JOHANN WOLFGANG GOETHE
UNIVERSITÄT
FRANKFURT AM MAIN

Fachbereich Informatik und Mathematik
Institut für Informatik

Diplomarbeit

Lokale Suche in variabler Tiefe

Dennis Luxen

19. Juli 2007

eingereicht bei Prof. Dr. Georg Schnitger
Professur Theoretische Informatik

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Frankfurt am Main, den 19. Juli 2007

(Dennis Luxen)

Inhaltsverzeichnis

1. Vorwort 3

2. Grundlagen 5
   - 2.1. Das Traveling Salesman Problem (TSP) 5
   - 2.2. Algorithmen 9
     - 2.2.1. Exakte Algorithmen 9
     - 2.2.2. Iterative Methoden 10
     - 2.2.3. Heuristische Methoden 11
   - 2.3. Heuristiken 12
     - 2.3.1. Nearest Neighbor 12
     - 2.3.2. Minimum-Spanning-Tree (Double-Tree) Heuristik 16
     - 2.3.3. Christofides-Heuristik 19
     - 2.3.4. Einfügeheuristiken 21
   - 2.4. Untere Schranken 23
     - 2.4.1. 1-Baum-Schranke 24
     - 2.4.2. Held-Karp-Schranke 24

3. Lokale Suche 29
   - 3.1. Nachbarschaft für das Traveling Salesman Problem 31
   - 3.2. Grundidee der Lin-Kernighan Heuristik 32
   - 3.3. Heuristik von Lin-Kernighan für das symmetrische TSP 35
     - 3.3.1. Die Prozedur nach Lin-Kernighan, vereinfacht dargestellt 35
     - 3.3.2. Der Algorithmus von Lin-Kernighan 36
   - 3.4. Die Implementierung von Johnson et al. 39
     - 3.4.1. Die innere Schleife 39
     - 3.4.2. Steuerung der inneren Schleife 42
     - 3.4.3. Experimentelle Ergebnisse der Implementierung 42
   - 3.5. Die Implementierung von Helsgaun 43
     - 3.5.1. Begrenzung der Nachbarschaft durch Kandidatenlisten 43
     - 3.5.2. Veränderte Löschregeln für Kanten in einer Tour 44
     - 3.5.3. Veränderte Bewegung durch die Nachbarschaft 45
     - 3.5.4. Wahl der Starttour 46
     - 3.5.5. Experimentelle Ergebnisse der Implementierung 47
   - 3.6. Die Implementierung von Neto 48
     - 3.6.1. Einordnung eines Clusters 48
     - 3.6.2. Längere Laufzeit bei geclustertern Instanzen 49

3.6.3. Effiziente Clusterkompensierung 50
3.6.4. Einfluss der Cluster-Distanz auf die Heuristik 52
3.6.5. Justierung der Kantenwahl 52
3.6.6. Experimentelle Ergebnisse der Implementierung 53
3.7. PLS-Vollständigkeit 53
3.7.1. Variante der Lin-Kernighan Heuristik nach Papadimitriou . . . 56
3.7.2. PLS-Vollständigkeit von Netos Implementierung 61

4. Experimentelle Analyse der Lin-Kernighan-Heuristik 63
4.1. TSPLIB 64
4.2. Aufbau der entstandenen Implementierung 65
4.3. Lin-Kernighan ohne Backtracking 65
4.4. Lin-Kernighan mit Backtracking 68
4.5. Lin-Kernighan ohne Fixieren entfernter Kanten 70
4.6. Lin-Kernighan ohne Fixieren hinzugefügter Kanten 72
4.7. Einfluss der Starttour auf die Güte 75
4.8. Fazit 77

5. MAX2SAT 79
5.1. Definition des Problems 79
5.2. Definition der Nachbarschaft 80
5.3. Algorithmus für MAX2SAT 80
5.4. Datenformat für MAX2SAT-Instanzen 83
5.4.1. Datenformat für Instanzen 83
5.5. Erzeugung der MAX2SAT-Instanzen 84
5.5.1. Idealer und realer Testinstanzgenerator 84
5.5.2. Erzeugung der Testinstanzen nach Motoki 85
5.6. Experimentelle Analyse 88
5.6.1. Vergleichsheuristik 89
5.7. Berechnungs- und Simulationsergebnisse 90
5.7.1. Experimentelle Ergebnisse der Heuristik 91
5.8. Fazit 94

6. Ausblick 95

A. Simulationsergebnisse für das TSP 97
A.1. Ohne Backtracking 97
A.2. Mit Backtracking 98
A.3. Varianten mit abweichender Kantenfixierung 99
A.3.1. Ohne Fixieren entfernter Kanten 99
A.3.2. Ohne Fixieren hinzugefügter Kanten 100
A.3.3. Ohne Fixieren hinzugefügter Kanten und mit randomisierter Starttour 101

B. Implementierungsdetails 103
B.1. Bedienung der TSPBench 103
B.2. Erweiterung des TSPBench 104
B.3. Bedienung der Anwendungsfamilie für MAX2SAT 105
B.4. Erweiterung der MAX2SAT Heuristiken 107

Abbildungsverzeichnis 109

Index 111

Literaturverzeichnis 113

1. Vorwort

Die vorliegende Arbeit untersucht eine unter dem Namen „Lokale Suche in variabler Tiefe" bekannte Optimierungsstrategie für algorithmisch schwierige Probleme. Diese Untersuchung wird unter anderem anhand des Traveling Salesman Problems durchgeführt, welches zu den klassischen Problemen der kombinatorischen Optimierung gehört. Der Schwerpunkt liegt auf dem Spezialfall des symmetrischen Traveling Salesman Problem, das in der Praxis viele Anwendungen hat. Beispielsweise kommt es in der VLSI Chipprooduktion oder der Ölbohrindustrie vor und betrifft auch ganz alltägliche Fragestellungen, wie die optimale Aufgabenreihenfolge für Servicetechniker der Deutschen Telekom AG im Ausseneinsatz. Das Problem an sich ist $\mathcal{NP}$-hart und unter der allgemein als wahr vermuteten Behauptung, dass $\mathcal{P} \neq \mathcal{NP}$ hat jeder neue Algorithmus, der das Problem optimal löst, eine Laufzeit, die schneller steigt als jedes Polynom. Diese Tatsache ist Haupttriebkraft bei der Suche nach guten Schätzverfahren, die zwar eine gute Lösung nur abschätzen, dies aber vergleichsweise schnell tun. In der Realität reicht eine gute Schätzung in vielen Fällen schon aus.

Ein gerne gewählter Ansatz zur Schätzung ist die lokale Suche. Diese benutzt die Vorstellung, dass es in der „Nähe" einer Lösung wohl noch eine bessere gibt und dann von dort aus weitere suchten, bis eine Lösung vorliegt, die ein lokales Optimum ist. Die Hoffnung ist hierbei, dass das lokale schon fast oder sogar ganz dem globalen Optimum entspricht. Mit wachsender Nachbarschaftsgröße wächst allerdings auch die Laufzeit und wird schnell unhandhabbar. 1970 stellen Shen Lin und Brian Kernighan [LK73] eine besonders effektive Variante dieses Suchverfahrens für das Traveling Salesman Problem vor. Ihre Arbeit ist eine Verallgemeinerung eines Verfahrens, das sie kurze Zeit zuvor schon erfolgreich am ebenfalls $\mathcal{NP}$-harten Problem der Graph-Partitionierung anwandten und veröffentlichten. Sie geben die Nachbarschaftsgröße nicht von vorn herein fest an, sondern lassen das Verfahren im Lauf selbst entscheiden. Die nach ihnen benannte Lin-Kernighan Heuristik zählt auch heute nach fast 35 Jahren noch zu den effektivsten Verfahren zur Approximation des Traveling Salesman Problems, so wie es in vielen Anwendungen auftritt.

Das Traveling Salesman Problem hat eine lange Historie. Bereits 1832 wird es im Buch *Der Handlungsreisende - Wie er sein soll und was er zu thun hat, um Aufträge zu erhalten und eines glücklichen Erfolgs in seinen Geschäften zu sein - Von einem alten Commis-Voyageur* erwähnt. Die Formalisierung des Problems und die intensive wissenschaftliche Forschung begannen in den dreißiger Jahren des letzten Jahrhunderts. Für eine detaillierte historische Betrachtung des Problems empfiehlt sich das Buch von

Applegate et al. [ABCC07].

Diese Arbeit untersucht neben einigen Varianten der Lin-Kernighan Heuristik auch die Frage, ob sich die Optimierungsstrategie auch auf andere schwere Probleme anwenden lässt. Namentlich ist dies das Problem MAX2SAT, ein Spezialfall aus der Familie der Erfüllbarkeitsprobleme. Dieses Optimierungsproblem fragt bei einer gegebenen Formel in 2CNF nach der maximalen Zahl erfüllbarer Klauseln. Dabei ist die Tatsache interessant, dass das zugrunde liegende Problem 2SAT, also die Frage ob eine Formel in 2CNF komplett erfüllbar ist, sogar einfach zu lösen ist. Ganz im Gegensatz zu MAX2SAT, von dem bekannt ist, dass es schwer ist.

Zu diesen Fragestellungen werden im Verlauf Lösungsverfahren entwickelt und experimentell untersucht. In diesem Zuge werden zusätzlich offene und thematisch verwandte Probleme identifiziert.

2. Grundlagen

2.1. Das Traveling Salesman Problem (TSP)

Das Traveling Salesman Problem gehört zu den bekanntesten Optimierungsproblemen. Das Problem lässt sich anschaulich sehr einfach formulieren: Gegeben sind eine Menge von Städten und die Distanzen zwischen diesen Städten. In der einfachsten Version unterliegen diese Distanzen keinen Randbedingungen. Sie sind beliebig. Gesucht ist nun eine Tour über alle Städte mit minimaler Gesamtlänge, d.h. ausgehend von einer Anfangsstadt wird eine Tour über alle Städte gesucht, die an ihrem Anfang auch wieder endet und jede Stadt genau einmal besucht. Dies bedeutet insbesondere auch, dass keine Stadt mehrfach besucht wird. Obwohl dieses Problem auf den ersten Blick sehr leicht anmutet, ist diese Leichtigkeit trügerisch. Es ist bis heute nicht gelungen, ein effizientes Lösungsverfahren zu entwickeln.

Optimale Lösungen für dieses allgemeine Traveling Salesman Problem sind von einem Lösungsalgorithmus allerdings auch nicht zu erwarten. Für das allgemeine TSP gibt es sogar ein negatives Resultat, was die Schätzung einer Lösung betrifft. Nur wenn gewisse Einschränkungen für die Kantengewichte exisiteren, werden Schätzverfahren möglich sein. Die Sprachversion des TSP, also die Frage, ob es eine Rundreise gibt, deren Kosten durch eine Konstante beschränkt sind, ist $\mathcal{NP}$-hart und gehört zu den meist erforschten Problemen in der Komplexitätstheorie. Allgemein wird davon ausgegangen, dass die Laufzeit aller $\mathcal{NP}$-harten Probleme exponentiell ist [Weg03]. Allerdings handelt es sich hier nur um eine Vermutung und ein Beweis ist noch zu erbringen. Nicht ohne Grund sind diese Fragestellung und ihre Verwandten ein Feld intensiver Forschung.

Für viele reale Anwendungen des TSPs sind Schätzungen aber ausreichend. So wäre ein Geschäftsmann zufrieden, wenn er für eine zeitaufwändige Dienstreise durch verschiedene Länder vielleicht nicht die schnellsten Flugverbindungen benutzt, aber auf jeden Fall sicher sein könnte, dass die geplante Route ihm nicht mehr Reisetage zumutet als wirklich notwendig. Die Suche nach Approximationen, für die bestimmte Güteaussagen gelten, ist also kein Vorgang von rein theoretischer Natur, sondern findet seine Berechtigung auch durch die praktischen Anwendungen.

Eine Approximation A produziert für eine zulässige Eingabe $x \in X$ eine ebenso *zulässige Lösung* $s(x) \in S$ (kurz: $s \in S$), wobei X die Menge aller Eingaben und S die Menge aller Lösungen ist. Jeder Lösung s wird ein Wert $f(x, s)$ zugewiesen. Für viele

Probleme ist dieser Wert ganzzahlig. Diese Güte einer Lösung ist anschaulich die Nähe, die sie zur optimalen Lösung $opt(x)$ für die Eingabe x hat. Für Optimierungsprobleme ist die *Approximationsgüte* einer Lösung s und zugehöriger Eingabe x definiert als

$$\max \left\{ \frac{opt(x)}{f(s,x)}, \frac{f(s,x)}{opt(x)} \right\} \quad . \tag{2.1}$$

Sie wird auch synonym als *Approximationsfaktor* bezeichnet. Eine Approximation kann also daran gemessen werden, ob ihre Güte durch eine Konstante k für alle möglichen Eingabeinstanzen garantiert, also nach oben beschränkt werden kann und wie nah k an 1 ist. Für $k = 1$ wird offensichtlich das Optimum gefunden und je näher die Konstante an 1 ist, desto besser. Existiert ein solcher konstanter Faktor k, dann wird auch von einer k-Approximation gesprochen.

Im allgemeinen Traveling Salesman Problem ist ein vollständiger, gewichteter Graph G mit n Knoten gegeben. Die Kanten können in Hin- und Rückrichtung unterschiedliche, ja beliebige Längen haben. Gesucht ist eine Tour über alle Knoten in G mit minimaler Summe ihrer Kantengewichte. Für das allgemeine TSP gibt es aber leider ein negatives Resultat von Sahni und Gonzalez [SG76], dass keine Hoffnung auf einen Approximationsalgorithmus besteht.

Satz 1 (Nichtapproximierbarkeit des allgemeinen TSP)
Solange nicht $\mathcal{P} = \mathcal{NP}$ gilt, gibt es keine k-Approximation des allgemeinen TSP für beliebiges $k \geq 1$.

Der Beweis beruht auf einer Reduktion auf das Hamiltonkreisproblem, das als $\mathcal{NP}$-vollständig bekannt ist [Weg03]. Es wird nun angenommen, dass es einen Algorithmus A gäbe, der eine k-Approximation sei und in Polynomialzeit liefe. Gegeben sei weiterhin ein Graph $G = (V, E)$ als Eingabe für die Entscheidungsvariante des Hamiltonkreisproblems gegeben. Aus diesem wird eine Instanz für das allgemeine TSP mit $n = |V|$ Knoten und den Kantengewichten $c : E \rightarrow \mathbb{Z}_+$ wie folgt konstruiert:

$$c(i,j) := \begin{cases} 1, & \text{für } (i,j) \in E \\ n \cdot k, & \text{für } (i,j) \notin E \end{cases}$$

G enthält genau dann einen Hamiltonkreis, wenn es in der konstruierten Instanz eine TSP Tour mit Länge n gibt. Zudem enthält G genau dann keinen Hamiltonkreis, wenn jede mögliche TSP Tour in der konstruierten Instanz länger als $n \cdot k$ ist. Eine k-Approximation für das allgemeine TSP könnte also das Hamiltonkreisproblem entscheiden. Dies würde allerdings $\mathcal{P} = \mathcal{NP}$ implizieren und führt zum Widerspruch der Annahme. □

Trotz dieses sehr ernüchternden Resultats, ist noch nicht alles verloren. Die meisten Instanzen, die in der Realität auftauchen, sind nicht nur symmetrisch, sondern erfüllen auch die Dreiecksungleichung, gehören somit zur metrischen Variante des TSP. Kurz gesagt, die Kantengewichte werden so eingeschränkt, dass sie eine Metrik definieren. Für diese Variante werden sich durchaus k-Approximationen finden lassen.

Definition 1 (Das metrische Traveling Salesman Problem)
Gegeben sei ein gewichteter, ungerichteter Graph $G = (V, E)$. $V = \{1, \ldots, n\}$ ist die Menge der Knoten und $E = \{(i, j) | i, j \in \mathbb{N}\}$ die Menge der Kanten. Der Grad eines Knoten ist die Anzahl der zu ihm inzidenten Kanten.

Jede Kante (i, j) bekommt über eine Kostenfunktion $c : V \times V \mapsto \mathbb{N}$ ein Gewicht $c(i, j)$ zugeordnet. Die Distanzmatrix D, mit $c(i, j) = D_{i,j}$ ist eine andere Schreibweise der Kostenfunktion, die der Distanz von Knoten i bis Knoten j entspricht. Die Kantenlängen sind symmetrisch und erfüllen die Dreiecksungleichung.

Ein Pfad ist eine Kantenmenge $\{(i_1, i_2), (i_2, i_3), \ldots, (i_{k-1}, i_k)\}$, mit $i_p \neq i_q$, $\forall\ p \neq q$.
Ein Kreis ist eine Kantenmenge $\{(i_1, i_2), (i_2, i_3), \ldots, (i_k, i_1)\}$, mit $i_p \neq i_q$, für $p \neq q$.
Eine Tour ist ein Kreis mit $k = n$. Sie besucht jeden Knoten genau einmal.

Für eine gegebene Untermenge $S \subseteq E$ ist die Länge $l(S)$ definiert als

$$l(S) = \sum_{(i,j) \in S} c(i, j).$$

Eine optimale Tour $\tau \in T$ in G ist ein Kreis minimaler Länge. Damit lässt sich nun das eigentliche Problem formulieren.

Gegeben ist ein gewichteter Graph G. Finde eine optimale Tour τ in G. Bestimme also:

$$\min_{\tau \in T} \sum_{(i,j) \in \tau} c(i, j) + c(v_{n,\tau}, v_{1,\tau}) \tag{2.2}$$

Ein Graph heisst weiterhin zusammenhängend, wenn es für jedes beliebige Knotenpaar (i, j) einen Pfad zwischen den Knoten gibt. Ein Baum ist ein zusammenhängender Graph ohne Kreis und ein Spannbaum ist ein Baum der Größe $n - 1$, also mit $n - 1$ Kanten aus G. Ein minimaler Spannbaum ist ein Spannbaum minimaler Länge.

Für den Fall, dass das Problem mit roher Rechengewalt, durch simples Ausprobieren aller Möglichkeiten, gelöst werden soll, ließe sich ein einfacher Algorithmus finden. Alle Lösungsmöglichkeiten würden aufgezählt und einfach nacheinander ausprobiert. Zum Schluß würde die beste gesehene Tour nur noch einmal ausgegeben. Allerdings ist die schiere Anzahl der möglichen Lösungen selbst im symmetrischen TSP viel zu groß, als dass sich dieses Vorgehen auch nur ansatzweise lohnen würde.

Die Anzahl möglicher Touren im symmetrischen Traveling Salesman Problem beträgt $(n - 1)!/2$. Dies lässt sich mit einem einfachen induktiven Argument zeigen:

Angenommen, es existiert bereits eine Aufzählung aller Touren für $n - 1$ Städte und diese habe die Kardinalität $T(n)$. Daraus lässt sich eine Aufzählung aller Touren für n

Städte leicht konstruieren. Eine n-Städte Tour τ_n entsteht, indem in eine $n-1$-Städte Tour τ_{n-1} ein Umweg zur n-ten Stadt eingebaut wird. Dafür gibt es nun offensichtlicherweise $n-1$ mögliche Stellen, die aufgetrennt werden können. Für diese einfache Konstruktion lässt sich auch eine Rekursionsgleichung angeben. Die Anzahl möglicher Touren über n Städte ist gleich

$$T(n) = T(n-1) \cdot (n-1) \ . \tag{2.3}$$

Dazu verdeutlicht Abbildung 2.1 noch einmal die Konstruktion an einem Beispiel.

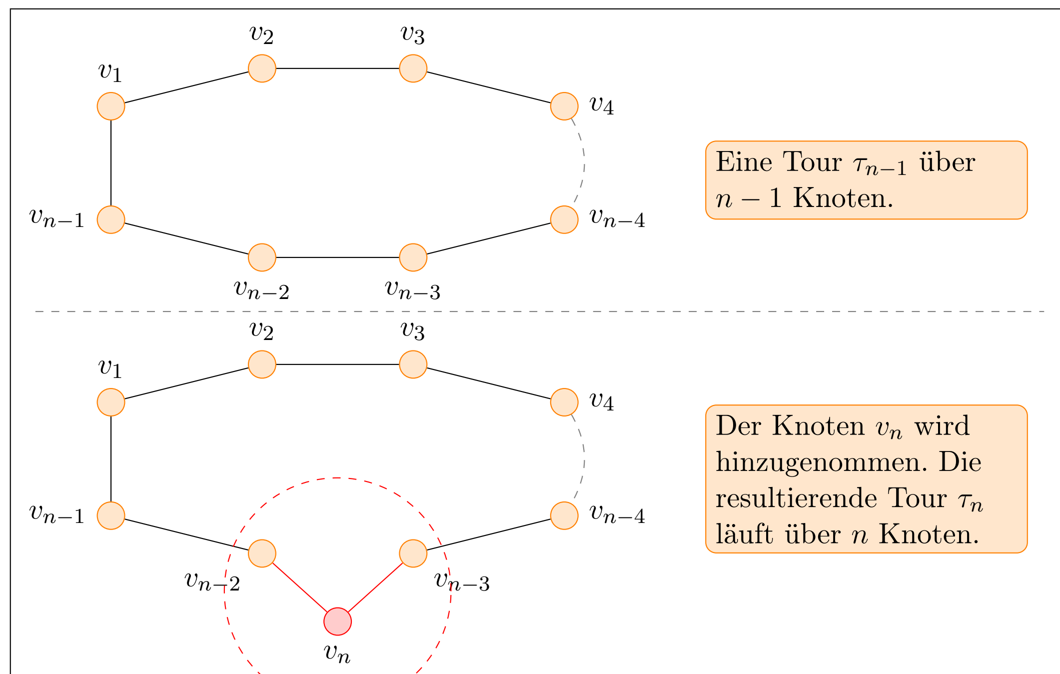

Abbildung 2.1.: Tour Konstruktion

Die kleinste Zahl an Städten, zu der sich Touren konstruieren lassen, ist offensichtlicherweise 3. Touren in umgekehrter Reihenfolge gelten ebenso wie Touren, deren Anfang um ein Offset verschoben ist, als äquivalent. Für $n = 3$ Städte gibt es eine Tour, für $n = 4$ Städte gibt es nach obiger Konstruktion schon 3 mögliche Touren und für $n = 5$ sind es bereits 12 mögliche unterschiedliche Touren. Für $n = 100$ ergibt sich die Schwindel erregende Zahl von $T(100) = 4.66631077 \cdot 10^{155}$ möglicher Touren. Zum Vergleich: Die geschätzte Anzahl Atome im Universum liegt in der Größenordnung von etwa 10^{78}. Es ergibt sich nun recht offensichtlich für n Städte $T(n) = 3 \cdot 4 \cdot 5 \cdot \ldots \cdot (n-1) = (n-1)!/2$. Aus obiger Betrachtung folgt also:

Satz 2 (Anzahl Touren im symmetrischen TSP)
Die Anzahl der Touren im symmetrischen Traveling Salesman Problem beträgt $T(n) = (n-1)!/2$.

Im weiteren Verlauf wird das metrische Traveling Salesman Problem zur Vereinfachung der Schreibweise oft nur als Traveling Salesman Problem bezeichnet. Beide Namen sind für diese Arbeit äquivalent, sofern im Text nicht anders angegeben.

2.2. Algorithmen

Die Gruppe der Lösungsalgorithmen für das Traveling Salesman Problem lässt sich grob in drei Gruppen einteilen:

- Exakte Algorithmen
- Iterative Methode
- Heuristische Methoden

2.2.1. Exakte Algorithmen

Exakte Algorithmen garantieren beweisbar das Ergebnis ihrer Berechnungen. Sie finden die optimale Lösung in einer vorher bekannten Anzahl von Schritten. Derzeit eignen sich exakte Verfahren für die Lösung von Instanzen mit einigen hundert Städten. Es gibt auch schon erste Berichte über die Lösung von Traveling Salesman Instanzen mit wenigen tausend Städten [Hel00].

Alle Methoden, die mit roher Gewalt optimale Lösungen durch Ausprobieren ausrechnen, gehören trivialerweise zu den exakten Algorithmen. Das naheliegendste Verfahren ist die Aufzählung aller Permutationen und die Ausgabe der günstigsten Rundreise. Dieser Algorithmus hat aber eine praktisch unbrauchbare Laufzeit von $O(n \cdot n!)$. Dieses naive Verfahren kann durch dynamisches Programmieren allerdings verbessert werden.

Die Grundidee ist die folgende. O.b.d.A. beginne eine optimale Rundreise im Knoten 1 und besuche dann den Knoten k. Dann muss der Rückweg von k nach 1 über die Knoten $\{2, \cdots, n\} - \{k\}$ auch optimal sein. Sei dazu die Distanzmatrix M gegeben und $g(i, S)$ als die Länge des kürzesten Weges, der bei Knoten i beginnt und alle Knoten in der Menge S genau einmal besucht, um dann im Knoten 1 zu enden. Diese Länge lässt sich durch folgende rekursive Gleichung für alle Teilmengen $S \subseteq \{2, \ldots, n\}$ und alle Knoten $i \notin S$ angeben:

$$g(i, S) = \begin{cases} m_{i1}, & \text{wenn } S = \emptyset \\ \min\limits_{j \in S}(m_{ij} + g(j, S - \{j\}), & \text{wenn } S \neq \emptyset \end{cases} \tag{2.4}$$

Daraus ergibt sich folgender Algorithmus, der die Einträge einer Tabelle g mit $g[i, S] = g(i, S)$ speichert.

Algorithmus 1 Lösung des TSP mit Hilfe des dynamischen Programmierens

Eingabe: Distanzmatrix M

Ausgabe: Optimale Lösung

1: **for** $i = 2$ to n **do**
2: $g[i, \emptyset] = m_{ij}$
3: **end for**
4: Berechne $g[1, \{2, \ldots, n\}]$

Alle entstehende Teilprobleme können mit Hilfe von Gleichung 2.4 gelöst werden. Das Verfahren hat höchsten $n \cdot 2^n$ Teilprobleme, die jeweils in linearer Zeit gelöst werden können. Der Zeitaufwand dieses Verfahrens zur Lösung des allgemeinen TSP ist also $O(n^2 \cdot 2^n)$ und der Platzbedarf für die Tabelle g beträgt $\Omega(n \cdot 2^n)$. Dies ist zwar immer noch kein praktisch nutzbarer Ressourcenbedarf, aber schon deutlich besser als der naive Lösungsalgorithmus.

2.2.2. Iterative Methoden

Eine weitere Möglichkeit zur Lösung des Traveling Salesman Problems ist die lineare Programmierung. Sie ist eines der Standardverfahren zur Lösung großer Probleminstanzen. Mit einer starken Anwendung im Operations Research entstanden, ist sie dort eines der Hauptverfahren zur Lösung linearer Probleme, indem sie eine sehr allgemeine Vorgehensweise zur Verfügung stellt, um auch Probleme lösen zu können, für die noch keine speziellen Verfahren bekannt sind.

Die lineare Programmierung geht so vor, dass sie das betrachtete Problem als System linearer Ungleichungen modelliert. Für das symmetrische Traveling Salesman Problem lässt sich eine solche Formulierung finden. Gegeben ist eine Menge von n Städten und ein Kostenvektor $c \in \mathbb{R}^{n(n-1)/2}$. Dies entspricht der Zeile einer Distanzmatrix. Eine Tour wird weiterhin als Inzidenzvektor $x \in S \subset \{0,1\}^{n(n-1)/2}$ aus der Menge der möglichen Touren S angegeben. Der Vektor gibt dabei in der i-ten Komponente an, ob Kante i in der Tour enthalten ist. Damit lassen sich die Kosten einer Tour berechnen durch $c^T x$ und eine äquivalente Formulierung des TSPs ist daher die Suche nach $\min\left(c^T x\right)$, mit $X \in S$. Diese Formulierung hat aber den Nachteil, dass sie so nicht ohne weiteres lösbar ist. Daher werden die Bedingungen relaxiert, was nichts anderes bedeutet als, dass sie anschaulich ein bißchen weniger streng beachtet werden. Das entstehende und einfacher zu handhabende Problem ist die Suche nach

$$\min\left(c^T x\right), \text{ mit } Ax \leq b \,. \tag{2.5}$$

$Ax \leq b$ ist ein lineares Ungleichungssystem, das von allen $x \in S$ erfüllt wird. Das heisst, dass jede Lösung des ursprünglichen Problems auch eine Lösung von Ungleichung 2.5 ist, aber nicht zwingend umgekehrt. Die neue Formulierung hat eventuell einen größeren Lösungsraum, da nur die Forderung erfüllt wird, dass die urpsürunglichen Lösungen im neuen Lösungsraum enthalten sind. Nun kann diese Formulierung

als lineares Programm aber beispielsweise mit dem Simplex-Algorithmus berechnet werden und das Ergebnis ist eventuell zu klein. Für eine berechnete Lösung x' gibt es zwei Möglichkeiten. Entweder ist $x' \in S$, womit eine optimale Tour gefunden wäre, oder es ist nur eine untere Schranke. Im zweiten Fall existiert wegen der Konvexität des Lösungsraum eine weitere lineare Ungleichung, die zwar von allen Punkten aus S erfüllt wird, aber eben nicht von x'. Diese Ungleichung wird als Schnittebene bezeichnet und wird dem Ungleichungssystem $Ax \leq b$ hinzugefügt. Dieses Vorgehen wird nun iterativ soweit fortgesetzt, bis der Lösungraum so weit beschnitten ist, dass nur noch die optimale Tour als Lösung gefunden wird. Das Problem ist also, geeignete Schnittebenen zu finden. Gerne werden hier sogenannte Hypergraphen-Schnitte verwendet, die Kanten erlauben, die über mehrere Kanten eines Graphen gehen. Ein Beispiel hierfür sind die *subtour inequalities*.

Es gibt effiziente Möglichkeiten, das Ergebnis der vorherigen Berechnungen bei Hinzunahme einer weiteren Ungleichung in dem Ungleichungssystem weiterzuverwenden und so iterativ zu einer Lösung des symmetrischen TSPs gelangen.

Viele Anwendung der Linearen Programmierung sind, wie eingangs erwähnt, im Operations Research beheimatet. Zu Ihnen zählen ganz alltägliche Dinge wie die optimale Verteilung von Containern auf Frachtschiffen unter Einbeziehung von Nebenbedingungen wie gesetzliche Bestimmungen, maximale Beladung oder Positionen, um bei Zwischenstationen der Reise schnell umgeladen werden zu können.

2.2.3. Heuristische Methoden

Im Vergleich zu den exakten Algorithmen liefern heuristische Verfahren nur „gute“ und nicht zwingend optimale Lösungen, dafür aber in relativ kurzer Zeit. Sie garantieren aber nicht, dass das Optimum während des Laufs gefunden wird. Diese Verfahren sind oft weniger komplex als exakte Algorithmen und die Laufzeiten sind um Größenordnungen kürzer. Man unterscheidet zwischen speziell auf das Problem zugeschnittenen Heuristiken und sogenannten *Meta-Heuristiken*. Letztere bieten einen generischen Ablauf, der in Teilschritten nur noch an das zu optimierende Problem angepasst werden muss. Bekannte Vertreter sind beispielsweise *Simulated Annealing* oder auch die *Evolutionären Algorithmen*.

Die Menge der heuristischen Verfahren speziell für das Traveling Salesman Problem wird gerne in die folgenden drei Unterklassen aufgeteilt:

- Algorithmen zur Tourkonstruktion
- Algorithmen zur Tourverbesserung
- Gemischte Verfahren

Algorithmen zur Tourkonstruktion berechnen eine Tour Knoten für Knoten. In jedem Schritt werden zu einer entstehenden Tour weitere Städte hinzugefügt. Algorithmen

zur Tourverbesserung optimieren eine bereits bestehende Tour Schritt für Schritt durch bestimmte Ersetzungen und Austausche. Die gemischten Verfahren vereinigen beide vorherigen Ideen. Der in einem solchen gemischten Verfahren benutzte Algorithmus zur Tourkonstruktion wird auch salopp als *Startheuristik* bezeichnet.

Die einfachste Herangehensweise konstruiert eine Tour für das metrische TSP in linearer Zeit. Gegeben sei eine Instanz $G = (V, E)$ für das metrische TSP. Daraus wird ein Eulerscher Graph durch eventuelles Verdoppeln von Kanten erzeugt, so dass jeder Knoten geraden Grad hat. Eine Eulertour $E(G)$ wird berechnet und mehrfache Vorkommen von Knoten auf dieser Tour werden bis auf das erste Vorkommen ignoriert. Da die Dreiecksungleichung gilt, ist die Gesamtlänge dieser konstruierten Tour höchstens $l(E(G))$.

Die besten Ergebnisse liefern allgemein die Tourverbesserungsalgorithmen, die ausgehend von einer *Starttour* nach Verbesserungen suchen. Interessanterweise gibt es auch Ansätze, exakte Algorithmen und heuristischen Verfahren zu kombinieren, um erfolgreich die Laufzeit auf der Suche nach dem Optimum zu beschränken.

2.3. Heuristiken

Nach dem einführenden Beispiel, über den Umweg einer Eulertour eine zulässige Lösung für das metrische TSP zu konstruieren, liegen mehrere einfache Heuristiken nahe. Eine der bekanntesten ist die *Nearest Neighbor Heuristik*.

2.3.1. Nearest Neighbor

Bei dieser Heuristik wird mit einem beliebigen Knoten begonnen. Der Grundgedanke ist, dass benachbarte Knoten auf der Tour auch möglichst nahe im Eingabegraphen beieinander liegen. Daher wird bei der Konstruktion der Tour per *Nearest Neighbor* immer der vom zuletzt hinzugefügten Knoten der Tour am nächsten liegende noch nicht besuchte Knoten im Graphen als neuer Knoten in die Tour aufgenommen. Dieses Schätzverfahren ist jedoch oft zu gierig. Die zuerst gefundenen Distanzen sind relativ kurz, aber zum Ende des Algorithmus werden sie immer länger. Die *Nearest Neighbor Heuristik* liefert nur recht schlechte Ergebnisse, wie erste Simulationen, als auch eine theoretische Worst-Case-Analyse zeigen.

Die Simulationen wurden mit je 5000 unabhängigen und randomisierten Läufen auf den Instanzen *lin318* und *berlin52* aus der TSBLIB [Rei91] durchgeführt. Die Ergebnisse der Läufe wurden in einer Liste gespeichert und mit dem Statistiktool *GNU R* [gnu07] aufbereitet.

In beiden Simulationen ist die Näherung an das Optimum nur sehr grob. Selbst das in den Testläufen jeweils gefundene Minimum weist eine deutliche Differenz zum bekannten Optimum auf. Für die Instanz berlin52 sind dies etwa $8,5\%$ und für lin318 bereits

	Optimum	Minimum	Median	Mittelwert	Maximum
lin318	42029	49201	52680	52580	58850
berlin52	7542	8181	9290	9374	10298

Tabelle 2.1.: Ergebnis der Simulationen auf den Instanzen lin318 und berlin52

das Zweifache mit knapp 17%. Dabei ist anzumerken, dass das gesehene Minimum der Testläufe nur selten auftaucht und, wie in Tabelle 2.1 zu sehen ist, im Mittel durchaus noch schlechtere Ergebnisse zu erwarten sind. Die Abbildungen 2.2 und 2.3 zeigen die Histogramme der Testläufe auf beiden Instanzen. Die häufigsten Ergebnisse sind deutlich schlechter als das Optimum und streuen zuden auch noch über einen großen Bereich. Wie es scheint, sinkt die Qualität der berechneten Lösungen mit steigender Größe der Eingabeinstanzen.

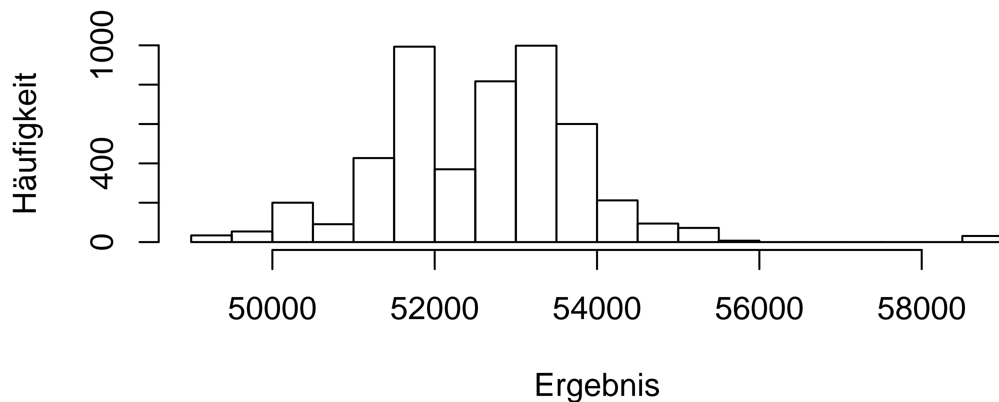

Abbildung 2.2.: Histogramm der 5000 Läufe auf der Instanz lin318

Neben der experimentellen gibt es für die Güte der Nearest Neighbor Heuristik auch noch eine theoretische Worst-Case-Analyse, die diese schlechten Ergebnisse erklärt. Insbesondere der Verdacht, dass die Nearest Neighbor Heuristik mit wachsender Knotenanzahl immer schlechtere Ergebnisse liefert, wird erhärtet.

