bekanntesten anderen Kurven sollen hier dennoch angesprochen werden. Kurven, die auf nicht-regelmäßigen Tilings beruhen oder geometrisch komplexere Gebilde als Drei- oder Vierecke nutzen, werden wegen erheblicher Nachteile in der Implementierung und Laufzeit nicht betrachtet.

5.5.1 Moscato-Norman-Kurven

Der Sinn der von Moscato und Norman 1994 veröffentlichten Kurven namens MPeano und MNPeano liegt darin, ETSP-Instanzen beliebiger Größe erstellen zu können, deren optimale Lösung bekannt ist. Dies gelingt auch: Für jede dieser Kurven ist eine Menge von Punkten auf der Kurve eindeutig definiert. Zu einer solchen Punktmenge ist die MPeano- bzw. die MNPeano-Kurve eine optimale Tour [42].

Die Autoren gehen davon aus, daß die zu MNPeano gehörende Punktmenge (asymptotisch für eine steigende Tiefe der Kurve) die Menge kleinster Kardinalität ist, für die diese Kurve optimal ist [38]. Dies ermöglicht eine Berechnung der Konstanten C, die in Kapitel 4.2.2 eingeführt wurde, für diese Punktmenge. C ergibt sich nach ihren Ausführungen zu 0,7148, was in einer akzeptablen Übereinstimmung mit anderen Abschätzungen liegt.

Zur Konstruktion der Kurve benutzen Moscato und Norman genau das Gegenteil einer Dreiecksbisektion, nämlich eine Dreiecksverdoppelung, und geben an, wie die Dreiecke, genauer die Punkte innerhalb der Dreiecke, zu verbinden sind. Abbildung 5.8 zeigt dieses Schema in den ersten Schritten.

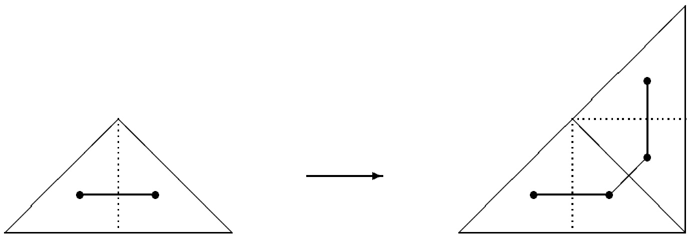

Abbildung 5.8: Konstruktion der Moscato-Norman-Kurven

Die MPeano- und MNPeano-Kurven sind gleich bis auf die zugehörigen Punktmengen. Auf die Unterscheidung dieser Mengen wird hier nicht eingegangen, da im Zusammenhang mit dieser Arbeit nur die Kurve von Interesse ist. Dank des Konstruktionsschemas ist mit Blick auf Abbildung 3.16 sofort ersichtlich, daß bei Verbindung zweier Dreieckstrukturen aus Abbildung 5.8 eine Sierpiński-Kurve beliebiger Tiefe entsteht. Die Festlegung von Punktkoordinaten hat dabei für $m \to \infty$ keinen Einfluß mehr. Die von Moscato und Norman vorgestellten Kurven sind also gleich und entsprechen zudem der Sierpiński-Kurve.

5.5.2 H-Index

Der ***H-Index*** beschreibt eine raumfüllende Kurve, die 1997 von Niedermeier, Reinhardt und Sanders vorgestellt wurde [41]. Ihren Namen trägt sie in Anlehnung an ihr Aussehen, das sich in jeder Tiefe mit der Aneinanderkettung von Buchstaben „H“ beschreiben läßt. In Tiefe 3 ist sie noch mit der geschlossenen Hilbert-Kurve identisch, danach verläßt sie allerdings das strenge rekursive Schema aus Kapitel drei.

Es trifft ab Tiefe 4 nicht mehr zu, daß ein Unterquadrat mit Index i in der nächsten Tiefe Unterquadrate mit Indizes $4i$ bis $4i + 3$ erzeugt; statt dessen entstehen völlig neue, manchmal unzusammenhängende Indexzuteilungen. So trägt das Startquadrat aus Abbildung 5.9 in höherer Tiefe (Abbildung 5.10) nicht die Indizes 0 bis 63, sondern die Indizes 0 bis 31 und 992 bis 1023. Eine dennoch recht übersichtliche H-Index-Routine, die den Trick benutzt, zwei Tiefen der Kurve gleichzeitig zu betrachten, ist dem Anhang zu entnehmen.

Beim Einsatz der H-Index-Kurve in der SFC-Heuristik wird die Reihenfolge der Städte aufgrund der Indexwechsel von Tiefe zu Tiefe durcheinandergewürfelt und keineswegs in jedem Schritt verbessert.

Ihre Konstruktion beschreiben die Autoren in [41] als „eine Variante der 2D-Sierpiński-Kurve“. Tatsächlich legt das Aussehen eine gewisse Verwandtschaft mit der Hilbert- und der Sierpiński-Kurve nahe. Die Implementierung des H-Indexes und der Einsatz in der SFC-Heuristik zeigen aber überraschenderweise, daß sich für steigende Tiefe die Ergebnisse des H-Indexes denen der Sierpiński-Kurve nicht nur annähern, sondern für $m \to \infty$ als diesen identisch angesehen werden können: Schon für eine Tiefe von $m = 16$ unterscheiden sich die Daten der Testfälle nur noch um höchstens

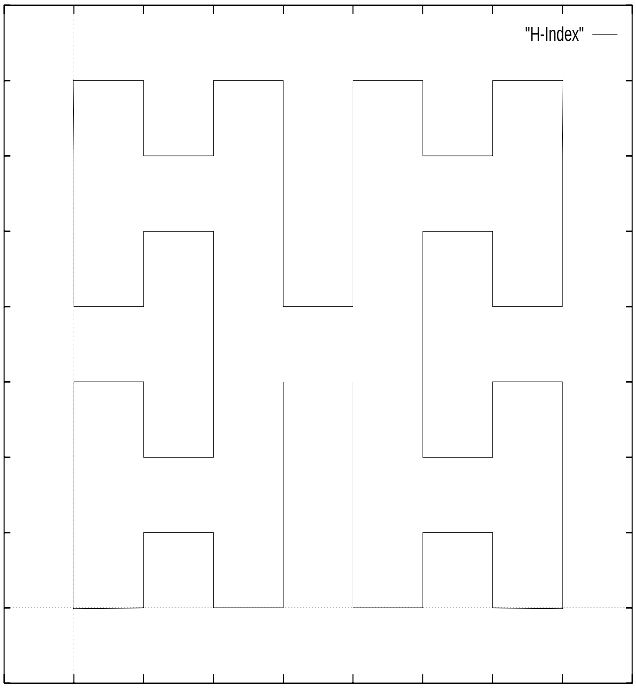

Abbildung 5.9: H-Index-Kurve in Tiefe 4

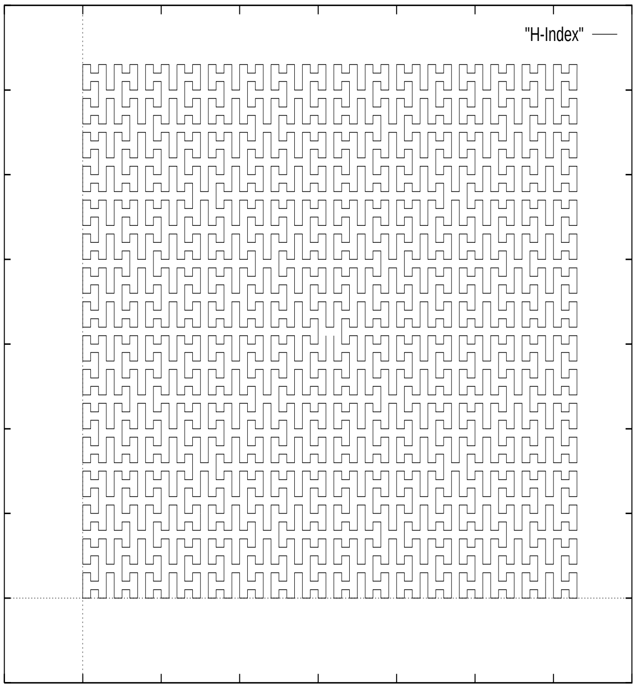

Abbildung 5.10: H-Index-Kurve in Tiefe 7

0,1 Prozentpunkte; auch die Ergebnisse auf randomisierten Instanzen sind praktisch gleich (siehe Anhang).

Ein möglicher theoretischer Nachweis, daß diese beiden Kurven tatsächlich identisch sind, würde diese Arbeit sprengen, sollte aber in mehrfacher Hinsicht von großem Interesse sein. Denn die „neue“ Kurve wurde eingeführt, um Lokalitätseigenschaften von raumfüllenden Kurven respektive Mesh-Indexings zu untersuchen. Niedermeier, Reinhardt und Sanders nehmen an, daß der H-Index optimal näheerhaltend unter allen SFC ist, und zeigen dies für alle geschlossenen Kurven, die aus Quadrat-Verfeinerungen entstehen und nicht über einen Punkt laufen [41]. Die Richtigkeit ihrer Annahme, verbunden mit der Gleichheit von H-Index und Sierpiński-Kurve, würde nahelegen, daß die Sierpiński-Kurve die beste Kurve für die SFC-Heuristik ist.

Zur Näheerhaltung einer raumfüllenden Kurve ist folgendes auszuführen: Faßt man eine SFC diskret auf, so beschreibt sie eine Abbildung K von einer Indexmenge $\{0, \ldots, k^2-1\}$ nach $\{0, \ldots, k-1\}^2$. Interpretiert man die rechte Seite als kartesisches Produkt – denkt man sich also diese Punktmengen in der euklidischen Ebene angeordnet – erhält man eine Indexzuteilung, wie sie in Kapitel drei beschrieben wird. Daher kann ein solches regelmäßiges *Mesh Indexing* mit einer diskreten raumfüllenden Kurve identifiziert werden.

Nun interessiert z. B. bei der Auslastung von Prozessoren eines Parallelrechners die Frage, wie weit $K(i)$ von $K(j)$ entfernt ist, wenn der Abstand von i und j bekannt ist. Gotsman und Lindenbaum zeigten für die Hilbert-Kurve die obere Schranke $d(K(i), K(j)) \leq \sqrt{(20/3) \cdot |i-j|}$, wenn $d(.,.)$ den euklidischen Abstand bezeichnet [18]. Niedermeier, Reinhardt und Sanders verbessern diese Schranke für die Hilbert-Kurve auf $d(K(i), K(j)) \leq \sqrt{6,01 \cdot |i-j|}$. Für den H-Index können sie eine noch bessere Abschätzung vornehmen, die unter den geschlossenen Quadrat-Verfeinerungen ohne Punktlauf sogar optimal ist [41]. Es gilt nämlich für den H-Index $d(K(i), K(j)) \leq \sqrt{4|i-j|-2}$. Für den Fall, daß der H-Index gleich der Sierpiński-Kurve ist, könnte man also feststellen, daß die beste Quadrat-Verfeinerung ausgerechnet eine Triangulierung ist.

5.5.3 Fraktale

Ohne auf die genaue Definition und die theoretischen Hintergründe eingehen zu können, sollen Fraktale dennoch nicht unberücksichtigt bleiben. Sie unterscheiden sich von „echten“ raumfüllenden Kurven dadurch, daß sie die Ebene nicht ausfüllen, da die zugrundeliegende Abbildung nicht surjektiv ist. Wenn dennoch von der Dimension eines Fraktals die Rede ist, handelt es sich hier um eine gesonderte Definition einer Ähnlichkeitsdimension (vgl. [51]). Tatsächlich haben alle bekannteren Fraktale (von Koch, Sierpiński-Gasket) ein zweidimensionales Lebesgue-Maß von 0 und füllen nicht wirklich einen Teil des Raumes aus.

Ein Fraktal kann z. B. so entstehen: Man betrachte eine $3{\times}3$-Quadrat-Verfeinerung wie bei der Peano-Kurve, nur entferne man in jedem Schritt von je neun Unterquadraten das mittlere. So entsteht schließlich eine Fläche mit vielen Löchern (daher heißt dieses *Mazurkiewicz-Continuum* im Dreidimensionalen auch Sierpiński-Schwamm). Da in jedem Schritt jeweils ein Neuntel der aktuellen Fläche entfernt wird, füllt die Kurve für $m \to \infty$ den Raum nicht mehr aus.

Um solche Gebilde dennoch in ihrer Komplexität unterscheiden zu können, bedient

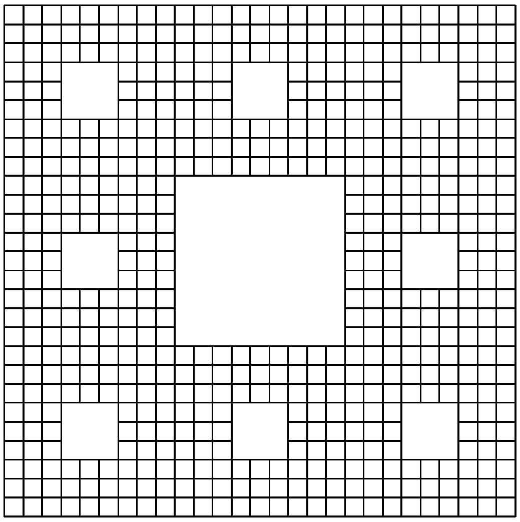

Abbildung 5.11: Mazurkiewicz-Continuum in Tiefe 4

man sich einer Ähnlichkeitsdimension, die sich im Fall des Mazurkiewicz-Continuums als Quotient von log(8) und log(3) zu 1,89 ergibt. Dabei steht die 3 für eine 3×3-Verfeinerung und die 8 für die entstehenden acht (statt neun) Unterobjekte. Würde man ein Fraktal betrachten, das auf einer 2×2-Verfeinerung beruht und bei dem jeweils auf eins von vier Unterquadraten verzichtet wird, ergäbe sich eine Dimension von $\log(3)/\log(2)$.

Für die Anwendung einer fraktalen Kurve beim TSP stellt sich unmittelbar folgendes Problem: Da die Fläche nicht ausgefüllt wird, werden Städte, die in „Löchern“ liegen, nicht besucht. Diesem Problem kann man begegnen, indem man die Koordinaten jeder Stadt auf das nächstgelegene Stück der Kurve rundet. Dies birgt allerdings die Gefahr einer deutlichen Tourverschlechterung und bedeutet einen größeren Aufwand als bei echten SFC. Andererseits werden so verschiedene Richtungsänderungen ermöglicht, die bei einer starren SFC nicht möglich sind (hier sind die Winkel zwischen den Unterobjekten zu 60°, 90° o. ä. vorgeschrieben). Außerdem ist sichergestellt, daß eine erste Anordnung der Punkte immer noch Nähebeziehungen erhält, so daß Verbesserungsschritte wie ein 2-opt-Tausch großen Erfolg haben.

Für diese Arbeit wurden zwei Fraktale entwickelt und untersucht. Auf die bekannteren Fraktale wie das oben beschriebene Mazurkiewicz-Continuum wurde nicht zurückgegriffen, da die Anordnung einer Kurve zusätzlich Stetigkeitsprobleme hervorruft.

Das erste Fraktal entsteht durch eine 2×2-Verfeinerung. Hier wird ab Tiefe 3 in jedem Schritt bei allen Quadraten, die durch eine Hauptdiagonale des Einheitsquadrates geschnitten werden, auf das äußerste Unterquadrat verzichtet. Abbildung 5.12 macht dieses Vorgehen deutlich. Die zu entfernenden Unterquadrate sind schraffiert dargestellt.

Eine stetige Anordnung ergibt sich kanonisch und ist ebenfalls in Abbildung 5.12 dargestellt. Führt man dieses Schema bis ins Unendliche fort, entsteht ein eher untypisches Fraktal, da die Kurve genau zwei Drittel des Einheitsquadrates ausfüllt und

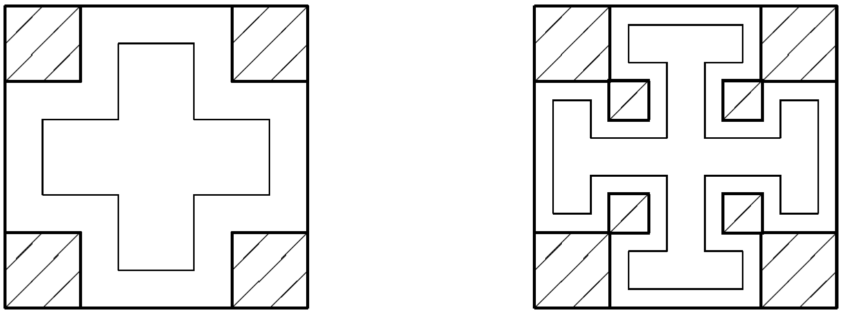

Abbildung 5.12: Fraktal 1 in Tiefe 3 und 4

damit für diese Fläche „echt“ raumfüllend ist[2].