Satz 3 (Approximationskomplexität der Nearest Neighbor Heuristik)
Die Nearest Neighbor Heuristik ist eine $\frac{1+\lceil log\ n\rceil}{2}$*-Approximation für das metrische Traveling Salesman Problem auf vollständigen, ungerichteten Graphen der Größe* n*.*

Bemerkenswert ist die Tatsache, dass die Anzahl der Knoten bei der Abschätzung des Worst-Case eine deutliche Rolle spielt. Der Beweis dieser Aussage lässt sich nach

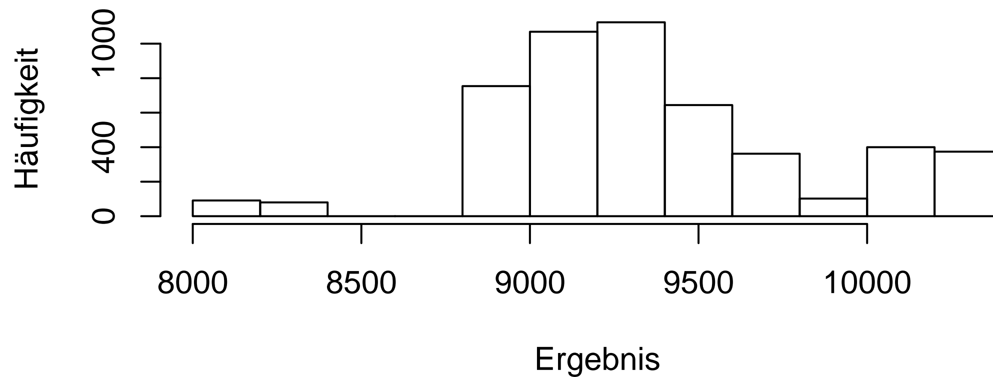

Abbildung 2.3.: Histogramm der 5000 Läufe auf der Instanz berlin52

[LLRKS85] herleiten:

Beweis: Die Lösung der *Nearest Neighbor Heuristik* ist eine Permutation π der Knoten, die der Reihenfolge der Rundreise entspricht. $(\pi_1, \ldots, \pi_n)$ entspricht in offensichtlicher Weise der Tour $t = (\pi_1, \ldots, \pi_n, \pi_1)$. Nun wird folgendes definiert:

$$l(\pi(s)) := c(\pi(s), \pi(s+1)),\ 1 \leq s \leq n-1$$

$$l(\pi(n)) := c(\pi(n), \pi(1))$$

Jedem Knoten i wird die Länge (Kosten) $c(i)$ derjenigen Kante der Tour $t \in T$ zugeordnet, die in t den Knoten i verlässt und zu seinem Tournachfolger führt. Da jede Kante der Tour in der Menge $\{l(1), \ldots, l(n)\}$ genau einmal vorkommt, entspricht die Länge der Tour

$$l(t) = \sum_{i=1}^{n} l(i). \tag{2.6}$$

Dadurch wird implizit die Funktion $l : \{1, \ldots, n\} \rightarrow \mathbb{N}$ definiert. Für diese Funktion lassen sich nun die drei folgenden Eigenschaften zeigen:

a. $c(i,j) \geq \min\{l(i), l(j)\}$, für alle $1 \leq i < j \leq n$

b. $l(i) \leq \frac{1}{2} \cdot l(t_{opt})$, für alle $1 \leq i < j \leq n$

c. $\sum_{i=1}^{n} l(i) \leq \frac{1}{2} \cdot (1 + \lceil \log n \rceil) \cdot l(t_{opt})$

Beweise zu

a: Sei (i, j) eine Kante in der Tour t. Es gibt also s und r mit $\pi(s) = i$ und $\pi(r) = j$. O.B.d.A sei $s < r$. Der Knoten i wird also von der Nearest Neighbor Heuristik vor dem Knoten j in die Tour eingefügt. Zu dem Zeitpunkt, zu dem $\pi(s)$ (Knoten i) fixiert wird, ist $\pi(r)$ (Knoten j) noch als freier Knoten vorhanden und wird irgendwann in die Liste der Tour mitaufgenommen. Die Kante (i, j) ist zu diesem zeitpunkt noch zu bestimmen. Die Heuristik wählt nun eine Kante $(\pi(s), \pi(s+1))$ mit der Länge $l(\pi(s)) = l(i)$. Es gilt $l(i) \leq c(i, j)$, denn entweder ist $\pi(s+1) = j$, wobei dann $l(i) = c(i, j)$ offensichtlich gelten muss, oder $\pi(s + 1) \neq j$, wobei dann $l(i) < c(i, j)$ gilt.

Der Fall $r < s$ ergibt sich analog.

b: Sei T_{opt} eine optimale Tour mit der Länge $l(T_{opt})$. Die Knoten i und j liegen irgendwo auf dieser Tour. Die optimale Tour kann in zwei Teilwege T^1_{opt} und T^2_{opt} zerlegt werden, die i und j verbinden. Teilweg T^1_{opt} läuft dabei von Knoten i nach j und T^2_{opt} von j nach i. Aus der Dreiecksungleichung folgt, dass die direkte Kante (i, j) nicht länger sein kann, als T^1_{opt} (bzw. T^2_{opt}). Die Summe beider Teilwege ist gerade $l(T_{opt})$, daher gilt $2 \cdot c(i, j) \leq l(T_{opt})$ und es folgt die Behauptung.

c: Es gibt offensichtlich eine Nummerierung der Knoten, so dass folgende Eigenschaft gilt: $l(1) \geq \ldots \geq l(n)$ Diese Nummerierung wird im weiteren Verlauf angenommen. Es gelte nun auch folgende Hilfsbehauptung:

$$l(T_{opt}) \geq 2 \cdot \sum_{k<i\leq 2k,n} l(i),\ k = 1 \ldots, n \tag{2.7}$$

Es wird über die $l(i)$-Werte in Blöcken im Intervall einer Zweierpotenz summiert. Diese Blöcke sind die folgenden: $[1]$, $[2]$, $[3, 4]$, $[5, 6, 7, 8]$, $[9, \ldots, 16]$, usw. bis zum letzten Block $[2^{L-1} + 1, \ldots, n]$ mit $L = \lceil \log n \rceil$. Für den ersten Block $[1]$ gilt nun $l(T_{opt}) \geq 2 \cdot l(1)$, was aus *b.* folgt. Für jeden der $L = \lceil \log n \rceil$ vielen anderen Blöcke wird höchstens $\frac{1}{2} \cdot l(T_{opt})$ zur Summe $\sum_{1\leq i\leq n} l(i)$ hinzuaddiert. Durch Aufsummierungen wird nun deutlich, dass folgende Beziehung gilt:

$$(L + 1) \cdot \frac{1}{2} l(T_{opt}) \geq l(1) + \sum_{1\leq s\leq L} \left(\sum_{2^{s-1}<i\leq min\{2^s,n\}} l(i) \right) = \sum_{1\leq i\leq n} l(i),$$

woraus die Behauptung *c.* folgt. Bleibt noch zu zeigen, dass Hilfsbehauptung 2.7 gilt:

Angenommen, es gilt $2k \leq n$. T_{opt} sei wieder eine optimale Tour mit $T_{opt} = (\pi(1), \ldots, \pi(n), \pi(1))$ und der Länge $l(T_{opt})$. Auf dieser Tour kommen die Knoten $1, \ldots, 2k$ in irgendeiner Reihenfolge vor. Es gibt daher $i_1 < \cdots < i_{2k}$ mit

$\{\pi(i_1), \ldots, \pi(i_{2k})\} = \{1, \ldots, 2k\}$. Werden nur diese Knoten betrachtet, so ergibt sich durch Weglassen der übrigen Knoten eine abgekürzte Rundreise:

$$K_k = (\pi(i_1), \ldots, \pi(i_{2k}), \pi(i_1))$$

Aus der Dreiecksungleichung folgt, dass die Gesamtlänge der $2 \cdot k$ vielen Kanten in K_k nicht größer sein kann als die Gesamtlänge der ursprünglichen Rundreise K, also:

$$l(T_{opt}) \geq \sum_{t=1}^{2k} w_{\{\pi(i_t),\pi(i_{t+1})\}} + w_{\{\pi(i_{2k}),\pi(i_1)\}}$$

Nun lässt sich Ungleichung $a.$ auf jede der Kanten in der obigen Abschätzung anwenden. Es folgt

$$l(T_{opt}) \geq \sum_{t=1}^{2k} \min\{l(\pi(i_t), l(\pi(i_{t+1}))\} + \min\{l(\pi(i_{2k}), l(\pi(i_1))\} \tag{2.8}$$

In Ungleichung 2.8 kommt jede Kante aus der gekürzten Rundreise K_k genau einmal vor und es wird der jeweils kleinere l-Wert des Knotens zur Summe hinzugezählt. Für jeden Knoten i in K_k wird also der Wert $l(i)$ entweder 0-, 1- oder zweimal zur Summe hinzuaddiert. Der Wert der Summe wird also nicht kleiner, als die doppelte Summe über die k kleinsten l-Werte. Nach der zu Anfang des Teils $c.$ angenommenen Ordnung sind dies gerade die Werte $l(k+1), \ldots, l(2k)$. Also folgt insgesamt:

$$l(T_{opt}) \geq \sum_{t=k+1}^{2k} 2 \cdot l(\pi(i_t))$$

Für den Fall $2k \leq n$ entspricht dies genau der Hilfsbehauptung 2.7. Der Fall $2k \geq n$ ergibt sich analog. □

Die *Nearest Neighbor Heuristik* für das metrische TSP ist also nur ein schlechtes Schätzverfahren, da es im schlimmsten Fall nur einen Approximationsfaktor von $O(log_2 n)$ garantiert. Dennoch ist es möglich, für das metrische Traveling Salesman Problem eine viel bessere Approximation zu finden, wie die folgende Heuristik zeigt.

2.3.2. Minimum-Spanning-Tree (Double-Tree) Heuristik

Die *Minimum-Spanning-Tree Heuristik* wird auch Double-Tree Heuristik genannt. Sie erzeugt mittels eines minimalen Spannbaums eine 2-Approximation für das metrische TSP. Das einfache Verfahren läuft wie folgt:

Algorithmus 2 Minimum-Spanning-Tree Heuristik

Eingabe: Graph G

Ausgabe: Approximation einer optimalen Tour

1: Finde einen minimalen Spannbaum T_{min} für G.
2: Bilde Multigraphen $G = (V, E_G)$, wobei jede Kante aus T_{min} doppelt vorkommt.
3: Finde einen Eulerkreis EK in G.
4: Kürze den Eulerkreis EK zu einem Hamiltonkreis HK ab und gib diesen aus.

Ein *Multigraph* $G = (V, E)$ ist eine Erweiterung eines regulären Graphen mit der zusätzlichen Eigenschaft, dass eine Kante mehrfach vorkommen kann. Formal lassen sich Kanten statt einer Menge als eine Funktion $m : V \times V \mapsto \mathbb{N}$ auffassen, die jedem Kantenpaar $\{u, v\}$ eine nicht negative, ganze Zahl $m(u, v)$ zuordnet. Diese Zahl wird als die *Vielfachheit* der Kante bezeichnet. Beispielsweise würde $m(u, v) = 0$ bedeuten, dass die Kante (u, v) im Multigraphen nicht vorkäme, $m(u, v) = 1$ würde auf eine gewöhnliche Einfachkante hindeuten und $m(u, v) > 1$ schließlich würde eine Mehrfachkante angeben. Ein Multigraph lässt sich nun als geordnetes Paar (V, m) begreifen, wobei wie gesagt $m : \binom{V}{2} \mapsto \{0, 1, \ldots\}$.

Definition 2 (Euler-Kreis und Euler-Graph)
Sei $G = (V, m)$ ein ungerichteter, zusammenhängender Multigraph.

1. *Eine Abfolge von Knoten $(v_0, v_1, \ldots, v_k)$ heisst Euler-Kreis, bzw. Euler-Tour in G, wenn $(v_{i-1}, v_i) \in V, 1 \leq i \leq k$ Kanten in G sind, $v_k = v_0$ gilt und jede Kante in G in der Abfolge genau einmal vorkommt.*

2. *G wird Euler-Graph oder Eulerscher Multigraph genannt, wenn dieser einen Euler-Kreis besitzt.*

Euler-Kreis und Hamilton-Kreis werden oft verwechselt. Der Unterschied liegt darin, dass ein Euler-Kreis eine Rundreise über alle **Kanten** des Graphen ist, wobei ein Hamilton-Kreis eine Rundreise über aller **Knoten** des Graphen darstellt. Interessanterweise ist es ohne Probleme möglich, effizient Eulersche Graphen zu erkennen. Ein wichtiger Fakt ist folgende Aussage:

Satz 4 (Eulerscher Multigraph)
Ein zusammenhängender Multigraph $G = (V, E)$ ist genau dann eulersch, wenn jeder Knoten geraden Grad besitzt.

Der Beweis [LLRKS85] der Äquivalenz erfolgt in zwei Richtungen.

„$\Rightarrow$“ Diese Richtung ist einfach zu zeigen. Der Eulerkreis verlässt offensichtlicherweise jeden Knoten genauso oft wie er ihn betritt. Bei jedem Betreten eines Knotens werden also zwei Kanten benutzt. Da am Ende der Rundreise alle Kanten benutzt sein müssen, hat jeder Knoten einen geraden Grad.

„$\Leftarrow$“ Hier wird eine Induktion über die Anzahl der Kanten in G geführt. Besitzt G eine leere Kantenmenge, dann besteht er nur aus einem Knoten v und dem dazugehörigen (entarteten) Eulerkreis (v, v). Besitzt G nur eine Kante, dann ist es unmöglich, dass dieser Graph einen Euler-Kreis besitzt. Bei zwei Kanten und geradem Grad an jedem Knoten muss G klarerweise aus zwei Knoten und einer Doppelkante bestehen. Dies ist dann der Eulerkreis.

Nun wird angenommen, dass G mindestens 3 Kanten besitzt. Ein sogenannter *einfacher Kreis* $K = (v_0, \ldots, v_l)$ ohne Knotenwiederholung lässt sich wie folgt konstruieren: Beginne mit einem beliebigen Knoten v_0. Da G zusammenhängend ist, hat v_0 einen Grad ≥ 2 und mithin existiert eine Kante (v_0, v_1). Diese wird nun markiert. Da nun v_1 geraden Grad hat, existiert wiederrum eine unmarkierte Kante (v_1, v_2). Ganz analog werden nun die weiteren Knoten $v_3, v_4, \ldots$ gefunden, bis zu irgendeinem Zeitpunkt $v_i = v_j$, mit $i < j$ gefunden wird. Der einfache Kreis wird dann aus der Knotenabfolge $(v_i, v_{i+1}, \ldots, v_j)$ gebildet.
Man löscht nun die Kanten des einfachen Kreises K aus dem Graphen G. Der Graph kann dadurch in die Zusammenhangskomponenten $G_1, \ldots, G_x$ zerfallen. Da alle Knoten v_i im Kreis geraden Grad haben, behalten die Knoten in den Zusammenhangskomponenten ebenfalls geraden Grad. An jedem Knoten wird ja auch nur eine gerade Anzahl Kanten gelöscht! Nun folgt mit der Induktionsvorraussetzung, dass jede Komponente $G_i, i = 1 \ldots x$ für sich wieder einen Euler-Kreis besitzen muss. Da aus dem einfachen Kreis K und den x Zusammenhangskomponenten auf naheliegende Weise ein gemeinsamer Euler-Kreis über alle Kanten konstruiert werden kann, folgt die Aussage. □

Der letzte Schritt des Algorithmus ist naheliegend. Aus der in Schritt 3 entstandenen Euler-Tour $(v_0, \ldots, v_{m-1})$ werden diejenigen Knoten v_j gestrichen, für die $v_i = v_j$ mit $i < j$ gilt. Es werden also bei mehrfachem Auftauchen eines Knoten alle Vorkommen bis auf das erste gestrichen. Aus der Dreiecksungleichung des metrischen TSP folgt, dass die resultierende Hamilton-Tour auf jeden Fall kürzer ist als die Euler-Tour. Interessanterweise lässt sich zeigen, dass die Spannbaum-Heuristik eine 2-Approximation ist.

Satz 5 (2-Approximation durch Minimum-Spanning-Tree)
Die Minimum-Spanning-Tree Heuristik ist eine 2-Approximation für das metrische Traveling Salesman Problem auf vollständigen, ungerichteten Graphen [LLRKS85].

Beweis: Sei der Graph G die Eingabe des Algorithmus. $K = (v_0, v_1, \ldots, v_n)$ (es gilt: $v_0 = v_n$) sei ein Hamilton-Kreis minimaler Länge l. $w(x)$ bezeichnet die Kosten der Traversierung einer Kantenliste x. Wird die Kante (v_0, v_1) aus dem Hamilton-Kreis entfernt, so entsteht ein offensichtlich einfacher Pfad über n Knoten. Dieser Weg ist also ein Spannbaum. Mit T' wird nun dieser einfache Pfad und mit T_{min} der minimale Spannbaum, den die Heuristik im ersten Schritt berechnet, bezeichnet. Da T_{min} ja ein minimaler Spannbaum ist, muss auch folgendes gelten:

$$w(T) \leq w(T') \leq w(K) = l$$

Da der Multigraph $G = (V, E_G)$ in Schritt 2 durch Verdoppeln der Kanten entsteht, muss ebenso gelten:

$$w(EK) = w(E_G) \leq 2 \cdot w(T) \leq 2 \cdot l$$

Da der Multigraph $G = (V, E_G)$ an jedem Knoten geraden Grad besitzt und mithin eulersch ist, folgt mit der Dreiecksungleichung:

$$w(HK) \leq w(EK) \leq 2 \cdot l \qquad \square$$

Die durch diesen Algorithmus gefundene Lösung ist im schlechtesten Fall also höchstens doppelt so lang wie die optimale Tour. Dennoch gibt es aber noch eine beweisbar bessere Schranke für das metrische Traveling Salesman Problem. Sie schafft es sogar auf einen Approximationsfaktor von 1.5.

2.3.3. Christofides-Heuristik

Im Jahr 1976 schlägt Christofides in [Chr76] folgende Heuristik zur Approximation des TSP vor. Sie ist ähnlich der Minimum-Spanning-Tree Heuristik und erzeugt aus einem Minimalen Spannbaum über ein perfektes Matching einen Eulerschen Graphen:

Algorithmus 3 Christofides-Heuristik

Eingabe: Graph G

Ausgabe: Approximation einer optimalen Tour

1: Finde einen minimalen Spannbaum T_{min} zu G.
2: Finde ein *perfektes Matching* M auf den Knoten des Spannbaums T_{min} mit ungeradem Grad.
3: Füge die entstehenden Kanten zu T_{min} hinzu. Der entstehende Graph $G_{T \uplus M}$ ist dann eulersch.
4: Konstruiere die Eulerkreis EK für T_{min}.
5: Kürze den Eulerkreis EK zu einem Hamiltonkreis HK ab und gib diesen aus.

Die Christofides-Heuristik unterscheidet sich also nur in der Konstruktion des Eulerschen Graphen von der Minimum-Spanning-Tree-Heuristik.

Definition 3 (Minimales perfektes Matching)
Gegeben sei ein Graph G. Eine Menge M von Kanten ist ein Matching, wenn jeder Knoten in G in höchstens einem Element $m \in M$ vorkommt. Ein perfektes Matching liegt dann vor, wenn jeder Knoten in G in genau einem Element von M vorkommt. Offensichtlicherweise liegt ein perfektes Matching genau dann vor, wenn die Knotenanzahl n gerade ist. Wenn die Kosten einer Paarung die Summe ihrer Kantengewichte sind, dann besitzt ein minimales perfektes Matching unter allen perfekten Paarungen eines Graphen minimale Kosten. Die Funktion $\sum_{e \in M} c(e)$ wird also minimiert.

Analog zum TSP interessiert bei der Suche nach einem minimalen, perfekten Matching der Fall, dass ein vollständiger, gewichteter Graph vorliegt. Das Problem der minimalen, perfekten Paarung liegt in $\mathcal{P}$. Lösungsalgorithmen finden sich in [PS82] und [CR99] und die Laufzeiten sind schlimmstenfalls in der Größenordung von $O(n^3)$.

Interessanterweise ermöglicht die wiederholte Anwendung der Dreiecksungleichung folgende Aussage:

Satz 6 ($\frac{3}{2}$-Approximation durch Christofides)
Die Christofides-Heuristik besitzt eine Approximationsgüte von $\frac{3}{2}$ für das metrische Traveling Salesman Problem auf vollständigen, ungerichteten Graphen.

Beweis: Damit Schritt 2 korrekt funktionieren kann, muss zunächst kontrolliert werden, ob der vollständige Graph auf den Knoten des Spannbaums T_{min} ein perfektes Matching besitzt, also die Anzahl dieser Knoten gerade ist. Dazu sei eben nun $V' = \{v \in \{1, \ldots, n\} | \ v \text{ hat ungeraden Grad}\}$. Nun gilt für jeden Graphen und damit auch für V':

$$\sum_{v \in V} deg(v) = 2|T_{min}|$$

Da nun die Summe der Grade eine gerade Zahl ergibt, muss als Schlußfolgerung die Anzahl der Knoten in V' (der Knoten mit ungeradem Grad in T_{min}) gerade sein.
In Schritt drei wird zu jedem Knoten, der in T_{min} ungeraden Grad hat, eine Kante hinzugefügt. Der Grad wird also um eins erhöht und damit gerade. Die Knoten im minimalen Spannbaum T_{min} mit geradem Grad bleiben davon unberührt. Mit Satz 4 folgt also, daß der entstehende Graph eulersch ist, denn jeder Knoten hat nun geraden Grad.

Aus dem Beweis zu Satz 5 ist bekannt, dass für einen Hamiltonkreis K mit minimaler Länge l folgende Beziehung gilt:

$$w(T_{min}) \leq w(K) = l \tag{2.9}$$

Das Gewicht des Matchings M lässt sich nun wie folgt betrachten. Der Hamilton-Kreis K erzeugt durch (simples) Übergehen der Knoten in $T_{min} \backslash V'$ einen Hamiltonkreis $K' = v_1, \ldots, v_m$ in der Knotenmenge V'. Aus der Dreiecksungleichung folgt:

$$w(K') = \sum_{1 \leq i \leq m} w(v_{i-1}, v_i) \leq w(K) = l$$

Nun lässt sich der Kreis K in zwei perfekte Matchings zerlegen, da er selbst gerade Länge besitzt:

$$M_1 = \{(v_0, v_1), (v_2, v_3), \ldots, (v_{m-2}, v_{m-1})\}, \text{ sowie}$$

$$M_2 = \{(v_1, v_2), (v_3, v_4), \ldots, (v_{m-1}, v_m)\}.$$

Da sowohl M_1, als auch M_2 perfekt sind und M ein minimales Matching ist, muss gelten:

$$w(M) \leq w(M_1) \text{ ,sowie } w(M) \leq (w(M_2)$$

Zusammengesetzt folgt:

$$2 \cdot w(M) \leq w(M_1) + w(M_2) = w(K') \leq l$$

Aus obiger Zeile und Ungleichung 2.9 folgt nun:

$$w(G_{T \uplus M}) \leq \frac{3}{2} \cdot l$$

Also gilt folgerichtig

$$w(HK) \leq w(EK) = w(G_{T \uplus M}) \leq \frac{3}{2} \cdot l. \qquad \square$$

Die Christofides-Heuristik garantiert also schlechtestenfalls um den Faktor 1.5 falsch zu liegen. Für diesen einfachen Algorithmus mit Laufzeit $O(n^3)$ ist dies eine durchaus beachtliche Leistung. Ein Tourkonstruktionsalgorithmus mit einem besseren Approximationsfaktor ist bis dato nicht gefunden worden und wäre ein beachtliches Ergebnis. Die meisten naheliegenden Verfahren sind schlechter, auch wenn ihre Grundidee sehr eingängig ist. Sie erweitern eine Liste von ursprünglich ein oder zwei Knoten derart, dass neue Knoten irgendwo in diese Liste eingefügt werden.

2.3.4. Einfügeheuristiken

Die Heuristiken in diesem Abschnitt besitzen alle eine Gemeinsamkeit. Sie starten die Tourkonstruktion von einer initialen (entarteten) Tour T aus, die nur aus einem Knoten mit Eigenschleife oder aus einem Kreis mit zwei Knoten besteht und fügen nach gewissen Regeln solange weitere Kanten ein, bis eine komplette Tour entsteht. Oberflächlich ähneln die Einfügeheuristiken der einfachen Nearest Neighbour Heuristik. Die folgenden Ergebnisse zu Laufzeit und Güte sind dem Buch von Lawler et al. [LLRKS85] entnommen.

Zum Verständnis ist es nützlich, eine erweiterte Kostenfunktion $c(i, k, j)$ zu betrachten, die das Einfügen des Knoten k in eine Tour zwischen die zuvor adjazenten Knoten i und j wiedergibt.

$$c(i, k, j) = c(i, k) + c(k, j) - c(i, j)$$

Dieser Schritt ist in jeder Einfügeheuristik zu finden und wird in Abbildung 2.4 skizziert.

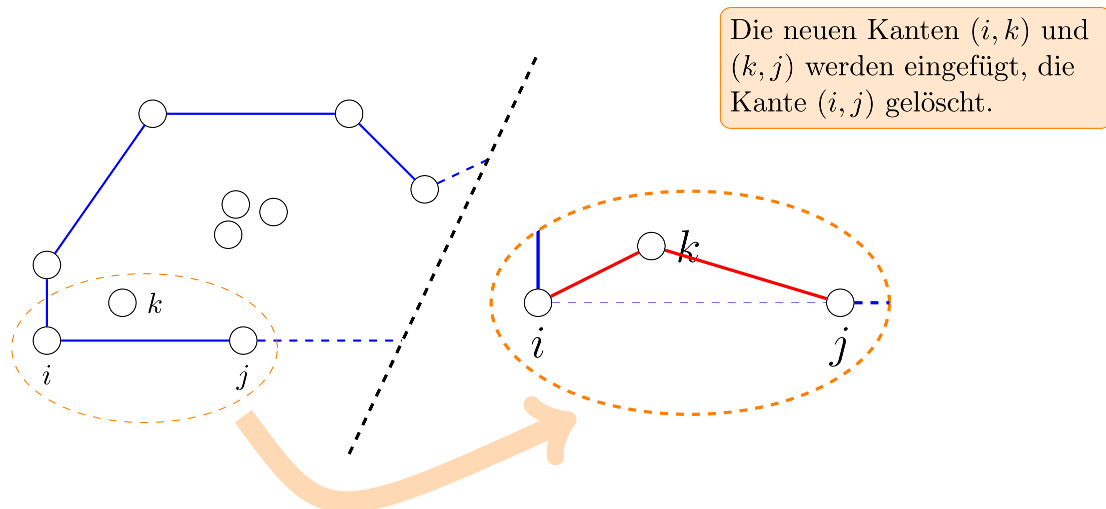

Abbildung 2.4.: Arbeitsschritt einer Einfügeheuristik

Auf der linken Seite ist in blau die zwischenzeitliche Tour vor dem Einfügeschritt skizziert. Rechts in der Vergrößerung zeigen zeigen sich in rot die neu hinzugefügten Kanten und darunter gestrichelt die gelöschte Kante in blau. Knoten, die noch nicht eingefügt wurden, heissen frei.

Random Insertion Heuristic

Der nächste in die Tour einzufügende Knoten wird zufällig unter den Knoten gewählt, die noch nicht in die Tour aufgenommen wurden. Die Einfügestelle, also die Kante, die aufgebrochen wird, ist diejenige, die die Kosten $c(i, j, k)$ minimiert. Das Verfahren endet, sobald der letzte freie Knoten in T eingefügt wurde. Die Laufzeit der Heuristik ist quadratisch und die approximierte Tour ist im schlimmsten Fall $log_2 n + 1 = O(log_2 n)$ mal länger als das Optimum.

Nearest Insertion Heuristic

In dieser Heuristik ist der nächste in die Tour T einzufügende Knoten derjenige freie Knoten $i \in V \backslash T$, der den geringsten Abstand zu einem Knoten j in der Tour hat. Die Einfügestelle ist diejenige, die die Kosten $c(i, j, k)$ minimiert. Das Verfahren bricht ab, wenn alle Knoten in die Tour aufgenommen wurden. Die Laufzeit ist ebenfalls quadratisch und die Heuristik erzeugt sogar eine 2-Approximation der optimalen Tour.

Farthest Insertion Heuristic

Die Farthest Insertion Heuristic dreht die Suche nach einem freien Knoten aus der vorherigen Heuristik um. Es ist also die Suche nach einem von T am weitesten entfernten Knoten. Der gefundene Knoten wird wie gehabt an der Stelle eingefügt, die $c(i, j, k)$ minimiert und das Verfahren endet, sobald die Menge der freien Knoten leer

ist. Die Laufzeit ist offensichtlich wieder quadratisch, aber die Güte des Verfahrens ist unbekannt. Es ist ein offenes Problem, wie gut hierbei die optimale Tour approximiert wird.

Cheapest Insertion Heuristic

Der nächste einzufügende freie Knoten $k \in V \backslash T$ ist derjenige, der $c(i, j, k)$ für alle Paare i, j, die in T adjazent sind, minimiert. Die Einfügestelle ist offensichtlich diejenige, für die $c(i, j, k)$ minimal ist. Das Verfahren endet, sobald alle Knoten in T eingefügt wurden. Die Laufzeit für eine naive Implementierung ist in $O(n^3)$, lässt sich aber durch geschickte Verwendung der passenden Datenstrukturen auf $O(n^2 \cdot log_2 n)$ begrenzen. Die gefundenen Lösungen sind maximal um den Faktor 2 schlechter als das Optimum.

Convex Hull Algorithmen

Für euklidische Instanzen lässt sich die Konstruktion einer Tour anschaulich auch umdrehen. Anstatt mit einem, bzw. zwei Knoten im Kreis zu beginnen und nach „aussen“ zu wachsen, wird mit einer konvexen Hülle begonnnen um dann Knoten aus dem „Inneren“ hinzuzufügen. Die Wahl der einzufügenden Knoten und die jeweiligen Einfügestellen werden wieder nach einem der vorherigen vier Einfügeschemata bestimmt. Das Verfahren ist also wie folgt:

Algorithmus 4 Convex-Hull Algorithmus

1: Berechne die konvexe Hülle für die Menge der n Knoten als Ausgangspunkt der Tourkonstruktion.
2: Benutze eine beliebige der vier vorherigen Einfügeheuristiken um die Tour zu vervollständigen.

Die Laufzeiten sind mit $O(n^3)$ für *cheapest insertion* und $O(n^2)$ für die anderen Einfügestrategien recht moderat. Das Verhalten der Convex-Hull Algorithmen im Worst-Case ist allerdings für keine Variante bekannt und immer noch ein offenes Problem.

2.4. Untere Schranken

Obwohl beispielsweise mit der TSPLIB [Rei91] Instanzensammlungen existieren, die jeweils Optima angeben, ist es im Allgemeinen schwer, eine Vorstellung zu entwickeln, ob eine gefundene Lösung optimal, nah am Optimum oder weit weg vom Optimum ist. Hier helfen untere Schranken, die nah an das Optimum heranreichen. Für eine gegebene Instanz I sollte, soweit möglich, die größtmögliche Zahl z als untere Schranke gefunden werden, so dass $l(t) \geq z$ für alle Touren t ist. Eine einfache Schranke ist die sogenannte 1-Baum-Schranke.

Diese Idee wird später erweitert um eine noch genauere Schranke zu finden, die sogenannte *Held-Karp-Schranke* [HK70].

2.4.1. 1-Baum-Schranke

Ein 1-Baum ähnelt den spannenden Bäumen für einen Graphen.

Definition 4 (1-Baum)
Gegeben ist ein vollständiger Graph $G = (V, E)$, mit $V = \{1, \dots, n\}$. Ein 1-Baum für G ist ein Spannbaum auf der Knotenmenge $V = \{2, \dots, n\}$, erweitert um zwei zusätzliche Kanten aus E, die inzident zum Knoten 1 sind.

Die Touren sind also gerade 1-Bäume, für die jeder Knoten im Baum den Grad 2 besitzt. Ein 1-Baum besitzt einen Kreis und der Knoten v_0 hat immer Grad 2. Ein minimaler 1-Baum, der an jedem Knoten Grad 2 besitzt, muss daher auch eine optimale Tour sein. Die Schranke lässt sich algorithmisch leicht bestimmen.

Algorithmus 5 1-Baum-Schranke

Eingabe: Graph $G = (V, E)$
Ausgabe: Untere Schranke für die optimale Tour
1: Wähle einen Knoten $v_0 \in V$.
2: Sei r die Länge eines minimalen Spannbaums M^* auf dem Graphen $G^* = (V - \{1\}, E)$.
3: Sei s die Summe der beiden kürzesten Kanten, die zu Knoten v_0 inzident sind. Also, $s = \min\Big(l(v_0, x) + l(v_0, y) \mid x, y \in V - \{v_0\}\Big)$.
3: Ausgabe: $r + s$

Eine beliebige Tour T muss offensichtlicherweise zwei zu v_0 inzidente Kanten e und f benutzen. Ohne diese beiden Kannten zerfällt T in einen Spannbaum M, der die Knoten $V - \{v_0\}$ aufspannt. Damit lässt sich folgendes abschätzen:

$$l(T) = c(e) + c(f) + l(R) \geq s + l(M*) = t.$$

Daher muss t also eine untere Schranke für T sein.

Ein 1-Baum kann anschaulich als Tour verstanden werden, die gelockerten Randbedingungen folgt. Diese Idee machen sich Held und Karp zu nutze.

2.4.2. Held-Karp-Schranke

Die Idee des 1-Baums wird von Held und Karp [HK70] über die Relaxierung der Kantengewichte erweitert. Angenommen $\pi = (\pi_1, \dots, \pi_n)$ sei ein Vektor reellwertiger Zahlen. Ein minimaler 1- Baum in Bezug auf π ist definiert als der minimale 1-Baum auf den geänderten Kantengewichten $\overline{c}(i, j) = c(i, j) + \pi_i + \pi_j$. Ist T_k dieser minimale 1-Baum in Bezug auf π, dann sei $w(\pi)$ definiert als

$$w(\pi) = \sum_{(i,j) \in T_k} \overline{c}_{ij} - 2 \sum_{i=1}^{n} \pi_i \ . \tag{2.10}$$

Der von der Held-Karp-Schranke gesuchte Wert ist $\max_\pi w(\pi)$ und dies entspricht anschaulich dem Vorgehen, dass die Schranke versucht den Vektor π so zu bestimmen, dass der minimale 1-Baum so groß wie möglich wird um immer noch eine untere Schranke zu sein.

Der gesuchte 1-Baum sollte so nah wie möglich an einer Tour sein, ohne die Kosten der optimalen Tour zu überschreiten. Da ein minimaler 1-Baum, der an jedem Knoten Grad 2 besitzt, offensichtlich eine optimale Tour ist, wird von Held und Karp ein iteratives Schema vorgeschlagen, das versucht schrittweise den Vektor π über die Relaxierung der Kantengewichte zu bestimmen.

Angenommen, für einen Graphen wird ein minimaler 1-Baum berechnet. Es ist klar, dass jeder Knoten entweder Grad 1, Grad 2 oder einen Grad größergleich 3 hat. Die gefunde Lösung darf die Eigenschaft, dass sie eine Tour ist verletzen (die Lösung ist jetzt ein Baum), aber dafür wird an den Stellen, an welchen die Eigenschaft verletzt wird, eine gewisse Bestrafung eingeführt. Diesem Vorgehen entspricht die Änderung der Kantengewichte und die Berechnung der $w(\pi)$. Hat ein Knoten v in einem minimalen 1-Baum in Bezug auf π einen Grad > 2, dann sollte sich ein π' finden lassen, für das gilt $w(\pi') > w(\pi)$. Dieses π' lässt sich finden, indem die u-te Komponente π_u des Vektors π erhöht wird. Der erste Summand auf der rechten Seite der Gleichung 2.10, nämlich $\sum_{(i,j)\in T_k} \overline{c}_{ij}$ wächst stärker als der zweite $-2\sum_{i=1}^{n} \pi_i$ abnimmt. Analog ist der Fall, dass ein Knoten u den Grad 1 hat. Es gibt ebenso ein π' für das gilt $w(\pi') > w(\pi)$, das sich dadurch finden lässt, dass diesmal π_u erniedrigt wird.

Im Schritt nach der Anpassung der Kantengewichte wird nun ein neuer minimaler 1-Baum in Bezug auf π' konstruiert. An Knoten, die im vorherigen Schritt Grad 1 hatten, ist es nun attraktiver, eine weitere inzidente Kante zum entstehenden Baum hinzuzunehmen, während durch die Relaxierung der Kantengewichte es an Knoten, deren Grad höher zwei wahr, unattraktiver wird, wieder mehr als 2 inzidente Kanten in den Baum aufzunehmen. Die Konstruktion des minimalen 1-Baums wird also immer weiter genötigt, nur Knoten mit Grad 2 zu wollen. Die gesuchte Lösung soll also möglichst einer Tour entsprechen, darf aber einige Defekte ausweisen.

Die Korrektheit des Verfahrens folgt aus zwei zu zeigenden Eigenschaften. Erstens, ist t_{opt} eine optimale Tour für eine betrachtete Instanz, dann ist t_{opt} auch für die relaxierten Kantengewichte optimal und zweitens, $w(\pi)$ ist immer eine untere Schranke für Kosten $l(t_{opt})$ der optimalen Tour.

Satz 7 (Invariante der optimalen Tour für Held-Karp)
Sei $\pi = (\pi_1, \ldots, \pi_n)$ ein Vektor reellwertiger Zahlen. Wenn t_{opt} eine optimale Tour für die ursprünglichen Kantengewichte $c(i,j)$ ist, dann ist t_{opt} auch eine optimale Tour für die relaxierten Kantengewichte $\overline{c}(i,j) = c(i,j) + \pi_i + \pi_j$.

Die Länge einer Tour t ist die Summe ihrer Kantenlängen, also

$$\sum_{(i,j)\in t} c(i,j) \tag{2.11}$$

und in Bezug auf die relaxierten Kantengewichte $\overline{c}(i,j)$ ist die Länge der Tour

$$\sum_{(i,j)\in t} c(i,j) + \pi_i + \pi_j \tag{2.12}$$

Jeder Knoten in t ist inzident zu zwei Kanten. Werden die Gleichungen 2.11 und 2.12 voneinander abgezogen, folgt

$$\sum_{(i,j)\in t} c(i,j) - \sum_{(i,j)\in t} c(i,j) + \pi_i + \pi_j = 2 \cdot \sum_{i=1}^{n} \pi_i \tag{2.13}$$

und dies ist gerade die Änderung, um die sich die Kosten jeder Tour ändern. Jede Tour ist also durch den Übergang von einem Gewichtsvektor π zu π' gleich betroffen und die zuvor optimale Tour muss auch weiterhin optimal sein, denn die bestehende Ordnung nach ihrer Länge bleibt intakt. □

Satz 8 (Untere Schranke durch Held-Karp)
Sei t_{opt} eine optimale Tour, dann ist $w(\pi)$ immer eine untere Schranke für die Länge der optimalen Tour.

Nun sei t_{opt} wieder eine optimale Tour und t_{HK} der von Held-Karp produzierte minimale 1-Baum in Bezug auf π. Dann gilt:

$$\begin{aligned} w(\pi) &= \sum_{(i,j)\in t_{HK}} \overline{c}(i,j) - 2 \cdot \sum_{i=1}^{n} \pi_i \\ &\leq \sum_{(i,j)\in t_{opt}} \overline{c}(i,j) - 2 \cdot \sum_{i=1}^{n} \pi_i \\ &= \sum_{(i,j)\in t_{opt}} (c(i,j) + \pi_i + \pi_j) - 2 \cdot \sum_{i=1}^{n} \pi_i \\ &= \sum_{(i,j)\in t_{opt}} c(i,j) \end{aligned}$$

□

Wie lassen sich nun die π_i bestimmen? Held und Karp schlugen die *subgradiente Optimierung* (engl. subgradient optimization) als erfolgreichste Technik vor, um den Vektor π zu bestimmen. Gegeben ist eine konkave Funktion f. Ein Vektor s heisst ein *Subgradient für f bei $\overline{u}$*, wenn für alle u gilt $f(\overline{u}) + s(u - \overline{u}) \geq f(u)$. Sei $T_1, \ldots, T_i$ die

Aufzählung aller 1-Bäume in Bezug auf π_k, $1 \leq k \leq t$, die Held-Karp erzeugt. Für $v_k = (d_{1k} - 2, d_{2k} - 2, \ldots, d_{nk} - 2)$, wobei d_{ik} der Grad des i-ten Knotens in T_k ist, konnten Held und Karp zeigen, dass es ein Subgradient für $w(\pi)$ ist.

Die Idee, Kanten, die inzident zu Knoten mit hohem Grad sind, zu verteuern, lässt sich auch formalisieren. Für die Erzeugung der Sequenz der Vektoren $(\pi^1, \pi^2, \ldots)$ wird von Held und Karp folgende Berechnungsregel angegeben:

$$\pi^{m+1} = \pi^m + \lambda^m \left(\frac{\overline{w} - w(\pi^m)}{||v_k||^2} \right) v_k^m \tag{2.14}$$

Dabei wird $0 < \lambda \leq 2$ gewählt. Die Variable $\overline{w}$ ist ein sogenannter Justierwert, damit gilt $w(\pi) < \overline{w} \leq \max_\pi w(\pi)$. Für $d_{ik} < 2$ ist damit $\pi^{j+1} < \pi^j$ und für $d_{ik} < 2$ gilt $\pi^{j+1} < \pi^j$.

Folgende Skizzen verdeutlichen die Funktionsweise der Held-Karp-Schranke an einem vereinfachten Beispiel (nach Pygim).

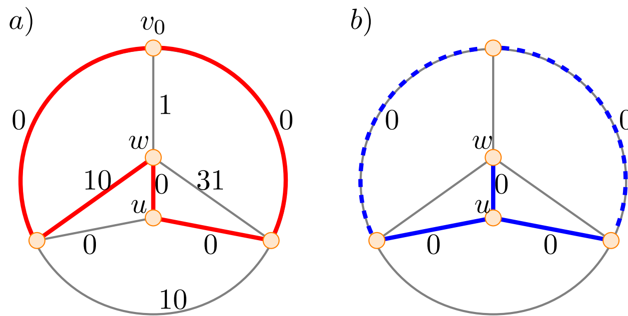

Abbildung 2.5.: Arbeitsschritt der Held-Karp-Schranke

Abbildung 2.5[1] zeigt eine Instanz für das symmetrische TSP und die offensichtlich optimale Tour t_{opt} in rot (a). Die Länge der optimalen Tour $l(t_{opt})$ ist 10. Mit v_0 als Knoten 1 ist der konstruierte 1-Baum (b) deutlich suboptimal.

[1] Die Kante (v_0, u) sei hier der Einfachheit halber vernachlässigt.

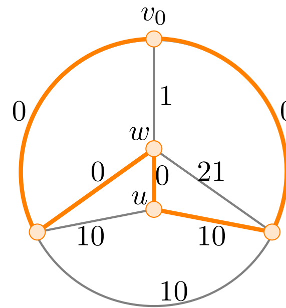

Abbildung 2.6.: Die Schätzung der Held-Karp-Schranke entspricht einer optimalen Tour

Die erste abgegebene Schätzung für die Länge einer optimalen Tour ist 0. In einem nächsten Schritt versucht die Held-Karp-Schranke nun, diese Schätzung zu verbessern und relaxiert die Kantengewichte, indem inzidente Kanten zu Knoten mit Grad > 2 teurer werden und inzidente zu Knoten mit Grad < 2 günstiger. Angenommen, die Kanten, die zum Knoten u inzident sind, werden nun mit zusätzlichen Kosten von 10 versehen und alle Kanten, die zum Knoten w inzident sind, werden im gleichen Zug um 10 verbilligt. Auf dem Graph mit diesen relaxierten Kanten wird ein neuer minimaler 1-Baum berechnet mit v_0 als Knoten mit der Nummer 1 und Knoten w als erster Knoten im minimalen Spannbaum. Abbildung 2.6 zeigt die entstehende neue Schätzung. Der Kantenzug des 1-Baums entspricht nun aber klar der optimalen Tour wie auch die abgegebene Schätzung $w(\pi) = 0 + 0 + 0 + 10 + 0 - 2(10 - 10) = 10$. Das Optimum wurde gefunden.

Die Leistung der Held-Karp-Schranke ist beachtlich. So berichten Johnson et al. [JMR96] von Ergebnissen auf zufälligen Instanzen, die bis unter ein Prozent an das Optimum heranreichen. Für Instanzen aus der TSPLIB wird das Optimum noch bis auf 2% approximiert und zudem berichten andere Autoren von ähnlich guten Ergebnissen auf weiteren Instanzen. Dennoch ist leider bis heute noch nicht genau verstanden worden, warum die Held-Karp-Schranke so nah an das Optimum herankommt. Auch ist bis heute noch keine praktikable Implementierung des Algorithmus bekannt, für die sich garantieren lässt, dass die Anzahl der Iterationen polynomiell beschränkt ist. Beides sind noch offene Probleme. Für die Schranke lässt sich garantieren, dass die abgegebene Schätzung immer mindestens 2/3 der Länge der optimalen Tour ist. Es gibt Instanzen, die so konstruiert werden können, dass die Schätzung auch wirklich diesen Faktor erreicht. Dennoch sind auf der überwältigenden Mehrheit der Instanzen sehr gute untere Schranken abschätzbarr. Daher wird die Schranke von Held-Karp immer dann gerne als Ersatz für ein Optimum benutzt, wenn dieses nicht bekannt ist.

3. Lokale Suche

Ein erfolgreiches Rezept für viele algorithmisch nur schwer handhabbare Probleme ist die *Lokale Suche*. Anstelle eines festen Verfahrens handelt es sich bei der Lokalen Suche eher um ein Optimierungsprinzip. Viele Algorithmen für schwierige Probleme, insbesondere das Traveling Salesman Problem, profitieren in der Praxis von diesem Prinzip.

Das Verfahren der Lokalen Suche ist ein sehr allgemeiner Ansatz für Optimierungen. Es geht von einer zulässigen, aber wahrscheinlich suboptimalen Lösung als Startpunkt aus und sucht nach einer Verbesserung durch die Modifaktion der aktuellen Lösung. Wird eine solche Verbesserung gefunden, dann startet die Lokale Suche erneut mit der zuvor gefundenen besseren Lösung als Eingabe. Dieser Prozess wird solange fortgesetzt, wie sich noch Verbesserungen finden lassen. Ansonsten bricht das Verfahren ab und gibt die zuletzt gefundene Verbesserung als Lösung aus.

Dieses Prinzip nutzt die Tatsache, dass jede Instanz eines Optimierungsproblems eine sogenannte Optimierungslandschaft besitzt. Angenommen es gibt eine Aufzählung aller möglichen Lösungen zu einer gegebenen, zu optimierenden Probleminstanz der Größe $|n|$. O.B.d.A. seien dies die möglichen Lösungen mit den Indizes von 0 bis $2^{|n|}$. Dann kann jeder Lösung x ein Wert $f(x)$ zugeordnet werden, der die Güte der Lösung x bewertet. Auch dieser liege zwischen 0 und einem Maximalwert y.

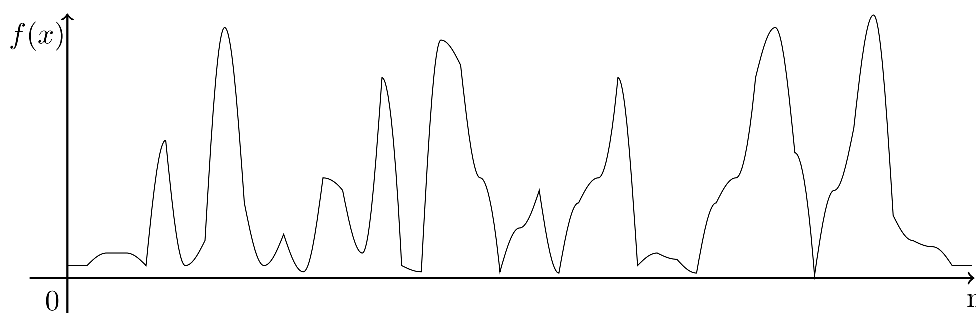

Abbildung 3.1.: Beispielhafte Optimierungslandschaft

Die Lokale Suche durchschreitet den Raum möglicher Lösungen nicht Punkt für Punkt, denn dies würde ja nur in exponentieller Zeit möglich sein. Es ist leicht vorstellbar, dass solch eine (gedachte) Optimierungslandschaft für eine Probleminstanz lokale Optima besitzt, deren zugehörige Lösungen in einer praktischen Anwendung voll ausreichend sind. Von einem bestimmten Startpunkt aus, also einer ersten zulässigen Lösung, wird

in der "Nähe" nach einer weiteren zulässigen Lösung gesucht, die möglicherweise auch noch eine bessere Bewertung besitzt. Die Menge derjenigen Lösungen, die von einer gegebenen Lösung durch einen Änderungsschritt, einer *atomaren Modifikation*, erreicht werden können, wird als die *Nachbarschaft* einer Lösung bezeichnet. Die Bestimmung dieser Nachbarschaft spezialisiert das Lokale Suchverfahren und ist von Problem zu Problem logischerweise unterschiedlich. Man mag versucht sein, die Nachbarschaft einer Lösung sehr gross zu wählen, um eine möglichst gute lokal optimale Lösung zu finden. Dies gibt sicherlich bessere Resultate als eine kleine Nachbarschaft. Die Laufzeit eines solchen Verfahrens erhöht sich aber. Ebenso gibt eine zu kleine Nachbarschaft schlechte Resultate, diese aber zügig. Wo keine guten Verhältnisse zwischen Laufzeit und Güte des Verfahrens theoretisch im vorhinein bestimmt werden können, geschieht dies oft experimentell. Die Idee der lokalen Suche lässt sich formalisieren:

Definition 5 (Lokale Suche)
Gegeben ist eine Eingabemenge X, der Instanzen $x \in X$. Weiterhin gibt es ein Polynom p. Für jedes $x \in X$ existiert eine Menge zulässiger Lösungen F_x.

Für jede Lösung $s \in F_x$ gibt es eine Menge an Nachbarn $N(s,x) \subseteq F_x$ und eine Maßzahl $M_x(s)$, die eine Lösung bewertet. Jedes $y \in N(s,x)$ ist durch eine sogenannte atomare Modifikation von s in polynomieller Zeit $p(|s|)$ aus erreichbar. Eine Lösung s heisst lokal optimal, wenn kein Element der Nachbarschaftsmenge $N(s,x)$ eine bessere Bewertung hat als x.

Die Bezeichnung *besser* leitet sich hier natürlich jeweils dabei davon ab, ob nun minimiert oder maximiert werden soll. Diese sehr allgemeine und auch flexible Vorgehensweise der Lokalen Suche hat den Vorteil, dass sie sich auf so ziemlich alle rechnerisch handhabbaren Probleme leicht anwenden lässt. Wichtig ist dabei, dass zwei fundamentale Fragen beantwortet werden:

- Was ist die erlaubte atomare Modifikation, sprich: Wie ist die Nachbarschaft definiert?
- Wann wird eine temporäre zulässige Lösung als Ausgangspunkt für eine weitere Suche gewählt?

Algorithmus 6 erläutert dieses Prinzip anhand einer Minimierung.

Algorithmus 6 Algorithmus Lokale Suche
Eingabe: Zulässige Lösung s für Probleminstanz x
Ausgabe: Lokal optimale Lösung
1: Finde Nachbarn $y \in N(s,x)$, so dass $M_x(y,x) < M_x(s,x)$.
2: Existiert kein solches y, dann Ende.
3: Setze s=y und weiter bei 1.

Wie lässt sich diese Definition nun auf das Traveling Salesman Problem anwenden? Die Elemente der Eingabemenge entsprechen in natürlicher Weise dem zur Probleminstanz

gehörenden Graphen. Eine zulässige Lösung ist eine Rundreise über alle Knoten und die einer Rundreise zugeordneten Maßzahl den Kosten, bzw. der Länge der Rundreise. Die Nachbarschaft einer Lösung s wird verschieden definiert, bestimmt sich aber meist über die Anzahl Kanten, die gelöscht, bzw. hinzugefügt werden, um eine Lösung in eine andere zu transformieren. O.B.d.A. wird angenommen, dass eine Eingabe für das Traveling Salesman Problem immer aus einem vollständigen Graphen besteht.

3.1. Nachbarschaft für das Traveling Salesman Problem

Ein einfaches Beispiel für eine Nachbarschaft in der Lokalen Suche ist das sogenannte *k-Opt-Verfahren* für das Traveling Salesman Problem. Es beruht auf der Idee der *k-Optimalität*:

Definition 6 (k-Optimalität)
Eine Tour T heisst k-optimal (kurz: k-opt), falls es unmöglich ist, eine kürzere Tour T' durch Tauschen von k vielen Kanten in der Tour gegen k beliebige andere Kanten, die nicht zur Tour gehören, zu erhalten.

Damit ist folgender Algorithmus naheliegend:

Algorithmus 7 k-Opt Algorithmus

Eingabe: Eine Eingabeinstanz $x \in X$
Ausgabe: Eine Tour
1: Wähle eine Starttour T.
2: **while** $(\exists T' \in N(T,x))(M_x(T') < M_x(T))$ **do**
3: $T = T'$.
4: **end while**

Die Nachbarschaft einer Tour T für dieses Verfahren sind alle Lösungen T', die sich von T nur durch maximal k Paare von Kanten unterscheiden. Diese Nachbarschaft wird im Suchschritt durchlaufen und sobald ein T' gefunden wird, das niedrigere Kosten hat, wird dieses als neuer Ausgangspunkt gewählt. Aus obiger Definition ist ersichtlich, dass für jede k-optimale Tour T gilt, dass diese für $1 \leq k' \leq k$ auch k'-optimal ist. Ebenso gilt, dass eine Tour T genau dann optimal ist, wenn sie n-optimal ist. Intuitiv folgt, dass je größer k nun gewählt wird, desto eher die resultierende Tour das gesuchte Optimum ist.

Eine einfach zu implementierende Variante ist das 2-Opt-Verfahren. Hier wird $k = 2$ gesetzt und wie folgt vorgegangen: Starte mit einer gegebenen Tour. Zwei Kanten der Tour werden mit zwei Kanten des Graphen, die nicht in der Tour enthalten sind, derart getauscht, dass die resultierende Tour kürzer ist. Dies wird wiederholt, solange sich Tauschpartner finden lassen. Die Laufzeit liegt in einem sehr annehmbaren Verhältnis zum erzielten Ergebnis und ein solcher Tausch entspricht dem Umdrehen der Reihenfolge einer Teiltour der ursprünglichen Lösung. Grafik 3.2 verdeutlicht den

Tauschvorgang.

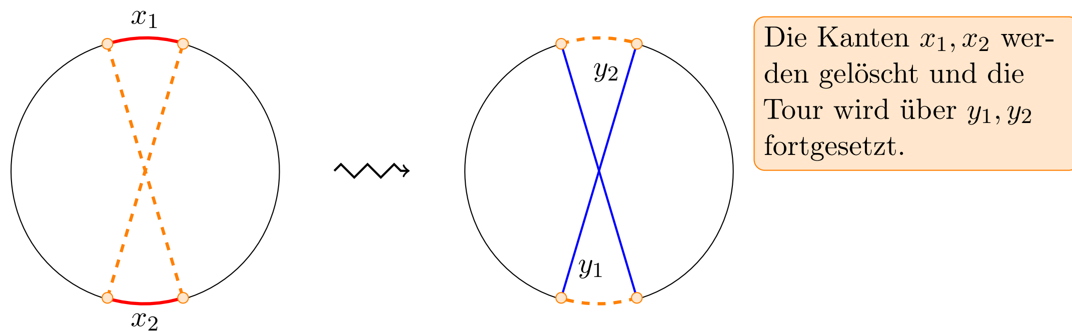

Abbildung 3.2.: Wahl der Kanten im 2-Opt-Verfahren

Es ist naheliegend, diese Suchidee auf größere Nachbarschaften der Länge 3, 4, etc. auszudehnen. Allerdings braucht diese Lösungsidee immer mehr Ressourcen. Sogar für kleine k wird die Laufzeit mit $\Theta(n^k)$ zu unhandlich. Daher wird dieses Verfahren auch nur sehr selten für $k \geq 4$ verwendet.

Chandra et al. [CKT94] berichten über die Qualität des k-Opt-Verfahrens. Für TSP-Instanzen, die die Dreiecksungleichung erfüllen, ist der Worst-Case Approximationsfaktor für 2-Opt höchstens $4\sqrt{n} = O(\sqrt{n})$. Für den k-Opt-Algorithmus gibt es unendlich viele Instanzen, die die Dreiecksungleichung erfüllen, für die ein Approximationsfaktor von mindestens $\frac{1}{4}n^{\frac{1}{2k}} = O\left(n^{\frac{1}{k}}\right)$ gilt. Weiter wird berichtet, dass Instanzen, die in $\mathbb{R}^m$ eingebettet sind, vom k-Opt-Vefahren mit einem Approximationsfaktor von schlechtestenfalls $O(\log n)$ gelöst werden.

Chandra et al. vermuten zwar für den k-Opt-Algorithmus eine Verbesserung der unteren Schranke von $\frac{1}{4}n^{\frac{1}{2k}}$ im Falle einer geschickt gewählten Starttour, können dies aber nur als offenes Problem formulieren.

3.2. Grundidee der Lin-Kernighan Heuristik

Als Nachteil des k-Opt-Verfahrens wird die starre Festlegung der Nachbarschaftsgröße angesehen. Es ist kaum machbar, k im Vorfeld der Optimierung zu bestimmen, um ein gutes Verhältnis zwischen Laufzeit und Qualität der Lösung zu erreichen. Lin und Kernighan führten als Abhilfe das *variable k-Opt-Verfahren* ein. Anstatt k im vorhinein zu bestimmen, entscheidet das Verfahren selbständig in jeder Iteration, wie weit die Schritttiefe k zu wählen ist. Für aufsteigende Werte von k wird jeweils neu entschieden, ob der Tausch von k vielen Kantenpaare eine Verbesserung bewirkt.

Ursprünglich formulierten Kernighan und Lin ihr variables Verfahren für das Problem der Graphpartitionierung und verallgemeinerten ihren Ansatz danach, um ihn anschließend auf das Traveling Salesman Problem anzuwenden. Bis heute gehört ihre Idee und deren Varianten immer noch zu den leistungsfähigsten Vorgehensweisen zur Optimierung der genannten Probleme.

Viele kombinatorische Probleme können abstrakt als 'Finde aus einer Menge S eine Untermenge T, die eine Bedingung C erfüllt und dabei die Funktion f minimiert' fomuliert werden. Ein allgemeiner Ansatz [LK73] für eine Heuristik ist dabei, ausgehend von einer initialen Lösung, die iterative Verbesserung:

1. Erzeuge eine initiale, eventuell zufällige Lösung. Dies ist die genannte Menge T, die die Eigenschaft C erfüllt.
2. Suche eine zulässige und verbesserte Lösung T', die über eine Transformation von T erreicht werden kann.
3. Wurde eine bessere Lösung gefunden, also ist $f(T') \leq f(T)$, dann ersetze T durch T' und wiederhole Schritt 2.
4. Kann keine verbesserte Lösung T' gefunden werden, dann ist T eine sogenannte *lokal optimale Lösung*. Bei Bedarf ab Schritt 1 wiederholen.

Es ist sinnvoll, die Initiallösung zufällig oder durch den Zufall beeinflusst zu bestimmen, da die Schritte 2 - 4 deterministisch ablaufen und sonst weitere Durchläufe, wie sie sich am Ende des letzten Schritts anbieten, keine neuen Ergebnisse produzieren können. Die Erwartungshaltung ist, dass unter den lokal optimalen die global optimale Lösung entweder auftaucht oder doch sehr gut angenähert werden kann. Wie oben erwähnt, wurde diese Vorgehensweise nicht zuerst auf das Traveling Salesman Problem angewandt. Im Jahr 1970 beschrieben die Autoren Kernighan und Lin in [KL70] ein effizientes Schätzverfahren, die sogenannte Kernighan-Lin Heuristik, zur Graphpartitionierung. Sie ist der Grundstein für die später vorgeschlagene Heuristik für das TSP:

Gegeben ist ein Graph $G = (V, E)$. Die Kanten E repräsentieren die Kosten zwischen den Knoten V. Also, $e = (i, j) \in E$ bestimmt die Kosten zwischen den Knoten i und j. Die Knoten von G sollen derart in die beiden Untermengen A und B partitioniert werden, dass die Summe über die kreuzenden Kanten zwischen A und B minimiert wird:

$$\min \sum_{\binom{i,j \in E}{i \in A, j \in B}} c(i, j)$$

Das Problem lässt sich analog für beliebig viele Partitionen definieren und hat viele praktische Anwendungen, zum Beispiel die Verteilung von elektronischen Komponenten in der Elektroindustrie auf mehrere Leiterplatten mit dem Ziel, möglichst weniger Kabelverbindungen zwischen den Platinen.

Die Autoren folgen in offensichtlicher Weise der Grundidee der lokalen Suche (eigene Übersetzung nach [KT06]). Das Verfahren läuft in n Phasen und bestimmt die Nachbarschaft einer Partition (A, B):

- Es wird eine zufällige Startlösung gewählt.
- In Phase 1 wird ein einzelner Knoten zum Austausch zwischen den Partitionen ausgewählt und zwar so, dass die daraus folgende Lösung so gut wie möglich ist. Es wird greedy gewählt. Der Tausch wird auch dann durchgeführt, wenn das Ergebnis schlechter ist als zuvor. Der getauschte Knoten wird markiert und die erreichte Lösung wird als Partition (A_1, B_1) bezeichnet.
- Zum Start der k-ten Phase, $k > 1$, exisitiert eine Lösung (A_{k-1}, B_{k-1}) und ebenfalls $k - 1$ markierte Knoten. Es wird ein einzelner unmarkierter Knoten zum Tausch ausgewählt in der Art, dass die Lösung maximiert wird. Auch hier wird in Kauf genommen, dass sich die Lösung verschlechtert. Das Resultat wird mit (A_k, B_k) bezeichnet und der vorher gewählt Knoten markiert.
- Nach n Schritten ist jeder Knoten markiert und wurde jeweils genau einmal von einer Parttion in die andere getauscht. Demnach ist die resultierende Partition (A_n, B_n) genau die Spiegelung von (A_1, B_1).
- Die $n - 1$ Partitionen $(A_1, B_1), \ldots, (A_{n-1}, B_{n-1})$ werden von der Kernighan-Lin Heuristik als die Nachbarn von (A, B) bezeichnet. Desweiteren ist eine Partition (A, B) ein lokales Optimum genau dann, wenn gilt: $w(A, B) \geq w(A_i, B_i)$, für $1 \leq i \leq n - 1$.
- Der beste Nachbar wird beim Durchschreiten der Nachbarschaft gemerkt und bildet den Ausgangspunkt für eine neue Suche.

Der einzige randomisierte Rechenschritt erfolgt bei der Wahl der ursprünglichen Partition. Alle anderen Rechenschritte sind deterministisch. Aus der Tatsache, dass nur der erste Rechenschritt nicht deterministisch ist, folgt, dass die Heuristik ihre Berechnung abbrechen kann, falls sich ein Zwischenergebnis wiederholt. Die weiteren Berechnungen würden sich, da sie deterministisch sind, ebenso wiederholen und keine weiteren fruchtbaren Neuerungen produzieren.

Das Verfahren für das Problem der Graphpartitionierung ist einfach zu implementieren, da die Datenstrukturen nicht komplex sind und während der Berechnung keine als Lösung unzulässigen, temporären Datenstrukturen entstehen. Diese Hürde wird im Zuge der Anwendung der Suchstrategie auf das Traveling Salesman Problem auftreten.

3.3. Heuristik von Lin-Kernighan für das symmetrische TSP

Nachdem einige Jahre lang die Optimierungsvariante des Traveling Salesman Problems mit nur mäßigem Erfolg erforscht wurde [BN68], präsentierten Lin und Kernighan 1971 [LK73] eine Heuristik, die ihr lokales Suchparadigma in variabler Tiefe nutzt. Aus Gründen der Gleichberechtigung werden seit jeher die Namen der Autoren in vertauschter Reihenfolge im Namen der Heuristik genannt. Unter anderem geschieht dies auch, um die Heuristik für das Traveling Salesman Problem gegenüber ihrem Verfahren zur Graphpartitionierung abzugrenzen.

3.3.1. Die Prozedur nach Lin-Kernighan, vereinfacht dargestellt

Wieder wird angenommen, dass eine Menge S und darin eine Untermenge T existieren. T ist wieder eine möglicherweise suboptimale, aber zulässige Lösung. Dann lässt sich wieder sagen, dass die Lösung T vom Optimum abweicht, da sie k viele Elemente $x_1, \dots, x_k$ enthält, die anschaulich gesprochen 'dort nicht hineingehören'. Um T nun in ein optimales T' zu überführen, müssen die $x_1, \dots, x_k$ durch geeignete Elemente $y_1, \dots, y_k \in S - T$ ersetzt werden. Das Problem wird also darauf reduziert, die geeigneten x_i und y_i, $1 \leq k \leq n$, zu bestimmen.
Da weder die Elemente $x_1, \dots, x_k$, noch $y_1, \dots, y_k$ oder auch nur ihre Anzahl k von vornherein bekannt sind, so sagen Lin und Kernighan, wäre es künstlich, k von Anfang an zu fixieren und damit nur k-Untermengen zu betrachten. Stattdessen wird zuerst versucht diejenigen x_1 und y_1 zu bestimmen, die am ehesten für einen Austausch in Betracht kommen. Sind x_1 und y_1 gewählt, werden sie temporär beiseite gelegt und die x_i und y_i, $1 < i \leq k$ aus den verbleibenden Kanten gewählt, bis ein Abbruchkriterium erreicht wird. k und die x_i, y_i werden also im Lauf der Heuristik näherungsweise bestimmt. Eine etwas formalere Sichtweise auf dieses Verfahren nach [BN68]:

1. Erzeuge eine zufällige oder vom Zufall beeinflusste erste Lösung T.
2. a) Sei $i = 1$.
   b) Wähle x_i und y_i im i-ten Schritt als das am ehesten auszutauschende Paar. Im Allgemeinen bedeutet dies, dass x_i und y_i so gewählt werden, dass sie die Verbesserung maximieren, wenn die $x_1, \dots, x_i$ gegen die $y_1, \dots, y_i$ getauscht werden.
   x_i wird aus $T \backslash \{x_1, \dots, x_{i-1}\}$ und y_i aus $S\,\{T \cup \{y_1, \dots, y_{i-1}\}\}$ gewählt.
   c) Wenn eine passende Abbruchregel zutrifft, dann gehe zu 3. Ansonsten setze $i = i + 1$ und weiter bei 2 b).
3. Wird die beste Verbesserung bei $i = k$ gefunden, dann tausche die $x_1, \dots, x_k$ mit den $y_i, \dots, y_k$ und nenne die neue Lösung T; weiter bei 2. Wird keine Verbesserung gefunden, dann gehe zu 4.
4. Wiederhole ab Schritt 1, falls gewünscht.

Lin und Kernighan schreiben dazu folgende Randbedingungen vor:

1. Eine Auswahlregel muss festlegen, welches Kantenpaar als das am ehesten auszutauschende festgelegt wird. Diese Regel muss aus den naheliegenden Gründen sowohl effizient als auch effektiv sein.

2. Eine einfach auszuwertende Funktion muss den zu erwartenden Gewinn eines Kantentauschs berechnen. Dazu sei g_i die Verbesserung, die mit dem Tausch der Kanten x_i und y_i erreicht wird, wenn $x_1, \ldots, x_{i-1}$ und $y_1, \ldots, y_{i-1}$ bereits gewählt wurden. Desweiteren sollte diese Gewinnfunktion additiv sein, also der Gewinn G des Gesamttauschs $x_1, \ldots, x_k$ mit $y_1, \ldots, y_k$ sollte gleich $g_1 + \ldots + g_k$ sein. Angenommen, diese Funktion ist additiv, dann muss die Heuristik nicht beim ersten negativen g_i stoppen, sondern erst, wenn $\sum_{i=1}^{k} \leq 0$ ist. So können dann lokale Optima auch überwunden werden.

3. Um die Heuristik für jedes k stoppen zu können, muss die temporäre Tour nach jedem Tausch im Sinne einer Lösung zulässig sein. Das heisst, dass nach jedem Tausch die Datenstruktur einer möglichen Lösung entsprechen muss.

4. Ein Abbruchkriterium muss angeben, ab wann keine erfolgversprechenden Kantenpaare mehr gefunden werden oder wenigstens, ab wann der Gesamtgewinn negativ wird.

5. Die Kantenmengen $X = \{x_1, \ldots, x_k\}$ und $Y = \{y_1 \ldots, y_k\}$ müssen disjunkt sein. Ist eine Kante in eine der beiden Mengen aufgenommen worden, kann sie im aktuellen Iterationsschritt nicht mehr aus dieser entfernt werden. Hier sei angemerkt, dass einige Varianten des Verfahrens diese Regel aufweichen werden.

Wurden bis zu einem Iterationsschritt m die Kantenpaare $x_1, \ldots, x_m$ und $y_1, \ldots, y_m$ mit den dazugehörigen Gewinnen $g_1, \ldots, g_m$ betrachtet, dann wird das endgültige $1 \leq k \leq m$ so gewählt, dass die Summe $\sum_{i=1}^{k} g_i$ maximiert wird und eine zulässige Lösung entsteht.

3.3.2. Der Algorithmus von Lin-Kernighan

Die Lin-Kernighan Heuristik enthält, wie oben erwähnt, mehrere Randbedingungen. Diese tragen in starker Weise sowohl zur Effektivität, als auch zur Effizienz des Verfahrens bei.

1. **Sequentielles Tauschkriterium** (sequential exchange criterion)**:**
   Die Kanten x_i und y_i müssen adjazent sein. Sie müssen also einen gemeinsamen Endpunkt besitzen. Die Sequenz $(x_1, y_1, x_2, y_2, \ldots, x_m, y_m)$ bildet eine Kette angrenzender Kanten. Resultiert der Tausch der Kantenmengen X und Y in einer Tour, dann wird von einem *sequentiellen Tausch* gesprochen. Anmerkung: Es gibt jedoch durchaus auch nichtsequentielle Verbesserungen, wie in Grafik 3.3 schematisch dargestellt.

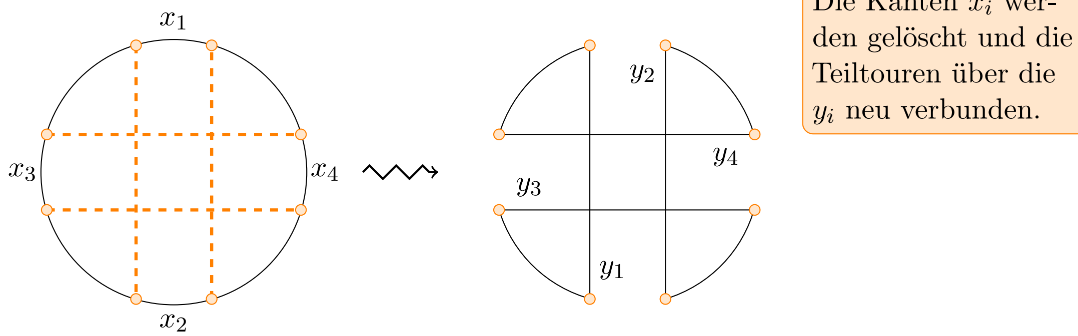

Abbildung 3.3.: Nichtsequentieller Tausch

Einige Implementierungen, unter anderem [Hel00],versuchen daher mit nichtsequentiellem Austauschen mehrer Kanten Verbesserungen zu erreichen, wenn die Heuristik in einem lokalen Optimum steckt und von daher andernfalls keinerlei Verbesserungen mehr zu erwarten sind.

2. **Zulässigkeitskriterium** (feasibility criterion)**:**
   Die Kante $x_i = (t_{2i-1}, t_{2i})$ muss so gewählt werden, dass die Konfiguration (die temporäre Zwischenlösung) einer Tour entspricht, falls die Kante $(t_{2i}, 1_1)$ eingefügt wird. Dieses Kriterium erlaubt dem Algorithmus zu jeder Iterationstiefe anzuhalten und beschränkt die Laufzeit, bzw. die Komplexität der Implementierung.

3. **Positivverbesserungskriterium** (positive gain criterion) **:**
   Dieses Kriterium erzwingt, dass y_i immer so gewählt werden muss, dass die zugehörige Gesamtverbesserung G_i positiv ist. G_i ist im i-ten Schritt: $G_i = \sum_{j=1}^{i} g_i$. In einem Suchschritt mag es mehrere Möglichkeiten geben um das Positivverbesserungskriterium zu erfüllen. Die naheliegende Regel ist, die nächste Kante „*greedy*“ zu wählen: g_{i+1} ist zu maximieren. Wenn kein positiver Zuwachs möglich ist, so ist derjenige mit dem wenigsten Verlust (immer noch das Maximum) zu wählen.

4. **Disjunktheitskriterium** (disjunctivity criterion)**:**
   Die Mengen X und Y müssen disjunkt sein. Wandert eine Kante von einer Menge in die andere, kann sie diese innerhalb eines Iterationsschritts nicht mehr verlassen. Bereits hinzugefügte (gelöschte) Kanten werden nicht wieder gelöscht (hinzugefügt). Dies vereinfacht die Implementierung, beugt Fehlern vor und bildet ein gutes Abbruchkriterium für den Iterationsschritt. Wie bereits oben erwähnt, wird es Varianten geben, die von dieser Randbedingung abweichen.

Auf den ersten Blick wirkt der folgende Algorithmus komplex, auf einen zweiten ist allerdings zu erkennen, dass er nah an der ursprünglichen Idee bleibt.

Algorithmus 8 Heuristik von Lin-Kernighan

Eingabe: Graph G

Ausgabe: Approximation der optimalen Tour

1: Erzeuge randomisiert eine Starttour T.

2: $G^* = 0$. Wähle einen Knoten t_1 und sei x_1 eine der zu t_1 adjazenten Kanten in T. $i = 1$.

3: Sei t_2 der andere Endpunkt von x_1.
Wähle die Kante y_1 von t_2 zu einem Knoten t_3 so, dass $g_1 > 0$.
Existiert ein solcher Knoten nicht, gehe zu 6d).

4: $i = i + 1$
Wähle x_i und y_i wie folgt:

a) x_i wird so gewählt, dass die Kanten dieser Konfiguration eine Tour bilden, falls die Kante (t_{2i}, t_1) eingefügt wird.

b) y_i ist eine noch wählbare Kante am Knoten t_{2i}. x_i und y_i sind adjazent. Existiert kein solches y_i, gehe zu Schritt 5.

c) Gelöschte (hinzugefügte) Kanten können nicht wieder hinzugefügt (gelöscht) werden

d) $G_i = \sum_{j=1}^{i} g_j > 0$

e) Damit die temporäre Konfiguration zulässig bleibt, muss die Wahl der Kante y_i zulassen, dass im $i+1$-ten Schritt ein x_{i+1} gewählt und mithin aus der Tour gelöscht werden kann.

f) Bevor y_i gewählt wird, ist zu überprüfen, ob ein Verbinden von t_{2i} und x_1 eine weitere Verbesserung bringt. Sei y_i^* diese Kante und sei $g_i^* = |y_i^*| - |x_i|$.
Ist $G_{i-1} + g_i^* > G^*$, dann setze $G^* = G_{i-1} + g_i^*$; $k = i$

5: Beende die Wahl der x_i und y_i, wenn entweder keine möglichen Kanten die Bedingungen 4c) - e) erfüllen oder wenn $G_i \leq G^*$.
Ist $G^* > 0$, dann starte bei Schritt 2 mit der mit G^* assoziierten Tour T' neu.

6: Gilt $G^* = 0$, so wird ein einfaches Backtracking wie folgt für die Iterationen $i = 1, 2$ durchgeführt.

a) Wiederhole die Schritte 4 und 5 und wähle die y_2's nach aufsteigender Länge, solange das Zuwachskriterium $g_1 + g_2 > 0$. Der erste gefundene Zuwachs wird gewählt.

b) Sind alle Möglichkeiten für y_2 in Schritt 4 b) ohne Erfolg untersucht worden, gehe zu Schritt 4 a) und wähle eine alternative Kante für x_2.

c) Schlagen die vorherigen Backtracking-Schritte fehl, wähle y_1 in Schritt 3 nach aufsteigender Länge.

d) Schlägt der vorherige Schritt fehl, wird die alternative Kante x_1 in Schritt 2 gewählt.

e) Schlagen alle vorherigen Backtracking-Schritte fehl, wird ein neues t_1 gewählt und zu Schritt 2 gegangen.

7: Wurden alle n Möglichkeiten für t_1 ohne Verbesserung durchgeführt, so terminiert die Heuristik. Bei Bedarf wird das Verfahren mit einer anderen Starttour T neu gestartet.

Eine wichtige Eigenschaft des Verfahrens ist die Tatsache, dass die Wahl einer Kante nicht von den Kosten der Kante (t_{2i}, t_1) abhängt. Diese werden geflissentlich ignoriert. Als integraler Bestandteil der Nachbarschaftsdefinition lassen sich nur so die qualitativ überlegenen Lösungen der Lin-Kernighan Heuristik finden.

3.4. Die Implementierung von Johnson et al.

In [JM95] berichten die Autoren Johnson et al. über Ihre Implementierung der Lin-Kernighan-Heuristik für das Traveling Salesman Problem. Diese Implementierung geht auf die ursprüngliche Beschreibung von Lin und Kernighan zurück, geht zusätzlich aber noch auf moderne Datenstrukturen ein. In weiten Teilen orientiert sich, wie noch geziegt wird, die, für diese Arbeit entstandene Implementierung an diesen Grundzügen. Grob gesprochen besteht die Implementierung von Johnson et al. aus zwei Teilen. Anschaulich sind dies eine innere Schleife und ein äußerer Mantel, der den inneren Teil steuert und bei Bedarf neu startet.

3.4.1. Die innere Schleife

Den Kern der Heuristik bildet eine innere Schleife, die von Johnson et al. auch *LK Search* genannt wird. Diese basiert auf sukzessiven 2-Opt Tauschen, also dem Tausch zweier Kantenpaare, wobei die Kosten der Kante (t_{2i}, t_1) ignoriert werden. Sowohl die vorliegende Implementierung, als auch die von Johnson et al. [JM95] und sowie die von Neto [Net99] speichern Lösungen aus Gründen der Handhabbarkeit als sogenannten *Verankerten Hamiltonpfad* anstatt eines Hamiltonkreises. Der *Anker* eines solchen Pfads P ist ein fixierter Knoten v_1, siehe dazu auch Abbildung 3.4.