Das zweite untersuchte Fraktal beruht ebenfalls auf einer 2×2-Verfeinerung. Allerdings wird hier bei jedem Quadrat auf genau ein Unterquadrat verzichtet. Daher verliert die Kurve ihre raumfüllende Eigenschaft; sie erreicht eine fraktale Dimension von $\log(3)/\log(2) \approx 1,58$. Die Anordnung ist ab Tiefe 5 nicht überall stetig möglich. Eine Darstellung dieser Kurve in Tiefe 3 und 4 ist der Abbildung 5.13 zu entnehmen.

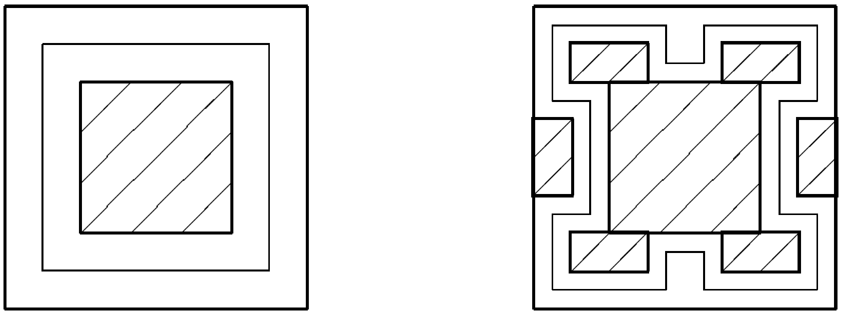

Abbildung 5.13: Fraktal 2 in Tiefe 3 und 4

Für den Einsatz in der SFC-Heuristik muß nun noch geklärt werden, wie den Städten, die durch die Kurve *nicht* berührt werden, ein Index zugeteilt wird. Dazu wird jedes zu entfernende Unterquadrat derart in zwei Dreiecke aufgeteilt, daß jedes Dreieck genau eine Seite mit einem Unterquadrat teilt, das aus dem gleichen (Ober)-Quadrat entstanden ist. Jeweils eine Koordinate der Städte wird dann auf diese eindeutig zugeordnete Seite gerundet, wie in Abbildung 5.14 dargestellt.

Dabei ist es durchaus möglich, daß Koordinaten mehrfach gerundet werden müssen, bevor schließlich ein Index verteilt werden kann. Die Indexzuteilung selbst ignoriert die entfernten Unterquadrate und erfolgt dann wie bei anderen SFC.

Fraktal 1 fungiert als ein Mittelding zwischen einer „echten“ raumfüllenden Kurve und einem flächenlosen Fraktal. Eingesetzt in der SFC-Heuristik, darf man also erwarten, daß es mittlere Werte erzielt. Dies ist auch der Fall: Fraktal 1 erreicht in den 20 Testfällen Abweichungen von 69 % bis 457 % vom Optimum; Fraktal 2 dagegen weicht zwischen 133 % und 1313 % ab (siehe Anhang). Auffallend ist eine umso stärkere Abweichung, je größer die Anzahl der Städte ist. Dies ist leicht erklärbar, da durch die starken Rundungen ganze Flächenstücke bei steigender Städteanzahl

[2]Die entfernte Fläche ergibt sich zu $\sum_{i=1}^{\infty} \frac{1}{4^i} = \frac{1}{3}$.

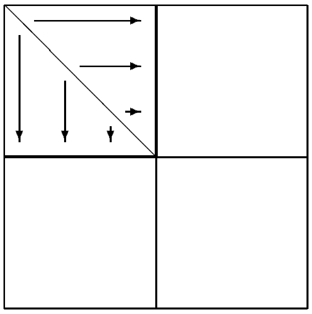

Abbildung 5.14: Mögliche Rundung bei Fraktalen

mehrfach durchreist werden, ohne daß sich die heuristische Anordnung noch nach der Geometrie des Problems richtet.

Eine geschicktere Art des Rundens der Koordinaten könnte positive Auswirkungen auf die Leistung der Heuristik haben. Sie wird sich aber sehr wahrscheinlich negativ in den Laufzeiten niederschlagen. Die untersuchten Fraktale sind durch die Laufzeit gegenüber anderen Kurven ohnehin stark benachteiligt. Da außerdem die Abweichungen enorm sind, kann zu ihrem Einsatz nicht geraten werden.

5.6 Bewertung

Dieses Unterkapitel dient zur Zusammenfassung der bisher gewonnenen Erkenntnisse und ihrer Bewertung. Um zudem die Performance anderer, hier nicht untersuchter SFC abschätzen zu können, wird ein Bewertungsschema aufgebaut, das schon vor der Implementierung eine Orientierungshilfe gibt.

5.6.1 Bewertung der Ergebnisse

Zunächst ist festzuhalten, daß es nicht gelingt, die Performance der von Bartholdi und Platzman vorgeschlagenen Sierpiński-Kurve beim Einsatz in der SFC-Heuristik deutlich zu unterbieten. An der Sierpiński-Kurve kommt man auch in anderer Hinsicht nicht vorbei: Viele verschiedene Ideen (Bartholdi-Platzman, Moscato-Norman, H-Index, Longest Edge Bisection) führen auf diese SFC.

Kurve	Random-Performance in %	Test-Performance in %
Sierpiński	34,3	42,4
Moore (geschl. Hilbert)	37,4	41,7
Wunderlich 1	38,3	44,7
6×6	39	47,2
4×4 - 1	39,5	42,7

Tabelle 5.9: SFC mit Random-Performance unter 40 %

Betrachtet man die 20 Testfälle aus der TSPLIB, können die geschlossene Hilbert-Kurve (Moore) und die hier neu dargestellte 4×4-1-Kurve in den Ergebnissen mithal-

Problem	beste Kurve	Abweichung in %
pla85900	Moore	32
pla33810	4×4-1	34,1
d18512	Sierpiński	30,5
d15112	Sierpiński	31,6
brd14051	Sierpiński	31,0
usa13509	Sierpiński	40,7
rl11849	4×4-1	47
pla7397	Sierpiński	39,2
rl5934	Wunderlich 1	49,9
rl5915	4×4-1	54,1
fnl4461	4×4-1	31,1
fl3795	4×4-1	56,5
pcb3038	Sierpiński	32,7
pr2392	Sierpiński	37,6
u2319	Moore	9,2
u2152	5×5-Peano	42,3
d2103	4×4-1	41,4
rl1889	Moore	55,5
u1817	5×5-Peano	40
vm1748	4×4-1	41,4

Tabelle 5.10: Performance der jeweils besten SFC auf den Testfällen

ten; auf randomisierten Instanzen hingegen ist die Sierpiński-Kurve eindeutig überlegen. Da diese Instanzen aussagekräftiger sind, wird die Sierpiński-Kurve mit ihrer Performance von unter 35 % als beste SFC angesehen.

Die Vorteile der genannten Kurven bestehen neben der guten Performance in einer recht einfachen Programmierbarkeit und mit Ausnahme der 4×4-1-Kurve in der Stabilität in randomisierten Fällen. Die Gründe für diese Ergebnisse sind die stetige Anordnung ohne Punktläufe, die grobe Struktur und die vielen Richtungswechsel der Kurven. Ein zusätzlicher Vorteil der Sierpiński-Kurve liegt in der Symmetrie; ein Drehen oder Spiegeln der Kurve ändert nichts an ihrer Performance. Allerdings hat sie in puncto Laufzeit geringe Nachteile gegenüber den Quadrat-Verfeinerungen.

Die unterschiedlichen Möglichkeiten, Daten in das Einheitsquadrat zu verschieben, können deutliche Vorteile erbringen. Bei bekannter geometrischer Struktur sollte man mit der Wahl der richtigen Verschiebevariante reagieren, in anderen Fällen beide Möglichkeiten testen, sofern man mit einer Lösung nicht zufrieden ist. Wie in Kapitel sieben noch festgestellt wird, kann auf das Austesten dieser Varianten verzichtet werden, wenn die Lösung durch Verbesserungsschritte optimiert wird.

5.6.2 Bewertungsschema für weitere Kurven

Die in dieser Arbeit untersuchten Kurven stellen nur eine winzige Auswahl von möglichen SFC dar. Kann man einer Kurve ansehen, welche Performance zu erwarten ist, ohne sie implementieren zu müssen? Um diese Frage positiv beantworten zu können, wird im folgenden ein Bewertungsschema aufgebaut, das die Grundzüge beliebiger SFC, die nach dem Konstruktionsprinzip aus Kapitel drei entstehen, einschätzt und eine frühzeitige Performance-Aussage ermöglicht. Dieses Schema hält sich eng an die in 5.3 aufgestellten Vermutungen.

In der nach oben offenen Bewertung soll „1“ die bestmögliche Note sein. Da es keinen Hinweis gibt, daß eine bessere Kurve als die Sierpiński-Kurve existiert, erhält diese den Wert eins[3]. Für gewisse Eigenschaften werden den Kurven nun Maluspunkte angerechnet:

1. Da Quadrat-Verfeinerungen den Triangulierungen unterlegen sind, erhalten sie einen Malus von 0,2.

2. Falls eine Kurve über k Punkte bei l Unterobjekten läuft, wird ein Malus von $5 * k/l$ addiert.

3. Läuft ein Leitmotiv eine ganze Länge des Einheitsquadrates ab, ohne die Richtung zu ändern, ergibt dies einen Malus von je 0,1.

4. Um die Schwächen feinerer Unterteilungen nachzuvollziehen, wird die Kurve einer $k \times k$-Verfeinerung mit einem Malus von $(k-2)/10$ belegt.

Diese recht einfachen Bewertungen reichen schon aus, um eine zufriedenstellende Einschätzung einer Kurve zu erhalten, wie die Überprüfung an den bekannten SFC zeigt: Nur die 6×6- und die Peano-Kurve werden nicht hundertprozentig genau eingeordnet.

Kurve	Quadrat	Punktlauf	Richtung	Grobheit	Gesamt	P-Index
Sierpiński					1	0,957
Moore	0,2				1,2	0,979
Wunderlich 1	0,2			0,1	1,3	0,985
4×4 - 1	0,2			0,2	1,4	0,994
Wunderlich 2	0,2		0,1	0,1	1,4	1,006
4×4 - 2	0,2			0,2	1,4	1,018
6×6		0,15		0,4	1,55	0,99
5×5	0,2	0,05		0,3	1,55	1,019
Peano	0,2		0,3	0,1	1,6	1,045
3×3 - 1		0,55	0,2	0,1	1,85	1,034
3×3 - 2		0,55	0,2	0,1	1,85	1,062
5×5-Peano	0,2		0,5	0,3	2	1,103
Dreieck 1		1,25			2,25	1,121
Dreieck 2		1,25			2,25	1,132

Tabelle 5.11: Ergebnisse des Bewertungsschemas

[3] Die Schwäche einer solchen Bewertung wird sofort deutlich; da keine Kurve besser als 1 abschneiden kann, ist die Sierpiński-Kurve sozusagen „gesetzt“.

Auf eine Implementierung einer neuen raumfüllenden Kurve kann nach der Bewertung durch obiges Schema vielleicht von vornherein verzichtet werden. Außerdem ermöglicht das Schema auch die Bewertung von komplizierteren Konstrukten (z. B. einer Verkettung von Verfeinerungsvarianten), die hier nicht betrachtet wurden.

Kapitel 6

Implementierung

Eine Umsetzung der SFC-Heuristik in ein Programm fällt nicht schwer. Dieses Kapitel soll helfen, einerseits den Rahmen-Algorithmus, in den verschiedene SFC eingesetzt werden können, und andererseits die Implementierung der einzelnen Kurven nachzuvollziehen. Dazu wird ein einfaches **C**-Programm vorgestellt. Ferner werden Möglichkeiten zur Parallelisierung des Algorithmus aufgezeigt, und zum Ende des Kapitels folgt eine Betrachtung der Laufzeit des Verfahrens.

6.1 Ein C-Programm

Für die SFC-Heuristik werden folgende Programmabschnitte benötigt: das Einlesen der Daten, das Verschieben der Daten in das Einheitsquadrat, die Zuordnung eines Indexes für jede Stadt, dazu meist die Umrechnung der Daten vom Dezimalsystem in ein anderes Zahlensystem, ein Sortierprogramm und schließlich die Längenberechnung der Tour. Ein leicht nachvollziehbares und lauffähiges **C**-Programm ist dieser Arbeit als Anhang A angefügt. Bei der Erstellung des Programms wurde Wert auf gute Lesbarkeit und Verständlichkeit gelegt. Es wird keinesfalls der Anspruch erhoben, daß es sich um einen optimal programmierten Code handelt. Verbesserungen und Beschleunigungen wurden für einige Untersuchungen in den folgenden Unterkapitel eingebaut. Diese vorzustellen, würde hier aber zu weit führen. Die nachfolgenden Abschnitte geben daher nur einen Einblick in die Implementierung des „einfachen Programms“.

6.1.1 Hauptprogramm

Der Ablauf des Hauptprogramms sieht so aus: Zunächst liest das Programm die Daten aus einer angegebenen Datei ein, in der in jeder Zeile ein Numerierungsindex (von 1 bis n) sowie die `x`- und `y`-Koordinaten einer Stadt stehen. Für die realen Testfälle aus der TSPLIB [48] können die dort hinterlegten Dateien benutzt werden. Es müssen lediglich die Textzeilen zu Beginn dieser Dateien gelöscht werden, da die Einleseroutine nur Zahlen erwartet.

Danach wird die Prozedur „Verschiebung in das Einheitsquadrat“ für den kompletten Datensatz aufgerufen (siehe 6.1.3). Es folgt für jede Stadt einzeln der Aufruf der Indexbestimmung durch die raumfüllende Kurve (siehe 6.1.4). Dieser Index wird im Vektor `neux` hinterlegt, welcher im darauffolgenden Schritt sortiert wird. Die Sortierung wird im Numerierungsvektor `z` nachvollzogen (siehe 6.1.5), wodurch die Längenbestimmung, die das Programm abschließt, erleichtert wird. Die nachfolgenden Abschnitte gehen auf einzelne Programmteile näher ein.

6.1.2 Variablendeklaration

Die Koordinaten der Städte werden in einem `x`- und einem `y`-Vektor abgespeichert. Zusätzlich braucht man einen `z`-Vektor, der der Reihenfolgebestimmung dient.

Nach der Einpassung in das Einheitsquadrat werden die neuen Koordinaten in die Vektoren `neux` und `neuy` abgespeichert. Die durch den Aufruf der raumfüllenden Kurve errechneten Indizes werden ebenfalls im `neux`-Vektor abgelegt.

Die Anzahl der Städte ist `n`, die Tiefe der Kurve wird unter dem Namen `m` gespeichert. Eine Hilfsvariable `d` bezeichnet die Art der benutzten Distanzfunktion. Für die realen Testfälle wird neben der euklidischen Distanz auch die nach oben gerundete euklidische Distanz benötigt. Unter der Variablen `v` wird die gewünschte Variante der Datenverschiebung vermerkt.

Schließlich ist es für die allermeisten Kurven hilfreich, die Koordinaten aus dem Dezimalsystem in ein anderes Zahlensystem, z. B. das Binärsystem, zu überführen, wie im Abschnitt 6.1.4 zu ersehen ist. Dafür werden in dieser Implementierung die Vektoren `bitx` und `bity` mit der Länge m (entsprechend der Tiefe der Kurve) bereitgestellt.

6.1.3 Datenverschiebung

Wie bereits mehrfach angesprochen, ist die Verschiebung der Koordinaten in das Einheitsquadrat eine wichtige Routine innerhalb eines SFC-Algorithmus. Es gibt zwei Arten, wie diese Verschiebung stattfinden kann:

1. Die Verschiebung findet in x- und y-Richtung unabhängig statt: Für beide Koordinatensätze wird jeweils ein Minimum und ein Maximum bestimmt. Die Koordinaten jeder Stadt werden umgeformt zu `neux[i]=(x[i]-minx)/(maxx-minx)` bzw. zu `neuy[i]=(y[i]-miny)/(maxy-miny)`, d. h. in beide Richtungen wird das jeweilige Intervall [`min`, `max`] in das Einheitsintervall überführt. Das Einheitsquadrat wird damit in voller Höhe und Breite ausgenutzt.