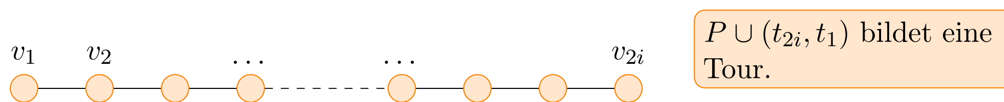

Abbildung 3.4.: Hamiltonpfad als Tour im TSP, nach [JM95]

Mit v_{2i} wird der Endknoten des Hamiltonpfads P_i bezeichnet, welcher zu Beginn der i-ten Iteration von LK Search existiert und die Tour im Graphen lässt sich jederzeit wieder auf einfache Weise rekonstruieren, indem der Pfad P um die Kante $\{v_{2i}, v_1\}$ erweitert wird. Die Durchführung des eigentlichen Kantentauschs, der in Abbildung 3.2 angedeutet wird, lässt sich mit dem Hamiltonpfad als Datenstruktur auf eine einfache Art und Weise realisieren. Dazu wird zunächst der Pfad um eine weitere Kante (v_{2i}, v_{2i+1}) zu einem 1-Baum ergänzt. Die Abbildung 3.5 verdeutlicht den Kantentausch im Hamiltonpfad.
Der entstandene 1-Baum ist aber nur ein Zwischenschritt, denn nach dem Kantentausch muss die Datenstruktur bekanntlich wieder eine zulässige Lösung sein. Dazu wird im nächsten Schritt die Kante $\{v_{2i+1}, v_{2i+2}\}$ gelöscht und mithin der Tausch abgeschlossen. Anschaulich wird im i-ten Iterationsschritt die Reihenfolge des Suffix'

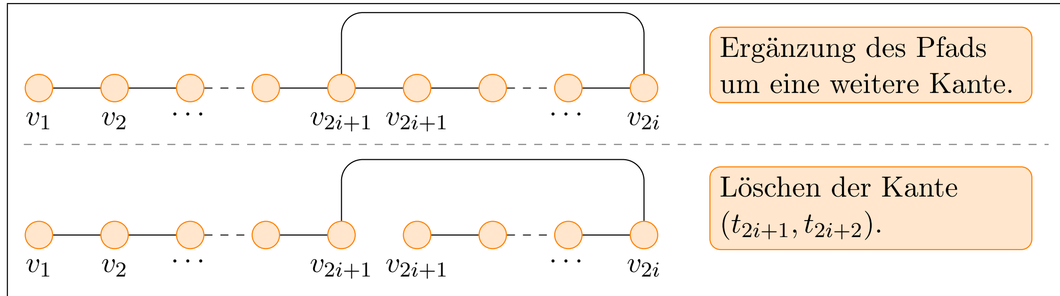

Abbildung 3.5.: Kantentausch mit Hamiltonpfad, nach [JM95]

$(v_{2i+1}, \dots, v_{2i})$ des Pfads P_i einfach umgedreht.

Satz 9 (Das sequentielle Tauschkriterium im Hamiltonpfad)
Die oben beschriebene Vorgehensweise des Kantentauschs mit dem Hamiltonpfad als Datenstruktur, bzw. Konfiguration, beachtet das sequentielle Tauschkriterium für alle Iterationsschritte $i \geq 2$.

Der Beweis lässt sich per Induktion führen. Zu Zeigen ist, dass die Kantensequenz $(x_1, y_1, x_2, y_2, \dots, x_m, y_m)$ eine Kette angrenzender Kanten bildet:

Verankerung bei i=2.
Zu Anfang der ersten Iteration entspricht der Hamiltonpfad als Tourkonfiguration der Starttour. Ist die Kante x_1 gewählt, wird die Tour solange zyklisch geshiftet, bis sie der Tour in Abbildung 3.4 entspricht. Die Kante x_1 wird damit gebrochen. Also, v_1 liegt auf der einen und v_2 auf der anderen Seite des Pfads. Nun wird $y_1 = (t_2, t_3)$ gewählt und der Tausch mittels 1-Baum ausgeführt. Die Kante $x_2 = (t_3, t_4)$ wird dadurch ebenfalls gelöscht. Der Tausch wird durch folgende Skizze deutlich:

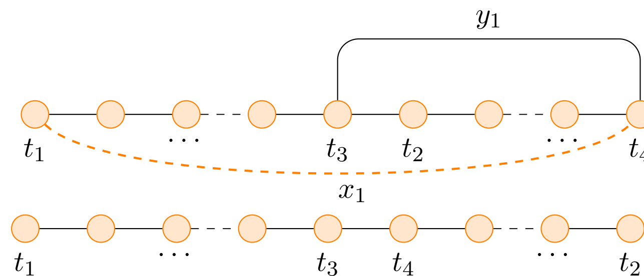

Abbildung 3.6.: Kantentausch für $i = 2$

Induktionsschritt.
Angenommen, die Aussage gilt für $i = n$, dann liegt folgender Hamiltonpfad vor:

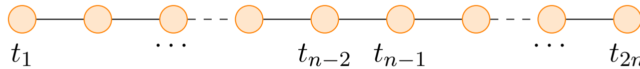

Abbildung 3.7.: Kantentausch für $i = n$

Nach Konstruktionsvorschrift muss die Kante x_{n+1} nun als (v_{2n}, v_{2n+1}) gewählt werden, wobei v_{2n} der Endknoten des Hamiltonpfads ist. Diese Kante erzeugt aus dem Pfad zwischenzeitlich einen 1-Baum. Um diesen wieder auf einen Hamiltonpfad zurückzuführen, muss eine Kante $(v_{2n+1}, v_{2(n+1)})$ aus dem 1-Baum gelöscht werden. Es ist nun offensichtlich, dass die zu Anfang bestehende Sequenz angrenzender Kanten $(x_1, y_1, x_2, y_2, \ldots, x_n, y_n)$ um den Suffix (x_{n+1}, y_{n+1}) erweitert wird. □

Die Vorteile eines Hamiltonpfads als zentrale Datenstruktur zur Repräsentation einer Rundreise liegen also auf der Hand. Neben dem sequentiellen Tauschkriterium wird auch das Praktikabiltätskriterium umgesetzt. Es gibt jedoch auch andere Datenstrukturen, wie die von Osterman und Rego [OR03] vorgeschlagenen Satellitenlisten.

In jedem Iterationsschritt der Johnson et al. Implementierung wird nun nach Kantenpaaren gesucht, die miteinander getauscht werden. Da ein Suffix des Pfads umgedreht wird, werden auch nur solche Tauschpartner berücksichtigt, die die Kante $\{v_{2i}, v_1\}$ löschen, bzw. vorneweg ignorieren. Die neue Kante $\{v_{2i}, v_{2i+1}\}$ muss dabei so gewählt werden, dass die Kosten des temporären 1-Baums geringer sind als die Kosten der besten bisher bekannten Tour. Diese Tour wird auch *Champion Tour* genannt.

Die Implementierung von Johnson et al. verzichtet auf die Tabuliste für gelöschte Kanten. Die Wiederaufnahme einer zuvor gelöschten Kante ist also ohne weiteres erlaubt. Eine der Listen alleine reicht aber offensichtlicherweise aus, um die Anzahl maximal erlaubter Tauschschritte auf N zu begrenzen und die Heuristik hat mehr Kanten zur Verbesserung zur Auswahl. Interessanterweise zeigt Papadimitriou in [Pap92] für den anderen Fall, dass nur hinzugefügte Kanten auf die Tabuliste kommen, ein erstaunliches komplexitätstheoretisches Ergebnis. Werden die hinzugfügten Kanten nicht auf eine Tabuliste gesetzt, ist es also erlaubt erzeugte Kanten wieder zu löschen, dann ist diese Variante *PLS-Vollständig*. Die spätere experimentelle Analyse wird auch diese Variante betrachten. Für den originalen Algorithmus und die Implementierung von Johnson et al. gilt dieses Ergebnis übrigens nicht. Die Beweisskizze und einige Schlussfolgerungen werden in Kapitel 3.7 präsentiert.

Die Wahl eines auszutauschenden Paars läuft wie folgt. Für jeden möglichen Tausch wird geprüft, ob die entstehende Tour inklusive Kante (t_{2i}, t_1) kürzer ist, als die Champion-Tour, die bisher beste bekannte Tour. Ist sie kürzer, dann wird sie für den späteren Gebrauch zwischengespeichert. Für den Praktiker ist hier zu beachten, dass solche temporären Zwischenspeicherungen anfällig sind für unerwünschte Seiteneffekte,

falls Simulationen parallel in verschiedenen Threads durchgeführt werden.

Eine Champion-Tour wird aber nicht notwendigerweise Ausgangspunkt in Schritt i. Stattdessen wird im i-ten Iterationschritt derjenige Tausch gewählt, der einen kürzesten neuen Pfad P_{i+1} erzeugt. Hierbei sind auch Schritte erlaubt, die zu einem schlechteren Ergebnis führen. Die Suche nach einem neuen kürzesten Pfad P_{i+1} bedeutet also nicht notwendigerweise, dass dieser kürzer wäre als P_i oder die dazugehörige Tour. Gerade diese Schritte in die falsche Richtung erlauben es der Heuristik, schlechte lokale Optima zu verlassen und nicht wie die einfacheren k-Opt-Verfahren mit vorher festgelegter Schrittweite in einem lokalen Optimum stecken zu bleiben.

3.4.2. Steuerung der inneren Schleife

Der Algorithmus speichert in jedem Schritt die Champion-Tour. Dies ist die bisher kürzeste gesehene Tour. Zu Anfang ist das die Lösung der Startheuristik. Alle Läufe der inneren Schleife gehen zu Anfang von der jeweiligen Champion-Tour aus. Wenn eine neue beste Tour gefunden wird, dann terminiert der aktuelle Lauf der inneren Schleife und der Algorithmus merkt sich diese als neue Champion-Tour. Die Steuerung terminiert den Algorithmus dann, wenn keine neue Champion-Tour von einem Ausgangspunkt aus gefunden werden kann. Damit entspricht die Steuerung der Johnson et. al Implementierung der Idee der Lin-Kernighan Heuristik.

3.4.3. Experimentelle Ergebnisse der Implementierung

Johnson et al. testen ihre Implementierung auf zufälligen euklidischen Instanzen und auf Instanzen mit zufälligen Distanzmatrizen. Sie vergleichen dabei die Ergebnisse mit einer Implementierung des 3-Opt-Verfahrens. Es wird berichtet, dass die Ergebnisse der Lin-Kernighan Implementierung etwa 1% besser sind als die 3-optimalen Touren. In ihrer Analyse schätzen sie die optimale Tour durch die Schranke von Held-Karp ab und kommen zu dem Schluß, dass die Ergebnisse von Lin-Kernighan etwa 2 Prozent über der unteren Schranke liegen und zwar unabhänig von der Instanzgröße.

Diese sehr guten Ergebnisse lassen sich aber nicht auf die Instanzen mit zufälligen Distanzmatrizen übertragen. Hier berichten Johnson et al. von viel schlechteren Ergebnissen beider Verfahren. Zwar scheint das 3-Opt-Verfahren an sich schon um den Faktor 10 schlechter zu sein, aber auch Lin-Kernighan Heuristik in ihrer Implementierung erzielt im Vergleich zu vorher nur mäßige Ergebnisse. Die getesteten Instanzen können umso schlechter approximiert werden, je größer die Instanz ist. Für kleine Instanzen mit 100 Knoten wird von fast optimalen Touren, die nur zwischen ein oder zwei Prozent über dem Optimum liegen, berichtet. Für Instanzen mit 4.500 Knoten wächst dies aber schon auf knapp 7 % an. Die Autoren vermuten daher, dass der Fehler in einer logarithmischen Größenordnung mit der Eingabegröße wächst. Sie können diese Vermutung aber nicht anderweitig belegen.

3.5. Die Implementierung von Helsgaun

Helsgaun berichtet in [Hel00] von einer modifizierten und hocheffizienten Implementierung der Lin-Kernighan-Heuristik. Gegenüber der ursprünglichen Variante von Lin und Kernighan erreichen die Erweiterungen noch bessere Touren, vor allem bei steigender Knotenzahl. Das ursprüngliche Verfahren erreicht für kleine Instanzen mit etwa 50 Knoten mit einer Wahrscheinlichkeit nahe bei 1 optimale Lösungen. Für größere Instanzen allerdings fällt die Wahrscheinlichkeit das Optimum zu erreichen auf nur noch 20 bis 30 Prozent, wie Helsgaun schreibt.

Die Leistungssteigerungen seiner Implementierung werden vor allem durch Änderungen an den Regeln, wie die lokale Suche in ihrer variablen Länge beschränkt und gelenkt wird, erreicht. Obwohl die ursprünglichen Regeln naheliegend und natürlich erscheinen, hindern sie das Verfahren, beste Ergebnisse zu erzielen. In vier Bereichen der Heuristik tragen Änderungen zur Leistungssteigerung der Implementierung bei:

- Begrenzung der in der Nachbarschaft betrachteten Knoten durch Kandidatenlisten,
- Veränderte Löschregeln für Kanten in einer zwischenzeitlichen Tour,
- Veränderung der Durchschreitens der Nachbarschaft, sowie
- Wahl der Starttour

Im folgenden Teil werden diese Punkte beleuchtet.

3.5.1. Begrenzung der Nachbarschaft durch Kandidatenlisten

In der ursprünglichen Variante [LK73] schlagen Lin und Kernighan vor, dass ein Suchschritt nur die fünf nächsten Nachbarn eines Knoten betrachtet. So liesse sich die Rechenzeit bei größeren Instanzen drastisch minimieren bei weitestgehend gleichbleibender Qualität des Ergebnisses. Der Nachteil, dass eine optimale Lösung gerade durch diese Beschränkung nicht gefunden werden kann, ist aber nicht von der Hand zu weisen: Besitzt die optimale Lösung einer Instanz eine Kante, die nicht mit den nächsten fünf Nachbarn verbunden ist, so wird das Verfahren kaum in der Lage sein, die optimale Lösung zu berechnen. Zum Beispiel besitzt die von Padberg und Rinaldi [PR87] präsentierte optimale Lösung für eine Instanz mit 532 Städten eine Kante, die von einem Endpunkt zum 22-nächsten Nachbarn verbunden ist. Die Heuristik müsste also die 22 nächsten Nachbarn betrachten um diese Kante finden zu können. Dies wiederrum würde den Platz- und Zeitbedarf der Heuristik aufblähen. Um diesen Nachteil aufzuwiegen, geht Helsgaun einen anderen Weg.

Der Grundgedanke ist aber ähnlich: Je kürzer eine Kante ist, desto wahrscheinlicher ist es, dass sie in einer optimalen Tour vorkommt. Um die *Nähe zweier Knoten* wird

ein Maß eingeführt, genannt *α-Nähe*, das in seiner Analyse auf minimale Spannbäume und 1-Bäume zurückgreift. Helsgaun berichtet aus experimenteller Erfahrung, dass eine optimale Tour bereits $70 - 80\%$ derjenigen Kanten enthält, die in einem minimalen 1-Baum zu finden sind. Daher eignet sich das Konzept der 1-Bäume um ein Maß für die Nähe zu entwickeln. Kanten, die zu einem minimalen 1-Baum gehören, bzw. Kanten, die *fast* zu einem minimalen 1-Baum gehören, haben gute Chancen Teil einer optimalen Tour zu sein.

Definition 7 (α-Nähe)
Sei T ein minimaler 1-Baum der Länge $l(T)$ und sei $T^*_{i,j}$ ein minimaler 1-Baum der die Kante (i,j) enthalten muss. Das Mass der α-Nähe ist definiert als

$$\alpha(i,j) = l(T^*_{i,j}) - l(T). \tag{3.1}$$

Es ist mit dieser Definition leicht zu sehen, dass bei gegebener Länge eines minimalen 1-Baums die α-Nähe gerade der Längenzuwachs ist, der notwendig ist, damit ein minimaler 1-Baum erzwungenermassen die Kante (i,j) enthält.

Das Mass der α-Nähe kann systematisch dazu benutzt werden, diejenigen Kanten zu identifizieren, die möglicherweise in einer optimalen Tour vorkommen. Diese vielversprechenden Kanten werden zur sogenannten *Kandidatenliste* hinzugefügt. Beachtet werden in dieser Liste dann eine fixe Anzahl k α-nächster Kanten und/oder diejenigen Kanten unter einem bestimmten Schwellenwert. Es ist zu erwarten, dass Kanten, die nach diesem Maß *weit* entfernt scheinen, von der Lin-Kernighan Heuristik getrost ignoriert werden können, ohne die Wahrscheinlichkeit, so die optimale Tour zu verpassen, stark zu erhöhen. Helsgaun berichtet, dass im Allgemeinen kleine Kandidatenlisten einer guten Lösung keinen Abstrich tun.

3.5.2. Veränderte Löschregeln für Kanten in einer Tour

Die Kandidatenliste beschränkt die Menge der Kanten Y, die zu einer Lösung hinzugefügt werden können. Analog lässt sich auch die Menge der Kanten X beschränken, die aus einer Tour zu löschen sind. Helsgaun wählt in seiner Implementierung eine einfache, aber dennoch leistungsfähige Regel, um diese Menge zu beschränken:

1. Die erste zu löschende Kante x_1 darf nicht zu der bisher besten bekannten (Champion-)Tour gehören, wobei die Starttour nicht als solche zählt. Ist noch keine Champion-Tour bekannt, so darf die Kante im Graphen der Eingabeinstanz nicht zum minimalen 1-Baum gehören.

2. Die letzte zu löschende Kante in einer Iteration, darf in aktuellen Iteration vorher nicht hinzugefügt worden sein.

Die erste Regel beschränkt den Suchraum bereits zu Anfang des Verfahrens (Schritt $i = 1$), wobei das ursprüngliche Verfahren dies erst bei $i > 2$ macht. Helsgaun berichtet weiter, dass seine Löschregeln effektiver als das Original und zudem noch leichter zu

implementieren sind.

Die zweite Regel begrenzt die Anzahl der Suchschritte. Sie ist eine Verallgemeinerung der ursprünglichen Regel, dass zuvor gelöschte (hinzugefügte) Kanten nicht wieder hinzugefügt (gelöscht) werden dürfen. Helsgaun ist nicht der einzige Autor, der eine Relaxierung des Disjunkheitskriteriums fordert. Im weiteren Verlauf werden Neto und Papadimitriou diese Regel aufweichen und auch die Analyse der für diese Arbeit erstellten Implementierung zeigt, dass sich dadurch im Allgemeinen bessere Touren finden lassen, auch wenn sich dabei die Laufzeit erhöht.

3.5.3. Veränderte Bewegung durch die Nachbarschaft

Die zentrale Eigenschaft der Lin-Kernighan Heuristik ist die Art und Weise, wie der Suchraum der lokalen Nachbarschaft durchschritten wird und welche grundlegenden Suchschritte erlaubt sind. Die Menge der erlaubten Schritte entspricht der (Unter−)Menge der möglichen r-Opt-Schritte, die betrachtet werden, um eine kürzere Tour zu erreichen.

Das ursprüngliche Verfahren betrachtet nur diejenigen r-Opt-Schritte, die in einen 2- oder 3-Opt-Schritt mit einer folgenden Sequenz von 2-Opt-Schritten zerlegbar sind, wobei diese Sequenz natürlich auch eine leer Folge sein darf. Weiterhin müssen die Handhabbarkeits- und die Zulässigkeitskriterien eingehalten werden. Zusätzlich werden im Original noch nicht-sequentielle Suchschritte erlaubt, wenn sonst keine Verbesserung gefunden werden kann.

Helsgauns Implementierung weicht diese Vorgaben auf. Hauptmerkmal ist, dass der neue grundlegende Suchschritt ein 5-Opt-Schritt ist. Dadurch wird die durchsuchte Nachbarschaft auf Kosten der Laufzeit weit ausgedehnt. Sobald allerdings während der Konstruktion eines Suchschritts festgestellt wird, dass ein Zwischenergebnis, also ein 2-, 3- oder 4-Opt-Schritt, schon eine Verbesserung bringt, wird der Suchschritt abgebrochen und dieses Zwischenergebnis als neuer Ausgangspunkt der Suche gewählt.

Die Vergrößerung der Nachbarschaft wird durch die Benutzung der Kandidatenliste allerdings teilweise kompensiert. Dadurch erhöht sich die Laufzeit nur um einen vergleichsweise kleinen Faktor. Helsgaun berichtet, dass aus Computerexperimenten heraus ersichtbar ist, dass der Algorithmus kein Backtracking (ausser für die Wahl der ersten Kante x_1) mehr benötigt. Durch das Weglassen verringert sich die Laufzeit weiterhein, ohne dass die Güte der Lösung einen Schaden nimmt. Der modifizierte Algorithmus ist zudem leichter zu implementieren.

Eine weitere Abweichung ist die Menge der nicht-sequentiellen Suchschritte, die Helsgaun betrachtet, wenn keine sequentiellen Verbesserungen mehr gefunden werden können. Anstatt der einfachen, nicht-sequentiellen 4-Opt-Schritte wird eine mächtigere Menge nicht-sequentieller Suchschritte betrachtet. Diese besteht aus den folgenden

Elementen:

- Jeder beliebige nicht zulässige 2-Opt-Tausch, der zwei disjunkte Teiltouren produziert, gefolgt von einem 2-, bzw. 3-Opt-Tausch, der wieder eine zulässige Lösung durch Zusammenfügen der Teiltouren erzeugt, gehört dazu, ebenso wie

- jeder beliebige nicht-zulässige 3-Opt-Tausch, der zwei disjunkte Teiltouren produziert, gefolgt von einem beliebigen 2-Opt-Tausch, der wieder eine zulässige Lösung durch Zusammenfügen der Teiltouren erzeugt.

Die einfachen nicht-sequentiellen 4-Opt-Tausche des ursprünglichen Verfahrens sind eine Teilmenge dieser mächtigeren Menge. Da hier eine Obermenge der ursprünglichen Möglichkeiten betrachtet wird, erhöht sich nach Helsgaun die Wahrscheinlichkeit einen besten Suchschritt zu finden. Weiterhin ist die Laufzeit der Suche in dieser Obermenge relativ gering zur Gesamtlaufzeit, da die Kandidatenlisten und die Vorgabe des Positivverbesserungskriteriums den Suchraum insofern einschränken, dass nur eine geringere Menge an Kanten in die Auswahl kommt, betrachtet zu werden.

Der Einfluss dieser nicht-sequentiellen Schritte ist aber in Helsgauns Implementierung noch nicht abschließend untersucht worden. Es lässt sich nun aus seinen Ergebnissen vermuten, dass sie einen spürbaren Einfluss auf die Güte der Touren der Lin-Kernignhan Heuristik im Allgemeinen und nicht nur speziell für Helsgauns Implementierung haben.

3.5.4. Wahl der Starttour

Im ursprünglichen Verfahren schreiben Lin und Kernighan keine spezielle Startheuristik vor. Sie gehen davon aus, dass eine zufällig bestimmte Tour als Startpunkt der Suche ausreicht. Sie kommen sogar zu dem Schluß, dass ein spezialisiertes Startverfahren Zeitverschwendung sei. Diese Überlegung kommt Teils aus der Tatsache, dass viele Konstruktionsheuristiken deterministisch arbeiten und von daher auch nur eine mögliche Lösung von der Lin-Kernighan Heuristik berechnet werden kann, denn die eigentlich Rechenschritte laufen ohne Zufallsbits ab.

Dazu gibt es aber gegenläufige Ergebnisse von Perttunen [Per94]. So wird herausgefunden, dass die sogenannte *Clark and Wright Savings Heuristik* als Startheuristik im Allgemeinen bessere Gesamtergebnisse erzeugt. Auch Reinelt [Rei94] kommt zu dem Schluss, dass zufällige Starttouren zu schlechteren Gesamtergebnissen führen, als speziell gewählte Starttouren dies tun. Er schlägt vor, lokal optimale Touren aus Ausgangspunkt zu benutzen, die auf die eine oder andere Weise immer noch große Fehler aufweisen, beispielsweise das Konstruktionsverfahren von Christofides, s. Kapitel 2.3.3. Das heisst aber nicht, dass es keine Starttouren gäbe, die genau das Gegenteil erreichen.

Helsgaun wendet mit einigem Erfolg folgende Startheuristik an:

Algorithmus 9 Startheuristik von Helsgaun

Eingabe: Graph G

Ausgabe: Starttour für Lin-Kernighan Heuristik

1: Wähle einen zufälligen Startknoten i.

2: Wähle einen weiteren, noch unmarkierten Knoten j wie folgt:

1. Die Kante (i, j) ist in der Kandidatenliste,
2. $\alpha(i, j) = 0$, und
3. Die Kante (i, j) gehört zum minimalen Spannbaum.

- Andernfalls wähle, wenn möglich, Kante (i, j) so, dass sie zur Kandidatenliste gehört.
- Andernfalls wähle j aus der Menge der unmarkierten Knoten.

3: Markiere i. Setze $i = j$.

4: Sind noch unmarkierte Knoten vorhanden, dann weiter bei 2.

Stehen in Schritt 2 mehr als ein Knoten zur Auswahl, dann wird unter den Möglichkeiten ein Knoten zufällig gewählt. Die Reihenfolge der Wahl der Knoten impliziert die Starttour. Helsgaun berichtet, dass dieses Verfahren schnell ist und die Menge der initialen Lösungen weit genug streut, um gute endgültige Lösungen zu erhalten. Eine theoretische Analyse dieser Startheuristik steht aber noch aus.

3.5.5. Experimentelle Ergebnisse der Implementierung

Die Implementierung liefert allgemein gesprochen sehr gute Ergebnisse. Sie ist so aufgebaut, dass jede Instanz einer gewissen Anzahl an unabhängigen *Läufen* unterworfen wird und innerhalb eines Laufs mehrere *Versuche* unternommen werden, mittels der modifizierten Lin-Kernighan Heuristik eine neue beste Tour zu berechnen. Die Versuche innerhalb der Läufe sind nicht unabhängig, da die Kanten in der Champion-Tour des aktuellen Laufs benutzt werden um die Suche innerhalb eines Versuchs zu beschränken. Die Anzahl der Läufe in den Simulationen variiert mit der Größe der zu bearbeitenden Instanz. Graphen mit weniger als 1000 Knoten werden 100 Läufen unterzogen, wobei größere Instanzen mit 10 Läufen berechnet werden. Die Anzahl der Versuche innerhalb der jeweiligen Läufe entspricht der Dimension, also der Anzahl der Städte einer Instanz. Ist das Optimum einer Instanz bekannt, so wird die Suche abgebrochen, sobald das Verfahren eine optimale Lösung präsentiert.

Die präsentierten Ergebnisse seiner Simulationen auf den Instanzen der TSPLIB [Rei91] zeigen die Güte der Helsgaun-Implementierung sehr deutlich. Die durchschnittliche Abweichung für alle Instanzen beträgt nur 0.001% und die größte gefundene Abweichung zum bekannten Optimum beträgt 0.072%. Die dazu benutzte CPU-Zeit ist ebenfalls erstaunlich gering: Auf einem Apple Power Macintosh G3 mit 300 MHz Taktfrequenz

braucht Helsgauns Implementierung je nach Instanz nur zwischen 0.2 und 3.6 CPU-Sekunden. Für 11 Probleme, die Helsgaun testet, waren zu diesem Zeitpunkt noch keine Optima, sondern nur beste Touren bekannt. Auch hier überzeugt die Implementierung. Für 4 dieser Probleme werden die bekannten Lösungen bestätigt und für die übrigen 7 werden sogar noch bessere Touren gefunden. Für die größte Instanz *pla85900* mit knapp 86.000 Knoten ist der Rechenaufwand aber schon nicht mehr vernachlässigbar. Es wird von zwei Wochen CPU-Zeit berichtet.

Die Größe einer Instanz alleine ist aber noch kein Indikator für die Laufzeit des Verfahrens. Wieder gibt es Instanzen, deren Berechnung länger dauert als größere Probleme. Auf dieses Phänomen wird im Zuge der Clusterkompensierung von Neto im folgenden Kapitel genauer eingegangen. Für zufällige Instanzen kommt Helsgaun zu dem Schluß, dass die Laufzeit in der Größenordnung von $O(n^{2.2})$ anzusiedeln ist.

3.6. Die Implementierung von Neto

Im Allgemeinen besitzt die Lin-Kernighan Heuristik eine sehr gute Laufzeit. Allerdings kann die Laufzeit bei speziellen Instanzen auch um ein Vielfaches schlechter ausfallen, als auf Instanzen gleicher Größe. Dies tritt auf, wie auch schon Johnson [JM95] berichtet, wenn die Knoten der Instanz nicht gleichmäßig, sondern in Clustern verteilt sind. Neto [Net99] führt in seiner Arbeit das Konzept der *effizienten Clusterkompensierung* (engl. efficient cluster compensation) ein. Ihr Ziel ist es, den Laufzeitnachteil bei Clustern in einer Instanz aufzuwiegen, ohne die Güte der Heuristik nachhaltig zu beeinträchtigen. Anschaulich beschneidet die effiziente Clusterkompensierung den Suchraum so, dass Regionen mit geringem Nutzen schnell wieder verlassen werden. Dazu wird ein *Lookahead* eingeführt, der die Suche stärker lenkt als von Lin und Kernighan eigentlich vorgesehen. Der Lookahead berücksichtigt die sogenannte *Cluster-Distanz* als Richtmaß für die Zugehörigkeit zweier Knoten zu verschiedenen Clustern.

3.6.1. Einordnung eines Clusters

Clusterung ist eine Eigenschaft der Distanz-, bzw. Kostenfunktion $C : V \times V \mapsto \mathbb{R}^+$. Anschaulich lässt sich sagen, dass eine Instanz Cluster besitzt, wenn die Knoten der Instanz so eingruppiert werden können, dass die Abstände zwischen den Knoten eines Clusters relativ klein sind im Vergleich zu den Abständen zwischen zwei Knoten in verschiedenen Clustern. Neto weist daraufhin, dass die genaue Definition für ein Cluster-Maß schwer zu geben ist. Stattdessen wird folgende „weiche“ Definition gegeben.

Definition 8 (γ-Maß für Cluster)
Sei $T = (V, E_T)$ ein minimaler Spannbaum für den Graphen $G = (V, E)$. Durch Löschen längster Kanten des Spannbaums werden die Knoten in G in mehrere Cluster partitioniert. Sei γ das Verhältnis von Länge einer längsten Kante zur Länge der Kante, deren Länge der Median aller Kantenlängen ist. Ist γ groß, dann ist die minimale

Distanz zwischen den Knoten eines Clusters viel kleiner als die minimale Distanz zwischen Knoten in verschiedenen Clustern. Ist γ klein, dann sind die Abstände zwischen den Clustern relativ gering.

Demnach ist das γ-Maß nur ein recht grober Indikator für Clusterung einer Instanz ist. Die gegebene Funktion und die naheliegenden Modifikationen können offensichtlich von einem schlauen Gegenspieler geschlagen werden. Aus diesem Grund verzichtet Neto auf eine quantitative Definition und vertraut weitestgehend auf die Intuition des Lesers.

3.6.2. Längere Laufzeit bei geclustertern Instanzen

Um eine Tour T in eine kürzere Tour T' zu transformieren, werden k Kanten wie gehabt aus der Tour gegen k Kanten, die hinzugefügt werden, getauscht. Wie bereits zuvor erwähnt, werden diese Austausche sequentiell vorgenommen und bilden einen Kreis gerader Länge im Graphen. Die Kanten dieses Kreises wechseln zwischen gelöschten und hinzugefügten Kanten. Ein solcher k-Tausch kann dann als eine Reihe von $2 \cdot k$ Kanten des Graphen aufgeschrieben werden: $t_1, t_2, \ldots, t_{2k-1}, t_{2k}$. Mit ungeraden Indizes sind die gelöschten und mit geraden Indizes sind die hinzugefügten Kanten beschriftet. Dies ist analog zu den Kanten x_i und y_i, also $t_1, t_2, \ldots, t_{2k-1}, t_{2k} = x_1, y_1, \ldots, x_k, y_k$.

Die Kostenfunktion $C : V \times V \mapsto \mathbb{R}^+$ ist nicht-negativ. Daher muss das Positivverbesserungskriterium nur überprüft werden, wenn eine Kante hinzugefügt wird. Die Berechnung der bisher erreichten Verbesserung, von Neto mit *cumulative gain* bezeichnet, ist:

$$cum_gain = \sum_{\binom{1 \leq l \leq j}{l \text{ ungerade}}} c(t_l, t_{l+1}) - \sum_{\binom{1 \leq l \leq j}{l \text{ gerade}}} c(t_l, t_{l+1}) \tag{3.2}$$

Dieses Mass ist klar äquivalent zur vorherigen Definiton und die bisher erreichte Verbesserung ignoriert die Kosten, den entstehenden (Tausch-)Kreis mit der Kante (t_{2i}, t_1) zu schließen, [LK73], da nach Lin und Kernighan die Schätzung sonst zu pessimistisch ist. In der Tat ist es so, dass die Ergebnisse der Heuristik stark nachlassen, wenn die Kante (t_{2i}, t_1) in die Berechnungen für das Positivverbesserungskriterium direkt mit einbezogen wird. Im Zuge der für diese Arbeit erstellten Implementierung zeigen durchgeführte Experimente, dass die Güte dann nur noch minimal über einem einfachen 2-Opt Verfahren liegt. Die Entscheidung diese *Kreisschlusskosten* zu ignorieren, ist also, wie schon zuvor erwähnt ein, integraler Bestandteil der Heuristik.

Diese vordergründig „gute" Ignoranz macht das Verfahren im Gegenzug aber anfällig für lange Laufzeiten. Eine Tatsache, die in Kapitel 3.7 für eine Komplexitätstheoretische Betrachtung der Heuristik noch einmal herangezogen wird.

Wieso sind Cluster nun schlecht? Sogenannte *Köderkanten* führen die Heuristik bei den Berechnungen zur bisher erreichten Verbesserung (cumulative gain) in die Irre

[Net99]. Genauer: Kanten, die relativ weit entfernte Cluster verbinden, ködern die Heuristik. Wird eine solche *Brückenkante* gelöscht, so wächst der Wert der bis dato erreichten Verbesserung überoptimistisch und die Heuristik besitzt nun ein hohes Budget, dass sie benutzen kann, um innerhalb eines Clusters herumzuwandern ohne dabei weder einen besonderen Anreiz zu finden zum ursprünglichen Cluster mit den Startknoten t_1 zurückzukehren, noch eine bedeutende Verbesserung zu finden. Die Kosten, eine solche Brücke hinzuzufügen, sind gross und es ist aufgrund der greedy-Wahl der nächsten Knoten t_{2i+1}, t_{2i+2} recht unwahrscheinlich, dass so der Weg auch noch zum Ursprungscluster zurückgefunden wird.

Natürlich gibt es die Möglichkeit per Zufall wieder einen Pfad zurück zu finden, indem eine Kante mit ungeradem Index aus der zwischenzeitlichen Tour gelöscht wird. Mit der Zeit werden aber immer bessere Zwischenlösungen gefunden und die Chancen auf solch eine Entdeckung sinken. Anschaulich formuliert, enthält eine optimale Tour in der Regel nicht allzuviele Brückenkanten zwischen verschiedenen Clustern. Da aber eine Lösung eine Tour über alle Knoten und damit ein zusammenhängender Subgraph ist, müssen Brückenkanten dennoch in jeder Instanz mit Clustern vorkommen.

Zusammenfassend lässt sich sagen: Ursache für das schlechte Laufzeitverhalten bei Instanzen mit Clustern ist die Tatsache, dass sowohl das Positivverbesserungskriterium, als auch die greedy Wahl der nächsten Kante, die Kosten für das Schließen eines Kreises per Kante (t_{2i}, t_1) ignorierten. Sie tun dies sowohl im aktuellen als auch den folgenden Iterationsschritten. Ein Kompromiss zwischen Lookahead und Ignorieren der Kante liegt nahe.

3.6.3. Effiziente Clusterkompensierung

Die Ignoranz der Kreisschlußkosten erlaubte dem ursprünglichen Verfahren eine große Freiheit auf der Suche nach guten Lösungen. Allerdings kann die Heuristik, wie gesehen, eine lange und fruchtlose Suche beginnen, wenn sie eine Kante zwischen zwei Clustern aus der zwischenzeitlichen Lösung löscht. Ein naheliegender Ausweg aus diesem Dilemma ist die Idee, dass die Heuristik eine vorrausschauende Komponente erhält. Diese Komponente soll es ihr erlauben, sich dem Lockruf von Brückenkanten entziehen zu können.

Neto sieht den Schlüssel dazu darin, die bisher erreichte Verbesserung (cumulative gain) als ein Maß zu benutzen, das angibt, wie segensreich eine gefundene Verbesserung nun wirklich ist. Das Löschen von Brückenkanten zwischen Clustern soll die Heuristik keine überpositive Verbesserung mehr vermuten lassen. Die Lin-Kernighan Heuristik trifft bei der Wahl der Knoten t_{2i+1} und t_{2i+2} eine sehr lokal begründete Entscheidung. Erster Knoten ist ein Element der Kandidatenliste von t_{2i} und die Wahl von t_{2i+2} hängt vor allem vom Zulässigkeitskriterium ab.

Für die Heuristik spielt es in der Bewertung eigentlich keine Rolle, ob eine betrachtete

Kante gelöscht oder hinzugefügt werden soll oder, ob sie eine Kante innerhalb eines Clusters oder zwischen verschiedenen Clustern betrachtet. Wenn eine Kante zwischen zwei Clustern liegt, dann sollte die Heuristik vermeiden, diese aus einer zwischenzeitlichen Lösung zu löschen. Ist im Gegenzug die betrachtete Kante aber eine möglichst lange Kante innerhalb eines Clusters, so macht es für die Heuristik durchaus Sinn, diese zu löschen. Das Positivverbesserungskriterium und die greedy-Wahl einer Kante müssten also globale Informationen darüber besitzen, wie die umliegende Landschaft in Bezug auf Cluster aussieht und dementsprechend ihre Entscheidungen anpassen. Die effiziente Clusterkompensierung passt daher die greedy Wahl einer Kante und die Berechnung der bisher erreichten Verbesserung an. Um nun Informationen darüber zu erhalten, ob eine Kante zwei Cluster verbindet und eine Brückenkante ist, wird die sogenannte Cluster-Distanz eingeführt.

Cluster-Distanz

Die Cluster-Distanz hilft der Heuristik, globale Informationen über die Cluster-Struktur einer Instanz zu sammeln. Sie entspricht der Länge einer längsten Kante, die auf einem Pfad traversiert werden *muss*, um von einem Knoten zu einem anderen zu gelangen.

Definition 9 (Cluster-Distanz)
Gegeben seien ein Pfad P in einem Graphen $G = (V, E)$ und zwei Knoten $u, v \in V$. Die längste Kante von P wird als Flaschenhalskante bezeichnet. Die Cluster-Distanz zwischen Knoten u und v, geschrieben als $C_G(u, v)$, ist die kleinste Flaschenhalskante aller Pfade von u nach v in G.

Die Cluster-Distanz zwischen den Knoten t_1 und t_{2i} ist eine geschätzte untere Schranke für die zukünftigen Kreisschlusskoten, die im Lauf der Optimierung entstehen können.

Anders formuliert: Angenommen, die Knoten t_1 bis t_{2i} sind von der Lin-Kernighan Heuristik bereits fixiert und der Kreis mit den alternierenden, gelöschten und hinzugefügten Kanten soll vervollständigt werden. Dann wird die t-Sequenz fortgeschrieben mit den Knoten $t_{2i+1}, \ldots, t_{2k}$. Der noch unbekannte Teil der t-Sequenz hat ungerade Länge und beginnt mit einer Kante $e_{2i} = (t_{2i}, t_{2i+1})$, die hinzugefügt wird, einer gelöschten Kante $e_{2i+1} = (t_{2i+1}, t_{2i+2})$ und so weiter bis zur hinzugefügten Kante $e_{2k} = (t_{2k}, t_1)$. Die Cluster-Distanz ist die Schätzung einer unteren Schranke für die Länge einer längsten Kante auf diesem Pfad. Die wirklichen Kreisschlusskosten, nämlich

$$c(e_{2i}) - c(e_{2i+1}) + \ldots + e_{2k-2} - e_{2k-1} + e_{2k},$$

müssen aber noch gefunden werden.

Wieso ist die Cluster-Distanz nun eine untere Schranke? Angenommen, der größte Summand der obigen Summe wäre $c(e_{2L})$ und alle Subtrahenden würden diesen aufheben, dann wären die Kreisschlusskosten dieses Pfades gerade $c(e_{2L})$. Daher ist die

Cluster-Distanz $C_G(t_{2i}, t_1)$ eine unter Schranke für die nach Wahl der Kante e_{2i-1} auftretenden Kreisschlusskosten. Da sich die Terme in der Regel nicht gegeneinander aufheben, ist die Cluster-Distanz keine exakte untere Schranke, sondern nur eine grobe untere Schranke für die später zu verbuchenden Kreisschlusskosten. Es sei angemerkt, dass dieser Betrachtung die Vorstellung zu Grunde liegt, nur eine längste Kante auf einem Pfad von t_{2i} nach Knoten t_1 sei von Interesse. Die Längen der Kanten vor oder nach dieser längsten Kanten gehören zum sogenannten Grundrauschen des Lookaheads [Net99] und sind somit zu vernachlässigen.

3.6.4. Einfluss der Cluster-Distanz auf die Heuristik

Im Hinblick auf die geschätzte untere Schranke für die entstandenen Kreisschlusskosten, ist es sinnvoll, diese auf die Art und Weise, wie die Heuristik eine Kante wählt, Einfluß zu lassen. Neto passt das Positivverbesserungskriterium an, indem es in der Bewertung der bisher erreichten Verbesserung im Mittel zwischen direkten Kreisschlusskosten und völliger Ignoranz der Cluster-Distanz liegt. Die erste Möglichkeit der direkten Kreisschlusskosten widerspricht klar der Idee von Lin und Kernighan und würde die Heuristik unbrauchbar machen. Gerade die Tatsache, dass die Kosten der Kante (t_{2i}, t_1) ignoriert werden, sichert ja erst die Güte der Heuristik. Die zweite Möglichkeit der Ignoranz wäre auch fruchtlos. Sie bedeutet keine Änderung gegenüber dem bekannten Positivverbesserungskriterium und ist, wie bereits erwähnt, anfällig für Köderkanten.

Demnach ist die greedy-Wahl der nächsten Kante anzupassen. Die Anpassung hilft ebenso, die Kante (t_{2i+1}, t_{2i+2}) zu wählen. Nun soll nach der greedy-Eigenschaft die Kante, die den besten Erfolg verspricht, ausgewählt werden. Auch hier hilft die Cluster-Distanz, keine Köderkante zu wählen.

Justierung des Positivverbesserungskriteriums

Die Bewertung der bisher erreichten Verbesserung wird also angepasst. Neto schlägt vor, das Positivverbesserungskriterium wie folgt anzupassen:

$$cum_gain(2i) - C_G(t_{2i}, t_1) > 0$$

3.6.5. Justierung der Kantenwahl

Die t-Folge mit den zu tauschenden Kanten wird mit demjenigen Paar (t_{2i+1}, t_{2i+2}) fortgesetzt, dass folgende Summe maximiert:

$$cum_gain(2i+2) - C_G(t_{2i+2}, t_1)$$

Es gilt dabei $cum_gain(2i+2) = cum_gain(2i) - c(t_{2i}, t_{2i+1}) + c(t_{2i+1}, t_{2i+2})$. Um nun $cum_gain(2i+2) - C_G(t_{2i+2}, t_1)$ bei fixiertem $t_1, \ldots, t_{2i}$ zu maximieren, muss die Summe

$$-c(t_{2i}, t_{2i+1}) + c(t_{2i+1}, t_{2i+2}) - C_G(t_{2i+1}, t_1)$$

maximiert werden.

3.6.6. Experimentelle Ergebnisse der Implementierung

Neto berichtet von sehr guten Ergebnissen der Heuristik sowohl hinsichtlich der Laufzeit als auch der Güte der berechneten Lösungen. Die Experimente laufen auf zwölf Instanzen der TSPLIB Sammlung, die über eine starke Clusterung verfügen. Eine Implementierung der Lin-Kerninghan Heuristik ohne effiziente Clusterkompensierung wird mit einer Variante mit Kompensierung in den Punkten Laufzeit und Güte der Ergebnisse verglichen. Wird effiziente Clusterkompensierung benutzt, so sind die Ergebnisse leicht schlechter in der Güte als ohne die Kompensierung. Hier handelt es sich um etwa ein halbes Prozent Abweichung. Die Laufzeiten sinken im gleichen Zug sehr stark und machen den wirklichen Vorteil der Implementierung klar. Clusterkompensierung senkt die Laufzeiten um einen Faktor zwischen $1,3$ und Faktor 10. Als Ausnahme, die die Regel bestätigt, gibt Neto die TSPLIB Instanz `fnl4461` an, die etwa 1 Prozent mehr Zeit zur Optimierung benötigt.

Ein ähnliches Verhalten zeigt sich auf zufälligen geometrischen Instanzen. Die Tourqualität ist leicht schlechter und zwar um etwa 0.02%. Die Laufzeiten sinken aber um etwa 25 bis 100%. Die Clusterkompensierung hat also auf Laufzeit und Güte der Heuristik Einfluss. Vom Standpunkt des Praktikers bietet sie den Vorteil, dass in der selben Zeit mehr Läufe der Heuristik auf einer Instanz stattfinden und so eventuell insgesamt bessere Ergebnisse gefunden werden können. Offen ist allerdings noch die Frage nach effizienten Variationen der Clusterkompensierung, die noch bessere Laufzeiten bei gleichbleibender oder besserer Tourqualität versprechen.

Für das Worst-Case-Verhalten gibt es ein Resultat, das zeigen wird, dass Netos Variante der Lin-Kernighan-Heuristik PLS-vollständig ist. Dies wird zeigen, dass auch Netos Variante nicht das Worst-Case-Verhalten der Lin-Kernighan Heuristik überwinden kann. Auf der anderen Seite wird das Ergebnis bestätigen, dass durch effiziente Cluster-Kompensierung keine fundamentale Verschlechterung in das Verfahren hineingetragen wird.

3.7. PLS-Vollständigkeit

Wie schwierig ist es, ein lokales Optimum zu finden? Die Antwort auf diese Frage lässt sich durch eine neue Komplexitätsklasse zur Einordnung lokaler Suchverfahren formalisieren. Johnson et al. [JPY88] präsentieren die Komplexitätsklasse PLS, die in natürlicher Art und Weise alle, wie die Autoren sie nennen, sinnvollen lokalen Suchstrategien erfasst. Der Name PLS steht für *polynomial-time local search*.

Definition 10 (Komplexitätsklasse PLS)
Ein Problem L in PLS ist entweder ein Maximierungs- oder ein Minimierungproblem und erfüllt mehrere Eigenschaften:

- *Für L gibt es eine Menge X_L der Instanzen, die kodiert als $\{0,1\}^n$ in polynomieller Zeit erkannt werden können.*
- *Für jede Eingabe $x \in X_L$ gibt es eine Menge zulässiger Lösungen $F_L(x)$, die o.B.d.A. auch als $\{0,1\}^n$ kodiert werden und alle in der Größe durch $p(|x|)$ nach oben beschränkt sind.*
- *Bei gegebenem x und für jedes s ist es in polynomieller Zeit möglich zu entscheiden, ob $s \in F_L(x)$. Die Zulässigkeit einer möglichen Lösung ist also effizient nachprüfbar.*
- *Für jede Lösung $s \in F_L(x)$ gibt es eine Kostenfunktion $c(s,x)$, die Lösung s in polynomieller Zeit eine Maßzahl zuordnet.*
- *Desweiteren gibt es für jedes $s \in F_L(x)$ eine Menge $N_L(s,x) \subseteq F_L(x)$, die sogenannte Nachbarschaft von s.*

Dazu gibt es drei Algorithmen A_L, B_L und C_L mit polynomiellen Laufzeiten, die folgende Aufgaben übernehmen:

1. *A_L berechnet bei gegebenem $x \in X_L$ eine Standardlösung $A_L(x) \in F_L(x)$.*
2. *B_L berechnet bei gegebenem x und s, ob $s \in F_L(x)$ und wenn ja, die zugehörigen Kosten $c(s,x)$.*
3. *C_L berechnet bei gegebenem x und s, ob es in der Nachbarschaft von s ein s' mit besseren Kosten im Sinne der Maximierung, bzw. Minimierung, gibt. Falls ja, dann wird diese Lösung generiert und falls nicht, das negative Ergebnis gemeldet.*

Damit liegt der klassische lokale Suchalgorithmus auf der Hand:

Algorithmus 10 Standardalgorithmus zur lokalen Suche

Eingabe: Instanz x des Problems L.
Ausgabe: Lokal optimale Lösung s für x

```
1: Benutze A_L um eine Startlösung s = A(x) zu erzeugen.
2: while lokales Optimum noch nicht erreicht do
3:    Wende C_L auf das Paar (x, s) an.
4:    if C_L findet einen besseren Nachbarn s' von s then
5:       s = s'
6:    end if
7: end while
```

Da die Lösungsmenge endlich ist, muss dieser Algorithmus terminieren, d.h. es gibt mindestens ein lokales Optimum für das Problem L in PLS.

Nach der obigen Definition ist PLS also eine Menge von lokalen Suchproblemen und

jedes Element dieser Menge ist genau genommen ein Paar $\{P, L\}$, wobei P ein abstraktes Problem ist, für das es den lokalen Suchalgorithmus L gibt. Die verkürzte Schreibweise identifiziert den lokalen Suchalgorithmus mit dem Namen des Problems. L definiert also die Nachbarschaft und $\{P, L\}$ findet lokale Optima für Instanzen von P mit Hilfe von L. Wie verhält sich diese Definition im vorliegenden Fall nun im Hinblick auf das Traveling Salesman Problem? Die Eingabe x muss dem Graphen, in welchem eine minimale Rundreise gefunden werden soll, entsprechen. Beispielsweise ist dies eine Distanzmatrix. Die Menge der zulässsigen Lösungen F_x sind alle Touren in diesem Graphen. Die Kosten, bzw. die Maßzahl, die einer Lösung zugeordnet werden, entsprechend der Länge einer Tour. Die Möglichkeit in polynomieller Zeit eine lokales Optimum aus einer gegebenen zulässigen Lösung zu berechnen, ist naheliegenderweise die jeweils benutzte Heuristik.

Um die PLS-Vollständigkeit zu zeigen wird auf das schon aus den Beweisen der $\mathcal{NP}$-Vollständigkeit einzelner Probleme bekannte ähnlich benutzte Mittel der Reduktion zurückgegriffen. Anschaulich wird die Instanz eines Problems durch den Algorithmus für ein anderes Problem gelöst. Das erste Problem ist offensichtlich dann algorithmisch genauso handhabbar wie das zweite. Dazu darf natürlich die „Umrechnung" von einem Problem in das andere nicht zu stark ins Gewicht fallen.

Eine PLS-Reduktion nimmt die folgende Gestalt an:

Definition 11 (PLS-Reduktion)
Seien Π_1 und Π_2 zwei lokale Suchprobleme. Eine PLS-Reduktion von Π_1 auf Π_2 besteht aus zwei in polynomieller Zeit berechenbaren Funktionen h und g mit den Eigenschaften, dass

1. *h die Instanz $x \in \Pi_1$ in eine Instanz $h(x) \in \Pi_2$ abbildet.*
2. *Sei s_2 die von Π_2 berechnete Lösung zu $h(x)$, dann bildet $g(s_2, x)$ diese auf eine Lösung s_1 für Π_1 ab.*
3. *Für alle Instanzen $x \in \Pi_1$ gilt: Ist s eine lokal optimale Lösung für die Instanz $h(x)$ Π_2, dann ist $g(s, x)$ lokal optimal für Π_1.*

PLS-Reduktionen sind konkatenierbar und lösen das Problem Π in polynomieller Zeit mittels L. Das Problem Π ist also, auf die lokale Suche bezogen, algorithmisch so schwierig wie das Problem L.

Definition 12 (PLS-Vollständigkeit)
Ein Problem L ist PLS-vollständig, wenn sich jedes $\Pi \in PLS$ auf L PLS-reduzieren lässt.

Es ist leicht zu sehen, dass diese Reduktion den in Definition 10 gerforderten Eigenschaften entspricht. Dazu lassen sie sich komponieren und erlauben die Schwierigkeit des einen Problems durch ein anderes auszudrücken. Zwei Fakten liegen nahe:

Fakt 1 *Sind* Π_1, Π_2 *und* Π_3 *Probleme aus PLS, derart, dass* Π_1 *PLS-reduzierbar auf* Π_2 *und* Π_2 *PLS-reduzierbar auf* Π_3*, dann ist auch* Π_1 *PLS-reduzierbar auf* Π_3*.*

Fakt 2 *Seien* Π_1 *und* Π_2 *Probleme in PLS, derart dass* Π_1 *PLS-reduzierbar auf* Π_2 *ist. Gibt es einen lokalen Suchalgorithmus, der lokale Optima findet, mit polynomieller Laufzeit für* Π_2*, dann gibt es auch lokalen Suchalgorithmus mit polynomieller Laufzeit für* Π_1*, der lokale Optima findet.*

Es lässt sich nun eine Variante der Lin-Kernighan Heuristik beschreiben, für welche Papadimitriou [SY91] zeigt, dass sie PLS-vollständig ist.

3.7.1. Variante der Lin-Kernighan Heuristik nach Papadimitriou

Papadimitriou definiert seine Varianter der Heuristik wie folgt:

Sei T eine Tour. In der Nachbarschaft zu T sind alle Touren T', die sich durch einen 2- oder 3-Opt-Schritt erreichen lassen. Diese Nachbarn werden dann genommen, wenn sie existieren. Zusätzlich gibt es für jedes Paar adjazenter Kanten (x_1, y_1) mit $l(y_1) < l(x_1)$ und $x_1 = (a, b) \in T$, sowie $y_1 = (b, c) \notin T$, eine Sequenz benachbarter Touren $T_1, T_2, \ldots$ wie folgt. Sei $H = T - x_1$ ein Hamiltonpfad mit dem fixierten Endpunkt a und dem freien Endpunkt b. Jede Tour, die durch einen Kantentausch im Hamiltonpfad, wie in Kapitel 3.4.1 beschrieben und unter Hinzufügen der Kante (t_{2i}, t_1) entsteht gehört zur Nachbarschaft. Zu jedem Hamiltonpfad H_i gehört also genau eine Tour T_i in der Nachbarschaft und umgekehrt. Die Kanten x_i, $1 \leq i \leq k$ werden eingefroren und dürfen in einer Tauschsequenz nicht wieder in die Touren der Nachbarschaft aufgenommen werden. In den Schritten $i > 1$ wird der Tausch im Hamiltonpfad derart durchgeführt, dass die Kosten von H_{i+1} minimiert werden. Die Sequenz der Touren in der Nachbarschaft bricht nach weniger als n^2 Schritten ab, da verlassene Kanten nicht wieder aufgenommen werden können.

Der hauptsächliche Unterschied zum originalen Verfahren von Lin und Kernighan ist die Tatsache, dass hinzugefügte Kanten die Tour auch wieder verlassen dürfen. Dies ist ursprünglich nicht vorgesehen. Lin und Kernighan sehen dagegen sogar vor, dass die Menge hinzugefügter und gelöschter Kanten immer disjunkt ist. Für das ursprüngliche Verfahren ist es aber noch offen, ob es PLS-vollständig ist. Etwas formaler lässt sich der Algorithmus wie folgt darstellen:

Algorithmus 11 Heuristik LK' nach Papadimitriou

Eingabe: Starttour T

Ausgabe: Approximation der optimalen Tour

1: Alle 2- und 3-Verbesserungen werden ausprobiert. Wird eine Verbesserung T' mit $c(T') \leq c(T)$ gefunden, dann wird diese ausgeführt.

2: Schritt 3 wird mit allen Paaren $x_1 \in T$ und $y_1 \notin T$ wiederholt, derart dass

a. x_1 und y_1 adjazent sind,

b. $c(y_1) < c(x_1)$

3: $i = i + 1$. Betrachte alle Kanten $y_i \notin T$ adjazent zu x_{i-1} und wähle diejenige, die das Positivverbesserungskriterium maximiert. Gibt es keine solche Kante, gehe zu Schritt 4.

4: Sei $T_j = T - \{x_1, \ldots, x_j\} \bigcup \{y_1, \ldots, y_j\}$, mit $j = 2, \ldots, i$. T_j ist die Tour, die entstehen würde, wenn die Heuristik bei $i = j$ abgebrochen würde. T_j ist so zu wählen, dass es minimal ist. Ist $c(T_j) < c(T)$, dann ist T_j der gesuchte Nachbar. Andernfalls wähle x_1 und y_1 neu und weiter bei 2.

5: Wenn alle Möglichkeiten für x_1 und y_1 ohne Erfolg probiert wurden, dann ist die aktuelle Tour ein lokales Optimum.

Aus dieser Definition der Nachbarschaft ist leicht zu erkennen, dass sie in PLS liegt. Um die PLS-Vollständigkeit zu zeigen, wird das Problem auf ein anderes bereits als PLS-vollständig bekanntes reduziert.

Satz 10 (PLS-Vollständigkeit von Lin-Kernighan)
Die von Papadimitriou spezifizierte Variante der Lin-Kernighan Heuristik ist PLS-vollständig.

Papadimitrious Beweis erfolgt über die Angabe der Konstruktion einer TSP-Instanz aus einer 2SATFLIP-Instanz heraus. 2SATFLIP ist als PLS-vollständig bekannt. Gegeben sind hier Klauseln $C_1, C_2, \ldots, C_m$. Jede Klausel besteht aus zwei Literalen (Variablen und Negationen) aus der Menge $X = \{x_1, x_2, \ldots, x_n\}$. Jeder Klausel C_i ist ein Gewicht $W_i > 0$ zugeordnet. Eine Lösung ist eine Belegung der n Variablen in dieser Formel. Die Kosten einer Lösung entsprechen der Summe der Gewichte derjenigen Klauseln, die von der Belegung erfüllt werden. Eine Belegung ist lokal optimal, wenn die Kosten nicht mehr durch einzelne Flips einer der n Variablen verbessert werden können. Die Nachbarschaft besteht also aus allen Belegungsvektoren s', die den Hamming-Abstand 1 zu s haben.

Im Beweis wird eine 2SATFLIP-Instanz I, genauer die Klauselmenge $C \in I$ in einen völlständigen Graphen $f(C)$ transformiert. Die lokal optimalen Touren entsprechen dabei lokal optimalen Belegungen für I. Nun entspricht jede Belegung b einer bestimmten Tour T_b. Solche Touren werden als sogenannte *Standardtouren* bezeichnet. Die Kanten des vollständigen Graphen können anhand ihres Gewichts in Klassen partitioniert

werden.

Der Graph $f(C)$ hat ein Skelett aus leichten Kanten, die in einen vollständigen Graphen mit ansonsten sehr hochgewichtigen Kanten eingebettet sind. Abbildung 3.8 zeigt dieses Subgraphenskelett schematisch. Jede Variable $x_i \in C$ wird durch ein Paar spezieller Pfade in $f(C)$ und eine Startkante vom Knoten S aus repräsentiert, in der Abbildung blau dargestellt. Neto [Net99] nennt diese Pfade anschaulich „Rippen". Jede Variable hat zwei Rippen. Rippe x_i entspricht der Belegung wahr und $\overline{x_i}$ der Belegung falsch. Die Standardtour für $f(C)$ traversiert nur eine Rippe pro Variable.

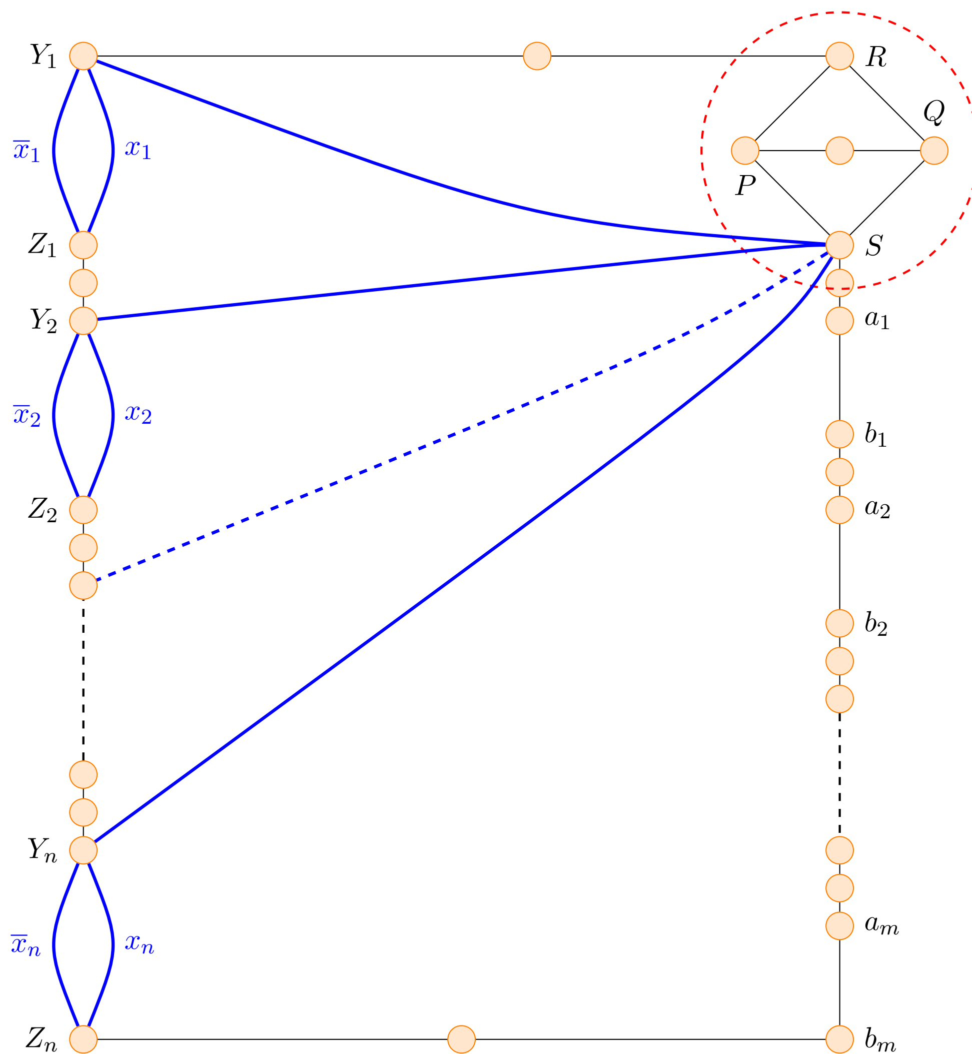

Abbildung 3.8.: Allgemeiner Aufbau der PLS-Reduktion nach [Pap92]

Jeder Klausel $C_i = (L_{j1}, L_{j2})$ sind eine Kante (a_j, b_j) und eine sogenannte ODER-

Komponente zugeordnet, die beidseitig mit den Rippen für L_{j1} und L_{j2} verbunden sind. Für jede aus zwei Literalen bestehende Klausel wird ein ODER-Subgraph eingefügt, der die Pfade, die zu den Klauseln gehören, verbindet. Ein solcher Subgraph ist nun mit dem restlichen Graphen über seine 4 Endpunkte und die passenden Rippenkanten verbunden. Die ODER-Komponenten sind dazu gedacht anzuzeigen, ob die jeweilige Klausel, zu denen sie verbunden sind, insgesamt und wie die Literale einzeln von der als Tour kodierten gewählten Belegung, erfüllt werden. Es ist leicht einzusehen, dass eine Tour einen solchen Subgraphen auf 3 verschiedene Weisen durchlaufen kann, um die Belegung anzuzeigen.

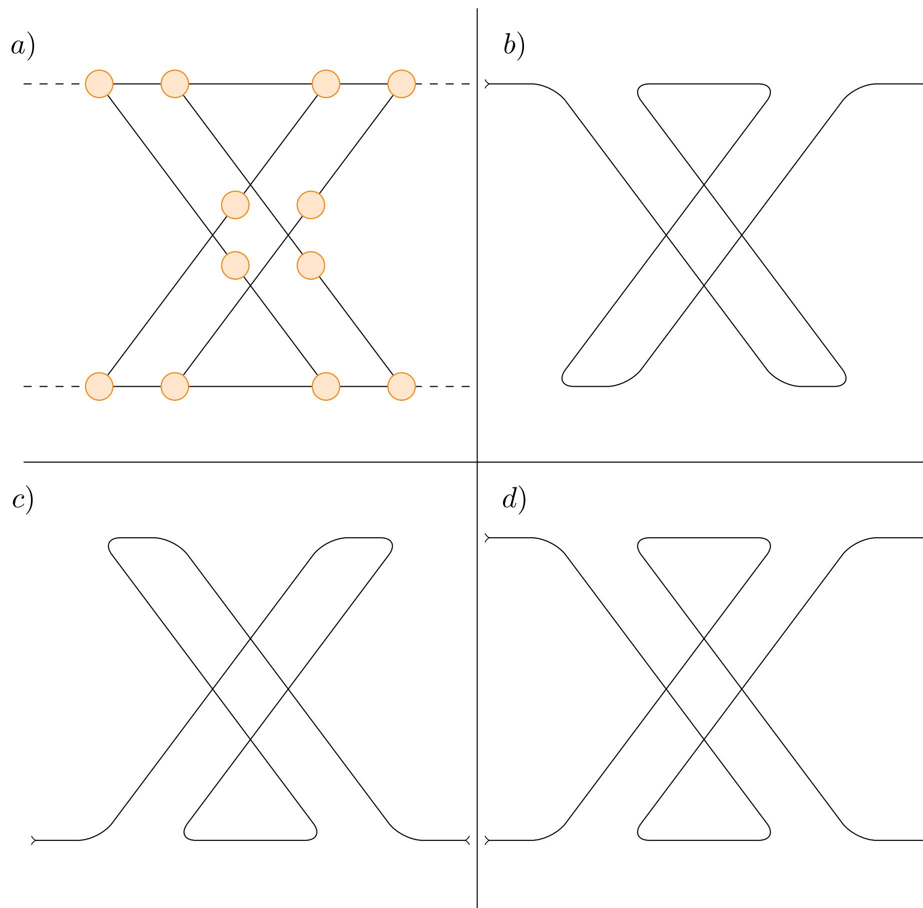

Abbildung 3.9.: Die Möglichkeiten, den ODER-Subgraphen zu traversieren, nach [Pap92]

In Abbbildung 3.9a) ist der allgemeine Aufbau des ODER-Subgraphen skizziert. In Abbildung 3.9b) ist das obere Literal mit WAHR, in 3.9c) das untere Literal und in Abbildung 3.9d) beide Literale mit WAHR belegt. Der Pfad, der den Literalen entspricht muss also eine ganze Sequenz von Komponenten sein.

Jede ODER-Komponente verbindet die Teilpfade, die für die Belegungen der Literale

einer Klausel stehen. Wenn die oben gezeigten Pfade die einzigen Traversierungsmöglichkeiten wären, dann würde der Klausel immer die Belegung WAHR zugewiesen, denn ein oder beide Literale der Klausel wären ja so belegt. Es gibt daher eine weitere Möglichkeit eine ODER-Komponente zu traversieren. Betrachtet wird die Klausel C_i. Diese kann von einem Pfad von a_i nach b_i durchlaufen werden. Zur Erinnerung: (a_i, b_i) ist auf der rechten Seite auf Abbildung 3.8 zu sehen und repräsentiert Klausel C_i. Die zugehörige Komponente kann sogar auf zwei Weisen angetroffen werden, aber die Kanten, die Knoten a_i und b_i sind aber mit höheren Kosten als die anderen versehen. Sie werden daher auch als *Strafkanten* bezeichnet und ihr Gewicht ist proportional zur Gewichtung W_i der Klausel C_i. Daher muss, falls die Komponente auf diese Art und Weise durchlaufen wird, eine teure Kante in die Tour aufgenommen werden. So wird ein größerer Preis proportional zu W_i bezahlt und dies entspricht ja auch gerade der Idee, dass eine Klausel nicht erfüllt wird.

In Abbildung 3.8 ist der sogenannte „Diamant" rot umrandet dargestellt. Die Kantengewichte werden so gewählt, dass er in einer Standardtour auf zwei Weisen traversiert werden kann. Entweder die Kanten (R, P) und (Q, S) oder die Kanten (R, Q) und (P, S). Angenommen, die Tour traversiert die Kanten (R, Q) und (P, S), dann könnte ein möglicher k-Tausch mit dem Löschen der Kante (P, S) (Köderkante) und dem Hinzufügen einer Startkante, beispielsweise zum Literal x_j beginnen. Dies kann anschaulich als *Flippen* einer Literalbelegung für x_j interpretiert werden. Die durch den Tausch entstandene Tour wird nun durch die Traversierung der anderen Kanten des Diamanten wieder geschlossen. Die Traversierung des Diamanten ändert sich also von einem Flip zum nächsten. Er macht es möglich einen Flip als Kantentausch zu modellieren. Es ist leicht zu sehen, dass eine solcher Tausch sich natürlich nur dann lohnt, wenn die dadurch implizierte Belegung der Klauseln besser ist als die vorherige.

Alle nicht erwähnten Kanten haben immense, also so hohe Kosten, dass die Heuristik geradezu abgeschreckt wird sie in die Tour aufzunehmen. Die immensen Kosten mancher Kanten erzwingen, dass die Heuristik diese nur entfernt und im weiteren Verlauf nicht wieder einfügen wird. Nach der Konstruktion wird es immer bessere Kandidaten geben, als diese Kanten.

Damit ist die Konstruktion schon fast abgeschlossen. Als letzter Schritt werden die Kantengewichte des Skeletts noch so gewählt, dass das gewünschte Verhalten eintritt. Der Graph kann nun wie folgt traversiert werden. Der Diamant wird auf eine der beiden Möglichkeiten durchlaufen. Für jede Variable wird entweder ein WAHR- oder ein FALSCH-Pfad durchlaufen. Für die ODER-Komponente gibt es die gezeigten vier Möglichkeiten. Die Tour wird über die rechte Seite des Graphen abgeschlossen.
Papadimitrou [Pap92] zeigt weiter, dass die so konstruierte *Standardtour* genau dann ein lokales Optimum darstellt, wenn die dadurch implizierte Belegung der Klauseln ein lokales Optimum wiederspiegelt. Desweiteren wird gezeigt, dass nur die Standardtouren lokal optimal sind. Es bleibt also folgender Fakt festzuhalten.

Fakt 3 *Die Standardtouren sind lokal optimal und auch die einzigen lokal optimalen Touren.*

3.7.2. PLS-Vollständigkeit von Netos Implementierung

Es ist leicht nachzuvollziehen, dass auch die Variante von Neto in PLS liegt. Die Definition der Nachbarschaft erfüllt klar die Forderungen, die die Zugehörigkeit zu PLS an sie stellt. Die Reduktion erfolgt so, dass Netos Variante auf die von Papadimitriou reduziert wird. Der Aufbau des Graphenskeletts ist dabei gleich. Änderungen finden nur an den Kantengewichten statt, um die Anpassungen, die Neto an der Gewichtsfunktion vornimmt wiederzuspiegeln. Weiter wird gezeigt, dass auch mit der angepassten Gewichtsfunktion die Standardtouren lokal optimal sind und ausser ihnen keine. Neto zeigt dadurch und unter Mithilfe von Fakt 1 folgendes:

Fakt 4 *2SATFLIP lässt sich auf die Variante der Lin-Kernighan Heuristik mit effizienter Clusterkompensierung von Neto reduzieren. Daher ist auch diese Variante PLS-hart.*

Neto fasst diese Ergebnisse treffend zusammen: Als Konsequenz bedeutet der Beweis der PLS-Vollständigkeit der Variante mit effizienter Cluster-Kompensierung, dass auch sie das gleiche Worst-Case-Verhalten der Lin-Kernighan Heuristik zeigt. Insbesondere gibt es eine Familie von Graphen, auf denen diese Variante ineffiziente Laufzeit besitzen wird. Auf der anderen Seite lässt sich aus dieser Tatsache aber auch schließen, dass die effiziente Clusterkompensierung die Leistungsfähigkeit der Lin-Kernighan Heuristik nicht fundamental verschlechtert. Zumindest aus dem Augenwinkel der PLS-Vollständigkeit betrachtet.

4. Experimentelle Analyse der Lin-Kernighan-Heuristik

Für diese Arbeit wurden verschiedene Varianten der Lin-Kernighan-Heuristik implementiert, um diese experimentell zu analysieren. Die Programme wurden in der Programmiersprache JAVA entwickelt. Diese entspricht dem aktuellen Stand der Softwaretechnik, ist nicht nur weit verbreitet, sondern auch plattformunabhängig. Für ein eingehendes Studium dieser Programmiersprache eignen sich [Fla98] und [WF01]. Die benutzte Version dieser Sprache ist 1.5. Die erwartete absolute Laufzeit der Programme ist langsamer, als zum Beispiel eine Umsetzung in C/C++. Dies rührt aus der Tatsache her, dass JAVA-Anwendungen in einer virtuellen Maschine ausgeführt und dazu vorher in eine Art Zwischencode, den sogenannten Bytecode, übersetzt werden. Dieser ist natürlich langsamer als nativer Maschinencode für eine spezielle Hardwareplattform, aber der Geschwindigkeitsnachteil wird andererseits durch mehrere andere Vorteile wieder aufgewogen. Zum einen sind JAVA-Anwendungen auf einer Vielzahl von Betriebssystemen lauffähig. Für fast jedes moderne Betriebssystem sind mittlerweile virtuelle Maschinen verfügbar. Zum anderen wird von einer speziellen Hardwarestruktur abstrahiert. Dadurch sollte der Code leichter zu warten und zu erweitern sein, wenn der technische Fortschritt bestimmte Hardware hinter sich lässt. Ein weiterer Punkt, der den direkten Geschwindigkeitsverlust aufwiegt, ist der Gebrauch von Just-In-Time Compilertechniken in den aktuellen virtuellen Maschinen für Java. Während des Laufs wird der Bytecode parallel zur Ausführung in Maschinencode der wirklich zugrunde liegenden Hardware übersetzt. Dies kostet zwar zusätzliche Rechenzeit, bei rechenintensiven Prozessen rückt dies jedoch in den Hintergrund und die Ausführungsgeschwindigkeit erreicht fast die einer direkt in Maschinensprache übersetzten Anwendung. Die Entwicklung der Programme und ihre experimentelle Analyse fanden auf einem Apple MacBook mit Dualcore Prozessor und 2GHz Taktfrequenz statt. Die virtuellen Maschine bekam 2 Gigabyte Hauptspeicher zur Verfügung gestellt.

Die implementierten Varianten unterscheiden sich von der ursprünglich formulierten Variante dadurch, dass die Backtrackingschritte abgeschaltet, bzw. dass die Art und Weise, welche Kanten die Heuristik betrachtet modifiziert wurden. Die Simulationen fanden auf 40 symmetrischen Probleminstanzen der TSPLIB der Größe kleiner 1.400 statt. Für alle benutzen Instanzen sind Optima bekannt und die benutzte Starttour ist die kanonische Tour um Starttouren zu erkennen, die eine Optimierung in ein relativ schlechtes lokales Optimum zwingt. Die in den Simulationen gesammelten Daten sind die gemessene Laufzeit und die Länge der approximierten Tour.

Für jede Probleminstanz wird der zugehörige Approximationsfaktor aus der Simulation berechnet. Über alle Daten einer Simulation werden zusätzlich mehrere Auswertungen berechnet. Für jeden Simulationslauf werden Minimum, Maximum, Median und arithmetisches Mittel der berechneten Approximationsfaktoren und der Laufzeiten berechnet. Laufzeiten und Güte der Lösungen sind in dieser Zusammenfassung unabhängig voneinander zu sehen, da die beste gefundene Approximation nicht notwendigerweise in der minimalen gemessenen Optimierungsdauer gefunden wurde.

Die gesammelten Daten werden weiter ausgewertet, um einen Überblick über das vermutete allgemeine Verhalten der jeweiligen Variante zu erhalten. Diese Auswertungen sind ein Histogramm, also wie häufig ein Approximationsfaktor in einer Simulationsreihe auftaucht, und ein Plot, der die Eingabegröße gegen die Güte aufgetragen zeigt, um einen Eindruck zu erlangen, ob es einen Zusammenhang zwischen Instanzgröße und Güte der Lösung gibt und wie stark diese jeweils ausgeprägt ist. Zusätzlich zeigt ein Plot der beobachteten Laufzeiten, welches Laufzeitverhalten eine bestimmte Variante hat. Kleinberg und Tardos [KT06] weisen korrekterweise daraufhin, dass es oft schwer ist eine grundsätzliche Aussage zu treffen, ob ein lokaler Suchalgorithmus nun gut oder schlecht ist, da theoretische Ergebnisse für lokale Suchverfahren oft zu unscharf seien. Wo keine theoretischen Ergebnisse direkt herleitbar sind, helfen Simulationen, um eine Vorstellung über ein 'allgemeines' Verhalten einer Heuristik zu erlangen.

In Anhang A finden sich die gesammelten Rohdaten jeweils als Tabelle für jede Variante und die auf ihr gelaufenen Simulationen.

4.1. TSPLIB

Ein wichtiges Element in der experimentellen Bewertung einer Heuristik ist eine anerkannte Instanzensammlung. Die TSPLIB [Rei91] ist eine Sammlung von Beispielinstanzen für das Traveling Salesman und artverwandte Probleme. Sie ist frei verfügbar und wird gerne zu Performance-Messung von Heuristiken benutzt. Die Instanzen haben unterschiedliche Größen (von 14 bis etwa 86.000 Städte) und die größte Klasse innerhalb der Sammlung sind Instanzen für das metrische TSP. Für alle Instanzen sind die besten, bzw. optimalen Touren bekannt, so dass die Leistung einer Heuristik gemessen werden kann.

Die weitaus größte Menge der Instanzen sind nicht als Distanzmatrizen gegeben, sondern jeder Knoten ist durch eine Menge von Koordinaten charakterisiert. Die Distanz zwischen zwei Knoten i und j wird dann zur Laufzeit oder während einer Vorberechnung bestimmt. Es gibt eine ganze Reihe an Metriken. Zu den am häufigsten genutzten zweidimensionalen Metriken gehören

- die Manhattan Metrik oder auch L_1 Metrik: $d(i,j) = |i_x - j_x| + |i_y - j_y|$,

- die euklidische Metrik oder auch L_2 Metrik: $d(i,j) = \sqrt{(i_x - j_x)^2 + (i_y - j_y)^2}$ oder auch

- die Maxnorm Metrik oder auch L_∞ Metrik: $d(i,j) = \max(|i_x - j_x|, |i_y, j_y|)$.

Die vorliegende Implementierung implementiert diese Metriken und versucht, sofern genügend Hauptspeicher zur Verfügung steht, eine Distanzmatrix im Speicher aufzubauen, um im Lauf schnellen Zugriff auf die Kantengewichte zu haben. Steht nicht genügend Speicher zur Verfügung, wird aus den Koordinatenangaben der metrischen Instanzen das jeweilige Kantengewicht *on-the-fly* berechnet. Dies verlangsamt die Anwendung aber um mindestens den Faktor 100, was sich bei großen Instanzen doppelt schwer auswirkt.