2. Die Verschiebung findet gleichmäßig statt, d. h. nur ein Skalierungsfaktor wird für beide Koordinatensätze gebraucht. Dies geschieht z. B. so: Nach der Berechnung von `minx, maxx, miny` und `maxy` wird geprüft, welches Koordinatenintervall größer ist. Ist dies beispielsweise das Intervall [`minx`, `maxx`], so werden `miny` und `maxy` derart verändert, daß das Intervall [`miny`, `maxy`] dieselbe Länge hat. Danach findet die gleiche Umrechnung statt wie unter (1). Im beschriebenen Fall würde das dazu führen, daß das Einheitsquadrat in voller Breite, nicht aber in voller Höhe genutzt wird, da nun `miny` kleiner ist als die kleinste `y`-Koordinate und `maxy` größer als die größte `y`-Koordinate. Die Geometrie der Punktanordnung bleibt damit erhalten.

Zu den Vor- und Nachteilen dieser Verschiebungsvarianten und den daraus erzielten Ergebnissen sei auf die entsprechenden Kapitel dieser Arbeit verwiesen.

In den randomisierten Instanzen kann die Datenerstellung und -verschiebung z. B. folgermaßen ablaufen: Mit Hilfe der `randomize`-Funktion in **C** werden eine ganzzahlige Zufallszahl zwischen 0 und 9.999 und eine weitere ganzzahlige Zufallszahl zwischen 0 und 10.000 erstellt. Die erste Zahl wird durch 10.000 geteilt, die zweite durch 100.000.000. Schließlich werden beide Zahlen addiert, und man erhält so eine Zufallszahl zwischen 0 und 1, die acht Stellen hinter dem Komma hat.

6.1.4 Implementierung einer SFC

Wenn von der Implementierung einer raumfüllenden Kurve gesprochen wird, ist eigentlich das Gegenteil gemeint, nämlich eine Art Umkehrabbildung zu einer solchen Kurve. Ziel ist es ja nicht, zu einem Punkt aus dem Einheitsintervall einen Kurvenpunkt im Einheitsquadrat zu finden, sondern jedem Datenpunkt aus dem Einheitsquadrat einen Urbildpunkt bzw. Urbild-Index zuzuordnen.

Um dies zu erreichen, wird die rekursive Struktur einer regelmäßigen Verfeinerung genutzt. Für alle so erzeugten Kurven fällt die Indexzuordnung nicht schwer. Das Vorgehen ist immer gleich: Nur das Leitmotiv und gegebenenfalls eine vom Leitmotiv abweichende Anfangstiefe 2 der Kurve werden der Untersuchung unterworfen; danach reicht ein rekursiver Aufruf der Routine zum Errechnen des Indexes in Tiefe m.

Wie wir noch sehen werden, erweist es sich in vielen Fällen als vorteilhaft, wenn die Koordinaten in `neux` und `neuy` zuvor in ein anderes Zahlensystem umgerechnet werden, welches der Verfeinerungsstruktur entspricht, d. h. ins Binärsystem für $2{\times}2$-Verfeinerungen usw. Die Koordinaten müssen dann meist nicht an die Routine weitergegeben werden. Eine solche Darstellung in m Stellen zur Basis b erhält man, wenn man die Zahl mit b^m multipliziert und sukzessive für $i = 1, \ldots, m$ durch b^{m-i} teilt und danach b^{m-i} subtrahiert. Der sich jeweils in der Division ergebende ganzzahlige Anteil wird im Vektor `bitx` bzw. `bity` an den Stellen $m-1$ bis 0 abgespeichert. Die Zahl 1 wird als $0{,}(b-1)(b-1)\ldots_b$ notiert.

Zurück zur raumfüllenden Kurve: Betrachtet wird zunächst der einfachste Fall, eine Quadrat-Verfeinerung. Jede regelmäßige Quadrat-Verfeinerung kann mit einer einfachen Fallunterscheidung behandelt werden. Im Beispiel der Peano-Kurve wird in einem ersten Schritt geprüft, in welchem der neun Unterquadrate aus Tiefe 2 der betrachtete Punkt liegt. Dieser bekommt dann seinen Index aus Tiefe 2 (zwischen 0 und 8) zugeordnet. Die o. g. Überprüfung geschieht am einfachsten durch eine `switch-case`-Anweisung oder eine artverwandte. Dafür wird die Zahlendarstellung von `neux` und `neuy` im Trialsystem verwendet: Die Möglichkeiten 0, 1 und 2 der führenden Ziffern, die in `bitx[m-1]` und `bity[m-1]` abgelegt sind, ergeben kombiniert die neun Indexvarianten für die Tiefe 2 der Peano-Kurve.

Dank der regelmäßigen Verfeinerung kann man aber nun folgendes feststellen: In jedem der gerade erzeugten Unterquadrate läuft das gleiche Verfahren für Tiefe 3 und die nächste Stelle in `bitx` bzw. `bity` ab. Denn dieses Unterquadrat wird seinerseits wieder in neun Unterquadrate aufgeteilt, in die der Punkt durch Abfragen der Werte von `bitx` resp. `bity` eingeordnet werden kann. Multipliziert man nun den Index der Tiefe 2 mit 9 und addiert man den neuen Index aus der feineren Tiefe 3 (wieder zwischen 0 und 8) hinzu, erhält man den Gesamt-Index für Tiefe 3 usw.

Nur noch ein kleiner Haken ist zu beachten: Nicht in jedem Unterquadrat des Peano-Leitmotivs findet *genau* das gleiche Verfahren statt. Denn nicht jedes Unterquadrat wird „unten links“ betreten und „oben rechts“ verlassen. Zur Behandlung dieses Problems gibt es zwei Möglichkeiten: Durch Manipulation der Daten kann jedes Quadrat so gedreht und in anderen Fällen auch gespiegelt werden, daß es wie das Leitmotiv aussieht. Solche Manipulationen sind manchmal etwas kryptisch, haben aber den großen Vorteil, daß nur eine `switch-case`-Anweisung programmiert werden muß; der Code ist dann sehr kurz. Zusätzlich zu obigen Informationen müssen dann entweder noch die Daten über Eingangs- und Ausgangspunkt des jeweiligen Unterquadrates in der Rekursion weitergeben werden, wie es bei der Moore-Kurve in Anhang A geschieht, oder jedes Unterquadrat wird grundsätzlich in das Quadrat transformiert, das den Ursprung enthält.

Die andere Variante ist programmtechnisch etwas umfangreicher, dafür aber sehr leicht zu programmieren und zu verstehen. Jede gedrehte oder gespiegelte Variante des Leitmotivs bekommt eine Kennziffer, die zusammen mit dem Index weitergegeben wird. Im Fall der Peano-Kurve heißt dies, daß die Indizes 0, 2, 6 und 8 die gleiche Kennziffer erhalten, da dort in der nächsthöheren Tiefe die Indizierung gleichartig ist, wie durch den Pfeil in Abbildung 6.1 dargestellt. Die Indizes 1 und 7, 3 und 5, sowie die 4 bekommen jeweils eine andere Kennziffer. In der Routine ist für jede Kennziffer eine eigene `switch-case`-Unterscheidung hinterlegt, was den Code gerade bei komplizierten 5×5-Verfeinerungen sehr lang macht.

2	3	8
1	4	7
0	5	6

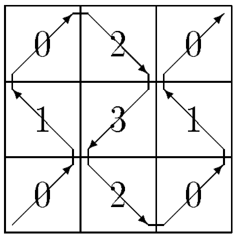

Abbildung 6.1: Peano-Kurve: Index (li.) und Rekursionskennziffer (re.)

Zu beachten ist, daß bei vielen Kurven die Tiefe 2 nicht mit dem Leitmotiv übereinstimmt, insbesondere dann, wenn eine geschlossene Kurve angestrebt wird, das Leitmotiv aber nicht geschlossen ist (Hilbert-Kurve). Die Tiefe 2 muß daher häufig separat programmiert werden; die Rekursion beginnt erst danach.

Für einfache Kurven wie die Hilbert- oder insbesondere die Lebesgue-Kurve bieten sich noch simplere Indexzuordnungen an. In *jeder* Tiefe k der Lebesgue-Kurve ergibt sich der Index zu `2*bitx[k-1]+bity[k-1]`, ohne Fallunterscheidungen oder Kennziffern. Ähnliche, nur leicht kompliziertere Formeln lassen sich auch für die Moore- oder die Peano-Kurve angeben.

Durch solche Arten der Implementierung wird auch ein Problem umgangen, das bereits kurz angeschnitten wurde: Punkten, die am Rand von Unterquadraten liegen, könnten theoretisch mehrere benachbarte Indizes zugeordnet werden. Manche Implementierungen der gleichen Kurve unterscheiden sich genau hier. Nach obigem Verfahren ist die Umrechnung in das zu benutzende Zahlensystem entscheidend. Der Wert 1/2 zum Beispiel ließe sich im Binärsystem als $0{,}1_2$ oder als $0{,}0111\ldots_2$ darstellen und würde in diesem Fall unterschiedliche Indizes zugeordnet bekommen. Da wir aber nur eine endliche Tiefe betrachten, ist nur die erste Darstellung möglich, welche infolgedessen auch benutzt wird. Daher erhält jeder Punkt eine *eindeutige* Zuordnung, auch wenn er mehrere Urbilder hat.

Bei Triangulierungen ist die Sache leider nicht so einfach. Eine `switch-case`-Fallunterscheidung der `bitx`- und `bity`-Komponenten ist noch nicht genau genug, da nun Dreiecke zugeordnet werden müssen. Eine Möglichkeit des Umgangs damit ist die Überweisung von `neux` und `neuy` in die Routine, so daß innerhalb der Fallunterscheidung dann zusätzlich eine Abfrage wie etwa `neuy>neux?` gemacht werden kann. Die Werte in `neux` und `neuy` müssen dabei in jeder Tiefe dem gerade betrachteten Dreieck angepaßt werden. Ein Beispiel dazu ist dem Anhang A zu entnehmen, da die dort benutzte Sierpiński-Kurve auf solche Art behandelt wird.

Wie aus den Kurvendarstellungen in Kapitel drei zu ersehen ist, ist es bei Triangulierungen kaum umgehbar, verschiedene Leitmotive zu nutzen, da die Eingangs- und Ausgangspunkte eines Dreiecks sowohl an der Hypotenuse als auch an einer Kathete liegen können. Dies erzwingt automatisch eine andere Indexzuordnung und damit mehrere Leitmotive. Ausnahmen sind z. B. die 6×6-Kurve und natürlich die Sierpiński-Kurve. Hier liegen Eingangs- und Ausgangspunkt immer nur an der Hypotenuse.

Da die Sierpiński-Kurve durch Longest Edge Bisection entsteht, sind für sie noch ganz andere Implementierungsmöglichkeiten gegeben. Für diese Arbeit wurden insgesamt fünf (!) verschiedene Implementierungen benutzt. Die vielleicht schönste davon geht auf Bartholdi und Platzman zurück und kann in [46] nachgelesen werden.

Wenn Triangulierungen schon eine erhebliche Aufwandserhöhung bedeuten, muß man davon ausgehen, daß kompliziertere Verfeinerungen noch schlechter handhabbar sind, was sich praktisch bestätigt. Einerseits fällt es sehr schwer, ein beliebiges geometrisches Objekt regelmäßig zu unterteilen, andererseits ist eine Indexzuordnung für ein kompliziertes Objekt entsprechend aufwendiger. Da es in der SFC-Heuristik um Schnelligkeit geht, können solche Verfeinerungen keinen Vorteil erbringen.

6.1.5 Sortierschritt

Als Sortierverfahren stehen viele sehr gute Möglichkeiten zur Verfügung. In dem in Anhang A dargestellten Code ist ein Quicksort-Verfahren angewendet worden, das sich an einen Code von Kernighan und Ritchie anlehnt [33]. Quicksort ist laut Theorie nicht das schnellste Sortierverfahren [34], erzielt praktisch aber gerade für randomisierte Daten recht gute Laufzeitwerte. In Kapitel 6.2 wird dargestellt, wie ein Sortierschritt parallelisiert werden kann, was erheblichen Einfluß auf die Laufzeit des Algorithmus hat, da das Sortieren die zeitkritische Aktion ist.

Kurz zum Wesen des Quicksort-Verfahrens: In jedem Aufruf wird eine gewisse Stelle, in diesem Fall die Vektor-Mitte, markiert. Eine Vergleichsoperation ordnet alle anderen Vektoreinträge oberhalb oder unterhalb dieser Stelle an, je nachdem, ob die Einträge größer oder kleiner als der Eintrag an der markierten Stelle sind. Nach einem Durchlauf ist der Eintrag der Marke richtig positioniert. Auf beiden Seiten der markierten Stelle wird nun das gleiche Verfahren für die kleiner gewordenen Intervalle rekursiv erneut aufgerufen, so daß die Einträge nun mit weiteren Werten verglichen werden. Dies wird so lange durchgeführt, bis nur noch ein Wert verglichen werden muß. Da jede markierte Stelle in ihrem Durchlauf an die richtige Position gerückt wird, ist das Verfahren beendet.

In der hier benutzten Implementierung wird die Sortierung der Indizes im Vektor `neux` im Numerierungsvektor `z` nachvollzogen. Dies hat den Vorteil, daß die Inhalte der Koordinatenvektoren `x` und `y` nicht bewegt werden müssen. Nach der Sortierung gibt `z[i]-1` die Position an, an der die Koordinaten der an `i`-ter Stelle eingeordneten Stadt stehen, $i = 0, \ldots, n-1$.

Das Quicksort-Verfahren hat eine Worst-Case-Laufzeit von $\mathcal{O}(n^2)$ [34]. Diese kann z. B. bei vorsortierten Vektoren auftreten. Sollten die Daten des betrachteten TSP vorsortiert sein, müßte der Anwender ein anderes Sortierverfahren wählen. Zu den Vor- und Nachteilen anderer Verfahren sei auf Knuth verwiesen [34].

Die wichtigsten Programmteile sind damit vorgestellt. Für die Erstellung eines Programms ist noch anzumerken, daß bei der Längenberechnung das Schließen der Tour nicht vergessen werden darf. Alle weiteren Komponenten des Codes in Anhang A sollten leicht verständlich sein.

6.2 Parallelisierung

Ein Programm zu parallelisieren bedeutet, p Prozessoren gleichzeitig an diesem Programm arbeiten zu lassen. Dabei wird im folgenden eine ***SPMD***-Parallelisierung für randomisierte Instanzen des TSP verfolgt. SPMD steht für Single Program Multiple Data, d. h. auf jedem Prozessor wird das gleiche Programm gestartet, welches individuell seine Daten verarbeitet [21]. Grundsätzlich können die Prozessoren entweder auf alle Daten zugreifen (***Shared Memory***) oder haben, wie im vorliegenden Fall, einen lokalen Speicher, in dem nur ein Teil der Daten liegt (***Distributed Memory***). Unter letztgenannten Voraussetzungen müssen die Prozessoren notwendigerweise miteinander kommunizieren, wenn am Ende ein Gesamtergebnis für alle Daten vorliegen soll. Solche Kommunikationen werden z. B. durch das ***Message Passing Interface (MPI)*** ermöglicht, welches hier eingesetzt wird. Für alle in diesem Unterkapitel eingeführten Definitionen wird auf Haase [21] verwiesen.

Ziel der Parallelisierung ist es, bei einem Einsatz von p Prozessoren nur noch $1/p$ der Laufzeit des Ein-Prozessor-Falls zu benötigen. Man sagt dann, der ***Speed-up*** als Quotient der Laufzeiten im Ein- und im p-Prozessor-Fall betrage p. Tatsächlich ist dies aber eine unerreichbare Idealgröße, da jedes Programm auch einen seriellen, nicht parallelisierbaren, Anteil hat (vgl. Amdahlsches Gesetz [21]). Hinzu kommen unter Umständen noch zeitaufwendige Kommunikationen zwischen den Prozessoren. Gerade bei wachsender Prozessorenzahl ist daher von einer sinkenden ***parallelen Effizienz*** (Speed-up/p) auszugehen.