4.2. Aufbau der entstandenen Implementierung

Kernstück der für die Arbeit erstellten Implementierung ist die ursprüngliche Heuristik, wie Lin und Kernighan sie in [LK73] vorschlagen. Dazu gehört ein voll implementiertes Backtracking, das auf jeden Fall die 2-Optimalität der Lösungen garantiert. Die Fixierung der Kanten, also das Disjunktheitskriterium, ist in verschiedenen Varianten implementiert und erlaubt die gängigen Variationen. So müssen bei Bedarf die gelöschten (hinzugefügten) Kante nicht fixiert werden. Die unterschiedlichen Leistungsgüten dieser Variationen des Verfahrens werden in den nun folgenden Kapiteln experimentell analysiert.

Die eingesetzte Datenstruktur für die Speicherung der Zwischentouren benutzt den Hamiltonpfad so, wie in Kapitel 3.4.1 vorgestellt. Die Implementierung ist softwaretechnisch modular und objekt-orientiert aufgebaut. Jede der Varianten der Heuristik ist in eine eigene Klasse gekapselt. So lassen sich die Komponenten leicht in anderen Anwendungen weiterverwenden. Anhang B erläutert zudem die Erweiterung der erstellten Anwendung.

4.3. Lin-Kernighan ohne Backtracking

Die erste Simulationsreihe testet die Lin-Kernighan-Heuristik mit deaktiviertem Backtracking. Die Ergebnisse sind in Tabelle 4.1 zusammengefasst. Die beste gefundene Approximation für eine optimale Tour weicht nur um den Faktor 1.002 ab. Allerdings sind Median, bzw. arithmetisches Mittel größer als 1.5. Die Christofides-Heuristik würde also im Mittel bessere Ergebnisse liefern.

	Approximation	Zeit
Minimum	1,002	2 ms
Median	1,540	1.723 ms
Arithm. Mittel	1,673	82 s
Maximum	3,800	1.841 s

Tabelle 4.1.: Zusammenfassung: Lin-Kernighan ohne Backtracking

Diese Ergebnis spiegelt sich im Gütehistogramm, Abbildung 4.3 links, für die abgelaufenen Simulationen wieder. Der Plot rechts zeigt die gemessenen Approximationsfaktor gegen die Größe der Instanz. In rot sind die geglätteten Approximationsfaktoren als Funktion der Größe eingezeichnet. Betrachtet man diese Glättung als Funktion der Güte, dann legt der Plot nahe, dass mit wachsender Instanzgröße die Approximationsfaktoren nicht nur 1.5 überschreiten, sondern sogar noch schlechter werden.

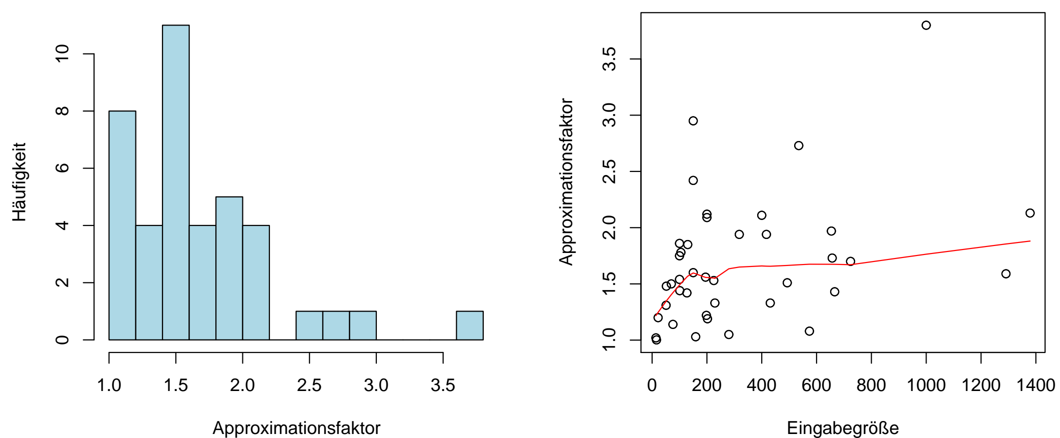

Abbildung 4.1.: Güte der Lösungen in der Simulation ohne Backtracking

Die Laufzeit der Variante des Verfahrens ohne Backtracking ist etwa in der Region quadratischen Wachstums. Abbildung 4.2 zeigt dazu einen Plot der beobachteten Laufzeiten mit dem Vergleichskurven der Funktionen $f(n) = n^2$ (orange) und $g(n) = n^{2.2}$ (rot).

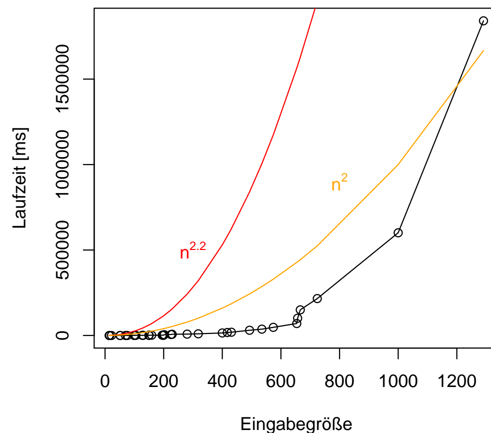

Abbildung 4.2.: Beobachtete Laufzeiten mit Vergleichsgrößen in der Simulation ohne Backtracking

In schwarz sind in der Abbildung die Verbindungsgeraden zwischen den einzelnen Datenpunkten eingezeichnet.

Wie Johnson et al. [JM95] bereits feststellen, vermuten manche Autoren, dass ein Weglassen der Backtracking-Schritte keine oder nur eine sehr geringe Auswirkung auf die Lösungsgüte hat. Die gemessenen Ergebnisse liegen in dieser Simulation aber teilweise weit unter dem, was einfachere Tourkonstruktionsalgorithmen, wie z.B. der von Christofides vorgeschlagene Algorithmus, bieten. Einziger Vorteil ist die verringerte Laufzeit im Vergleich zu den folgenden Varianten, die in der beobachteten Lösungsqualität aber durchaus einfacher zu haben wäre. Daher lohnt sich diese Variante nicht wirklich für einen praktischen Einsatz, sie zeigt aber den Einfluss, den fehlendes Backtracking auf die Lösungsgüte hat. Die erzeugten Lösungen sind zum Beispiel nicht einmal 2-Optimal. Abbildung 4.3 zeigt den Ausschnitt einer gefundenen Lösung für die Probleminstanz *berlin52*.

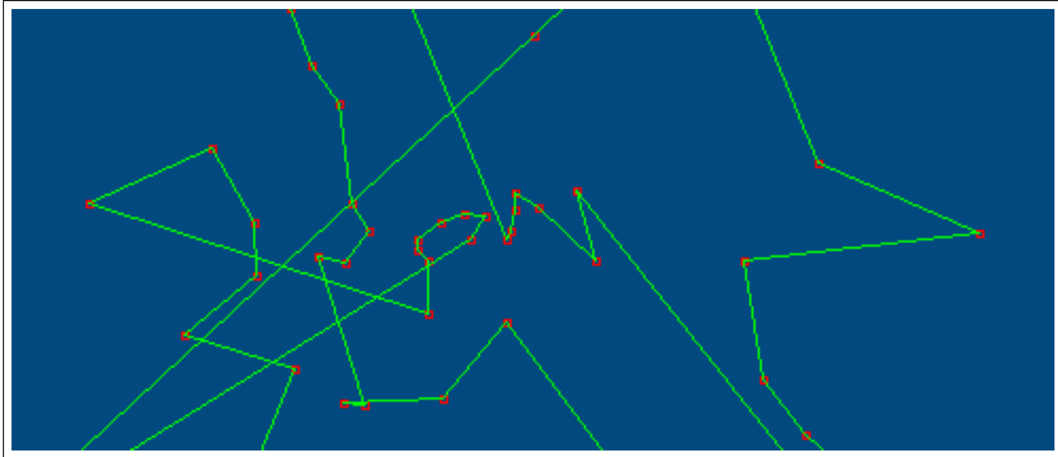

Abbildung 4.3.: Ausschnitt aus dem Plot zu einer ohne Backtracking gefundenen Lösung für die Probleminstanz *berlin52*.

Der gezeigte Ausschnitt ist offensichtlich nicht einmal 2-optimal, da es Kanten gibt, die sich kreuzen. Ausnahmen in diesem Verhalten sind die Instanzen *burma14* und *ulysses16* aus der TSPLIB, die auch mit deaktiviertem Backtracking mit den Faktoren von $1,02$ und $1,002$ gut optimiert werden. Dies scheint an der geringen Größe der Instanzen zu liegen. Eine leistungsfähige Implementierung kommt aber nicht umhin, die vorgesehenen Backtracking-Schritte zu benutzen.

4.4. Lin-Kernighan mit Backtracking

Die ursprüngliche Variante der Heuristik friert innerhalb eines Suchschritts eingefügte und gelöschte Kanten ein und führt ein einfaches Backtracking auf den Niveaus 1 und 2 durch, um mindestens die 2-Optimalität garantieren zu können. Einmal fixierte Kanten in einem Schritt können nicht mehr wieder gelöscht, bzw. eingefügt werden. Der Suchraum wird einerseits beschnitten, um die Laufzeit gering zu halten, behält aber andererseits seine Mächtigkeit bei, ohne exponentiellen Blowup zu erfahren.

Die Zusammenfassung der Ergebnisse in Tabelle 4.2 zeigt die Leistungsfähigkeit der traditionellen Variante. Für eine Instanz wird das Optimum gefunden und die mittlere Güte liegt mit einem Approximationfaktor von etwa $1,06$ knapp über dem Optimum. Der schlechteste gefundene Approximationsfaktor liegt bei $1,15$ über der zugehörigen optimalen Tour. Bei dieser Instanz handelt es sich interessanterweise um das relativ kleine Problem *kroC100*.

Wie sich im späteren Verlauf noch zeigen wird, kommt hier der Einfluss der kanonischen Tour als Starttour ins Spiel. Kapitel 4.7 wird daher genauer auf den Einfluss der Starttour auf die Güte der Approximationen eingehen.

	Approximation	Zeit
Minimum	1,000	2 ms
Median	1,060	5.000 ms
Arithm. Mittel	1,063	1.340 s
Maximum	1,150	21.484 s

Tabelle 4.2.: Zusammenfassung: Lin-Kernighan mit Backtracking

Die Güte der Lösungen ist mit aktiviertem Backtracking deutliche besser als noch zuvor ohne. Bis auf drei gefundene Lösungen sind die Abschätzungen maximal um den Faktor $1,10$ vom Optimum entfernt. Der Plot in Abbildung 4.4 rechts gibt darüber Auskunft. Die großen Ausreisser in der Güte sind auch nicht, wie man eigentlich vermuten könnte, bei den großen Instanzen zu finden, sondern eher im Mittelfeld.

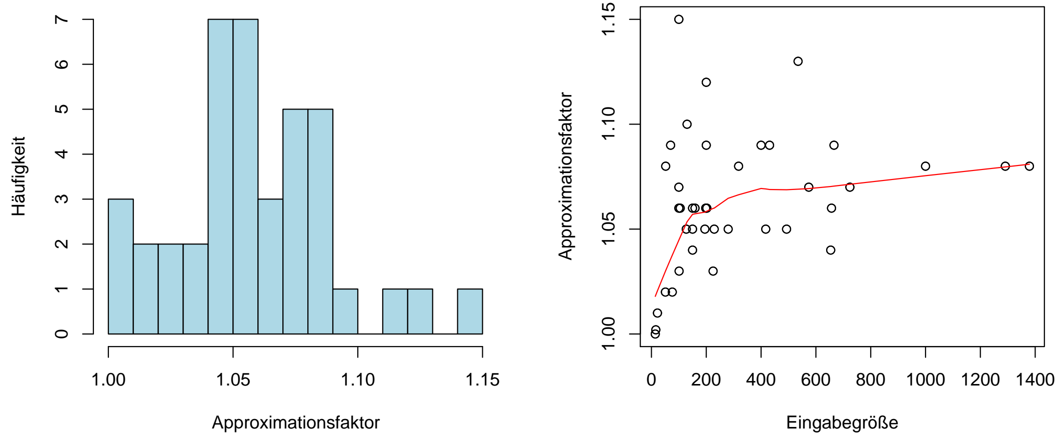

Abbildung 4.4.: Güte der Lösungen in der Simulation mit Backtracking

Die Laufzeit der Variante mit Backtracking ist, wie erwartet, größer als mit abgeschalteten Backtracking. Allerdings ist der Anstieg recht moderat. Abbildung 4.5 zeigt den Plot der beobachteten Laufzeit, die in der Region von $O(n^{2.3})$ zu liegen scheint. Die Verbreiterung des Suchraums bedingt also einen Anstieg der Laufzeit, der aber durchaus ertragbar ist.

Um einen Vergleich zu haben, sind in den Plot der Laufzeit zwei Vergleichsgrößen mit $f(n) = n^{2.3}$ und $g(n) = n^{2.4}$ eingezeichnet.

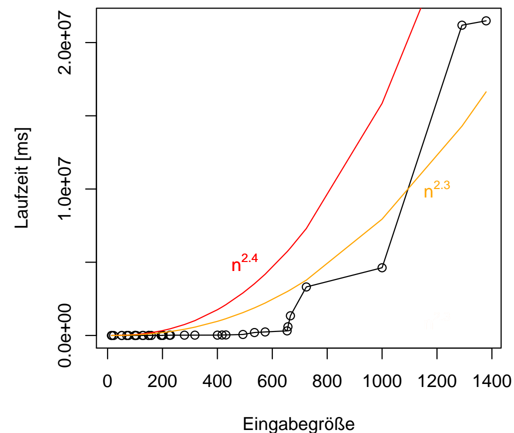

Abbildung 4.5.: Wachstum der Laufzeit mit Vergleichsgrößen in der Simulation mit Backtracking

Die Ergebnisse dieser Variante zeigen, welchen grossen Anteil die Backtracking-Schritte an den allgemein guten Lösungen der Heuristik haben. Nachdem das Aussschalten der Backtracking-Schritte keine ernstzunehmende Variante des Verfahrens darstellt, lässt sich jedoch über die Erlaubnis, benutzte Kanten teilweise wiederzuverwenden die mittlere Lösungqualität noch ein wenig steigern. Die in den folgenden Abschnitten untersuchten Varianten zeigen dies.

4.5. Lin-Kernighan ohne Fixieren entfernter Kanten

Eine erste Variation des Disjunktheitskriteriums der Heuristik von Lin-Kernighan ist die Erlaubnis, entfernte Kanten in einer Iteration wieder verwenden zu dürfen. Dadurch wird der Suchraum vergrößert und die Hoffnung ist, der Heuristik noch mehr lohnende Verbesserungen anbieten zu können. Das so veränderte Verfahren wird auch weiterhin nach maximal n Schritten eine Iteration abbrechen, denn auch weiterhin werden nur n neue Kanten aufgenommen. Es stehen lediglich mehr Kandidaten zur Verfügung. Tabelle 4.5 liefert die bereits zuvor benutzte Zusammenfassung der zugehörigen Statistiken.

	Approximation	Zeit
Minimum	1,000	2 ms
Median	1,060	13.618 ms
Arithm. Mittel	1,059	757 s
Maximum	1,150	254.517 s

Tabelle 4.3.: Zusammenfassung: Lin-Kernighan mit Backtracking und ohne Fixieren der entfernten Kanten

Die allgemeine Güte der Lösungen verbessert sich minimal gegenüber der ursprünglichen Variante der Heuristik. Die Durchschnittsgüte der berechneten Schätzung steigt von 1.063 auf 1.059, während der Median gleich bleibt. Obwohl die Vorstellung naheliegt, dass mit Vergrößerung des Suchraums auch die Laufzeit wächst, zeigen die Simulationen ein nicht ganz einheitliches Bild. So nimmt für einige Instanzen sogar die Laufzeit ab, während sie sich für andere erhöht. Die Laufzeit verbleibt für kleinere Instanzen zuerst im Bereich von $O(n^{2.3})$, wie der Plot der Laufzeiten in Abbbildung 4.6 nahelegt.

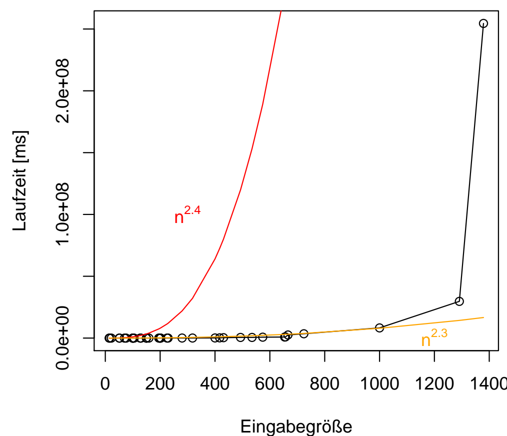

Abbildung 4.6.: Wachstum der Laufzeit mit Vergleichsgrößen in der Simulation mit Backtracking und ohne Fixieren der entfernten Kanten.

So bleibt die Laufzeit bei den kleineren Instanzen zwar durchaus moderat, aber die Berechnung der Instanz *nrw1379* verschlingt grob zehnmal soviel Rechenzeit wie zuvor, als noch alle geänderten Kanten fixiert wurden. Beachtlich ist der Sprung in der Laufzeit allemal, denn bis zur nächstkleineren untersuchten Instanz *d1291* scheint $O(n^{2.3})$ ein guter Schätzwert der asymptotischen Laufzeit zu sein. Der Sprung danach in eine Region von $O(n^3)$ lässt vermuten, dass es sich hier um eine Singularität handelt. Es stellt sich aber trotzdem die Frage, ob die Laufzeit für große n nicht vielleicht doch mit einer anderen Größenordnung wächst.

Für die Güte der berechneten Lösungen zeigt sich aber ein eindeutigeres Bild wie die Abbildung 4.5 nahelegt. Die Güte der Lösung für die kleinen Instanzen ist am Besten und scheint sich im Mittel für größere Instanzen in der Nähe des Approximationsfaktors von $1,10$ zu stabilisieren. Im direkten Vergleich mit der traditionellen Variante fällt auf, dass die Ergebnisse zwar leicht besser sind, aber wieder existieren einige Ausreisser mit mittlerer Größe.

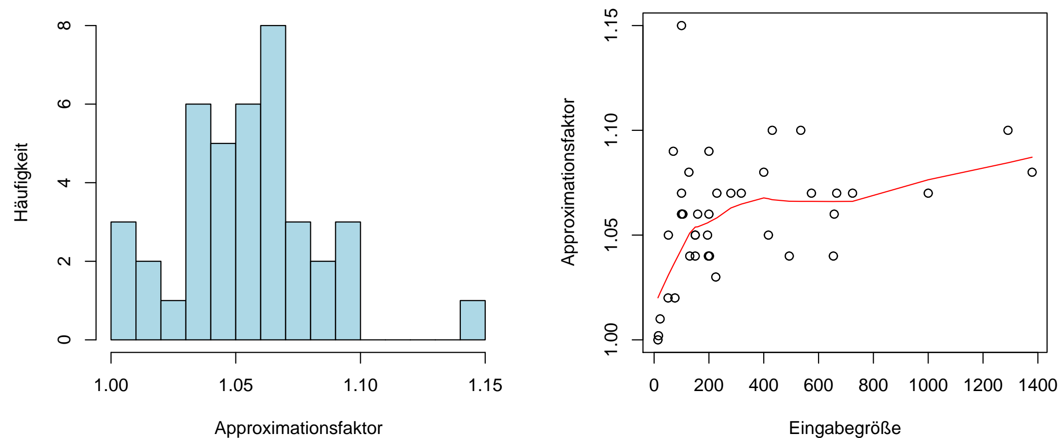

Abbildung 4.7.: Güte der Lösungen in der Simulation mit Backtracking und ohne Fixieren entfernter Kanten.

Der schlechteste gefundene Approximationsfaktor ist in dieser Simulationsserie mit 1.15 wieder bei der Instanz *kroC100* zu finden. Sie erscheint nur als Einzelereignis unter 40 getesteten Instanzen. Dies ist Abbildung 4.5 sowohl links im Histogramm der Approximationsfaktoren, als auch rechts im Plot erkennbar. Interessanterweise ist die kanonische Tour als Starttour für dieses schlechte Ergebnis verantwortlich. In 10 unabhängigen Läufen mit zufälligen Starttouren verbessern sich die Approximation auf Werte zwischen 1.05 und 1.08. Auch hier sein noch einmal auf Kapitel 4.7 und den dort untersuchten Einfluß der Starttour auf die Güte der jeweiligen Lösung hingewiesen.

4.6. Lin-Kernighan ohne Fixieren hinzugefügter Kanten

Die nächste Variante des Disjunktheitskriteriums dreht die Fixierung der Kanten genau um. Hinzugefügte Kanten dürfen jetzt wieder gelöscht werden und nur zuvor gelöschte Kanten dürfen nicht wieder aufgenommen werden. Sie entspricht im Kern der PLS-vollständigen Variante nach Papadimitriou [Pap92] aus Kapitel 3.7. Die Idee, warum dies getan wird, ist dieselbe wie schon zuvor. Der Heuristik soll eine größere Nach-

barschaft angeboten werden und so die Chance auf bessere Zwischen- und dadurch auch Endtouren erhöht werden. Die Erwartungshaltung an die Laufzeit ist, dass sie zwar generell erhöht ist, aber keine dramatischen Sprünge zu verzeichnen sind. Auch in dieser Variante muss eine Iteration nach n Tauschoperationen enden, denn die Zahl entfernter Kanten ist maximal n.

Die Auswertung der Simulationen auf dieser Variation zeigt verbesserte Lösungen gegenüber den anderen Heuristiken. Die kurze Zusammenfassung in Tabelle 4.4 gibt einen ersten Überblick über diese Ergebnisse.

	Approximation	Zeit
Minimum	1,000	2 ms
Median	1,060	1.678 ms
Arithm. Mittel	1,056	2.072 s
Maximum	1,100	46.140 s

Tabelle 4.4.: Zusammenfassung: Lin-Kernighan ohne Fixierung hinzugefügter Kanten

Im Mittel ist die gefundene Lösung besser als die vorherige Variante mit alleinigem Fixieren der hinzugefügten Kanten. Zudem sinken die beobachteten Laufzeiten wieder deutlich. Beachtlich ist, dass die größte berechnete Instanz *nrw1379* nur noch etwa ein sechstel der Zeit benötigt. Der Plot in Abbildung 4.8 zeigt die beobachteten Laufzeiten und dazu wieder die bekannten Kurven der Funktionen $f(n) = n^{2.3}$ (orange) und $g(n) = n^{2.5}$ (rot) zum Vergleich.

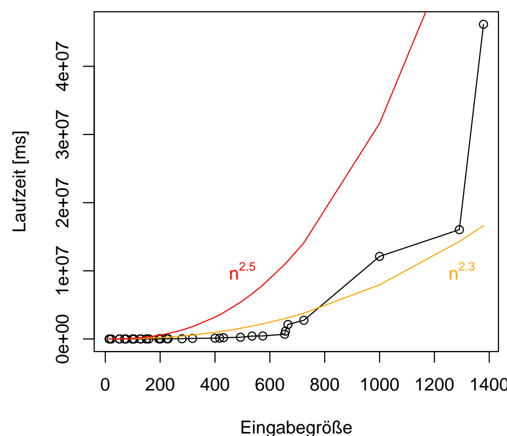

Abbildung 4.8.: Wachstum der Laufzeit mit Vergleichsgrößen in der Simulation mit Backtracking und ohne Fixieren hinzugefügter Kanten.

Es zeigt sich ein sehr ähnliches Verhalten wie auch schon in der Simalation zuvor. Es

lässt sich ein deutlich besseres Verhalten der Laufzeit im Vergleich zur vorherigen Simulation beobachten. Zuerst wächst die benötigte Zeit als Funktion der Eingabgröße moderat an und ist sehr nahe an der Funktion $n^{2.3}$. Auch hier ist ein Anstieg der Laufzeit für die Instanz *nrw1379* zu sehen. Dieser ist wie bereits erwähnt aber um Größenordnungen geringer als noch zuvor. Ein Sprung auf n^3 ist nicht zu sehen, eher in Richtung $n^{2.5}$. Dennoch ist dies aber nicht zu vernachlässigen.

Die Güte der Heuristik hat in dieser Variante keinen deutlichen Ausreisser mehr. Abbildung 4.9 zeigt, dass die Ergebnisse aller Approximationsfaktoren im Intervall von $[1,00 : 1,10]$ liegen. Interessant ist, dass das Histogramm einen deutlichen Schwerpunkt auf der Höhe des Medians zeigt. Rechts im Plot der Eingabegröße gegen den Approximationsfaktor zeigt die geglättete Funktion, dass bei wachsender Eingabegröße nur wenige schlechtere Ergebnisse zu erwarten sind. Die Funktion scheint, sich einem Grenzwert in der Nähe von $1,09$ anzunähern.

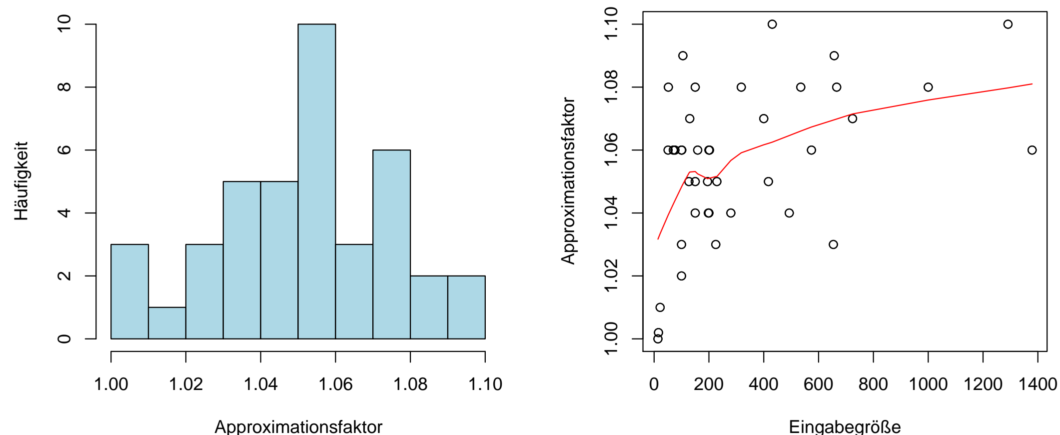

Abbildung 4.9.: Güte der Lösungen in der Simulation mit Backtracking und ohne Fixieren hinzugefügter Kanten.

Einige wenige kleine Instanzen werden sehr gut approximiert und das Gros der berechneten Instanzen liegt um den Median von $1,06$ verteilt. Die Größe der Instanzen scheint aber keinen besonderen und wenn, nur einen geringen Einfluss auf die Güte der Lösung zu haben. Dennoch lassen sich diese schon guten Lösungen noch weiter verbessern durch einen einfachen Arbeitsschritt. Statt durch die Nummerierung der Knoten eine feste Starttour zu erzwingen, wird diese Nebenbedingung gelockert.

4.7. Einfluss der Starttour auf die Güte

Ursprünglich schlagen Lin und Kernighan [LK73] vor, dass die Heuristik auf einer zufälligen Starttour beginnt. Dies soll gute Lösungen garantieren. Die kanonische Tour in den vorangegangenen Simulationen zu benutzen hatte die Idee besonders gute, bzw. schlechte Starttouren zu identifizieren und diese sehr leicht reproduzieren zu können. In diesem Abschnitt wird nun zum Vergleich untersucht, wie sich die Heuristik auf zufälligen Starttouren verhält. Dazu wird aus der Menge der möglichen Permutationen der Knoten eine Permutation uniform gezogen. Die Initialtour durchläuft die Knoten in der Reihenfolge, wie sie die gezogene Permutation aufzählt.

Jede der 40 Instanzen wurde in drei unabhängigen Läufen berechnet und das jeweils beste Ergebnis für eine Instanz in die Wertung aufgenommen. Dadurch soll verhindert werden, dass eine „schlechte“ Zufallszahl das Ergebnis beeinträchtigt. Es zeigte sich in den Simulationen, dass auf allen Instanzen die drei Ergebnisse eng beieinander lagen und sich im schlimmsten Fall nur um $1,5\%$ unterschieden. Die kurze Zusammenfassung in Tabelle 4.5 zeigt in gewohnter Übersicht die Eckdaten der Simulationsergebnisse.

	Approximation	Zeit
Minimum	1,000	4 ms
Median	1,045	1.635 ms
Arithm. Mittel	1,044	2.034 s
Maximum	1,080	45.140 s

Tabelle 4.5.: Zusammenfassung: Lin-Kernighan ohne Fixieren hinzugefügter Kanten mit randomisierten Starttouren

Es ist offen ersichtlich, dass eine zufällige Starttour bessere Ergebnisse liefert als die kanonische Starttour. Der Median und auch das arithmetische Mittel der gefundenen Approximationsfaktoren sinken im Vergleich zu den kanonischen Starttouren deutlich. Sie gleichen sich zudem weiter an, was für ersichtlich homogenere Ergebnisse spricht. In absoluten Zahlen liegen sie etwa ein Prozent unter den Werten der Simulationen mit kanonischer Starttour. Das Maximum der beobachteten Approximationsfaktoren sinkt auf 1,08, während sich das Gros der gemessenen Werte bei $1,05 \pm 0,01$ befindet.

Der Plot rechts in Abbildung 4.10, der Eingabegröße gegen Approximationsfaktor aufträgt, zeigt deutlich, dass die kleinen Instanzen mit weniger als etwa 200 Knoten sehr gut approximiert werden. Die aufgetragene Glättung hat nun auch im unteren Bereich einen gleichmäßigen Verlauf. Für die größeren Instanzen entspricht er aber in etwa den vorherigen Beobachtungen. Im Vergleich zur selben Abbildung der vorherigen Variante (Abb. 4.9 rechts) fällt deutlich auf, dass vor allem die kleinen Instanzen besser approximiert werden. Zuvor hatte die geglättete Linie einen lineareren Verlauf, der nun so erst ab einer Eingabegröße von etwa 400 bis 600 Städten erkennbar ist. Auch in dieser Simulation scheint sich die Glättung einem Grenzwert im Intervall $[1,08 : 1,10]$ anzunähern.

Das Histogramm bestätigt diese Beobachtungen. Es hat einen deutlichen Schwerpunkt um den Median, wie in Abbildung 4.10 zu erkennen ist. Auf der rechten Seite der Abbildung ist zu erkennen, dass die einzelnen Datenpunkte weniger streuen als noch zuvor. Die Punkte sind im Mittel näher an der Linie der geglätteten Daten. Eine mögliche Interpretation ist, dass die zufällige Wahl der Starttour mit hoher Wahrscheinlichkeit die „Spitzen" aus den Simulationsergebnissen nimmt.

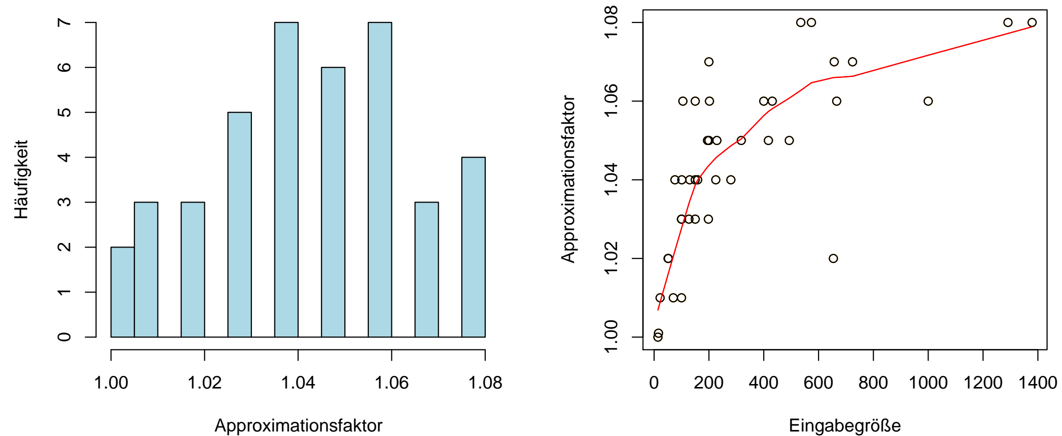

Abbildung 4.10.: Güte der Lösungen in der Simulation mit Backtracking mit randomisierter Starttour und ohne Fixieren hinzugefügter Kanten.

Die Freiräume zwischen den Balken des Histogramms entstammen der erhöhten Auflösung der Darstellung gemessen an der Intervalllänge.

Die Laufzeit ist verglichen mit der vorherigen Variante, wie erwartet, fast unverändert. Die Plots in den Abbildungen 4.8 und 4.11 sind daher auch weitestgehend deckungsgleich und weichen nur um vernachlässigbare Differenzen voneinander ab. Hier sei auf die Ergebnistabelle im Anhang unter Abschnitt A.3.3 verwiesen.

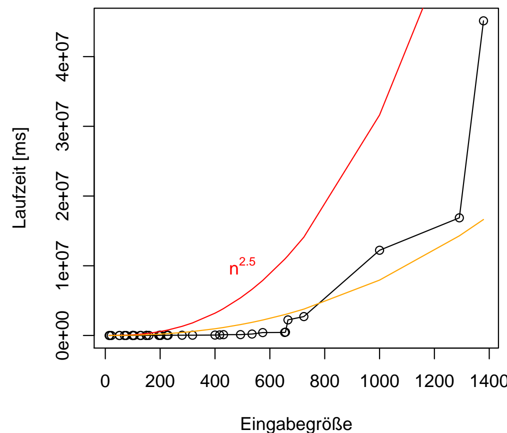

Abbildung 4.11.: Wachstum der Laufzeit mit Vergleichsgrößen in der Simulation mit Backtracking und ohne Fixieren hinzugefügter Kanten.

4.8. Fazit

Die besten Ergebnisse der Simulationen zeigt die zuletzt überprüfte Variante als leichte Abwandlung der ursprünglichen Heuristik von Lin und Kernighan. Das gelockerte Disjunktheitskriterium in der Form, dass hinzugefügte Kanten auch wieder aus der Tour gelöscht werden dürfen, bringt die besten Approximationen. Zugleich ist die beobachtete Laufzeit im Vergleich zu den guten Ergebnisse sehr akzeptabel.

Mit randomisierten Starttouren können diese Ergebnisse noch einmal leicht verbessert werden. Allerdings steigt die tatsächliche benötigte Laufzeit mit wachsender Instanzgröße allgemein so an, dass , mit keinem Simulationslauf Instanzen von mehr als 1.400 Knoten berechnet wurden. Helsgaun [Hel00] berichtet in diesem Zusammenhang von Laufzeiten jenseits von 15 Tagen für große Instanzen. Zu solchen Fällen würde sich der Einsatz leistungsfähiger Rechner lohnen und eine leistungsfähige parallelisierte Variante der Lin-Kernighan-Heuristik wäre ein erstrebenswertes Ziel. Hier gibt es erste Ansätze wie etwa verteiltes Simulated Annealing [SSFS02] mit der Nachbarschaft der Lin-Kernighan Heuristik, aber noch keine überzeugenden Verfahren, die bei vergleichbarer Rechenarbeit und -zeit ähnlich gute Ergebnisse wie die der sequentiellen Heuristik selbst bieten. Zudem gibt es in der Literatur auch keine überzeugende parallele Modellierung der Heuristik, die diese guten Ergebnisse verspricht. Die Standardtechniken sind hier die Zerteilung der Instanz in Subgraphen, die sich wie ein Puzzle zur Gesamtinstanz zusammen setzen lassen, oder die parallele Berechnung mehrerer Temperaturen, bzw. Abkühlschemata im Simulated Annealing.

Ein wichtiges Ergebnis der Simulationen ist, dass die Backtracking-Schritte unabkömmlich sind und maßgeblich zu den guten Ergebnissen der Heuristik beitragen. Interessant ist weiterhin die Tatsache, dass zufällige Starttouren als Ausgangspunkt der Suche für die Qualität der berechneten Touren sprechen.

5. MAX2SAT

Die Idee der lokalen Suche in variabler Tiefe lässt sich auch auf weitere Probleme übertragen. Die Gruppe der Erfüllbarkeitsprobleme ist seit vielen Jahren Gegenstand der Forschung mit klassischen Ergebnissen, wie zum Beispiel dem Satz von Cook [Coo71]. Ein wichtiger Anwendungsbereich für das Erfüllbarkeitsproblem ist die Forschung auf dem Feld der künstlichen Intelligenz. Im folgenden Teil wird die Optimierungsvariante des Erfüllbarkeitsproblems betrachtet, die nach einer Belegung sucht, die die Anzahl erfüllter Klauseln einer Formel in 2CNF maximiert. Die Erwartungshaltung ist nach den guten Ergebnissen des Suchparadigmas für das Traveling Salesman Problem, dass auch bei anderen schwierigen Problemen ähnlich gute Ergebnisse erzielbar sind.

5.1. Definition des Problems

Sei eine Menge X, bestehend aus den booleschen *Variablen* $x_1, x_2, \ldots, x_n$, gegeben. Jede dieser Variablen kann entweder den Wert 1 (für *wahr*) oder 0 (für *falsch*) annehmen. Ein *Literal* ist eine Variable $x_i \in X$, bzw. eine negierte Variable $\overline{x_i}$. Eine *Klausel* ist eine Disjunktion mehrerer Literale, also eine logische ODER-Verknüpfung, bspw. $(x_i \vee x_j \vee x_k)$. Eine *CNF* (**C**onjunctive **N**ormal **F**orm) ist eine Konjunktion von Disjunktionen, also eine logische UND-Verknüpfung der Klauseln. Eine *k-CNF*-Formel ist nun eine aussagenlogische Formel in CNF derart, dass die Anzahl der Variablen in jeder Klausel höchsten k beträgt. Das heisst, dass es durchaus auch Klauseln geben darf, in denen weniger als k Literale vorkommen.

Eine *Wahrheitsbelegung* b von X (kurz: *Belegung*) ist eine Abbildung $b : X \mapsto \{0, 1\}$. Jeder Variable wird entweder 0 oder 1 zugewiesen. Eine Klausel, die unter einer gegebenen Belegung als *wahr* ausgewertet wird, heisst *erfüllt.* Eine aussagenlogische Formel, für die eine Belegung existiert, die alle Klauseln erfüllt, heisst *erfüllbar* und eine Belegung heisst *maximal erfüllend*, wenn sie eine maximal mögliche Anzahl an Klauseln erfüllt. Eine maximal erfüllende Belegung ist also offensichtlicherweise eine optimale Belegung. Die Anzahl der erfüllten Klauseln unter einer optimalen Belegung wird mit $OPT(F)$ und die Anzahl der erfüllten Klauseln unter einer beliebigen Belegung b wird mit $EVAL(F, b)$ bezeichnet. Die Gesamtzahl der Klauseln in einer Formel wird im Allgemeinen mit m und die Anzahl der Variablen mit n bezeichnet. Das Problem MAX2SAT ist nun wie folgt definiert:

Definition 13 (MAX2SAT)
Gegeben: Aussagenlogische Formel F in 2CNF.
Gesucht: Belegung der Variablen in F, die die Anzahl erfüllter Klauseln maximiert.

Damit ergibt sich der Approximationsfaktor für dieses Maximierungsproblem als

$$\frac{OPT(F)}{EVAL(F,b)} \ .$$

MAX2SAT ist ein $\mathcal{NP}$-vollständiges Problem [Pap94] und gerade deswegen interessant, weil das zugrunde liegende Problem 2SAT sehr einfach gelöst werden kann und in $\mathcal{P}$ liegt. Es gibt daher die Ansicht, dass es sich bei MAX2SAT um ein relativ einfaches unter den schweren Problemen handelt, obwohl es natürlich offensichtlich schwieriger ist als 2SAT.

In der Tat gibt es bereits gute Approximationsalgorithmen, wie beispielsweise das Verfahren von Feige und Goemans [FG95]. Der von Ihnen vorgeschlagene Algorithmus approximiert MAX2SAT-Instanzen mindestens bis auf den Faktor $1,074$, wobei dieses Ergebnis nicht auf das allgemeine MAXkSAT übertragbar ist. Hier sind solch gute Approximationsfaktoren noch offen. In diesem Kapitel wird nun die Frage experimentell untersucht, wie gut sich lokale Suche in variabler Tiefe für MAX2SAT schlägt.

5.2. Definition der Nachbarschaft

Zu jedem lokalen Suchverfahren muss eine Nachbarschaft definiert werden. Ein oft gebräuchlicher Ansatz für aussagenlogische Formeln ist die *k-Flip Nachbarschaft*. Für eine gegebene und im Sinne des Problems zulässige Lösung (hier: eine Belegung) y für MAX2SAT gehört jede andere Lösung y' zur k-Flip Nachbarschaft $N_k(y)$, wenn sich y und y' höchstens in k Positionen, d.h. in maximal k Bits der zugehörigen Belegung, unterscheiden, also:

$$N_k(y) = \{y' \mid y \text{ und } y' \text{ unterscheiden sich in maximal } k \text{ Bits}\}$$

Diese Definition folgt der Idee, dass sich eine zwar zulässige, aber nicht optimale Lösung y von einer optimalen Lösung y_{opt} durch das Flippen von k Bits unterscheidet. Nun sind leider weder die Größe von k noch die jeweiligen zu flippenden Bits a priori bekannt. Ein Lösungsverfahren, dass nach dieser Suchstrategie eine Probleminstanz optimiert, kann daher nur versuchen, die k Bits auf geschickte Weise sukzessiv zu identifizieren. Die Variabilität der Nachbarschaft wird dadurch erzeugt, dass bestimmte Bits nicht betrachtet werden. Dies entspricht in natürlicher Weise der Idee von Lin und Kernighan aus Kapitel 3.2.

5.3. Algorithmus für MAX2SAT

Der Algorithmus erhält als Eingabe eine aussagenlogische Formel in 2CNF. Ausgehend von einer Ausgangslösung wird sequentiell durch einzelne Bitflips, also eine 1-Flip-Nachbarschaft, eine bessere Lösung in der Nachbarschaft der Ausgangslösung gesucht. Wird zwischenzeitlich eine bessere als die bisher beste bekannte Lösung gefunden, so

wird diese als die neue beste bekannte Lösung, der sogenannte *Champion*, vermerkt. Innerhalb einer Iteration werden die geflippten Bits fixiert, sie werden also nicht mehr weiter innerhalb einer Iteration berührt. Dies entspricht der Idee der variablen Tiefe. Sie bezieht sich eindeutig auf die Anzahl fixierter Bits innerhalb einer Iteration, denn eine solche endet, sobald eine Lösung y_l gefunden wurde und kein besserer Nachbar für y_l mehr gefunden werden kann, ohne gesperrte Bits wieder freizugeben. Eine neue Iteration beginnt jeweils mit der zuletzt gefundenen besten Lösung. Die Reihe der Iterationen bricht dann ab, wenn in der letzten Iteration keine weitere Verbesserung, also kein neuer Champion gefunden werden kann.

Algorithmus 12 Algorithmus zur Approximation von MAX2SAT

Eingabe: Ausssagenlogische Formel in 2CNF
Ausgabe: Lokal optimale Lösung

```
 1: Erzeuge Initiallösung
 2: repeat
 3:    Wähle den Champion als Ausgangspunkt der Suche.
 4:    Lösche die Tabuliste.
 5:    while noch freie Variablen verfügbar do
 6:       Suche greedy nach einem besten Flip einer freien Variable.
 7:       Flippe diese Variable und friere sie ein.
 8:       Speichere neuen Champion falls vorhanden.
 9:    end while
10: until kein neuer Champion mehr gefunden
```

Dieser Aufbau erinnert stark an die Implementierung der Lin-Kernighan Heuristik von Johnson et al. Die innere Schleife durchforstet die Nachbarschaft auf der Suche nach neuen Championlösungen. Die äußere Schleife steuert die innere und bricht das Verfahren ab, sobald innerhalb einer Iteration kein neuer Champion mehr gefunden wird.

Die Verbesserung, die durch das Umdrehen (Flippen) einer Variablenbelegung erzeugt wird, entspricht der Anzahl der Klauseln, die zusätzlich, bzw. weniger erfüllt werden, also $g_i = EVAL(F, b_i) - EVAL(F, b_{i-1})$. Natürlich kann diese Zahl auch negativ sein. Es werden in der Suche nach einem besten Nachbarn auch zwischenzeitliche Verschlechterungen zugelassen.

Es lässt sich ein erstes theoretisches Resultat zeigen:

Satz 11 (2-Approximation durch Algorithmus 12)
Die vom vorgeschlagenen Algorithmus erzeugte Lösung ist eine 2-Approximation.

Der *Beweis* erfolgt durch Widerspruch und benutzt die Eigenschaft, dass die maximale Anzahl erfüllter Klauseln höchsten m ist. Es wird gezeigt, dass das Verfahren immer mindestens $\frac{m}{2}$ Klauseln erfüllt.

Sei F eine beliebige Formel in 2CNF und sei t desweiteren eine Belegung für F. Angenommen, die Formel F enthalte unter der Belegung t weniger als $m/2$ erfüllte Klauseln, dann muss es mindestens eine Variable geben, die so gewählt wurde, dass sie weniger als die Hälfte derjenigen Klauseln erfüllt, in denen sie vorkommt. Gäbe es eine solche Variable nicht, so wären automatisch mindestens $m/2$ Klauseln erfüllt.

Da die Heuristik nach Vorraussetzung terminiert, gibt es einen letzten Champion, der gefunden wird, nach Vorraussetzung weniger als $m/2$ Klauseln belegt und der in der letzten Iteration nicht verbessert wurde. Die Variablenbelegungen, die von der Heuristik als erstes geändert werden, sind diejenigen, die eine direkte Verbesserung bringen. Eine solche Verbesserung ist aber nur möglich, wenn für die zu flippende Variablenbelegung mehr als die Hälfte der Klauseln unerfüllt sind, in welchen sie vorkommt.

Es gibt nun eine am schlechtesten belegte Variable x_i, für die am wenigsten und weniger als $\frac{m}{2}$ Klauseln erfüllt sind. Der Algorithmus wählt aber eine Verbesserung *greedy*, d.h. die Variable x_i oder eine ebenso schlechte Variable wird geflippt, denn dieser Flip bringt offensichtlich die beste Verbesserung. Das heisst aber, dass mit x_i oder mit jeder anderen Variablen eine neue Champion-Tour gefunden würde und der Algorithmus noch nicht abbrechen dürfte, sondern eine weitere Iteration der inneren Schleife begänne. Das steht aber im Widerspruch zu der Vorraussetzung, dass der Algorithmus mit einer Belegung abbricht, die weniger als $\frac{m}{2}$ erfüllte Klauseln enthält. Dies kann also nicht sein und der vorgeschlagene lokale Suchalgorithmus in variabler Tiefe ist also eine 2-Approximation. □

Zudem lässt sich auch die Zugehörigkeit zur Klasse PLS zeigen. Der vorgeschlagene lokale Suchalgorithmus in variabler Tiefe gehört zur Komplexitätsklasse PLS. Das Verfahren wird mit dem Namen des Problems und einem kleinen Zusatz identifiziert: MAX2SAT-LS. Die Tatsache, dass MAX2SAT-LS in PLS liegt, lässt sich direkt aus der Definition von PLS schließen. Die Heuristik erfüllt jede der geforderten Bedingungen:

1. Die Menge der Eingaben X_Π ist die Menge aller Formeln in 2CNF und die zulässigen Lösungen ist die Menge der zu einer Eingabe $x \in X_\Pi$ möglichen Belegungen. Bei n Variablen, die zu x gehören sind dies 2^n viele.

2. Die Überprüfung einer Lösung auf Zulässigkeit ist trivial.

3. Da die Zahl n der Variablen gegeben ist, lässt sich eine Initialbelegung offensichtlich als ein Element aus $\{0, 1\}^n$ problemlos finden.

4. Die Anzahl erfüllter Klauseln in x lässt sich offensichtlich leicht berechnen.

5. Eine einzelne Suche nach einem im Sinne der Heuristik besseren Nachbarn nimmt klar polynomielle Zeit in Anspruch. D.h. es wird in polynomieller Zeit ein Nachbar gefunden oder der Algorithmus gibt auf.

Für die einfache 1-Flip Nachbarschaft ist aus der Literatur [AL97] bekannt, dass sie PLS-vollständig ist. Dieses Ergebnis lässt sich aber nicht direkt auf lokale Suche in

variabler Tiefe, wie vorgeschlagen, reduzieren, denn die einfache 1-Flip Nachbarschaft könnte in einer längeren Suchphase dort ein lokales Optimum finden, wo die lokale Suche in variabler Tiefe einen Flip durch Einfrieren der Belegung sperrt.

5.4. Datenformat für MAX2SAT-Instanzen

Um eine MAX2SAT-Instanz zu optimieren, ist es notwendig die Testinstanzen nicht nur zu erzeugen, sondern auch zu speichern. Zur Sicherung wird ein einfaches Datenformat entwickelt, dass sich grob am Datenformat der TSPLIB orientiert. Die für diese Arbeit benutzten Testinstanzen werden nach der Arbeit von Motoki [Mot05], bzw. Kapitel 5.5 erzeugt.

Im Gegensatz zum Traveling Salesman Problem existiert für MAX2SAT keine anerkannte Instanzensammlung zur standardisierten experimentellen Analyse. Die benutzten Testinstanzen müssen deswegen für die experimentelle Analyse generiert werden.

Ein *Instanzgenerator* für MAX2SAT ist im Allgemeinen ein Algorithmus, der bei Eingabe der Variablenzahl n, bzw. Klauselzahl m eine aussagenlogische Formel in 2CNF und eine maximal erfüllende Belegung für die erzeugte Formel ausgibt. Ein einfaches Verfahren ist die zufällige Generierung von Testinstanzen. Für jede der zu erzeugenden Klauseln wird zufällig bestimmt, welche der $4 \cdot \binom{n}{2}$ Möglichkeiten, sie aus n Variablen zu bilden, gewählt wird. Allerdings wird hier die Erzeugung der maximal erfüllenden Belegung einfach vernachlässigt. Zufällige Instanzen haben aber den Nachteil, dass mit steigender Anzahl Klauseln im Verhältnis zur Variablenanzahl irgendwann hochwahrscheinlich jede beliebige Belegung maximierend wird.

Ob nun zufällig oder nach einem bestimmten Schema, jede erzeugte Instanz wird in einem speziellen Format gespeichert.

5.4.1. Datenformat für Instanzen

Das Format zur Speicherung der Instanzen orientiert sich am Format der TSPLIB. Für jede Klausel existiert eine einzelne Zeile und zusätzlich können der Name der Instanz und eine Kommentarzeile innerhalb der Datei gespeichert werden. Folgendes Beispiel erläutert das Format:

```
NAME:Beispielinstanz
COMMENT:Erläuterung zur Instanz
CLAUSE_SECTION
1 1 -2
2 -3 5
3 6 7
4 -7 1
EOF
```

In der ersten Zeile wird der Name der Instanz unabhängig vom tatsächlichen Dateinamen angegeben. In der zweiten Zeile kann ein Kommentar angegeben werden, etwa zur weiteren Erläuterung der Instanz. Beide Angaben sind optionale Beschreibungen und eher zur Visualisierung gedacht. Nach dem Marker `CLAUSE_SECTION` folgen in m Zeilen die einzelnen Klauseln, jede Klausel in ihrer eigenen Zeile. Schlussendlich folgt noch ein `EOF` um das Dateiende zu markieren.

Der Aufbau der dreispaltigen Klauselzeilen ist schlicht. Die erste Spalte gibt den Index der Klausel an. Prinzipiell wäre auch dieser Egal, denn die Reihenfolge der Klauseln spielt eigentlich keine Rolle. Der Index wird einfach hochgezählt, also gibt die i-te Klauselzeile das Aussehen der Klausel C_i an. Spalte 2 gibt den Index der Variable im ersten Literal und Spalte 3 den Index der Variable im zweiten Literal der Klausel an. Ein vorangestelltes Minuszeichen signalisiert die Negation. So lässt sich jede Formel in 2CNF auf einfache und leicht lesbare Art und Weise beschreiben, speichern und austauschen.

Die oben beschriebene Beispielinstanz lautet also: $(x_1 \vee \overline{x_2})(\overline{x_3} \vee x_5)(x_6 \vee x_7)(\overline{x_7} \vee x_1)$.

5.5. Erzeugung der MAX2SAT-Instanzen

Die zufällige Erzeugung von Testinstanzen für ein $\mathcal{NP}$-hartes Optimierungsproblem ist, wie bereits oben beschrieben, einfach und lässt sich problemlos implementieren. Sie hat allerdings den Nachteil, dass keine maximal erfüllende Belegung parallel mitgeneriert wird und mit steigendem Klauselverhältnis wird jede Belegung maximierend. Eine Aussage über die Approximationskomplexität der Heuristik ist daher mit zufälligen Formeln eigentlich kaum zu treffen.

5.5.1. Idealer und realer Testinstanzgenerator

Im Allgemeinen sollte ein idealer Generator für Testinstanzen eines schwierigen Problems so aufgebaut sein, dass aus der Eingabe von bestimmten Instanzparametern und Zufallsbits eine Testinstanz und eine zugehörige maximierende Lösung generiert werden.

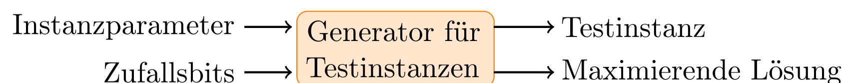

Abbildung 5.1.: Idealer Instanzgenerator

Die Laufzeit muss dazu polynomiell in der Länge der zu erzeugenden Instanz beschränkt sein. Nun ist aber nicht davon auszugehen, dass $\mathcal{NP} = \text{co-}\mathcal{NP}$ gilt. Als Resultat ist es nicht möglich, Testinstanzen aus der gesamten Menge aller möglichen Instanzen mit einer zugehörigen optimalen Lösung in polynomieller Zeit zu generieren.

Daher werden die Anforderungen an einen solchen Generator beschränkt. Den verträglichsten Kompromiss bietet die Einschränkung, nicht die gesamte Menge aller prinzipiell möglichen Instanzen erzeugen zu können, sondern nur eine Untermenge davon. Dafür geschieht dies aber mit der gleichzeitigen Bestimmung einer optimalen Lösung. Ein solcher Instanzgenerator hat also folgende Eigenschaften. Die erzeugten Testinstanzen

- kommen immer mit einer zugehörigen optimalen Lösung,
- sind nur aus einer Untermenge aller Instanzen gewählt und
- werden effizient in polynomieller Zeit im Vergleich zur Instanzgröße generiert.

Die Forderung nach polynomieller Laufzeit ist klar, denn ein Generator, der exponentielle Laufzeit besäße, wäre trivial. Zu einer zufällig erzeugten Instanz wird in exponentieller Laufzeit eine optimale Lösung durch ausprobieren aller möglichen Lösungen berechnet. Desweiteren sollten die generierten Instanzen nicht trivial, sondern schwer zu lösen sein.

5.5.2. Erzeugung der Testinstanzen nach Motoki

Für eine beliebige gegebene Boolesche Formel F in 2CNF über der Variablenmenge $X = \{x_1, \ldots, x_n\}$ existiert der sogenannte *Implikationsgraph* $G_F = (V, E)$, wobei $V = X \cup \{\overline{x_i} | x_i \in X\}$ die Knontenmenge und $E = \{(v_i, v_j) \mid v_i, v_j \in V$ und $(\overline{v_i} \vee v_j) \in F\}$ die Menge der Kanten sind. Zu beachten ist, dass für jede gerichtete Kante $(x_i, x_j) \in E$ auch die *Komplementärkante* $(\overline{x_j}, \overline{x_i}) \in E$ existiert, da ja die Symmetrie $(b \vee a) = (a \vee b)$ gilt. G_F ist also ein gerichter Graph, der möglicherweise auch Mehrfachkanten enthält.

Nun ist leicht zu sehen, dass eine Formel F in 2CNF genau dann nicht erfüllbar ist, wenn G_F einen Kreis enthält, in welchem sowohl eine Variable v als auch ihre Negierung $\overline{v}$ vorkommen. Existiert in einem Graph G_F nun ein solcher Kreis C, dann existiert gleichzeitig aber auch ein zweiter Kreis, der aus den Komplementärkanten des Kreises C besteht. Ein solches Paar von Kreisen wird *Widerspruchsdoppelkreis (WDK, engl. contradictory bicycle)* genannt.

Ausgehend von dieser Konstruktion lässt sich auch zeigen, dass das Problem 2SAT in $\mathcal{P}$ liegt. Die Eigenschaft, dass für kein i, $1 \leq i \leq n$, die Literale x_i und $\overline{x_i}$ in einer starken Zusammenhangskomponente liegen, ebenso wie die Konstruktion des Implikationsgraphen aus der gegebenen Formel, sind recht offensichtlich in Polynomialzeit zu berechnen.

In der Booleschen Algebra ist der Ausdruck $l(u) \Rightarrow l(v)$, wie bereits erwähnt, äquivalent zu dem Ausdruck $(\overline{l(u)} \vee l(v))$. Dies ist auch der Ursprung für die Bezeichnung des konstruierten Digraphen als sogenannter Implikationsgraph.

Für eine beliebige Belegung t über einer Variablenmenge X sei $\mathcal{B}_t$ die Menge der Booleschen Formeln in 2CNF derart, dass jedes $F \in \mathcal{B}_t$ folgende drei Eigenschaften erfüllt:

1. F besitzt genau eine Klausel, die von der Belegung nicht erfüllt wird.
2. G_F hat einen Widerspruchsdoppelkreis.
3. Wird eine Klausel aus F gelöscht, so ist der verbleibende Teil der Formel erfüllbar.

Weiterhin wird mit $\mathcal{C}_t$ die Menge aller Klauseln über der Variablenmenge X in 2CNF bezeichnet, die durch die Belegung t erfüllt werden. Nach dem dritten Gesichtspunkt ist F logischerweise eine minimale, unerfüllbare Formel. Die Konstruktion lässt sich leicht vornehmen.

O.B.d.A. sei $t = 1^n$ und $B \in \mathcal{B}_t$ eine beliebige Formel. Es ist klar, dass jede Klausel, die von t nicht erfüllt wird, aus zwei negativen Literalen bestehen muss und es ist auch klar, dass ein minimales B exakt eine solche Klausel enthält. Diese Klausel wird nun derart in Kanten des Implikationsgraphen überführt, dass die durch sie implizierte Kante in G_B von einem positiven Literal zu einem negativen Litearal führt. Jeder Zykel eines WDK für $t = 1^n$ kann in zwei Pfade zerlegt werden. *Ein* Pfad läuft nur durch positive Literale und *der* andere nur durch negative. Abbildung 5.2 verdeutlicht diesen Sachverhalt.

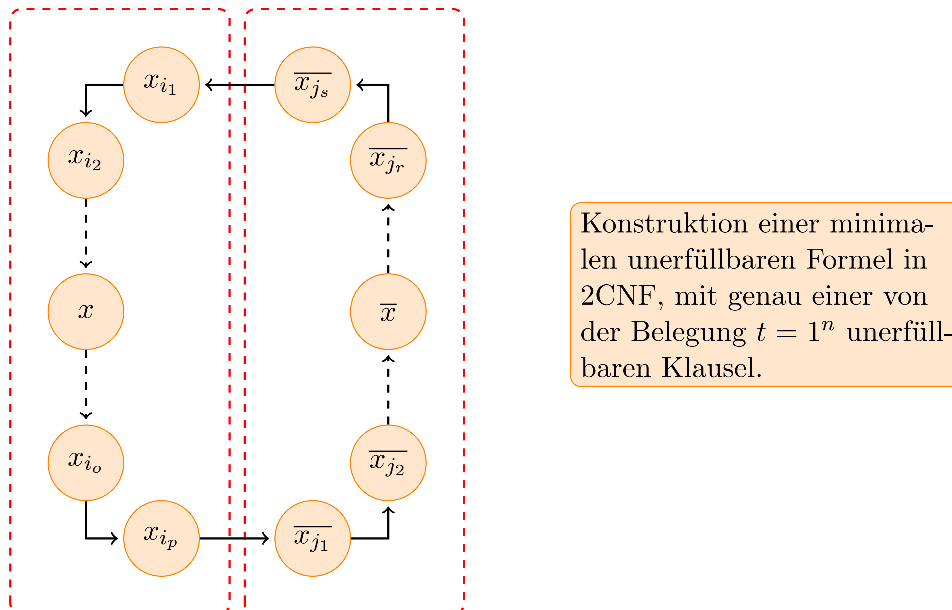

Abbildung 5.2.: Minimaler Widerspruchskreis für $t = 1^n$

Klar ist, dass beide Kreise eine gemeinsame Variable besitzen um einen Widerspruchsdoppelkreis zu bilden. Da keine Klausel trivial ist, also ein Literal und das dazugehörige Komplement enthält, muss die letzte Variable des einen Pfads unterschiedlich zur ersten Variable des anderen Pfads sein. Die triviale Klausel $(x_i \vee \overline{x_i})$ kommt also gar nicht vor.

Damit lässt sich die Formel B für $t = 1^n$ generieren: Zwei Variablensequenzen $x_{i_1}, \ldots, x_{i_p}$ und $x_{j_1}, \ldots, x_{j_s}$, deren jeweils letzte Variable der einen sich von der ersten Variable der anderen Sequenz unterscheiden, werden als Implikation $x_{i_1} \rightarrow \ldots \rightarrow x_{i_p} \rightarrow x_{j_1} \rightarrow \ldots \rightarrow x_{j_s} \rightarrow x_{i_1}$ betrachtet und daraus die äquivalenten Klauseln in 2CNF generiert. Für Belegungen $t \neq 1^n$ kommt noch ein letzter Arbeitsschritt hinzu. Für jeden Index i, $1 \leq i \leq n$, für den gilt, dass $t_i = 0$ ist, wird das im vorherigen Schritt erzeugte Literal l_i negiert. Auf naheliegende Weise wird so sichergestellt, dass die erzeugte Implikation für $t \neq 1^n$ die obigen drei Bedingungen erfüllt.

Die Elemente in $\mathcal{B}_t$ lassen sich einfach randomisiert erzeugen und ein Algorithmus mit linearer Laufzeit in Anzahl der Klauseln zur Erzeugung einer gesamten Testinstanz für MAX2SAT ist damit naheliegend:

Algorithmus 13 Erzeugung der Testinstanzen nach Motoki

Eingabe: Anzahl der Variablen n

Ausgabe: Testinstanz und zugehörige maximierende Belegung

1: Erzeuge leere Formel F.
2: Bestimme die maximierende Belegung $t \in \{0, 1\}^n$.
3: Wähle die Anzahl nicht erfüllter Klauseln k (≥ 0).
4: **for** $i = 1$ to k **do**
5: Erzeuge zufällig eine 2CNF Formel aus $\mathcal{B}_t$ über der Variablenmenge X aus und füge diese zu F hinzu.
6: Füge r zufällige Klauseln aus $\mathcal{C}_t$ zu F hinzu, wobei r eine zufällige natürliche Zahl ist.
7: **end for**

Für eine beliebige Belegung t und eine natürliche Zahl k besteht die erzeugte Formel F aus k nicht notwendigerweise disjunkten Formeln aus $\mathcal{B}_t$ und einer Anzahl von Klauseln aus $\mathcal{C}_t$. Unter der Belegung t müssen also mindestens k Klauseln als logisch *falsch* ausgewertet werden, da G_F ja k WDKs enthält. Andererseits enthält F aber auch nur k viele solcher Kreise und daher werden auch nur maximal k viele der Klauseln in F als logisch *falsch* ausgewertet. Daher muss t eine optimale Belegung und die Anzahl nicht erüllbarer Klauseln k sein.

Werden keine Klauseln aus $\mathcal{C}_t$ hinzugefügt, so hat die erzeugte Instanz mindestens eine zweite optimale Lösung $\bar{t}$, nämlich das Komplement zu t. Zur Erinnerung: Wegen der Symmetrie entspricht eine Implikation zwei Disjunktionen.

Neben der Kontrolle der Erfüllbarkeitseigenschaften bietet der vorgeschlagene Generator eine bedeutsame Eigenschaft. Ist $\mathcal{I}$ die Menge der vom Generator erzeugten Instanzen, dann ist es schwierig zu erkennen, ob eine Instanz aus dieser Menge ist oder nicht.

Satz 12 *Die Menge der erzeugten Instanzen $\mathcal{I}$ ist $\mathcal{NP}$-vollständig.*

Der Beweis erfolgt in zwei Schritten. Im ersten wird gezeigt, dass die Menge in $\mathcal{NP}$ liegt und im zweiten Schritt folgt die $\mathcal{NP}$-Härte.

Die Menge $\mathcal{I}$ liegt in $\mathcal{NP}$, denn die Belegung t, die Angabe der k Formeln aus $\mathcal{B}_\mathrm{t}$ und die Angabe der benutzten Untermenge von $\mathcal{C}_t$ bilden gemeinsam einen Zeugen für eine gegebene Instanz. Ein Orakel könnte diesen Zeugen erraten und mit ihm ließe sich in Polynomialzeit die Zugehörigkeit überprüfen.

Die Härte der erzeugten Instanzenmenge wird über eine Reduktion von 3SAT gezeigt. Sei F_{3CNF} eine beliebige Booleschen Formel in 3CNF über der Variablenmenge X mit den 3-Klauseln $c_1, \ldots, c_m$. Für alle i, $1 \leq i \leq m$ wird die i-te Klausel $C_i = (l_{i,1} \vee l_{i,2} \vee l_{i,3})$ auf eine 2CNF Formel B_i über der Variablenmenge $X \cup Y_i$, $Y_i = y_{i,1}, y_{i,2}$ abgebildet. Für B_i gilt die feste Konstruktionsvorschrift:

$$B_i = (\overline{l_{i,1}} \vee y_{i,1})(\overline{y_{i,1}} \vee l_{i,2})(\overline{l_{i,2}} \vee \overline{y_{i,2}})(y_{i,2} \vee l_{i,3})(\overline{l_{i,3}} \vee \overline{y_{i,1}})(y_{i,1} \vee y_{i,2})(\overline{y_{i,2}} \vee l_{i,1})$$

Die Variablen der Menge Y_i tauchen nur in B_i auf und ihre Belegung ist beliebig. Zu beachten ist, dass B_i genau dann einen Widerspruchsdoppelkreis, wenn C_i erfüllbar ist. Daher muss jede Wahrheitsbelegung über der Variablenmenge $X \cup Y_i$ mindestens eine der Klauseln als logisch *falsch* auswerten. Sei nun $F_{2cnf} = \bigwedge_i B_i$ die Konjunktion aller B_i. Die Anzahl der Klauseln in B_i ist $7 \cdot m$ und die der Variablen $n + 2 \cdot m$. Die Abbildung lässt sich also in Polynomialzeit konstruieren. Die Anzahl der WDKs ist genau gleich der minimalen Anzahl nicht zu erfüllender Klauseln, wenn F_{3cnf} erfüllbar ist. Das heisst, dass es genauso schwierig ist, generierte Instanzen zu erkennen, wie das Problem 3SAT zu entscheiden. $\square$.
Zudem scheint sich zeigen zu lassen, dass die Optimierung jeder generierten Instanz ebenfalls $\mathcal{NP}$-hart ist [Mot07].

5.6. Experimentelle Analyse

Zur experimentellen Analyse wurde eine Anwendung in der objektorientierten Programmiersprache JAVA realisiert. Die Analyse erfolgt auf zwei Weisen. Zum einen wird überprüft, wie gut die Testinstanzen approximiert werden. Zum anderen wird beobachtet, wie sich der vorgeschlagene Algorithmus im Vergleich zu einer simplen Heuristik verhält, die jede Variable genau einmal betrachtet.

5.6.1. Vergleichsheuristik

Die Heuristik, mit der das vorgeschlagene Verfahren verglichen wird, ist einfach aufgebaut. Die Liste der n Variablen wird sequentiell durchlaufen. Für jede Variable $x_i \in X$ wird geprüft, ob die Belegung x_i oder $\neg x_i$ insgesamt mehr Klauseln erfüllt und anschließend die objektiv bessere von beiden gewählt.

Algorithmus 14 Einfache Heuristik für MAX2SAT

Eingabe: Ausssagenlogische Formel in 2CNF
Ausgabe: Belegung b
1: Erzeuge eine Startlösung $b \in \{0,1\}^n$, hier $t = 0^n$.
2: **for** $i = 1$ to n **do**
3: Wähle diejenige Belegung für x_i, die mehr Klauseln erfüllt.
4: **end for**

Für Algorithmus 14 lässt sich nach ähnlichem Muster wie schon zuvor die theoretische Worst-Case-Schranke über die Güte der Lösung zeigen.

Satz 13 (2-Approximation durch Algorithmus 14)
Die von der Vergleichsheuristik erzeugte Lösung ist eine 2-Approximation.

Der *Beweis* wird per Induktion über die Anzahl der Variablen geführt. Für den Fall, dass es nur eine Variable gibt, ist der Fall klar. Offensichtlich gilt hier der Satz.

Angenommen, dass die Induktionsannahme und der Satz für $n-1$ $(n > 1)$ gelten. Der Algorithmus durchläuft in der FOR-Schleife in Schritt 2 alle Variablen. Wir betrachten den Fall, dass die letzte Variable geprüft wird und bereits die anderen $n-1$ Variablenbelegungen bestimmt wurden. Die letzte Variable heisse nun x.

Mit c_x, bzw. $c_{\overline{x}}$, seien die Anzahlen der Klauseln, in denen die Literale x, bzw. $\overline{x}$, vorkommen, bezeichnet, die in den vorherigen Schritten noch nicht als *wahr* ausgewertet wurden. O.B.d.A. sei $c_x \geq c_{\overline{x}}$, daher wird x mit einem logischen *wahr* belegt. Nun sei c_{n-1} die Anzahl der Klauseln, die durch die gewählten Belegungen der vorherigen $n-1$ Variablen, ausgewertet werden. Durch die Induktionsannahme sind mindestens die Hälfte, also $\geq \frac{c_{n-1}}{2}$, davon bereits wahr. Im n-ten Schritt werden nun zusätzlich $c_x + c_{\overline{x}}$ weitere Klauseln betrachtet. Da der Algorithmus $c_x \geq c_{\overline{x}}$ bestimmt, werden wieder mindestens die Hälfte dieser zusätzlichen und mithin mindestens die Hälfte aller Klauseln als erfüllt ausgewertet. □

Interessanterweise lässt sich für einen Spezialfall des MAX2SAT-Problems für diese Heuristik ein noch besserer Approximationsfaktor finden und zwar dann, wenn in jeder Klausel genau zwei Literale vorkommen. In jedem der n Schritte, die das Verfahren durchläuft, wird überprüft, welche Belegung der betrachteten Variable x_i mehr neue Klauseln erfüllt. Die zuvor erfüllten Klauseln sind deswegen uninteressant, da sie ja unabhängig von x_i erfüllt werden. Die Ausgabe am Ende ist eine Belegung b. Wird

die Formel F unter b ausgewertet, so zerfällt die Klauselmenge in 3 Teilmengen. Diese Mengen seien SAT_0, die die Klauseln enthält die kein erfülltes Literal haben, SAT_1, die die Klauseln mit genau einem erfüllten Literal enthält und SAT_2, die die Klauseln enthält, derer beider Literale erfüllt sind. Klar ist, dass $m = (|SAT_0|+|SAT_1|+|SAT_2|)$ gilt.

Es gibt eine erste wichtige Feststellung: Zu jedem Zeitpunkt $0 < i \leq n$ wird x_i so belegt, dass die Zahl Klauseln, die neu erfüllt werden mindestens so gross ist, wie die Zahl an Klauseln, denen ein nicht erfülltes Literal zugewiesen wird. Es gibt also Klauseln im Lauf des Verfahrens, die zwei erfüllte Literale enthalten, solche, die ein erfülltes Literal enthalten, wobei das zweite entweder unbestimmt oder als logisch 0 ausgewertet wird und schließlich noch solche Klauseln, die kein erfülltes Literal enthalten, sei es weil diese entweder mit logisch 0 ausgewertet werden oder deren Belegung noch nicht bestimmt wurde. Nachdem alle Belegungen bestimmt wurden, entspricht dies den Mengen $|SAT_i|$, $0 \leq i \leq 2$.

Stoppt die Heuristik nun, dann gibt es genau $|SAT_0|$ Klauseln, derer beider Literale mit logisch 0 ausgewertet werden. D.h. es ist mindestens $2 \cdot |SAT_0|$ vorgekommen, dass einer Klausel ein nicht-erfülltes Literal „zugewiesen" wurde. Dann müssen aber mit der ersten Feststellung mindestens $2 \cdot |SAT_0|$ Klauseln erfüllt werden. Die maximale Zahl an Klauseln, die nicht erfüllt werden, kann daher $m \cdot 3^{-1}$ nicht übersteigen und es ergibt sich der folgende Satz:

Satz 14 ($\frac{3}{2}$-Approximation durch Algorithmus 14)
Die vom vorgeschlagenen Algorithmus erzeugte Lösung ist eine $\frac{3}{2}$-Approximation.

□

Das Ergebnis lässt sich analog auf die vorgeschlagene lokale Suche in variabler Länge übertragen. Auch hier werden in jedem Schritt mindestens genausoviele Klauseln neu erfüllt, wie anderen Klauseln ein nicht erfülltes Literal zugewiesen wird. Beide Heuristiken scheinen, was die Güte der schlechtesten zu erwartenden Ergebnisse betrifft, laut den gefundenen Worst-Case-Schranken vergleichbar zu sein. Allerdings kann diese Schranke zu grob sein um einen qualifizierten Vergleich durchzuführen, denn die lokale Suche in variabler Tiefe darf ja von Haus aus länger laufen und könnte demnach in den zusätzlichen Phasen deutlich mehr Klauseln erfüllen. Um diesen Sachverhalt zu überprüfen, kommt die experimentelle Analyse ins Spiel.

5.7. Berechnungs- und Simulationsergebnisse

Der Simulationslauf wird mit generierten Instanzen aus Motokis Generator durchgeführt. Die Implementierung des Generator sieht als Eingabe die Anzahl der Klauseln vor. Die Anzahl nicht erfüllbarer Klauseln wird randomisiert bestimmt, ebenso wie

die Länge der Widerspruchskreise. Es werden insgesamt 11 Instanzen mit wachsender Variablenzahl generiert. Die ersten 5 Instanzen wachsen angefangen bei 10 Variablen jeweils um 10 weitere an, um dann in Schritten von 50 Variablen bis auf eine Zahl von 350 zu steigen. Ziel der Simulationen ist, zu überprüfen, wie gut sich die lokale Suche in variable Tiefe im Vergleich zur einfachen Vergleichsheuristik auf diesen Instanzen, für die ja die Optima bekannt sind, schlagen. Weiterhin ist zu überprüfen, ob sich die Qualität der berechneten Lösungen mit wachsender Variablenzahl, bzw. mit wachsendem Klausel-/Variablenverhältnis ändert.

5.7.1. Experimentelle Ergebnisse der Heuristik

Tabelle 5.1 zeigt die Ergebnisse der Simulationen auf den generierten Instanzen. Der Wert k gibt die Anzahl der nicht erfüllbaren Klauseln unter einer maximierenden Belegung an. Damit ist das Optimum bekannt. Zusätzlich sind in den Spalten *Erg. 1* und *Erg. 2* zuerst die Ergebnisse der einfachen Vergleichsheuristik und dann die der lokalen Suche in Variabler Tiefe angegeben. Danach folgen Approximationsfaktoren und die benötigte CPU-Zeit.

#	Var.	Klauseln	k	Erg. 1	Fkt.	Zeit	Erg. 2	Fkt.	Zeit
1	10	171	14	145	1,08	1ms	146	1,07	17ms
2	20	313	13	260	1,15	2ms	264	1,13	32ms
3	30	535	14	444	1,17	5ms	457	1,14	131ms
4	40	991	19	820	1,18	12ms	826	1,17	386ms
5	50	2.093	32	1.734	1,18	27ms	1.739	1,18	810ms
6	100	4.835	38	3.947	1,21	117ms	3.998	1,19	14.272ms
7	150	7.398	39	6.040	1,21	260ms	6.099	1,20	72.446ms
8	200	3.304	13	2.728	1,20	155ms	2.772	1,18	27.558ms
9	250	13.256	42	10.736	1,23	758ms	10.827	1,22	225.889ms
10	300	35.338	96	28.246	1,24	2.442ms	28.514	1,23	1.185708ms
11	350	75.927	174	60.958	1,24	6.063ms	61.232	1,23	1.959.990ms

Tabelle 5.1.: Simulationen auf Instanzen von Motoki

Beide Heuristiken haben eine sehr ähnliche Güte. In den Simulationen nimmt die Güte mit wachsendem Klausel- zu Variablenverhältnis bis zu einem Grenzwert ab, der 1.25 nicht zu überschreiten scheint. Instanzen mit kleiner Klauselzahl werden gut approximiert. Die gemessenen Approximationsfaktoren schwanken zwischen $1,08$ und $1,24$ für die einfache Vergleichsheuristik, bzw. leicht besser für die lokale Suche in variabler Tiefe mit 1.07 und 1.23. Die Abstände sind aber gering ausgeprägt. Die größte gemessene Differenz der Güte beider Heuristiken für eine Instanz ist $0,02$. Dies Differenz weitet sich auch nicht mit wachsender Instanzengröße. Deswegen bleibt nur der Schluß übrig, dass beide Verfahren eine vergleichbare Leistung ausfweisen. Dies bleibt deutlich hin-

ter der Erwartungshaltung nach den guten Ergebnissen der Lin-Kernighan-Heuristik zurück. Die Vermutung war, dass die lokale Suche deutlich bessere Ergebnisse zu Tage fördert als die einfache Vergleichsheuristik.

In Abbildung 5.3 werden die Approximationsfaktoren im Vergleich zur Instanzgröße gezeigt. Links ist die einfache Vergleichsheuristik in rot abgebildet und rechts die lokale Suche in variabler Tiefe in blau. In einem nahezu identischen Kurvenverlauf nähern sich beide Verfahren ihren maximalen Approximationsfaktoren an.