Ein weiterer wichtiger Begriff in diesem Zusammenhang ist der ***Scale-up***: Wenn ein Prozessor in einer bestimmten Zeit eine Problemgröße n bearbeitet, beschreibt der Scale-up den Zuwachs an Problemgröße, wenn p Prozessoren genauso lange arbeiten [21]. Auch hier wäre ein Wert von p ideal.

Große Probleme beim Parallelisieren macht häufig die Auslastung der Prozessoren. Es kann nur dann eine gute parallele Effizienz erreicht werden, wenn jeder in etwa gleich viel arbeitet. Das Verteilen der Arbeit auf die Prozessoren (***Load Balancing***), z. B. mit Hilfe von raumfüllenden Kurven ([17], [45], [56]), ist daher von großem Interesse. In unserem Fall stellt sich dieses Problem als zweitrangig heraus, wie noch zu sehen sein wird.

Wie können Prozessoren miteinander kommunizieren? MPI ermöglicht das *Mes-*

sage Passing durch die Befehle `MPI_Send` bzw. `MPI_Recv`, mit denen ein Prozessor eine angegebene Menge von Daten sendet bzw. von einem Prozessor eine angegebene Menge von Daten empfängt. Jeder dieser Befehle muß ein eindeutiges Pendant haben. Eine ausführliche Einführung in MPI ist an dieser Stelle nicht möglich. Zusätzliche Hinweise können z. B. der MPI-Homepage [39] oder dem Buch von Gropp et al. [20] entnommen werden.

Die Parallelisierung des vorliegenden Algorithmus fällt in weiten Teilen relativ leicht: Die Daten werden gleichmäßig auf alle Prozessoren verteilt; das Load Balancing ist also (zunächst) kein Problem. Prozessor k mit $k = 0, \ldots, p-1$ behandelt die Knoten $(n/p) * k$ bis $(n/p) * (k+1) - 1$. Das Erstellen und Transformieren der randomisierten Daten findet ohne gravierende Änderung statt; es muß allerdings beachtet werden, daß jeder Prozessor unterschiedliche und unabhängige Randomzahlen erzeugt. Jeder Prozessor muß statt n Knoten nun lediglich n/p Knoten bearbeiten. Nach diesem Vorgang beginnt die Messung der Laufzeit der eigentlichen Heuristik. Dazu werden alle Prozessoren mit Hilfe eines `MPI_Barrier`-Befehls so lange angehalten, bis auch der letzte Prozessor an dieser Stelle angekommen ist. Eine derartige Synchronisation der Prozessoren ist auch nach anderen Programmteilen empfehlenswert. Der Aufruf der Indexbestimmung findet wiederum für die n/p Knoten eines Prozessors unverändert statt.

Eine erhebliche Änderung des ursprünglichen Programms ist erst im Sortierschritt erforderlich. Ein rein lokales Sortieren der Indizes hätte keinen Sinn, da eine globale Ordnung gesucht ist. Die Prozessoren müssen also in Kontakt miteinander treten. Eine mögliche Umsetzung des Sortierschritts, in Anlehnung an ein einstufiges *Bucketsort*-Verfahren, sieht so aus: Sei max der größtmögliche Index, der einem Knoten zugeordnet werden kann. Wenn nun jeder Prozessor k die Knoten mit Indizes zwischen $k * (max/p)$ und $(k + 1) * (max/p) - 1$ lokal sortiert, ergibt sich fortlaufend eine richtige globale Ordnung. Die betreffenden Knoten sind aber noch über alle lokalen Speicher verteilt und keineswegs dem „richtigen“ Prozessor zugeteilt.

Um dies zu erreichen, legt jeder Prozessor jeweils p Eimer (*Buckets*) an, in denen er die Knoten sammelt, die an den gleichen Prozessor geschickt werden müssen. Das Errechnen der Eimernummer für jeden Knoten erfolgt in linearer Zeit in Form einer Schleife über alle lokale Knoten. Wenn alle Knoten abgearbeitet und die Eimer gefüllt sind, teilen sich die Prozessoren untereinander die jeweilige Anzahl der Knoten mit, die sie verschicken müssen, damit alle Knoten tatsächlich beim richtigen Prozessor ankommen. Danach ist jedem Prozessor bekannt, wieviele Knoten er an Prozessor k verschickt und wieviele er im Gegenzug bekommt. Diese Informationen reichen aus, um schließlich den Transfer der Knotendaten (Index, x- und y-Koordinaten) durchzuführen. Dabei ist zu beachten, daß eine geschickte Reihenfolge des Transfers gewählt wird, damit kein Prozessor unnötig auf Daten warten muß. So sollte im Schritt $i = 1, \ldots, p-1$ jeder Prozessor k an Prozessor $k + i \pmod p$ senden und von Prozessor $k - i + p \pmod p$ empfangen. Jeder Prozessor muß dann noch die neuen Knoten durchnumerieren.

Das bis dato perfekte Load Balancing kann hier ein wenig beeinträchtigt werden, da in diesem Schritt ein Prozessor mehr Arbeit erhalten kann als ein anderer. In den randomisierten Instanzen ist die erwartete Anzahl der zu bearbeitenden Knoten allerdings weiterhin gleich.

Die oben beschriebenen Schritte erfolgen in linearer Zeit. Die Kosten des Setups jeder Kommunikation und auch des Transportes der Daten sind dennoch nicht unerheblich. Bei zu hoher Prozessoranzahl lohnt sich die Parallelisierung nicht mehr, da der Aufwand für Kommunikation und Transport höher ist als die Einsparung durch Arbeitsteilung im Rest des Algorithmus, wie im folgenden Unterkapitel zu ersehen ist.

Jeder Prozessor kann nun in das eigentliche (lokale) Sortieren einsteigen. Hierfür kann z. B. wieder das bereits beschriebene Quicksort-Verfahren eingesetzt werden. Die darauffolgende Berechnung der Gesamtlänge der Tour ist fast ohne Eingriff möglich. Nach lokaler Berechnung der Länge fehlen nur die Verbindungsstücke zwischen den Randknoten zweier aufeinanderfolgender Prozessoren. Diese können in die Berechnung einbezogen werden, indem jeder Prozessor k seinem Vorgänger $k-1$ den nach der Sortierung *ersten* Datensatz sendet (Prozessor 0 sendet dabei an Prozessor p). Besagter Vorgänger speichert diese Daten als *letzten* Datensatz ab. Jeder Prozessor kann damit ein Verbindungsstück berechnen.

Mit Hilfe des `MPI_Reduce`-Summenbefehls können alle lokalen Längen aufaddiert werden. Der `Reduce`-Befehl hat außerdem die Wirkung, daß erneut auf den langsamsten Prozessor gewartet wird. Somit ist auch eine Zeitnahme möglich, und der parallele Algorithmus ist beendet.

Der Gesamtaufwand der Kommunikation je Prozessor, A_{par}, läßt sich folgendermaßen darstellen:

$$A_{par}(n,p) = c_1 \cdot p \cdot A_1 + c_2 \cdot \frac{n \cdot (p-1)}{p^2} \cdot A_2, \tag{6.1}$$

wobei A_1 den Aufwand für einen Set-up der Kommunikation (es finden je Prozessor $\mathcal{O}(p)$ `MPI_Send`-Aufrufe statt) und A_2 den Aufwand für das Senden eines Datums beschreibt. Dabei versendet jeder Prozessor alle seine $\mathcal{O}(n/p)$ Daten, mit Ausnahme derer, die er selbst weiterverarbeitet. Dies sind $\mathcal{O}(n/p^2)$. Insgesamt werden daher $\mathcal{O}(n/p - n/p^2)$ Daten verschickt. Konstante Terme, wie sie z. B. bei der Kommunikation für die Längenbestimmung auftreten, sind nicht berücksichtigt. In der hier benutzten Implementierung ergeben sich c_1 zu 4 und c_2 zu 3, wobei es sich dabei um Sendevorgänge im `double`-Format handelt.

Zusammen mit dem Aufwand A_{heu} für die Heuristik, der je Prozessor

$$A_{heu}(n,p) = c_3 \cdot \frac{n}{p} \cdot \log \frac{n}{p} + c_4 \cdot \frac{n}{p} \tag{6.2}$$

beträgt (auch hier sind konstante Terme nicht berücksichtigt), ergibt sich der Gesamtaufwand des parallelen Programms. Wie sich dies numerisch widerspiegelt, wird im nächsten Unterkapitel untersucht.

6.3 Laufzeiten

In diesem Unterkapitel wird die Laufzeit der SFC-Heuristik vom Beginn des Aufrufs der SFC bis einschließlich der Längenbestimmung bei Einsatz der beiden besten Kurven dargestellt und analysiert. Der Einfluß der Parallelisierung auf die Rechenzeit wird ebenfalls herausgearbeitet.

Die Rechnungen wurden durchgeführt auf einem ***Cray***-Rechner des Neumann Institute for Computing des Forschungszentrums Jülich. Es handelt sich dabei um einen

T3e-1200-Parallelrechner (DEC/COMPAQ Alpha 21164) mit je 600 MHz Prozessorleistung und 512 MB RAM pro Prozessor. Die Datenübertragung bei Parallelisierung beträgt maximal 500 MB pro Sekunde in alle drei Dimensionen. Wie bereits erwähnt, wurde ein C-Programm und für die Parallelisierung MPI benutzt. Das Betriebssystem der Cray ist Unicos/MK.

6.3.1 Ein-Prozessor-Fall

Die Laufzeiten im Ein-Prozessor-Fall bei Einsatz der Moore- bzw. der Sierpiński-Kurve sind den Tabellen 6.1 und 6.2 zu entnehmen. In der Spalte „Quotient1“ ist für n Städte der Quotient zwischen der Laufzeit bei n und bei $n/2$ Städten vermerkt, entsprechend in „Quotient2“ für die Sortierzeit.

Städteanzahl/Tsd.	Laufzeit/Sek.	Quotient1	Sortierzeit/Sek.	Quotient2
100	0,9255	–	0,15485	–
200	1,8253	1,972	0,31888	2,059
400	3,6719	2,012	0,68793	2,157
800	7,4206	2,021	1,48067	2,152
1.600	15,047	2,028	3,20433	2,164
3.200	31,098	2,067	6,706	2,093
6.400	64,796	2,084	14,9256	2,226
12.800	133,41	2,059	32,5734	2,182

Tabelle 6.1: Ein-Prozessor-Laufzeit bei Einsatz der Moore-Kurve

Städteanzahl/Tsd.	Laufzeit/Sek.	Quotient1	Sortierzeit/Sek.	Quotient2
100	1,0536	–	0,15296	–
200	2,0975	1,991	0,33029	2,159
400	4,2178	2,011	0,7097	2,149
800	8,532	2,023	1,54149	2,172
1.600	17,17	2,012	3,22625	2,093
3.200	35,394	2,061	6,67244	2,068
6.400	74,019	2,091	15,3943	2,307
12.800	151	2,04	32,5713	2,116

Tabelle 6.2: Ein-Prozessor-Laufzeit bei Einsatz der Sierpiński-Kurve

Zwischen den Ergebnissen der beiden Kurven ist kaum ein Unterschied festzustellen, wenn man davon absieht, daß die Moore-Kurve geringfügig schneller ist als die Sierpiński-Kurve. Dieser Effekt verliert sich für steigendes n etwas. Dies liegt daran, daß die im Vergleich zu den restlichen Programmteilen größere Komplexität des Sortierschrittes an Bedeutung zunimmt. Für 100.000 Städte nimmt er nur 16,7 % der Gesamtlaufzeit bei Einsatz der Moore-Kurve ein, für 12.800.000 Städte immerhin 24,4 %. Die Laufzeit des Sortierschrittes steigt nämlich wie erwartet in höherer als linearer Form an, wie man der Spalte „Quotient2“ entnehmen kann. Ein Quotient von 2 entspricht einer linearen Laufzeit. Im Gesamtalgorithmus ist dieser Einfluß ebenfalls, wenn auch zunächst gedämpft, wahrnehmbar (Spalte „Quotient1“). Für weniger als 1.000.000 Städte ist der lineare Teil der Heuristik, hier insbesondere die Indexbestimmung mittels der raumfüllenden Kurve, so zeitbestimmend, daß er die eigentliche $\mathcal{O}(n \cdot \log\, n)$-Komplexität überlagert. Dies bedeutet jedoch gleichzeitig, daß die SFC-Heuristik ihre Zeitvorteile gegenüber anderen Heuristiken erst bei großen Städtezahlen vollständig ausspielt.

In obigen Tabellen sind andere Kurven als die Sierpiński- und die Moore-Kurve nicht dargestellt; sie wurden selbstverständlich dennoch untersucht. Folgende Informationen sind für alle Kurven zusammenzufassen:

- Quadrat-Verfeinerungen haben einen Zeit-Vorteil gegenüber den aufwendiger zu programmierenden Triangulierungen. Kompliziertere Verfeinerungen sind aufgrund weiterer Laufzeitverschlechterungen abzulehnen.
- Feinere Unterteilungen haben Zeit-Nachteile gegenüber gröberen.
- Die vorgegebene Tiefe m hat einen direkten Einfluß auf die Laufzeit. Die für den Aufruf der SFC benötigte Zeit ist linear in m. Für kleinere Problemgrößen kann durch Verringerung der Tiefe erheblich Zeit eingespart werden. Für die Moore- und die Sierpiński-Kurve wird empfohlen, ab 1.000.000 Städten mit einer Tiefe von mindestens 15 zu rechnen, für 100.000 Städte reicht erfahrungsgemäß eine Wahl von $m = 12$ aus. Gegenüber einer Wahl von $m = 15$ spart man so 20 % Rechenzeit für den Aufruf der SFC. Analoges gilt, mit entsprechend angepaßtem m, für die anderen Kurven.

6.3.2 Mehr-Prozessor-Fall

Die SFC-Heuristik erweist sich als ausgezeichnet parallelisierbare Heuristik. In Tabelle 6.3 sind die Laufzeiten bei Einsatz der Sierpiński-Kurve und Nutzung von bis zu 32 Prozessoren aufgeführt. Die geringfügige Abweichung der Daten zum Ein-Prozessor-Fall in Tabelle 6.2 geht auf hier hinzukommende MPI-Aktionen zurück.

Prozessoranzahl	1	2	4	8	16	32
Städteanzahl/Tsd.						
100	1,093	0,602	0,352	0,2	0,153	0,121
200	2,266	1,13	0,59	0,325	0,198	0,129
400	4,4	2,256	1,125	0,609	0,334	0,217
800	8,713	4,476	2,281	1,131	0,599	0,332
1.600	17,63	8,863	4,474	2,252	1,149	0,607
3.200	35,96	18,44	8,915	4,49	2,241	1,157
6.400		37,33	18,13	9,037	4,54	2,259
12.800			37,59	18,27	9,045	4,584

Tabelle 6.3: Mehr-Prozessor-Laufzeit in Sek. bei Einsatz der Sierpiński-Kurve

Prozessoranzahl	1	2	4	8	16	32
Städteanzahl/Tsd.						
100	1	1,82	3,11	5,47	7,14	9,03
200	1	2,01	3,84	6,97	11,44	17,57
400	1	1,95	3,9	7,22	13,17	20,28
800	1	1,95	3,82	7,7	14,55	26,24
1.600	1	1,99	3,94	7,83	15,34	29,04
3.200	1	1,95	4,03	8,01	16,05	31,08

Tabelle 6.4: Speed-up bei Einsatz der Sierpiński-Kurve

Zur Interpretation der Laufzeiten dient Tabelle 6.4, in der die Speed-up-Werte dargestellt sind. Speed-ups, die bei p Prozessoren über p hinausgehen, wie es zum Teil der Fall ist, sind keineswegs auf Ungenauigkeiten zurückzuführen. Je nach Architektur des Speichers ist es möglich, daß eine geringere Städteanzahl die Cache-Nutzung und somit die Laufzeit beschleunigt.

Wie erwartet, kann nicht immer ein Speed-up von p bei Einsatz von p Prozessoren erreicht werden. Für große Städtezahlen werden allerdings hervorragende Werte erreicht: Für 3.200.000 Städte beträgt der Speed-up bei 32 Prozessoren über 31. Dem Erfolg der Parallelisierung sind jedoch auch Grenzen gesetzt. Ab einer gewissen Prozessorzahl wird die Kommunikation zwischen den Prozessoren prozentual so zeitaufwendig, daß der Zeitgewinn durch die Parallelisierung nicht voll ausgeschöpft werden kann. Man beachte dazu den Aufwand zur Parallelisierung in Gleichung (6.1): Mit steigendem p fällt der lineare Term in p stärker ins Gewicht.

Dies erkennt man z. B. in der ersten Zeile der Tabelle 6.4: Für 100.000 Städte sinkt die parallele Effizienz bei Einsatz von 16 oder mehr Prozessoren unter 50 %. Hat jeder Prozessor jedoch 100.000 Städte oder mehr zu bearbeiten, wird eine parallele Effizienz zwischen 96 und 98 % erzielt. Dies läßt die Interpretation zu, daß die SFC-Heuristik für eine Parallelisierung sehr gut geeignet ist.

Betrachtet man die Diagonalen in Tabelle 6.3, kann man auch den Scale-up-Erfolg sehen: Ausgehend von z. B. einem einzelnen Prozessor, der 100.000 Städte bearbeitet, können p Prozessoren das p-fache an Städten in fast der gleichen Zeit bearbeiten. Auch dies unterstreicht die gute Parallelisierbarkeit der SFC-Heuristik.

Für noch größere Städtezahlen bestätigen sich obige Interpretationen bei massivem Einsatz von Parallelisierung ebenfalls. In Tabelle 6.5 sind die Laufzeiten für 102.400.000 Städte vermerkt, wenn bis zu 256 Prozessoren genutzt werden. Auch hier ergibt sich ein herausragender Speed-up: Bei Verachtfachung der Prozessorzahl von 32 auf 256 sinkt die Laufzeit auf *weniger* als ein Achtel der Ausgangslaufzeit.

Prozessoranzahl	32	64	128	256
Laufzeit in Sek.	38,9	19,4	9,35	4,74

Tabelle 6.5: Laufzeit bei Einsatz der Sierpiński-Kurve für 102.400.000 Städte

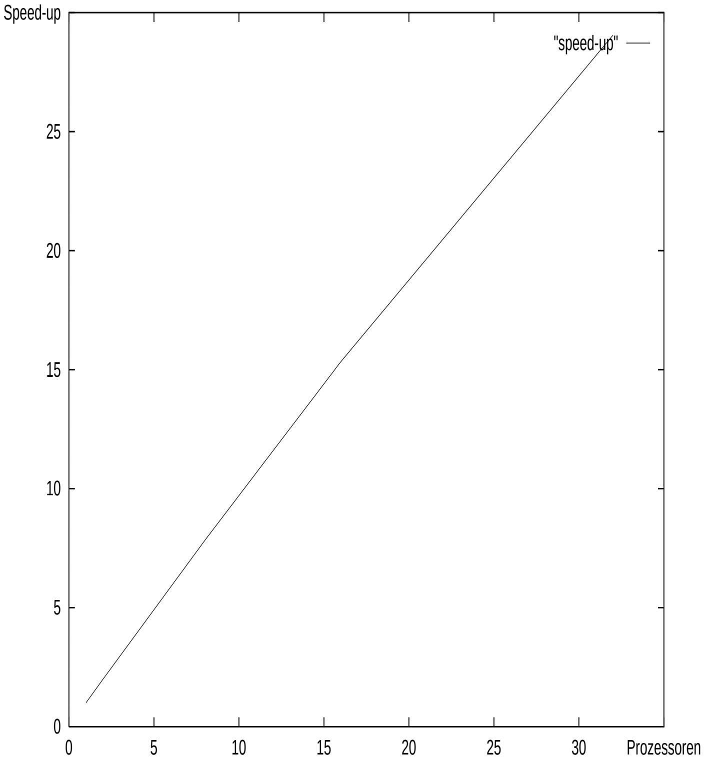

Abbildung 6.2: Speed-up bei 1.600.000 Städten

Zusammenfassend läßt sich feststellen, daß gerade bei großer Städtezahl die SFC-Heuristik ihre approximative Lösung in sehr schneller Zeit erstellt. Ein Problem von 1.000.000 Städten kann, oben genannte Rechnerleistung vorausgesetzt, in rund zehn Sekunden bearbeitet werden; bei Parallelisierung mit z. B. 16 Prozessoren erhält man eine Lösung bereits nach 75 Hunderstelsekunden.

Kapitel 7

Anwendungen der SFC-Heuristik

Welche TSP-Heuristik als approximativer Löser tatsächlich eingesetzt wird, hängt fast ausschließlich von der Ressource Zeit ab. Je mehr Zeit zur Verfügung steht, um so ausgeklügelter und besser kann eine Lösung gefunden werden, die hohen Ansprüchen bezüglich der Genauigkeit genügt. Für die SFC-Heuristik muß man jedoch zwei Einsatzgebiete unterscheiden: Da ist einmal die ausschließliche Nutzung dieser Heuristik, die in sehr schneller Zeit eine Lösungsapproximation mit 35 bis 40 % Abweichung erbringt; zum zweiten kann sie aber auch als Startschritt für Verbesserungsheuristiken verwendet werden, die je nach Zeitvorgabe eine fast beliebig gute Näherung erzielen. Auf solche Heuristiken wie z. B. Lin-Kernighan wird in Kapitel zwei ausführlich eingegangen.

Wie groß die zeitlichen oder leistungsmäßigen Unterschiede in einer Verbesserungsheuristik sind, wenn entweder mit einer recht guten (Nearest Neighbour), einer mittelschlechten (SFC) oder einer ganz schlechten (Random) Lösung gestartet wird, hängt von der Verbesserungsheuristik selbst ab und kann wegen der Fülle der Möglichkeiten hier nicht untersucht werden. Daher soll sich in diesem Kapitel das Augenmerk auf Problemstellungen richten, die in subquadratischer Laufzeit eine Lösung benötigen. Dazu wird in Unterkapitel 7.1 eine Verbesserung der SFC-Heuristik vorgestellt, die eine Lösung in linearer Laufzeit deutlich verbessert und außerdem die Zuverlässigkeit erhöht. Im darauffolgenden Unterkapitel werden schließlich einige Anwendungsgebiete erläutert.

7.1 Verbesserung der SFC-Heuristik

7.1.1 2-opt-Tausch

Das Prinzip des 2-opt-Tauschs ist in Kapitel zwei bereits dargestellt. Normalerweise arbeitet ein 2-opt-Tausch in $\mathcal{O}(n^2)$ Schritten, da jede Kante mit jeder anderen Kante verglichen wird. Bringt das Vertauschen zweier Kanten eine Tourverbesserung, so findet (unter Beachtung gewisser Regeln) ein solcher Tausch statt.

Setzt man sich zum Ziel, eine subquadratische Laufzeit zu erzielen, darf nicht jede Kante mit jeder verglichen werden. Dies wäre in einer Lösung, die durch die SFC-

Heuristik gegeben ist, auch unsinnig, da diese Heuristik Nähebeziehungen zwischen Punkten im allgemeinen gut erhält. Das bedeutet, daß ein Tausch zweier Kanten, die in der Tour weit voneinander entfernt sind, meist keine Verbesserung erbringt, da auch die inzidenten Knoten dazu tendieren, im Einheitsquadrat einen großen Abstand voneinander zu haben, und ein Tausch somit nur eine Verschlechterung erzeugen würde.

Was in der Lösung der SFC-Heuristik allerdings häufig auftritt und auch in manchen Abbildungen in dieser Arbeit zu sehen ist, sind Kreuzungen innerhalb der Tour. Eine sich kreuzende Tour kann aber leicht dadurch verbessert werden, daß sie kreuzungsfrei gemacht wird. Und da solche Kreuzungen meist durch fast benachbarte Kanten entstehen, kann ein Großteil des Verbesserungspotentials des 2-opt-Tauschs bereits durch Vergleich jeder Kante mit ihren engsten Tournachbarn ausgenutzt werden.

So verliert man in der Performance gegenüber einem echten 2-opt-Tausch nicht viel, gewinnt aber in der Laufzeit eine ganze Potenz von n, wenn statt n nur noch eine konstante Anzahl von Nachbarn untersucht wird.

Da die Laufzeit von großem Interesse ist, wurde der 2-opt-Tausch für die vorliegende Arbeit auf diese Art umgesetzt: Noch vor der Längenberechnung wird für jeden Knoten überprüft, ob ein Tausch der folgenden Kante mit einer zu den folgenden Nachbarn inzidenten Kante einen Gewinn bringt. Der größtmögliche Gewinn für jeden Knoten wird realisiert.

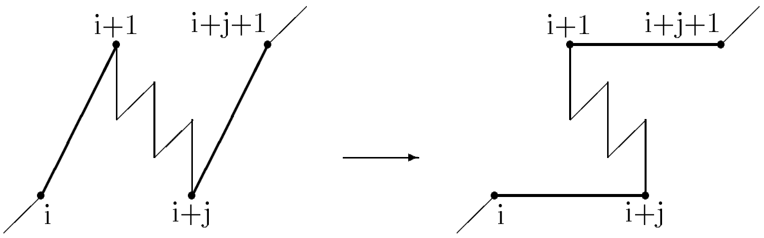

Abbildung 7.1: 2-opt-Tausch z. B. mit $j = 2, \ldots, 50$

Allein auf diese Art wird aber noch recht viel Verbesserungspotential verschenkt. Daher wird obiges Vorgehen in einer Schleife mehrfach (etwa dreimal) durchlaufen. Dies ist immer noch billiger als eine starke Erweiterung der Menge der zu überprüfenden Nachbarn. Außerdem entsteht nach jedem Tausch eine neue Situation, die vielleicht Einsparpotentiale an anderer Stelle erzeugt. Ein mehrfaches Durchlaufen ist also angebracht, wie insbesondere die numerischen Ergebnisse zeigen.

Ein k-opt-Tausch mit $k > 2$ hat bei der geringen Zahl der betrachteten Nachbarn wenig Sinn, da die Verbesserungen selten größer sind als im 2-opt-Tausch. Dafür ist der Rechenaufwand und damit die Laufzeit vergleichsweise zu hoch. Außerdem neigt ein solcher k-opt-Tausch nach eigenen Untersuchungen eher dazu, sich in einem lokalen Optimum zu verlaufen als ein mehrfach angewandter 2-opt-Tausch.

7.1.2 Numerische Ergebnisse

Benutzt man einen 2-opt-Tausch wie im vorangegangenen Unterkapitel als Verbesserungsschritt für die SFC-Heuristik, erhält man Lösungen, die nur noch um höchstens

25 % oder, bei etwas größerem Aufwand, um höchstens 20 % vom Optimum entfernt sind. Diese Abweichung ist fast nicht mehr abhängig von

- der Anzahl der Städte,
- dem Schwierigkeitsgrad der Instanz,
- der benutzten Kurve.

In diesem Sinne ist eine Kombination von SFC und 2-opt eine sehr zuverlässige und immer noch schnelle TSP-Heuristik. Im einzelnen:

Die Hauptuntersuchungen wurden auf randomisierten Instanzen von 10.000 Städten mit einer Auswahl der in Kapitel drei vorgestellten Kurven durchgeführt. Das größte Interesse galt der besten, der Sierpiński-Kurve. In Tabelle 7.1 sind einige Ergebnisse zu dieser Kurve dargestellt:

Durchläufe	Nachbarn	P-Index	Abw. in %
0	0	0,957	34,3
1	1	0,917	28,7
1	5	0,892	25,2
1	50	0,883	23,9
1	100	0,882	23,8
2	1	0,911	27,9
2	5	0,878	23,2
2	50	0,856	20,2
2	100	0,852	19,6
3	5	0,872	22,4
3	30	0,855	20
3	50	0,848	19
10	100	0,84	17,9

Tabelle 7.1: SFC-Heuristik mit Sierp.-Kurve und 2-opt-Tausch bei 10.000 Städten

Ausgehend von einem Performance-Index von 0,957, der eine Abweichung von etwa 35 % bedeutet, kann bei nur 2 Durchläufen des 2-opt-Tauschs mit je 5 Nachbarn eine Verbesserung auf 23 % (P-Index 0,878) erreicht werden. Die für den 2-opt-Tausch benötigte Zeit ist von der Anzahl der Veränderungen in der Tour und damit von der Instanz abhängig. Die Laufzeit entspricht in obigem Fall etwa der Zeit, die die SFC-Heuristik für die Indexzuteilung benötigt. Die Gesamtlaufzeit mit Sortierschritt wird also nicht einmal verdoppelt.

Daher macht in diesem Zusammenhang auch das zweifache Durchlaufen der Heuristik mit den beiden Datenverschiebungsvarianten keinen Sinn mehr, wenn ein 2-opt-Tausch angefügt wird; dessen Wirkung ist im allgemeinen größer.

Erhöht man die Anzahl der Durchläufe oder die der betrachteten Nachbarn, läßt sich das Ergebnis auf bis zu 18 % verbessern. Dafür ist allerdings ein erheblicher Aufwand notwendig. Nach zu vielen Durchläufen verirrt sich die kombinierte Heuristik in ein lokales Optimum, das mit diesen Ideen nicht weiter verbessert werden kann. Gleiches kann schon frühzeitiger geschehen, wenn zu viele Nachbarn betrachtet werden. Dies ist also keineswegs immer von Vorteil.

Der Effekt des mehrfachen Durchlaufens verringert sich stark; nach drei Durchläufen haben etwa 90 bis 98 % der Veränderungsschritte stattgefunden. Das dreifache Wiederholen des 2-opt-Tauschs kann also für eine gute Kurve als ausreichend angesehen werden. Der beste Trade-Off zwischen Laufzeit und Performance liegt bei zwei Durchläufen mit je fünf Nachbarn (23 % Performance bei knapp doppelter Laufzeit gegenüber der einfachen Heuristik). Soll die 20 %-Performance-Marke erreicht werden, müssen bei drei Durchläufen etwa 30 Nachbarn einbezogen werden, was allerdings zu einer Vervielfachung des Aufwands des 2-opt-Schritts führt.

Die obigen Zahlen werden auch von Instanzen mit $n = 100.000$ Städten genau bestätigt. Es ist insbesondere *nicht* notwendig, eine von n abhängige Zahl von Nachbarn zu betrachten, wodurch sichergestellt wird, daß der 2-opt-Tausch tatsächlich in linearer Zeit durchgeführt wird. Die Anzahl der stattfindenden Tauschoperationen ist natürlich von n abhängig und liegt zwischen $n/5$ und $n/4$, je nach gewählten Parametern und der betrachteten Instanz.

Das Bild ist bei Einsatz anderer Kurven oder auf realen Testfällen sehr ähnlich. Kurven mit nur geringfügig schlechterer Performance als die Sierpiński-Kurve erzielen fast wortgetreue Ergebnisse. Mit schlechter werdender Performance verschlechtern sich auch die 2-opt-Ergebnisse leicht. Jedoch selbst die fraktalen Kurven, die für große Instanzen eine beliebig schlechte Performance aufweisen, können bis auf 20 % Abweichung gedrückt werden; hier allerdings bei erheblich größerem Aufwand (etwa bei zehn Durchläufen mit 100 Nachbarn). Aufgrund dieses unnötig hohen Aufwandes sollten Fraktale nicht eingesetzt werden.

Durchläufe/Nachbarn	4x4-1	Moore	6x6	3x3-1	Fraktal1	Fraktal2
0/0	0,965	0,977	0,983	1,035	2,49	3,21
1/1	0,926	0,937	0,934	0,983	2,1	2,76
1/5	0,907	0,902	0,903	0,927	1,53	2,03
1/50	0,89	0,89	0,895	0,911	1,12	1,48
2/5	0,897	0,887	0,888	0,903	1,29	1,69
2/50	0,869	0,867	0,857	0,869	0,996	1,12
3/5	0,895	0,885	0,884	0,899	1,22	1,57
3/30	0,862	0,86	0,856	0,856	0,943	1,06
10/100	0,849	0,843	0,836	0,841	0,856	0,872

Tabelle 7.2: P-Indizes der SFC-Heuristik mit 2-opt-Tausch bei 10.000 Städten

Natürlich kann es bei der ein oder anderen realen Instanz immer noch passieren, daß eine Kurve deutlich besser abschneidet als eine andere oder daß die Performance doch über 25 % liegt. Die hier erreichten Ergebnisse lassen dennoch den Schluß zu, daß die SFC-Heuristik in Verbindung mit einem 2-opt-Tausch eine sehr zuverlässige und schnelle Heuristik ist. Die Frage nach der einzusetzenden Kurve verliert in diesem Zusammenhang an Gewicht. Vorteile der Sierpiński- und der Moore-Kurve bestehen wegen ihrer leichten Implementierbarkeit und guten Performance aber weiterhin.

7.2 Einsatzgebiete

Der größte Vorteil der SFC-Heuristik ist ihre Schnelligkeit. Daher bieten sich Anwendungen an, deren Instanzen so groß sind, daß andere Heuristiken sie in quadratischer Laufzeit praktisch gar nicht bewältigen können. Dazu muß jedoch angemerkt werden, daß große Travelling Salesman Probleme selten unter Zeitdruck gelöst werden müssen. Man denke z. B. an die Programmierung des ***Sternenteleskops Hubble*** zur Erfassung von 18 Millionen Sternen. Heuristiken mit quadratischer Laufzeit müssen trotz Parallelisierung mehrere Tage an diesem Problem arbeiten [49]. Dies ist aber unproblematisch, da ein solches Projekt über Jahre geplant wird. Ein einmaliges Lösen des TSP darf ruhig sehr lange dauern.

Also muß man neben den klassischen Einsatzgebieten einer schnellen Heuristik nach Anwendungen suchen, die eine besonders schnelle Berechnung (Echtzeit) oder mehrere, schnell aufeinanderfolgende Lösungen benötigen. Die „klassische“ Anwendung der SFC-Heuristik ist die ***Radarerfassung von Raketen*** [6]. In den Achtziger Jahren planten die USA eine weltraumbasierte Abwehrstation, die feindliche Raketen oder Kometenstücke anvisieren und Anweisungen zur Abwehr an die Erde senden sollte. Dieses SDI-Projekt ist bekanntlich nicht realisiert worden; seit einigen Monaten steht ein ähnliches Projekt namens NMD (*National Missile Defense*) aber wieder auf der Tagesordnung (vgl. [4]). Die Erfassung von Raketen entspricht der Lösung eines TSP: Jede Rakete muß anvisiert und markiert werden. Daß dem Anwender in diesem Fall an einer Echtzeitlösung gelegen ist, ist verständlich.

Eine weitere Weltraumanwendung ist die Erfassung ganz unmilitärischer Objekte, wie z. B. Autos, durch einen ***Satelliten.*** Ist es aus irgendeinem Grund notwendig, alle Objekte, für die der Satellit zuständig ist, nacheinander anpeilen zu müssen, ist dies wieder ein TSP. Ein solcher Fall ist denkbar, wenn der Satellit ausgefallen ist und neu initialisiert werden muß, wenn die Position aller Objekte dargestellt und mitverfolgt werden soll oder wenn an alle Teilnehmer des Systems Daten gesendet werden sollen. In letztgenanntem Fall werden zwei weitere Eigenschaften der SFC-Heuristik genutzt, die besonders wertvoll sind: Erstens ist es sehr schnell möglich, neue Objekte hinzuzufügen – man benötigt nur einen neuen Index, der eingeordnet werden muß. Noch einfacher ist das Streichen von Objekten, da dieses die Ordnung der anderen Objekte nicht verändert. Zweitens wird der Algorithmus bei wiederholter Anwendung wegen einer räumlichen Veränderung der beweglichen Objekte deutlich schneller sein, da die Daten bereits vorsortiert sind und sich dadurch die zeitkritische Aktion des Sortierens beschleunigt. Außerdem kann eine Beschleunigung des Algorithmus erreicht werden, indem der neue Index eines bewegten Objektes nicht von Grund auf neu errechnet wird. Man kann statt dessen auf frühere Indexberechnungen zurückgreifen.

Auch für sehr kleine Probleme ist die SFC-Heuristik von Interesse, z. B. wenn kein Computer zur Verfügung steht oder man diesen nicht bedienen kann. Bartholdi, Platzman, Collins und Warden haben für die täglich neu zu erstellende Route von ***„Essen auf Rädern“*** ein Karteikartensystem entwickelt, das diese Fahrtroute im Sinne der SFC-Heuristik ohne jegliche Rechnerunterstützung durch manuelles Sortieren erstellt (zitiert in [6]).

Eine weitere Anwendung ist ein klassisches TSP-Problem: das Bohren von ***Leiterplatten***, das schon in Kapitel zwei vorgestellt wurde. Müssen dort nämlich sehr viele Löcher unterschiedlicher Größe gebohrt werden und ist der Auftragsumfang nicht besonders groß, kann die SFC-Heuristik eingesetzt werden. Ebenso ist diese Heuristik in Verbindung mit einem 2-opt-Tausch für weitere klassische Einsatzgebiete von Interesse, die schon in Kapitel zwei beschrieben wurden.

Weiter könnte es Minimierungsprobleme geben, in denen das ***TSP als Nebenbedingung*** auftaucht, die immer wieder neu gelöst werden müßte. Dies würde nie exakt stattfinden, sondern nur angenähert werden. Da die Performance der SFC-Heuristik in Verbindung mit einem 2-opt-Tausch sehr stabil ist, kann man die Nebenbedingung gut abschätzen.

Immense Datenmengen fallen z. B. bei Telefongesellschaften oder im Internet an. Dem Autor dieser Arbeit ist aber leider kein Fall bekannt, wo in solchen Umgebungen Handlungsreisendenprobleme gelöst werden müssen. Der Einsatz der SFC-Heuristik wäre hier sicherlich sehr interessant.

Kapitel 8

Zusammenfassung und Ausblick

Zum Ende dieser Arbeit werden die gewonnenen Erkenntnisse zusammengefaßt. Darauf folgt eine Reihe von Anregungen für die weitere Forschung.

8.1 Zusammenfassung

Das Euklidische Travelling Salesman Problem, also die Suche nach einer kürzesten Rundfahrt durch n Städte, ist im allgemeinen nicht in annehmbarer Zeit lösbar. Eine Heuristik von Bartholdi und Platzman, die eine approximative Lösung generiert, ordnet die Städte in der Reihenfolge, in der eine raumfüllende Kurve sie besucht und heißt daher Space-Filling-Curve-Heuristik. Diese Heuristik ist leicht implementierbar und mit einer Laufzeit von $\mathcal{O}(n \cdot \log n)$ sehr schnell; sie kann außerdem parallelisiert werden, so daß die Laufzeit noch verbessert wird.

Die Performance der Heuristik hängt besonders von der benutzten raumfüllenden Kurve ab, nach der sich der Handlungsreisende richten soll. Als beste aus einer Reihe von implementierten Kurven erweisen sich die geschlossene Hilbert-(Moore-) und die Sierpiński-Kurve, die auch von Bartholdi und Platzman genutzt wird. Auf 20 realen Testfällen zwischen 1748 und 85.900 Städten erhält man Lösungen, die im Schnitt um gut 40 % vom Optimum abweichen. Hier schneidet die Moore-Kurve am besten ab.

Betrachtet man randomisierte Instanzen zwischen 10.000 und 1.400.000 Städten, erreicht die Sierpiński-Kurve die besten und außerdem sehr stabile Werte von unter 35 % Abweichung. Für den Vergleich mit der nicht bekannten optimalen Lösung einer beliebigen randomisierten Instanz wird eine Abschätzung ihrer Länge benutzt. Dies ist möglich, da in dem betrachteten Fall der Quotient aus Tourlänge zu $\sqrt{n}$ für $n \to \infty$ konstant ist. Die neuesten Abschätzungen dieser Konstante werden durch die vorliegende Arbeit bestätigt, wohingegen einigen Angaben von Bartholdi und Platzman zur Performance bestimmter Kurven auf randomisierten Fällen widersprochen werden muß.

Der Versuch, die Performance der Sierpiński- und der Moore-Kurve durch andere Kurven zu unterbieten, scheitert. Auch eine Reihe von neu erstellten Kurven ist dazu nur in Einzelfällen in der Lage. Die beiden oben genannten Kurven erfüllen alle Kriterien, die in dieser Arbeit für eine gute Performance aufgestellt werden: Sie haben eine grobe Struktur, viele Richtungswechsel im Leitmotiv, vermeiden Punktläufe, sind geschlossen und nutzen eine einfache Art der Verfeinerung. Dabei zeigt sich, daß

Triangulierungen einen leichten Vorteil gegenüber Quadratverfeinerungen haben. Andere Formen der Parzellierung (Tilings), genau wie kompliziertere Formen von Kurven (Fraktale), sind entweder aufgrund ihrer Komplexität oder ihrer Performance abzulehnen. Ein Bewertungsschema hilft, Kurven einzuschätzen, die nach gewissen Kriterien aufgebaut werden, ohne sie implementieren zu müssen.

Auch andere bekannte raumfüllende Kurven, über die schon wichtige theoretische Informationen gewonnen wurden, schneiden nicht besser ab als die Sierpiński-Kurve. Insbesondere sind die Moscato-Norman-Kurven gleich der Sierpiński-Kurve. Für den H-Index legen die numerischen Ergebnisse dieser Arbeit nahe, daß auch dieser gleich der besten raumfüllenden Kurve ist.

Deutliche Veränderungen der Performance sind außerdem durch unterschiedliche Anordnungen der Problemdaten möglich. Ein zweimaliges Durchlaufen des Algorithmus kann hier von Interesse sein; eine Verbindung der Heuristik mit einem Verbesserungsschritt wird aber bevorzugt.

Dabei bietet sich eine Kombination der SFC-Heuristik mit einem speziellen 2-opt-Tauschverfahren an, da die raumfüllende Kurve Nähebeziehungen zwischen den Städten recht gut erhält und somit viele Kandidaten für eine Verbesserung durch einen 2-opt-Tausch sehr nahe Tournachbarn sind. Dadurch kann der 2-opt-Tausch in linearer Komplexität realisiert werden und verbessert in einer Laufzeit, die geringer ist als die der SFC-Heuristik selbst, eine Lösung auf etwa 20 bis 25% Abweichung. Damit erreicht man in subquadratischer Laufzeit die gleiche oder zum Teil eine bessere Performance als andere Heuristiken in quadratischer Laufzeit (vgl. [47]).

Eine solche, hier neu vorgestellte Kombination ist für alle Anwendungen interessant, in denen bisher Heuristiken quadratischer Laufzeit benutzt werden. Eine Verbindung der SFC-Heuristik mit einem solchen 2-opt-Tausch mildert auch die hohen Ergebnisschwankungen der reinen SFC-Heuristik, die sowohl bei realen als auch bei randomisierten Fällen zu beobachten sind.

Dank der hervorragenden Parallelisierbarkeit der Heuristik approximiert sie Lösungen von Handlungsreisendenproblemen mit einer Million Städten bei Einsatz von 16 Prozessoren á 600 MHz bereits in weit unter 1 Sekunde. Für kleine Probleme sind Lösungen in (Fast-)Echtzeit möglich. Dadurch eröffnet sich ein neues Feld von Anwendungen, das bisher noch nicht bedient werden konnte.

8.2 Ausblick

Auch wenn die SFC-Heuristik in dieser Arbeit recht umfassend untersucht wurde, gibt es eventuell noch weitere Verbesserungsmöglichkeiten: Zum einen ist noch nicht abschließend geklärt, ob die Sierpiński-Kurve die tatsächlich beste raumfüllende Kurve ist, und zum anderen sollte es möglich sein, den hier dargestellten Algorithmus noch zu beschleunigen. Eine Möglichkeit liegt darin, anstelle einer Indexzuteilung mit anschließendem Sortierschritt eine Abbildung zu betrachten, die lediglich beantwortet, ob Knoten 1 vor oder nach Knoten 2 in der Tour einzuordnen ist. Diese Frage kann in vielen Fällen beantwortet werden, ohne daß für alle Knoten die gleiche (und meist hohe) Tiefe der Kurve betrachtet werden muß; dies spart Rechenzeit.

Offen ist ebenfalls, wo die kurvenabhängigen Ergebnisschwankungen der Heuristik bei randomisierten Fällen herrühren.

Nicht untersucht wurde die Frage, ob eine Lösung, die durch die SFC-Heuristik plus 2-opt-Tausch generiert wurde, Vor- oder Nachteile gegenüber anderen Lösungen hat, wenn sie als Startlösung einer (zusätzlichen) TSP-Verbesserungsheuristik eingesetzt wird. Eine allgemeine Untersuchung hätte den Rahmen dieser Arbeit gesprengt. Jedoch kann eine Beantwortung dieser Frage durchaus von Interesse für jemanden sein, der sich mit einer speziellen Verbesserungsheuristik beschäftigt.

In dieser Arbeit blieben höherdimensionale TSP unberücksichtigt. Die Heuristik von Bartholdi und Platzman kann für diese Fälle ebenfalls eingesetzt werden, da es möglich ist, auch höherdimensionale raumfüllende Kurven zu erstellen [51]. Für Triangulierungen (als Pendant zur erfolgreichen Sierpiński-Kurve) fällt dies aber wesentlich schwerer als für Quadratverfeinerungen. Die Leistung der Heuristik wird außerdem dadurch beeinträchtigt, daß es im k-dimensionalen auch k verschiedene Stetigkeitsgrade einer Kurve gibt: Im Zweidimensionalen kann sie über eine Kante oder über einen Punkt der Unterteilungsobjekte laufen, im Dreidimensionalen zusätzlich über eine Fläche usw. Die Erkenntnisse über zweidimensionale Kurven lassen vermuten, daß nur eine höchstdimensionale Nachbarschaft aufeinanderfolgender Unterteilungsobjekte eine gute Kurve generieren kann. Die Anordnung wird damit immer schwieriger.

Erkenntnisse aus Unterkapitel 5.5 könnten Einfluß auf weitere theoretische Untersuchungen haben. Da einerseits die Moscato-Norman-Kurven gleich der Sierpiński-Kurve sind und andererseits diese mit hoher Wahrscheinlichkeit gleich der H-Index-Kurve ist, ist eine Verbindung der theoretischen Ergebnisse zu MNPeano und H-Index möglich. Die optimale Näheerhaltung durch den H-Index und die Minimalität der MNPeano-Punktmengen suggerieren die Möglichkeit einer exakten Abschätzung der Konstanten C aus Gleichung (4.1), was einen lange währenden Abschätzungswettlauf beenden könnte. Der Weg dorthin ist zwar lang; eine Verbindung der oben aufgeführten Informationen war aber bis dato noch nicht möglich. Gleichzeitig könnten die Eigenschaften des H-Indexes auch die Frage nach der optimalen Kurve für die Bartholdi-Platzman-Heuristik beantworten. Ein vollständiger Beweis, daß der H-Index gleich der Sierpiński-Kurve ist, wäre also von großem Interesse.

Zum Abschluß besteht die Hoffnung, daß die in dieser Arbeit erstellten neuen raumfüllenden Kurven in anderen Gebieten der Forschung von Nutzen sein können.

Anhang A

Programme

Im Anhang A.1 ist ein lauffähiger C-Code für die Umsetzung der SFC-Heuristik unter Verwendung der Sierpiński-Kurve aufgelistet. In den weiteren Unterkapiteln dieses Anhangs finden sich einzelne Routinen für andere (raumfüllende) Kurven. Die Moore-Routine geht auf einen MATLAB-Code von Gerhard Zumbusch, Institut für Angewandte Mathematik, Universität Bonn, zurück. Das Quicksort-Verfahren ist an ein Programm von Kernighan und Ritchie angelehnt [33].

A.1 SFC-Algorithmus mit Sierpiński-Kurve

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
/*---------------------------*/
/* POTENZ */
void power (int m, int basis, double *pow)
/* hier wird der Vektor pow mit Potenzen aufgefüllt */
{
int i;
pow[0]=1;
for (i=1;i<m;i++)
   pow[i]=pow[i-1]*basis;
}
/*---------------------------*/
/* TAUSCHEN */
void tausch (double *neux, double *z, long i, long j)
/* hier wird neux[i] mit neux[j] und z[i] mit z[j] getauscht */
{
double temp1, temp2;
temp1=neux[i],temp2=z[i];
neux[i]=neux[j], z[i]=z[j];
neux[j]=temp1, z[j]=temp2;
}
/*---------------------------*/
```

```
/* DEZIMAL IN BINAER */
void dezinbin (double *pow, double x, int *dual, int m)
/* hier wird der Vektor dual mit der Dualdarstellung gefüllt */
{
int i;
x=x*pow[m];
for(i=m-1;i>=0;i--){
   dual[i]=(int)(x/pow[i]);
   if(dual[i]==2)
      dual[i]==1;
   x-=dual[i]*pow[i];
   }
}
/*---------------------------*/
/* QUICKSORT */
void quicksort (double *neux, double *z, long links, long rechts)
/* Sortierung findet in neux und z statt */
{
long i,letzter;
if(links>=rechts)
   return;
tausch(neux,z,links,(links+rechts)/2);
letzter=links;
for(i=links+1;i<=rechts;i++){
   if(neux[i]<neux[links])
      tausch(neux,z,++letzter,i);
   }
tausch(neux,z,links,letzter);
quicksort(neux,z,links,letzter-1);
quicksort(neux,z,letzter+1,rechts);
}
/*---------------------------*/
/* MIN */
double min (double *a, long n)
/* das minimale Element von a wird bestimmt */
{
long i;
double minwert=a[0];
for (i=1;i<n;i++) {
   if (a[i]<minwert)
      minwert=a[i];
   }
return (minwert);
}
/*---------------------------*/
/* MAX */
double max (double *a, long n)
```

```
/* das maximale Element von a wird bestimmt */
{
long i;
double maxwert=a[0];
for (i=1;i<n;i++) {
   if (a[i]>maxwert)
      maxwert=a[i];
   }
return (maxwert);
}
/*---------------------------*/
/* SIERPINSKIKURVE NACH 1.SCHRITT */
long sierpinski (double *pow, double neux, double neuy, int *x,
int *y, int m, int k, long index)
{
long i,j,l;
double pot=1./pow[k];
switch(index){
case 0:
   if (x[m+1]==1) {
      if (neuy>neux-pot)
         i=1,l=1;
      else
         i=0,l=0;
   }
   else {
      if (pot-neux>neuy)
         i=3,l=0;
      else
         i=2,l=3;
   }
   break;
case 1:
   if (y[m+1]==1) {
      if (neuy-pot>neux)
         i=3,l=1;
      else
         i=2,l=0;
   }
   else {
      if (pot-neuy>neux)
         i=0,l=1;
      else
         i=1,l=2;
   }
   break;
case 2:
```

```
   if (x[m+1]==1) {
      if (2*pot-neuy>neux-pot)
         i=2,l=1;
      else
         i=3,l=2;
   }
   else {
      if (neuy-pot>neux)
         i=0,l=2;
      else
         i=1,l=3;
   }
   break;
case 3:
   if (y[m+1]==1) {
      if (2*pot-neux>neuy-pot)
         i=1,l=0;
      else
         i=0,l=3;
   }
   else {
      if (neux-pot<neuy)
         i=2,l=2;
      else
         i=3,l=3;
   }
   break;
}
if(m>0) {
   neux=neux-x[m+1]*pot;
   neuy=neuy-y[m+1]*pot;
   j=sierpinski(pow,neux,neuy,x,y,m-1,k+1,l);
   i=pow[2*m]*i+j;
   }
return i;
}
/*---------------------------*/
/* SIERPINSKIKURVE IM 1.SCHRITT */
long sierpinski1 (double *pow, double neux, double neuy, int *x,
int *y, int m)
{
long i,j;
if (neux+neuy < 1) {
   if (neuy>neux)
   i=1;
   else
   i=0;
```

```
}
else {
   if (neuy>neux)
   i=2;
   else
   i=3;
}
j=sierpinski(pow,neux,neuy,x,y,m-1,1,i);
i=pow[2*m]*i+j;
return i;
}
/*---------------------------*/
/* HAUPTPROGRAMM */
void main (void)
{
long n=85900,i;
/*n=#punkte, i=laufindex*/
int m,d,v;
/*m=tiefe, d=distanz index, v=verschiebung*/
int bitx[16],bity[16];
/*bitdarstellung*/
double x[85900], y[85900], z[85900];
/*koordinaten und index*/
double neux[85900], neuy[85900], pow[32];
/*neue koord nach umrechnung, pow fuer potenz*/
double minx, maxx, miny, maxy, a1, a2;
/*hilfsgroessen fuer umrechnung*/
double x2,y2,abs=0;
/*laengenberechnung*/
FILE *f;
/*zum einlesen der daten*/
printf(``Tiefe der Kurve eingeben: (kleiner 17)\n'');
scanf(``%i'',&m);
printf(``Distanzfunktion: 0=eucl2d, 1=ceil2d\n'');
scanf(``%i'',&d);
printf(``Verschiebung: 0=unabhaengig, 1=gleichmaessig\n'');
scanf(``%i'',&v);
power(2*m,2,pow);
/*belegung eines potenzvektors*/
/*----------------------------*/
/* EINLESEN DATEN AUS TSPLIB */
f=fopen(``pla85900.tsp'',``r'');
if (!f)
   printf(``Dateifehler\n'');
else {
   for (i=0;i<n;i++) {
   fscanf(f,``%Lf\t %Lf\t %Lf\n'',&z[i],&x[i],&y[i]);
```

```
   }
fclose(f);
}
/*---------------------------*/
/* BERECHNUNG MIN. EINSCHLIESSENDES QUADRAT */
minx=min(x,n);
maxx=max(x,n);
miny=min(y,n);
maxy=max(y,n);
if(v){
a1=(minx+maxx)/2.0;
a2=(miny+maxy)/2.0;
if ((maxx-a1)>=(maxy-a2)) {
   miny=a2-(maxx-a1);
   maxy=a2+(maxx-a1);
   }
else {
   minx=a1-(maxy-a2);
   maxx=a1+(maxy-a2);
   }
}
/*---------------------------*/
/* TRANSLATION UND DILATATION AUF EINHEITSQUADRAT */
for (i=0; i<n; i++) {
   neux[i]=((double)(x[i])-minx)/(maxx-minx);
   neuy[i]=((double)(y[i])-miny)/(maxy-miny);
   }
/*---------------------------*/
/* AUFRUF DER JEWEILIGEN SFC */
for (i=0; i<n; i++){
   dezinbin(pow,neux[i],bitx,m);
   dezinbin(pow,neuy[i],bity,m);
   neux[i]=sierpinski1(pow,neux[i],neuy[i],bitx,bity,m-1);
/* oder: neux[i]=moore1(pow,bitx,bity,m-1,0,0,1,0); */
/* oder: neux[i]=hindex1(bitx,bity,m-1); oder andere Aufrufe */
   }
/*---------------------------*/
/* SORTIERAUFRUF */
quicksort (neux,z,0,n-1);
/*---------------------------*/
/* LAENGENBERECHNUNG */
if(d) {
   for(i=0;i<n-1;i++) {
      x2=x[(long)(z[i])-1]-x[(long)(z[i+1])-1];
      y2=y[(long)(z[i])-1]-y[(long)(z[i+1])-1];
      abs+=(long)(ceil(sqrt(x2*x2+y2*y2)));
      }
```

```
   x2=x[(long)(z[n-1])-1]-x[(long)(z[0])-1];
   y2=y[(long)(z[n-1])-1]-y[(long)(z[0])-1];
   abs+=(long)(ceil(sqrt(x2*x2+y2*y2)));
   /*kurve wird geschlossen*/
}
else {
   for(i=0;i<n-1;i++) {
      x2=x[(long)(z[i])-1]-x[(long)(z[i+1])-1];
      y2=y[(long)(z[i])-1]-y[(long)(z[i+1])-1];
      abs+=(long)(sqrt(x2*x2+y2*y2)+0.5);
      }
   x2=x[(long)(z[n-1])-1]-x[(long)(z[0])-1];
   y2=y[(long)(z[n-1])-1]-y[(long)(z[0])-1];
   abs+=(long)(sqrt(x2*x2+y2*y2)+0.5);
   /*kurve wird geschlossen*/
}
printf(``Laenge ist %li\n'',(long)(abs));
}
/*-----------ENDE------------*/
```

A.2 Routine für Moore-Kurve

```
/*---------------------------*/
/* MOOREKURVE NACH 1.SCHRITT */
long moore (double *pow, int *x, int *y, int m, int anfx, int anfy,
int endx, int endy)
{
int xb,yb;
long i,j;
if(anfx==endx){
   yb=y[m]^anfy;
   xb=(x[m]^anfx)^yb;
   }
else {
   yb=x[m]^anfx;
   xb=(y[m]^anfy)^yb;
   }
i=xb+2*yb;
if(m>0){
   if(i==0)
      j=moore(pow,x,y,m-1,anfx,anfy,endx^1,endy^1);
   if(i==1)
      j=moore(pow,x,y,m-1,anfx,anfy,endx,endy);
   if(i==2)
      j=moore(pow,x,y,m-1,anfx,anfy,endx,endy);
```

```
   if(i==3)
      j=moore(pow,x,y,m-1,anfx^1,anfy^1,endx,endy);
   i=pow[2*m]*i+j;
   }
return i;
}
/*---------------------------*/
/* MOORE IM 1.SCHRITT */
long moore1 (double *pow, int *x, int *y, int m, int anfx, int anfy,
int endx, int endy)
{
int xb,yb;
long i,j;
if(anfx==endx){
   yb=y[m]^anfy;
   xb=(x[m]^anfx)^yb;
   }
else {
   yb=x[m]^anfx;
   xb=(y[m]^anfy)^yb;
   }
i=xb+2*yb;
if(m>0) {
   if(i==0)
      j=moore(pow,x,y,m-1,anfx^1,anfy^1,endx^1,endy^1);
   if(i==1)
      j=moore(pow,x,y,m-1,anfx,anfy,endx,endy);
   if(i==2)
      j=moore(pow,x,y,m-1,anfx,anfy,endx,endy);
   if(i==3)
      j=moore(pow,x,y,m-1,anfx^1,anfy^1,endx^1,endy^1);
   i=pow[2*m]*i+j;
   }
return i;
}
/*---------------------------*/
```

A.3 Routine für H-Index

```
/*---------------------------*/
/* H-INDEX NACH 1.SCHRITT */
long hindex (int *x, int *y, int m, int k, long k1, long k2)
{
long i,j;
switch(k){
```

```
case 0:
switch(x[m]){
case 0:
   switch(y[m]){
      case 0:
         i=k2-1,k1=k1*4+6,k2=k2*4-3,k=0;
         break;
      case 1:
         i=k2,k1=k2*4-2,k2=k2*4+1,k=1;
         break;
      }
break;
case 1:
   switch(y[m]){
      case 0:
         i=k1+1,k2=k1*4+5,k1=k1*4+2,k=3;
         break;
      case 1:
         i=k1,k2=k2*4+3,k1=k1*4,k=0;
         break;
      }
break;
}
break;
case 1:
switch(x[m]){
case 0:
   switch(y[m]){
      case 0:
         i=k1,k2=k1*4+5,k1=k1*4+2,k=0;
         break;
      case 1:
         i=k1+1,k1=k1*4+6,k2=k2*4-3,k=1;
         break;
      }
break;
case 1:
   switch(y[m]){
      case 0:
         i=k2,k1=k1*4,k2=k2*4+3,k=1;
         break;
      case 1:
         i=k2-1,k1=k2*4-2,k2=k2*4+1,k=2;
         break;
      }
break;
}
```

```
break;
case 2:
switch(x[m]){
case 0:
   switch(y[m]){
      case 0:
         i=k1,k1=k1*4,k2=k2*4+3,k=2;
         break;
      case 1:
         i=k1+1,k2=k1*4+5,k1=k1*4+2,k=1;
         break;
      }
break;
case 1:
   switch(y[m]){
      case 0:
         i=k2,k1=k2*4-2,k2=k2*4+1,k=3;
         break;
      case 1:
         i=k2-1,k1=k1*4+6,k2=k2*4-3,k=2;
         break;
      }
break;
}
break;
case 3:
switch(x[m]){
case 0:
   switch(y[m]){
      case 0:
         i=k2-1,k1=k2*4-2,k2=k2*4+1,k=0;
         break;
      case 1:
         i=k2,k1=k1*4,k2=k2*4+3,k=3;
         break;
      }
break;
case 1:
   switch(y[m]){
      case 0:
         i=k1+1,k1=k1*4+6,k2=k2*4-3,k=3;
         break;
      case 1:
         i=k1,k2=k1*4+5,k1=k1*4+2,k=2;
         break;
      }
break;
```

```
}
}
if (m>0) {
   j=hindex(x,y,m-1,k,k1,k2);
   return j;
   }
return i;
}
/*---------------------------*/
/* H-INDEX IM 1.SCHRITT */
long hindex1 (int *x, int *y, int m)
{
long i,j;
long k,k1,k2;
switch(x[m]){
case 0:
   switch(y[m]){
      case 0:
         i=0,k=0,k1=0,k2=3;
         break;
      case 1:
         i=1,k=1,k1=4,k2=7;
         break;
      }
break;
case 1:
   switch(y[m]){
      case 0:
         i=3,k=3,k1=12,k2=15;
         break;
      case 1:
         i=2,k=2,k1=8,k2=11;
         break;
      }
}
if (m>0) {
   j=hindex(x,y,m-1,k,k1,k2);
   return j;
}
return i;
}
/*---------------------------*/
```

Anhang B

Zahlenmaterial

Es folgt eine Auflistung der Performances (in %) bzw. der P-Indizes aller in Kapitel drei dargestellten Kurven. In den realen Testfällen wird von den zwei Arten der Datenverschiebung die bessere dabei erzielte Performance aufgeführt. Dies ist grundsätzlich die gleichmäßige Verschiebung; die Fälle, in denen die unabhängige Verschiebung erfolgreicher ist, sind mit einem „*“ markiert. Für die randomisierten Testfälle wurden zehn (ab 500.000 Städte: fünf) Durchläufe gemittelt.

B.1 Reale Testfälle

B.1.1 Klassische Kurven

Problem	Hilbert	Moore	Lebesgue	Peano	Wunderl.1	Wunderl.2	Sierp.
pla85900	32,4*	32*	94,5*	40,5*	37,6	39,4	36,1*
pla33810	40,9*	40*	96,4*	48,3*	42,8	43	40,3
d18512	33,2	33,4	97,8	44,3	36,5	38,4	30,5
d15112	35,4	34,8	99,4	44,6	36,1	38,8	31,6
brd14051	37,4	35,2	98,6	45,4	37,5	41,1	31
usa13509	42,2	43,9	103,9	55,1	45,5	47,5	40,7
rl11849	49,8	47,6	103,5*	49,1*	49,5	52,5	49,4
pla7397	41,9*	39,4	94,6*	43*	40,6	41,5*	39,2
rl5934	60,7	56,3	107,2*	61*	49,9	63,4	59,9*
rl5915	60,8	59,9	109,5*	57,2*	60,6	65,2	58,4*
fnl4461	37,3	34,4	98,3	43,6	35,7	40,2	32,1
fl3795	68,3*	63,9*	108,2*	84,8*	65,3*	75,9	67,2*
pcb3038	35,8*	32,7*	90,9	47	33,9*	39,9*	32,7
pr2392	42,7*	40,7*	97	60,1	47,8*	50,2*	37,6*
u2319	11,2	9,2*	68,8*	21,7	17,7	16	11,8
u2152	46,9	43,6	102,9*	42,9*	46,2	47,1	47,3
d2103	46,1*	46,5*	95,5*	58*	51,9	56,5*	50,4
rl1889	62,6	55,5	103,2*	62*	58,5*	67,1	57,7*
u1817	48,1	42,5	98,3*	49,2*	50,6	52,7	50,4*
vm1748	49,2*	43,4	94,9*	49,5*	49,5*	55,9*	43,7*
Schnitt	44,1	41,7	98,2	50,4	44,7	48,6	42,4

B.1.2 Neuere Kurven

Problem	4x4-1	4x4-2	5x5	5x5-Peano	Dreieck1	Dreieck2
pla85900	41*	41,8*	35,5	41,4*	61,6	58,9
pla33810	34,1	41,9	41,4*	57,7*	67	67,3
d18512	39,5	39,9	37	47	52,1	54,2
d15112	36,6	41	38,5	50,2	54,9	56,1
brd14051	40,5	40,9	37,6	49,7	54,1	53,7
usa13509	44,9	47,6	50,5	59,7	64,7	64,2
rl11849	47	55,2	47,7*	50,2*	75,6	80
pla7397	40,9	45,5	42,1*	58,9*	64,9*	66*
rl5934	55,2*	62,7	58,2*	61,7*	86,7*	94,4
rl5915	54,1*	64,6	56,7*	59,4*	88,3*	96,9*
fnl4461	31,1	40,4*	42,7	55	54,7	55,1
fl3795	56,5*	71,3*	64,3	84,6	93,9*	104,5
pcb3038	39,3	41,8	41,6*	57,4	56,4*	50,9*
pr2392	42,1*	52,8*	53,7	67,1	61,7*	60,1*
u2319	15*	14	16,3	31,6*	29,2*	29,9
u2152	48,3	55,3	44,7*	42,3*	72,3	74
d2103	41,4*	43,8*	48,6*	60,2*	78,5*	82,1*
rl1889	57,9	65,1	58,8*	62,7*	86,5	94,5*
u1817	46,6*	52,4	42*	40*	71,6	72,2
vm1748	41,4	51,8	45,1*	51*	65,1*	69,8*
Schnitt	42,7	48,5	45,2	54,4	67	69,2

Problem	3x3-1	3x3-2	6x6	Fraktal1	Fraktal2	H-Index	beste Lösung
pla85900	46,2	48,3	40,5*	358,1	1313*	36,1*	142.307.500**
pla33810	52*	54,1*	44,1*	286,7	1291*	40,3	66.005.185**
d18512	42,1	45,4*	37	117,5	679*	30,5	645.198**
d15112	43,5	45,6	36,8	113,1*	651*	31,6	1.572.982**
brd14051	42,2	46,6	37,3	134,7	595*	31	469.358**
usa13509	51	55,6	45,8	87,8	1053*	40,7	19.982.859
rl11849	62,6*	62	53,2	158	548*	49,4	923.288
pla7397	49,5*	52,2*	43,2	457,2	259*	39,2	23.260.728
rl5934	70,9	69,9	58,6*	136,7	498*	59,8*	556.045
rl5915	72,2	72*	59,6*	143,8	485*	58,4*	565.530
fnl4461	44,3	45,2	38,3	104,1	345*	32	182.566
fl3795	83,8*	75,7*	68,9*	293,9	683*	67,3*	28.772
pcb3038	44,9*	45,4*	39,7	99,1	182*	32,8	137.694
pr2392	55,6*	56,1*	50,5	106,6	189*	37,6*	378.032
u2319	18,6*	20,7*	14,4	69,4	133*	11,8	234.256
u2152	59	57	52,1	75,5	210*	47,3	64.253
d2103	62,6*	58,6*	54,8*	108,3	338*	50,4	80.330
rl1889	66,8*	73,6*	60,6*	99,6	208*	57,6*	316.536
u1817	58,2*	58,8	53,6*	72,8	198*	50,4*	57.201
vm1748	51,6*	58,5	53,7	76,7	137*	43,7*	336.556
Schnitt	53,9	55	47,2	155	500	42,4	

**: nur Lower Bound (Stand: 01.07.2000)

B.2 Randomisierte Instanzen

B.2.1 Klassische Kurven

Städte	Hilbert	Moore	Lebesgue	Peano	Wunderl.1	Wunderl.2	Sierpiński
10.000	0,989	0,977	1,448	1,054	0,996	1,011	0,957
14.000						1,008	
17.000				1,048			
20.000	0,987	0,977	1,448	1,051	0,982	1,01	0,957
30.000	0,984	0,978	1,452	1,054	0,99	1,014	0,957
40.000	0,982	0,978	1,453	1,056	0,996	1,016	0,957
48.000					0,999		
50.000	0,983	0,979	1,452	1,054	0,998	1,015	0,957
80.000	0,982	0,979	1,452	1,047	0,99	1,007	0,956
100.000	0,981	0,979	1,452	1,044	0,985	1,004	0,957
125.000				1,041		1,002	
150.000					0,976		
200.000	0,98	0,979	1,454	1,044	0,978	1,006	0,958
300.000	0,98	0,979	1,454	1,051	0,987	1,011	0,957
400.000	0,98	0,979	1,455	1,051	0,993	1,012	0,958
500.000	0,98	0,979	1,454	1,051	0,995	1,01	0,957
800.000	0,98	0,979	1,454	1,042	0,984	1,004	0,957
1.000.000	0,98	0,979	1,454	1,039	0,979	1,002	0,957
1.100.000				1,038			
1.200.000	0,98	0,979	1,455	1,039	0,976	1,001	0,957
1.400.000	0,98	0,979	1,455	1,04	0,974	1,002	0,957

B.2.2 Neuere Kurven

Städte	4x4-1	4x4-2	5x5	5x5-Peano	Dreieck1	Dreieck2
10.000	0,965	1,016	1,031	1,144	1,116	1,132
20.000	0,989	1,027	1,01	1,09	1,118	1,131
30.000	1,015	1,034	1,004	1,072	1,119	1,131
35.000				1,071		
40.000	1,024	1,029	1,001	1,072	1,12	1,132
45.000	1,025					
50.000	1,024	1,027	1,005	1,08	1,119	1,132
80.000	1	1,015	1,019	1,106	1,12	1,131
100.000	0,987	1,012	1,027	1,118	1,12	1,131
130.000		1,008				
180.000	0,964		1,035	1,143		
200.000	0,965	1,013	1,034	1,142	1,121	1,131
300.000	0,982	1,022	1,027	1,122	1,12	1,132
400.000	1,002	1,027	1,018	1,1	1,121	1,133
500.000	1,016	1,028	1,013	1,084	1,121	1,132
700.000	1,024					
800.000	1,023	1,024	1,004	1,065	1,12	1,132
900.000			1,002	1,065		
1.000.000	1,015	1,018	1,003	1,067	1,121	1,132
1.200.000	1,004	1,015	1,005	1,072	1,121	1,132
1.400.000	0,993	1,011	1,009	1,079	1,121	1,132

Städte	3x3-1	3x3-2	6x6	Fraktal1	Fraktal2	H-Index
10.000	1,035	1,057	0,983	2,49	3,21	0,957
15.000	1,038					
20.000	1,038	1,061	0,996	3,13	4,44	0,957
30.000	1,031	1,062	1,001			0,957
35.000		1,065				
40.000	1,029	1,064	1,001			0,957
50.000	1,029	1,062	1,001	4,43	6,68	0,957
80.000	1,034	1,059	0,993			0,957
100.000	1,036	1,058	0,989	5,89	9,31	0,957
130.000	1,039					
200.000	1,034	1,061	0,979			0,957
250.000			0,977			
300.000	1,03	1,063	0,979			0,957
400.000	1,029	1,064	0,982			0,957
500.000	1,03	1,063	0,988	12,18	20,19	0,957
800.000	1,034	1,06	0,999			0,957
1.000.000	1,038	1,059	1,001	16,8	28,4	0,957
1.200.000	1,038	1,059	1,002			0,957
1.400.000	1,038	1,06	1,003			0,957

Literaturverzeichnis

[1] J. Alber, R. Niedermeier (1997), „On Multi-Dimensional Hilbert Indexings“, Preprint, Institut für Informatik, Universität Tübingen.

[2] D. Applegate, W. Cook und A. Rohe (1999), „Chained Lin-Kernighan for large traveling salesman problems“, Preprint, CAM, Rice University, Houston.

[3] S. Arora (1996), „Polynomial Time Approximation Schemes for Euclidean Traveling Salesman and other Geometric Problems“, in: *Proc. 37. IEEE Symposium on Foundations of Computer Science*, 2-12.

[4] J. Barry und E. Thomas (2000), „A Shot in the Dark“, in: *Newsweek International Magazine 24/2000*, 16-18.

[5] J.J. Bartholdi und L.K. Platzman (1982), „An $\mathcal{O}(N \log N)$ Planar Travelling Salesman Heuristic Based on Spacefilling Curves“, in: *Operations Research Letters 1*, 121-125.

[6] J.J. Bartholdi und L.K. Platzman (1988), „Heuristics Based on Spacefilling Curves for Combinatorial Problems in Euclidean Space“, in: *Management Science 34*, 291-305.

[7] J. Beardwood, J.H. Halton und J.M. Hammersley (1959), „The shortest path through many points“, in: *Proc. Cambridge Philosophical Society 55*, 299-327.

[8] D. Bertsimas und M. Grigni (1989), „Worst-case examples for the spacefilling curve heuristic for the Euclidean traveling salesman problem“, in: *Operations Research Letters 8*, 241-244.

[9] G. Cantor (1878), „Ein Beitrag zur Mannigfaltigkeitslehre“, in: *Crelle Journal 84*, 242-258.

[10] Web-basierter Kurs „Symmetry in Art, Biology and Mathematics“ des Center for the Mathematical Sciences, Wisconsin:
http://www.cms.wisc.edu/~cvg/course/491/modules/crystal/Crystals/node4.html

[11] S.A. Cook (1971), „The Complexity of Theorem-Proving Procedures“, in: *Proc. 3. Annual ACM Symposium on Theory of Computing*, 151-158.

[12] G.B. Dantzig, D.R. Fulkerson und S.M. Johnson (1954), „Solution of a Large Scale Traveling-Salesman Problem“, in: *Operations Research 2*, 393-410.

[13] R. Diestel (1996), *Graphentheorie*, Springer-Verlag, Berlin.

[14] V. Drakopoulos und L. Dalla (1999), „Space-filling Curves Generated by Fractal Interpolation Functions“, in: Iliev, Kaschiev, Margenov, Sendov, Vassilevski (Hrsg.), *Recent advances in numerical methods and applications*, World Scientific, Singapore.

[15] S. Finch (2000), Webseite über Konstanten für das TSP: http://www.mathsoft.com/asolve/constant/sales/sales.html

[16] T. Gerstner (1999), „Multiresolution Visualization and Compression of Global Topographic Data“, zur Veröffentlichung eingereicht bei *GeoInformatica 2000.*

[17] M. Griebel und G. Zumbusch (1998), „Parallel multigrid in an adaptive PDE solver based on hashing“, in: *Proc. ParCo '97*, Elsevier, 589-599.

[18] C. Gotsman und M. Lindenbaum (1996), „On the metric properties of discrete space-filling curves“, in: *IEEE Transactions on Image Processing 5*, 794-797.

[19] R.P. Grimaldi (1999), *Discrete and Combinatorial Mathematics*, Addison-Wesley, Reading, 4. Edition, 529.

[20] W. Gropp, E. Lusk, A. Skjellum (1994), *Using MPI*, The MIT Press, Cambridge.

[21] G. Haase (1999), *Parallelisierung numerischer Algorithmen für partielle Differentialgleichungen*, Teubner-Verlag, Stuttgart.

[22] M. Held und R.M. Karp (1970), „The Traveling Salesman Problem and Minimum Spanning Trees“, in: *Operations Research 18*, 1138-1162.

[23] D. Hilbert (1891), „Über die stetige Abbildung einer Linie auf ein Flächenstück“, in: *Math. Annalen 38*, 459-460.

[24] C.A.J. Hurkens (1991), „Nasty TSP Instances for Classical Insertion Heuristics“, Research Report, University of Technology, Eindhoven.

[25] D.S. Johnson und C.H. Papadimitriou (1985), „Computational Complexity“, in: [30], 37-85.

[26] D.S. Johnson (1990), „Local Optimization and the Traveling Salesman Problem“, in: G. Goos and J. Hartmanis (Hrsg.), *Automata, Languages and Programming*, Lecture Notes in Computer Science 442, Springer-Verlag, Heidelberg, 446-461.

[27] D.S. Johnson, L.A. McGeoch und E.E. Rothberg (1996), „Asymptotic experimental analysis for the Held-Karp traveling salesman bound“, in: *Proc. 7. Annual ACM-SIAM Symposium on Discrete Algorithms*, ACM, 341-350.

[28] S. Kirkpatrick, C.D. Gelatt Jr. und M.P. Vecchi (1983), „Optimization by Simulated Annealing“, in: *Science 222*, 671-680.

[29] S. Kirkpatrick (1984), „Optimization by Simulated Annealing: Quantitative Studies“, in: *Journal of Statistical Physics 34*, 975-986.

[30] E. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan und D.B. Shmoys (Hrsg.) (1985), *The Traveling Salesman Problem*, John Wiley & Sons, Chichester.

[31] H. Lebesgue (1905), „Sur les fonctions représentables analytiquement“, in: *Journal de Math. 6*, 139-216.

[32] S. Lin und B.W. Kernighan (1973), „An Effective Heuristic Algorithm for the Traveling-Salesman Problem“, in: *Operations Research 21*, 498-516.

[33] B. Kernighan und D. Ritchie (1990), *Programmieren in C*, Carl Hanser Verlag, München, 2. Auflage.

[34] D.E. Knuth (1973), *The Art of Computer Programming, Volume 3: Sorting and Searching*, Addison-Wesley, Reading.

[35] B.B. Mandelbrot (1977), *Fractals, Form, Chance, and Dimension*, W.H. Freeman and Co., San Francisco.

[36] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller und E. Teller (1953), „Equation of State Calculation by Fast Computing Machines“, in: *Journal Chem. Physics 21*, 1087-1092.

[37] E.H. Moore (1900), „On certain crinkly curves“, in: *Trans. American Math. Society 1*, 72-90.

[38] P. Moscato und M.G. Norman (1994), „An Analysis of the Performance of Traveling Salesman Heuristics on Infinite-Size Fractal Instances in the Euclidean Plane“, als Postscript-File bei CeTAD, Universidad Nacional de La Plata; zur Veröffentlichung vorgesehen in *ORSA Journal on Computing.*

[39] MPI-Homepage: http://www.mcs.anl.gov/mpi/index.html

[40] E. Netto (1879), „Beitrag zur Mannigfaltigkeitslehre“, in: *Crelle Journal 86*, 263-268.

[41] R. Niedermeier, K. Reinhardt und P. Sanders (1997), „Towards Optimal Locality in Mesh Indexings“, Technical Report, IB 12/97, Universität Karlsruhe. Kurzfassung in: *Proc. 11. Symposium on Fundamentals of Computation Theory*, Lecture Notes in Computer Science 1279, Springer-Verlag, 364-375.

[42] M.G. Norman und P. Moscato (1995), „The Euclidean Traveling Salesman Problem and a Space-Filling Curve“, in: *Chaos, Solitons and Fractals 6*, 389-397.

[43] G. Peano (1890), „Sur une courbe qui remplit toute une aire plaine“, in: *Math. Annalen 36*, 157-160.

[44] A.G. Percus und O.C. Martin (1996), „Finite size and dimensional dependence in the Euclidean traveling salesman problem“, in: *Physical Review Letters 76*, 1188-1191.

[45] J.R. Pilkington, S.B. Baden (1994), „Partitioning with Spacefilling Curves“, CSE Technical Report CS94-349, University of California, San Diego.

[46] L.K. Platzman und J.J. Bartholdi (1989), „Spacefilling Curves and the Planar Travelling Salesman Problem“, in: *Journal of the ACM 36*, 719-737.

[47] G. Reinelt (1994), *The Traveling Salesman - Computational Solutions for TSP Applications*, Springer-Verlag, Heidelberg.

[48] G. Reinelt (1995), „TSPLIB95 - A Traveling Salesman Problem Library“, unter http://www.iwr.uni-heidelberg.de/iwr/comopt/software/TSPLIB95.

[49] A. Rohe (1997), „Parallel Lower and Upper Bounds for Large TSPs“, in: *ZAMM 77*, Supplement 2, 429-432.

[50] D.J. Rosenkrantz, R.E. Stearns und P.M. Lewis (1977), „An Analysis of Several Heuristics for the Traveling Salesman Problem“, in: *SIAM Journal on Computing 6*, 563-581.

[51] H. Sagan (1994), *Space-Filling Curves*, Springer-Verlag, New York.

[52] H. Samet (1990), *The Design and Analysis of Spatial Data Structures*, Addison-Wesley, Reading.

[53] I.J. Schoenberg (1938), „The Peano-Curve of Lebesgue“, in: *Bull. American Math. Society 44*, 519.

[54] W. Sierpiński (1912), „Sur une nouvelle courbe continue qui remplit toute une aire plaine“, in: *Oeuvres Choisies*, Tome II (1975), Éditions Scientifiques de Pologne, Warschau.

[55] W. Wunderlich (1973), „Über Peano-Kurven“, in: *Elem. Math. 28*, 1-10.

[56] G. Zumbusch (1999), „Dynamic loadbalancing in a lightweight adaptive parallel multigrid PDE solver.“, in: *Proc. 9. SIAM Conference on Parallel Processing for Scientific Computing (PP99).*

Der *Travelling Salesman* auf der Umschlaginnenseite ist eine Collage verschiedener Vorlagen von Thomas Gerstner, IAM, Universität Bonn.