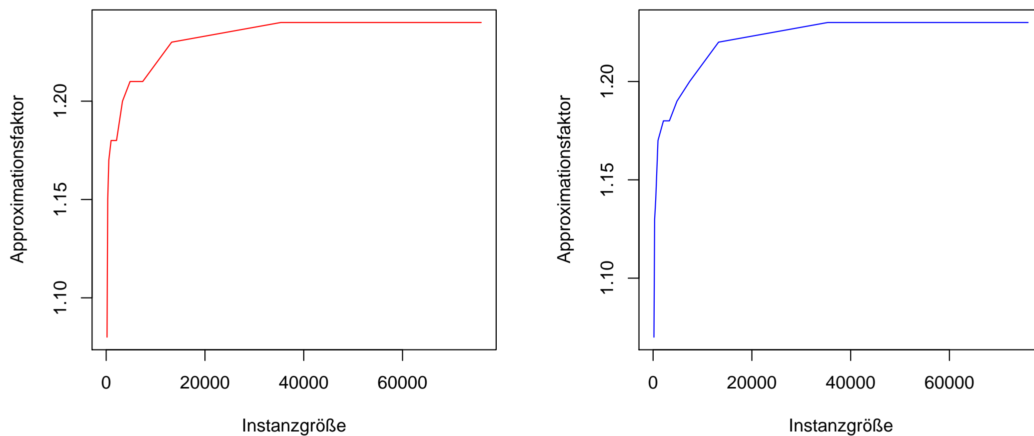

Abbildung 5.3.: Güte der Lösungen beider Verfahren. Links die einfache Verglichsheuristik in rot, rechts die lokale Suche in variabler Tiefe in blau

Diese Abbildung zeigt deutlich, dass das Verhalten der beiden Verfahren sehr ähnlich ist. Allein der minimale Unterschied in den Approximationsfaktoren ist unterschiedlich. Die Ergebnisse legen die Vermutung nahe, dass die Instanzen aus Motokis Generator schwierig zu approximieren sind. Immerhin gibt es ein bekanntes globales Optimum, dass deutlich nicht erreicht wird. Es scheint zudem naheliegend, dass die Optimierung von Motoki-Instanzen nicht trivial und nicht leichter als die Entscheidungsvariante des Problems 3SAT ist. Eine rigorose Analyse steht zwar derzeit noch aus, ist aber in Arbeit und laut Angaben des Autors auch kurz vor der Veröffentlichung [Mot07].

Die beiden Heuristiken sind aber nicht in allen Punkten austauschbar. Deutlich unterschiedlich sind die absoluten Laufzeiten beider Verfahren ausgeprägt. Während die einfache Vergleichsheuristik selbst große Instanzen in sehr kurzer Zeit bearbeitet, braucht die lokale Suche in variabler Tiefe ein Vielfaches dieser Zeit. Abbildung 5.4 zeigt Plots der beobachteten Laufzeiten. Im linken Plot ist die Laufzeit der einfachen Vergleichsheuristik aufgetragen und rechts die lokale Suche in variabler Tiefe. Zusätzlich ist zum

Vergleich im rechten Plot noch die Laufzeit der einfachen Vergleichsheuristik dargestellt.

Die Laufzeiten beider Verfahren besitzen mehr oder weniger lineares Wachstum. Im Vergleich zur lokalen Suche in variabler Tiefe wirkt sich das Wachstum der Laufzeit der einfachen Vergleichsheuristik aber fast konstant aus. Das heisst, dass die Konstante der einen Laufzeit verschwindend gering ist, wenn sie mit der der anderen Laufzeit verglichen wird.

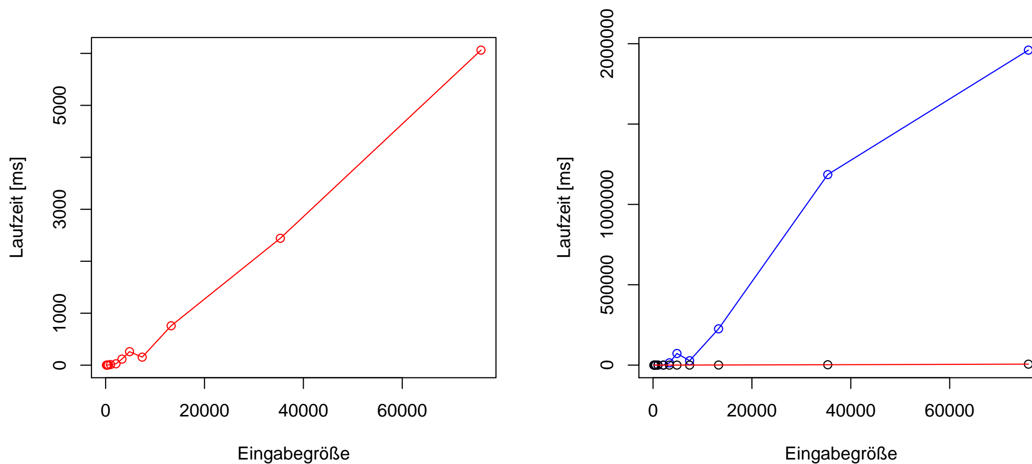

Abbildung 5.4.: Güte der MAX2SAT-Heuristiken. Einfache Vergleichsheuristik in rot und die lokale Suche in variabler Tiefe in blau

Die meiste Zeit verbringt die lokale Suche in variabler Tiefe also auf der mehr oder weniger fruchtlosen Jagd nach besseren Belegungen, wobei viel Zeit damit verbracht wird, den nächsten Flip greedy zu wählen. Die Iterationstiefe ist relativ gering. Selbst für die größte Instanz mit knapp 76.000 Knoten werden nur 4 Iterationen benötigt um zur gefundenen Lösung zu gelangen.

Interessant ist auch die Tatsache, dass eine nicht triviale Änderung der Nachbarschaft keine siginfikant verbesserten Ergebnisse bringt. In einem kompletten Simulationslauf wurde die betrachtete Nachbarschaft derart verändert, dass anstatt jedem möglichen Einzelflip nur diejenigen Variablen betrachtet wurden, die in Klauseln als unerfüllte Literale vorkommen. Nach der obigen Notation sind dies die Variablen, die in SAT_0 auftauchen und die in den unerfüllten Literalen in SAT_1. Zusätzlich wird ein einfaches Backtracking zugelassen.

Wird in einer Phase (einem Lauf der inneren Schleife) keine Verbesserung gefunden,

dann wird nicht abgebrochen, sondern diese Phase wiederholt und der erste Flip diesmal mit der zweitbesten Variable durchgeführt, usw. Die Heuristik bricht erst dann ab, wenn kein neuer Champion mehr gefunden wird und alle Variablen erfolglos als erster Flip ausprobiert wurden. Im Vergleich mit der Nachbarschaft aus der ersten lokalen Suche in variabler Tiefe ist diese veränderte Nachbarschaft nicht mit wirklich besseren gefundenen lokalen Optima versehen als die Nachbarschaft im ausführlichen getesteten Verfahren. Die beobachteten Ergebnisse lagen nur in einem Fall besser als die zuvor vorgeschlagene Heuristik. Selbst die beobachteten Laufzeiten waren bis auf minimale Differenzen identisch. Die größte Formel mit knapp 76.000 Klauseln konnte nur mit einer zusätzlich erfüllten Klausel mehr optimiert werden. Ein vernachlässigbarer Zugewinn.

5.8. Fazit

Die schlechten Ergebnisse der lokalen Suche in variabler Tiefe für MAX2SAT überaschen. Sie bleiben deutlich hinter den Erwartungen zurück, die die guten Ergebnisse der Lin-Kernighan Heuristik aufgebaut hatten. Die Erwartungshaltung war, dass die einfache Vergleichsheuristik deutlich schlechter optimiert. Es stellt sich aber heraus, dass beide Verfahren ebenbürtig sind. Von der lokalen Suche in variabler Tiefe sind keine um irgend eine bedeutende Größenordnung besseren Ergebnisse zu erwarten. Dies lässt vermuten, dass diejenigen Optimierungsstrategien, die die 1-Flip-Nachbarschaft als Basisschritt haben, eine grundlegende Gemeinsamkeit besitzen. Wie es scheint, liegen lokale Optima bzw. die Lösung der einfachen Vergleichsheuristik sehr nah nebeneinander. Eine Formalisierung dieses Sachverhalts und eine theoretische Analyse, wie nah diese vermutete „Nähe“ wirklich ist, sind noch offene Fragestellungen.

Die lokale Suche in variabler Tiefe bietet sich durch die greedy Wahl des nächsten Knotens aber nicht für eine tatsächliche praktische Anwendung an. Sie verschlingt viel Rechenzeit, die leider nicht in einem angemessen besseren Ergebnis mündet, bzw. in der Größenordnung schlicht einfacher zu haben ist.

6. Ausblick

Diese Arbeit untersuchte experimentell die Leistungsfähigkeit der lokalen Suche in variabler Tiefe. Obwohl sehr gute Ergebnisse für das Traveling Salesman Problem erzielt werden, kann diese Leistung nicht auf jedes andere beliebige schwere Problem übertragen werden. Für das Problem MAX2SAT zeigt die lokale Suche in variabler Tiefe nur Leistungen, die in der Größenordnung eines ganz primitiven Optimierungsschemas liegen.

Es stellt sich daher die Frage, aus welchen Gründen das Verfahren bei einem Problem sehr gut arbeitet und bei einem anderen nur mäßige Ergebnisse liefert. Auch stellt sich die Frage, bei welchen Problemen dies nun der Fall ist. Hier bieten sich neben weiteren experimentellen Analysen auch theoretische Arbeiten an, die diesen Sachverhalt untersuchen. Die experimentellen Analysen sind dabei ein erster Schritt, um schnell zu einer Vorstellung über die Güte der lokalen Suche in variabler Tiefe für ein spezielles Problem zu gelangen.

Speziell für MAX2SAT bietet sich an zu klären, wieviel die lokale Suche in variabler Tiefe denn wirklich besser sein kann als die einfache Vergleichsheuristik. Desweiteren ist von Interesse, ob sich das simple Vergleichsverfahren mit ähnlichem Erfolg auch in anderen Bereichen als dem Feld der Erfüllbarkeitsprobleme einsetzen lässt.

Für die einfache Vergleichsheuristik zeichnete sich in den Analysen ab, dass sie eine $(4/3 \pm \epsilon)$-Approximation zu sein scheint. Ein Beweis dieser Tatsache würde die Leistungsfähigkeit des Verfahrens noch weiter untermauern.

Mit Hilfe dieser Erkenntnisse wäre eine genauere Charakterisierung der Leistungfähigkeit der untersuchten Optimierungsstrategien möglich. Dies könnte zu leistungsfähigeren Heuristiken oder auch zu dem Wissen führen, dass für bestimmte Problemgruppen andere Ansätze in der Algorithmenentwicklung zielführender sind.

A. Simulationsergebnisse für das TSP

In diesem Abschnitt werden die gesammelten Rohdaten der mit der für diese Arbeit erstellten Implementierung ohne Auswertung wiedergegeben.

A.1. Ohne Backtracking

Instanz	Ergebnis	Zeit	Faktor	Instanz	Ergebnis	Zeit	Faktor
a280	2.710	70ms	1,05	kroA100	39.602	1.723ms	1,86
ali535	552.600	150.089ms	2,73	kroA150	77.713	5.630ms	2,95
berlin52	11.199	70ms	1,48	kroA200	61.449	14.632ms	2,09
bier127	168.843	1.918ms	1,42	kroB100	38.208	825ms	1,75
burma14	3.390	2ms	1.02	kroB150	63.416	1.582ms	2,42
ch130	11.352	1.421ms	1,85	krob200	62.601	7.952ms	2,42
ch150	10.551	4.689ms	1,60	kroC100	32.023	1.617ms	1,54
d198	19.379	17.498ms	1,22	lin105	25.141	705ms	1,78
d493	53.084	1.012.538ms	1,52	lin318	81.806	30.567ms	1,95
d657	84.958	368.077ms	1,73	nrw1379	120.960	15.647.987ms	2,13
d1291	81.083	–	1,59	p654	68.262	–	1,97
dsj1000	$7,1059 \cdot 10^7$	1.841.076ms	3,80	rat195	3.642	47.632ms	1,46
eil51	562	828ms	1,31	rd400	32.292	216.211ms	2,11
eil76	614	1.374ms	1,14	st70	1.024	1.829ms	1.50
eil101	911	1.307ms	1,44	tsp225	6.029	9.840ms	1.53
fl417	23.110	532s	1,94	u159	43.381	3.621ms	1,03
gr202	48.118	19.087ms	1,19	u574	40.197	–	1,08
gr229	179.819	7.144ms	1.33	u724	71.296	601.072ms	1,70
gr431	228.475	37.222ms	1.33	ulysses16	6.875	3ms	1,002
gr666	423.710	101s	1,43	ulysses22	8,427	16ms	1,20

A.2. Mit Backtracking

Instanz	Ergebnis	Zeit	Faktor	Instanz	Ergebnis	Zeit	Faktor
a280	2.710	69ms	1,05	kroA100	22.612	1.516ms	1,06
ali535	229.145	201.073ms	1,13	kroA150	27.589	10.036ms	1,04
berlin52	8.215	72ms	1,08	kroA200	32.082	9.890ms	1,09
bier127	1275.239	3.009ms	1,05	kroB100	23.748	1.846ms	1,07
burma14	3323	2ms	1,00	kroB150	27.922	5.884ms	1,06
ch130	6.726	1.387ms	1,10	kroB200	33.038	19.526ms	1,12
ch150	6.912	1.912ms	1,05	kroC100	23.990	2.313ms	1,15
d198	16.727	6.029ms	1,06	lin105	15.332	1.950ms	1,06
d493	36.966	391ms	1,05	lin318	45.622	32.681ms	1,08
d6575	2.261	446.044ms	1,06	nrw11379	60.559	$24,7 \cdot 10^6$ms	1,08
d1291	54.997	$21,1 \cdot 10^6$ms	1,08	p654	36.226	3.313ms	1,04
dsj1000	$2,03 \cdot 10^7$	4.625.022ms	1,08	rat195	2.442	20.845ms	1,05
eil51	438	280ms	1,02	rd400	16.716	377.022ms	1,09
eil76	550	494ms	1,02	st70	741	254ms	1,09
eil101	652	1.070ms	1,03	tsp225	4.056	37.424ms	1,03
fl417	12.468	596.021ms	1,05	u159	44.737	3.502ms	1,06
gr202	42.674	4.117ms	1,06	u574	39.572	314.529ms	1,07
gr229	142.050	22.777ms	1,05	u724	45.205	$21,4 \cdot 10^6$ms	1,07
gr431	187.812	1.516ms	1,06	ulysses16	6.875	4ms	1,002
gr666	321.841	1.346.098ms	1,09	ulysses22	7.114	21ms	1,01

A.3. Varianten mit abweichender Kantenfixierung

A.3.1. Ohne Fixieren entfernter Kanten

Instanz	Ergebnis	Zeit	Faktor	Instanz	Ergebnis	Zeit	Faktor
a280	2.761	38.545ms	1,07	kroA100	22.612	1.413ms	1,06
ali535	223.872	958.054ms	1,10	kroA150	27.589	11.905ms	1,04
berlin52	7.960	1.37ms	1,05	kroA200	32.179	7.769ms	1,09
bier127	128.665	4.235ms	1,08	kroB100	23.748	1.993ms	1,07
burma14	3.323	2ms	1,00	kroB150	27.627	7.093ms	1,05
ch130	6.402	4.222ms	1,04	kroB200	31.478	24.839ms	1,06
ch150	6.917	3.742ms	1,05	kroC100	23.990	2.910ms	1,15
d198	16.535	7.568ms	1,06	lin105	15.332	2.485ms	1,06
d493	36.708	527.098ms	1,04	lin318	45.031	57.137ms	1,07
d6575	52.282	2.536.633ms	1,06	nrw11379	60.654	$25,7 \cdot 10^7$ms	1,08
d1291	57.036	$29,1 \cdot 10^6$ms	1,10	p654	36.226	3.499ms	1,04
dsj1000	$2,003 \cdot 10^7$	82.625.022ms	1,07	rat195	2.442	3.499.975ms	1,05
eil51	438	117ms	1,02	rd400	16.413	341.995ms	1,08
eil76	550	595ms	1,02	st70	741	635ms	1,09
eil101	669	1.090ms	1,06	tsp225	4.056	60.234ms	1,03
fl417	12.468	743.863ms	1,05	u159	44.737	4.540ms	1,06
gr202	42.038	6.680ms	1,04	u574	39.572	519.546ms	1,07
gr229	144.568	18.851ms	1,07	u724	45.282	123.787ms	1,07
gr431	189.752	187.782ms	1,10	ulysses16	6.875	19ms	1,002
gr666	314.982	841.262ms	1,07	ulysses22	7.114	34ms	1,01

A.3.2. Ohne Fixieren hinzugefügter Kanten

Instanz	Ergebnis	Zeit	Faktor	Instanz	Ergebnis	Zeit	Faktor
a280	2.705	38.958ms	1,04	kroA100	21.901	1.368ms	1,02
ali535	219.648	$1.1{\cdot}10^7$ms	1,08	kroA150	28.073	5.317ms	1,05
berlin52	8.182	76ms	1,08	kroA200	31.312	17.003ms	1,06
bier127	124.715	2.914ms	1,05	kroB100	22.843	1.032ms	1,03
burma14	3.323	3ms	1,00	kroB150	27.239	3.566ms	1,04
ch130	6.579	4.536ms	1,07	kroB200	30.699	23.217ms	1,04
ch150	7.081	5.239ms	1,08	kroC100	22.152	698ms	1,06
d198	16.430	16.566ms	1,04	lin105	15.718	618ms	1,09
d493	36.708	423.952ms	1,04	lin318	45.559	118.239ms	1,08
d657	53.361	698.160ms	1,09	nrw11379	60.128	$1{,}6{\cdot}10^7$ms	1,06
d1291	56.031	$46{,}1{\cdot}10^6$ms	1,10	p654	35.788	$2{,}7{\cdot}10^7$ms	1,03
dsj1000	$2,02 \cdot 10^7$	$12{\cdot}10^6$ms	1,07	rat195	2.442	16.241ms	1,05
eil51	449	42ms	1,05	rd400	16.353	82.300ms	1,07
eil76	574	370ms	1,06	st70	719	306ms	1,06
eil101	672	889ms	1,06	tsp225	4.056	24.234ms	1,03
fl417	12.774	191.828ms	1,07	u159	44.737	2.390ms	1,06
gr202	42.674	3.244ms	1,0ms6	u574	39.242	254.618ms	1,06
gr229	142.050	17.717ms	1,05	u724	45.022	122.870ms	1,07
gr431	188.624	32.854ms	1,10	ulysses16	6.875	3ms	1,002
gr666	319.514	443.256ms	1,08	ulysses22	7.114	12ms	1,01

A.3.3. Ohne Fixieren hinzugefügter Kanten und mit randomisierter Starttour

Instanz	Ergebnis	Zeit	Faktor	Instanz	Ergebnis	Zeit	Faktor
a280	2.684	37.717ms	1,04	kroA100	22.116	1.401ms	1,03
ali535	220.360	124.973ms	1,08	kroA150	27.731	9.321ms	1,04
berlin52	7.718	200ms	1,02	kroA200	30.975	16.876ms	1,05
bier127	121.988	3.218ms	1,03	kroB100	22.924	1.096ms	1,03
burma14	3.323	4ms	1,00	kroB150	26.961	10.557ms	1,03
ch130	6.398	6.328ms	1,04	kroB200	31.499	30.287ms	1,07
ch150	6.942	5.538ms	1,06	kroC100	21.067	1.392ms	1,01
d198	10.332	15.839ms	1,03	lin105	15.257	2.396ms	1,06
d493	36.773	417.921ms	1,05	lin318	44.160	118.808ms	1,05
d657	53.138	436.013ms	1,07	nrw11379	60.654	16.877.105ms	1,04
d1291	55.204	$4.5 \cdot 10^6$ms	1,08	p654	35.681	$2.71 \cdot 10^7$ms	1,02
dsj1000	1.994.387	$12 \cdot 10^6$ms	1,06	rat195	2.441	16.995ms	1,05
eil51	436	110ms	1,02	rd400	16.284	84.000ms	1,06
eil76	562	603ms	1,04	st70	687	1.951ms	1,09
eil101	660	737ms	1,04	tsp225	4.091	25.056ms	1,04
fl417	12.479	208.031ms	1,05	u159	44.098	2.490ms	1,04
gr202	42.625	9.88ms	1,06	u574	39.927	60.300ms	1,08
gr229	141.952	19.812ms	1,05	u724	77.932	236.749ms	1,07
gr431	183.141	39.827ms	1,06	ulysses16	6.870	8ms	1,001
gr666	313.739	478.147ms	1,06	ulysses22	7.100	33ms	1,01

B. Implementierungsdetails

Dieser Abschnitt stellt die im Rahmen der Arbeit erstellten Anwendungen zur Simulation vor. Damit können die getätigten Experimente für das metrische Traveling Salesman Problem und für MAX2SAT wiederholt werden.

Die Programme sind auf der, dieser Arbeit beiliegenden, CD-ROM verfügbar. Sie wurden mit Java erstellt und sind so unter jedem gängigen Betriebssystem ausführbar, das eine Java Virtual Machine der Version 1.5 oder höher bereitstellt. Ist diese installiert, so sind die Anwendungen direkt startbar durch einen Doppelklick auf den Dateinamen. Zuerst wird die Anwendung `TSPBench` für das Traveling Salesman Problem und dann die Gruppe der Anwendungen für MAX2SAT vorgestellt.

B.1. Bedienung der TSPBench

Die Anwendung präsentiert direkt nach dem Start die graphische Benutzeroberfläche, wie in Abbildung B.1 dargestellt. Startet die Anwendung einwandfrei, dann zeigt sie dies in der Konsole mit der Meldung [`System`] `Ready` an. Metrische Instanzen der TSPLIB werden über das Menü unter dem Punkt *Datei->Öffnen* ausgewählt. Die Anwendung nach dem Einlesen der Instanz eine skalierte Ausgabe der Koordinaten der einzelnen Städte als rote Punkte.
Im Fenster *Optionen* kann die zu verwendende Heuristik gewählt werden und mit *OK* wird die Berechnung gestartet. Der Benutzer erhält über das Statusmenü detaillierte Informationen zur Instanz und dem aktuellen Optimierungslauf. Im einzelnen sind dies:

- Instanzname: Dies ist der vom Dateinamen unabhängige Name der Instanz.
- Kommentar: Jede TSPLIB Instanz besitzt ein besonderes Feld im Datensatz, das Herkunft und Zweck der Instanz erläutert.
- Aktuelle Lösung: Dieses Feld gibt die Länge der aktuellen Championtour an und fällt im Lauf einer Optimierung.
- Iterationsdauer: Gibt die Dauer der Berechnung aus, wobei nur Zeiten berücksichtigt werden, die über 1ms liegen.
- Weitere Informationen sind im Status- und im Konsolenfenster ersichtlich.

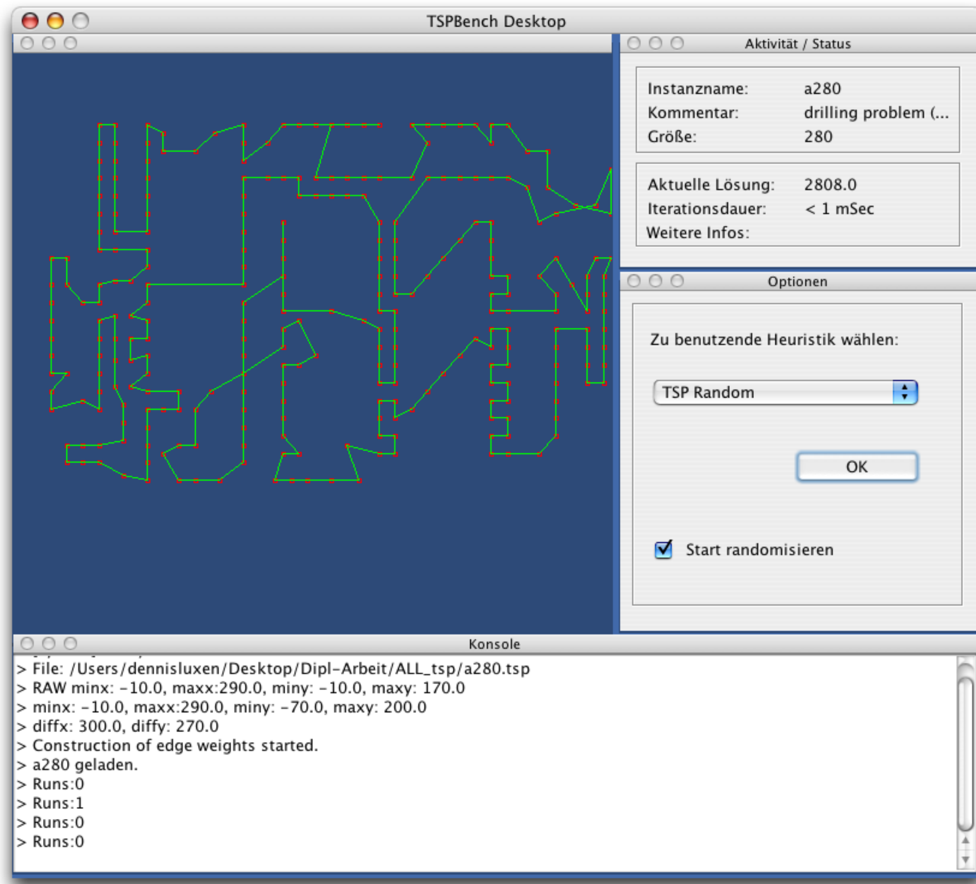

Abbildung B.1.: GUI der TSPBench

B.2. Erweiterung des TSPBench

Die Anwendung wurde von Beginn an nach dem Model-View-Controller Prinzip aufgebaut. Es ist daher verhältnismäßig einfach, sie um weitere Heuristiken zu erweitern. Jede Heuristik implementiert das Interface *util.Heuristic* als Model. Das Grundgerüst ist der Quellcode für die Berechnung der kanonischen Tour und bei jeder Heuristik gleich.

Der Aufbau ist gradlinig. Dem Kontruktor werden Objektreferenzen für den Eingabegraphen und die Ergebnisliste übergeben. Die Methode *setActor* speichert den globalen Controller und wird beim Instanzieren vom Controller selbst aufgerufen um eine Referenz auf sich im Objekt zu speichern.

Die Methode *run()* ist das Kernstück einer Heuristik. Wird diese aufgerufen, startet die Berechnung. Der Algorithmus merkt sich die Startzeit und legt dann eine Objektkopie der Eingabeinstanz an. Diese ist intern als eine Liste implementiert und speichert

die Knoten in der kanonischen Reihenfolge. Dieser Teil ist natürlich durch die jeweilige Implementierung zu ersetzen. Danach wird die Tour den Methoden zur Darstellung übergeben und die Methode *run()* endet.

```
package heuristics;

import util.*;
import application.Interactor;

public class TSPCanonicalTour implements Heuristic {
    private Tour myTour;
    private Interactor actor;
    private SimpleGraph orig;
    private List<Double> resultList;

    public TSPCanonicalTour(SimpleGraph g, List<Double> resultList){
        this.orig = g;
        this.resultList = resultList;
    }

    public void setActor(Interactor actor) {
        this.actor = actor;
    }

    public void run() {
        long startTime = System.currentTimeMillis();
        myTour = (Tour)orig.clone();

        myTour.setGraph(orig);
        actor.setTour(myTour);
        actor.setDuration(System.currentTimeMillis()-startTime);
        resultList.add(myTour.computeCost());
    }

}
```

Listing B.1: Heuristik Template

B.3. Bedienung der Anwendungsfamilie für MAX2SAT

Für MAX2SAT ist eine Gruppe von drei Anwendungen entstanden. Dies sind zwei Anwendungen für die implementierten Heuristiken und eine Implementierung von Moto-

kis Instanzgenerator. Die graphischen Benutzeroberflächen sind schlicht und gestatten die Konzentration auf das Wesentliche. Abbildung B.2 zeigt die Implementierung von Motokis Instanzgenerator und Abbildung B.3 stellvertretend für beide Heuristiken die Implementierung einer Heuristik. Die Unterschiede in den Oberflächen ist nur von kosmetischer Natur. Die Bedienung ist naheliegend. Nach Eingabe der Zahl der Variablen

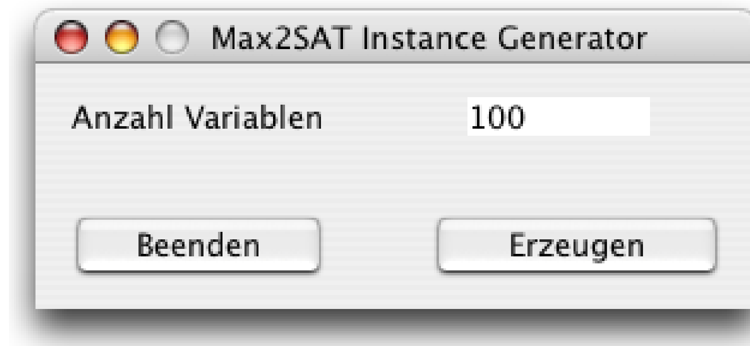

Abbildung B.2.: GUI der Implementierung von Motokis Generator

der Instanz berechnet die Anwendung automatisch alles weitere und fordert den Benutzer noch noch auf, einen Ort und Namen für die Speicherung der Formel anzugeben.

Ähnlich offensichtlich ist die Bedienung der Heuristikimplementierungen. Der Button *Laden* öffnet einen Dateibrowser zur Auswahl der Instanz. Im Feld Status wird angegeben, ob die Instanz nur geladen oder bereits gelöst wurde. Nach einem Klick auf *Lösen* startet die Berechnung. Die Angabe, wieviel Klauseln erfüllt werden, erscheint

Abbildung B.3.: GUI der Lösungsprogramme

sobald die Berechnung beendet wurde. Die Benutzerführung ist also einfach und klar gestaltet.

B.4. Erweiterung der MAX2SAT Heuristiken

Das Grundgerüst der implementierten Lösungsalgorithmen lässt sich wiederverwenden. Die Algorithmen wurden nach dem Model-View-Controller Prinzip aufgebaut. Kern jeder Implementierung ist eine Klasse mit dem Namen *Heuristic*. Ihr prinzipieller Aufbau lässt sich leicht anhand der einfachen Vergleichsheuristik erklären.

```
package instsolve;

import java.util.BitSet;

public class Heuristic {

    private BitSet solution;
    private Interactor ia;

    public Heuristic(int size, Interactor ia){
        solution = new BitSet(size);
        this.ia = ia;
    }

    public void solve(Formel f) {
        long laufzeit = System.currentTimeMillis();
        mw.getClauseArea().setText(""+f.goodNess(solution));

        for(int i=0;i<g.getVarSize;i++){
            int satisfied = f.goodNess(solution);
            solution.flip(i);
            if(satisfied>=f.goodNess(solution)){
                solution.flip(i);
            }
        }
        laufzeit = System.currentTimeMillis() - laufzeit;
        ia.setGoodness(""+f.goodNess(solution));
        System.out.println("Laufzeit: "+laufzeit+" ms");
    }
}
```

Listing B.2: Heuristik Template

Der Konstruktor erhält als Parameter die Instanzgröße und eine Referenz auf den Controller der Anwendung. Damit wird eine erste initiale Lösung erzeugt. Diese ist 0^n, denn ein BitSet in Java wird standardmäßig in jeder Komponente auf *false* gesetzt.

Der Aufruf der Methode *solve* startet die Berechnung mit der Eingabeinstanz als

Parameter. Als erstes wird die aktuelle Systemzeit gespeichert und die Anzeige der GUI auf die Güte der Startlösung initialisiert. In der folgenden for-Schleife wird nun für jede Variable berechnet, welche Belegung mehr Klauseln erfüllt und diese dann gewählt. Danach wird über den Interactor noch die Anzeige in der GUI aktualisiert und die Laufzeit in Milisekunden berechnet und auf die Konsole ausgegeben.

Die Methode *solve* ist von der jeweiligen Implementierung mit Leben zu füllen.

Abbildungsverzeichnis

2.1. Tour Konstruktion 8
2.2. Histogramm der 5000 Läufe auf der Instanz lin318 13
2.3. Histogramm der 5000 Läufe auf der Instanz berlin52 14
2.4. Arbeitsschritt einer Einfügeheuristik 22
2.5. Arbeitsschritt der Held-Karp-Schranke 27
2.6. Die Schätzung der Held-Karp-Schranke entspricht einer optimalen Tour 28

3.1. Beispielhafte Optimierungslandschaft 29
3.2. Wahl der Kanten im 2-Opt-Verfahren 32
3.3. Nichtsequentieller Tausch 37
3.4. Hamiltonpfad als Tour im TSP, nach [JM95] 39
3.5. Kantentausch mit Hamiltonpfad, nach [JM95] 40
3.6. Kantentausch für $i = 2$ 40
3.7. Kantentausch für $i = n$ 41
3.8. Allgemeiner Aufbau der PLS-Reduktion nach [Pap92] 58
3.9. Die Möglichkeiten, den ODER-Subgraphen zu traversieren, nach [Pap92] 59

4.1. Güte der Lösungen in der Simulation ohne Backtracking 66
4.2. Beobachtete Laufzeiten mit Vergleichsgrößen in der Simulation ohne Backtracking 67
4.3. Ausschnitt aus dem Plot zu einer ohne Backtracking gefundenen Lösung für die Probleminstanz *berlin52*. 68
4.4. Güte der Lösungen in der Simulation mit Backtracking 69
4.5. Wachstum der Laufzeit mit Vergleichsgrößen in der Simulation mit Backtracking 70
4.6. Wachstum der Laufzeit mit Vergleichsgrößen in der Simulation mit Backtracking und ohne Fixieren der entfernten Kanten. 71
4.7. Güte der Lösungen in der Simulation mit Backtracking und ohne Fixieren entfernter Kanten. 72
4.8. Wachstum der Laufzeit mit Vergleichsgrößen in der Simulation mit Backtracking und ohne Fixieren hinzugefügter Kanten. 73
4.9. Güte der Lösungen in der Simulation mit Backtracking und ohne Fixieren hinzugefügter Kanten. 74
4.10. Güte der Lösungen in der Simulation mit Backtracking mit randomisierter Starttour und ohne Fixieren hinzugefügter Kanten. 76

4.11. Wachstum der Laufzeit mit Vergleichsgrößen in der Simulation mit Backtracking und ohne Fixieren hinzugefügter Kanten. 77

5.1. Idealer Instanzgenerator 84
5.2. Minimaler Widerspruchskreis für $t = 1^n$ 86
5.3. Güte der Lösungen beider Verfahren. Links die einfache Verglichsheuristik in rot, rechts die lokale Suche in variabler Tiefe in blau 92
5.4. Güte der MAX2SAT-Heuristiken. Einfache Vergleichsheuristik in rot und die lokale Suche in variabler Tiefe in blau 93

B.1. GUI der TSPBench 104
B.2. GUI der Implementierung von Motokis Generator 106
B.3. GUI der Lösungsprogramme 106

Index

α-Nähe, 44
γ-Maß für Cluster, 48
k-Opt-Verfahren, 31
k-Optimalität, 31
1-Baum, 24

Algorithmen zur Tourkonstruktion, 11
Algorithmen zur Tourverbesserung, 12
Anzahl Touren im STSP, 7
Approximationsfaktor, 6

Champion Tour, 41
Christofides-Heuristik, 19
Cluster-Distanz, 51
Cluster-Distanz, Einfluß, 52
Convex Hull Algorithmus, 23

Disjunktheitskriterium, 37
Double-Tree Heuristik, 16

Effiziente Clusterkompensierung, 48, 50
Einfluss der Starttour, 75
Euklidische Metrik, 65
Euler-Graph, 17
Euler-Kreis, 17
Euler-Tour, 17
Eulerscher Multigraph, 17
Exakte Algorithmen, 9

Flaschenhalskante, 51

Gemischte Verfahren, 12
Graphpartitionierung, 33

Hamiltonpfad, 39

Idealer Instanzgenerator, 84

k-Flip Nachbarschaft, 80
Kandidatenliste, 44
Kandidatenlisten, 43
Kreisschlusskosten, 49

Lin-Kernighan-Heuristik, vereinfacht, 35
Lokale Suche, 30

Manhattan Metrik, 64
Matching, perfektes, 19
MAX2SAT, 79
MAX2SAT Datenformat, 83
MAX2SAT-Heuristik, 81, 89
Maxnorm Metrik, 65
Meta-Heuristiken, 11
Minimum-Spanning-Tree Heuristik, 16
Multigraph, 17

Nachbarschaft für MAX2SAT, 80
Nachbarschaft in der lokalen Suche, 30
Nearest Neighbor, 12

Optimierungslandschaft, 29

PLS, 53
PLS-Reduktion, 55
PLS-Vollständigkeit, 55
polynomial-time local search, 53
Positivverbesserungskriterium, 37

Sequentielles Tauschkriterium, 36
Startheuristik, 12

Testinstanzen nach Motoki, 85
Traveling Salesman Problem, 5
TSPLIB, 12, 64

Zulässigkeitskriterium, 37

Literaturverzeichnis

[ABCC07] Applegate, David L. ; Bixby, Robert E. ; Chvatal, Vasek ; Cook, William J.: *The Traveling Salesman Problem.* Princeton University Press, 2007

[ACG$^+$02] Ausiello, G. ; Crescenzi, P. ; Gambosi, G. ; Kann, V. ; Marchetti-Spaccamela, A. ; Protasi, M.: *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties.* Springer Verlag, 2002

[AL97] Aarts, Emile ; Lenstra, Jan K.: *Local Search in Combinatorial Optimization.* John Wiley and Sons, 1997

[BM71] Bellmore, Mandell ; Malone, John C.: Erratum: Pathology of Traveling-Salesman Subtour-Elimination Algorithms. In: *Operations Research* 19 (1971), nov, Nr. 7, S. 1766. – ISSN 0030–364X

[BN68] Bellmore, M. ; Nemhauser, G. L.: The Traveling Salesman Problem: A Survey. In: *Operations Research* 16 (1968), may, Nr. 3, S. 538–558. – ISSN 0030–364X

[Chr76] Christofides, N.: Worst-case analysis of a new heuristic for the traveling salesman problem. In: *Technical Report 388, Carnegie Mellon University, Graduate School of Industrial Administration* (1976)

[CKT94] Chandra ; Karloff ; Tovey: New Results on the Old k-Opt Algorithm for the TSP. In: *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1994

[Coo71] Cook, Stephen A.: The complexity of theorem-proving procedures. In: *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing.* New York, NY, USA : ACM Press, 1971, S. 151–158

[CR99] Cook, W. ; Rohe, A.: Computing minimum-weight perfect matchings. In: *INFORMS Journal on Computing* 11 (1999), 138–148. `citeseer.ist.psu.edu/cook98computing.html`

[FG95] Feige, Uriel ; Goemans, Michel X.: Approximating the Value of Two Prover Proof Systems, with Applications to MAX2SAT and MAXDICUT. In: *Proceedings of the Third Israel Symposium on Theory of Computing and Systems*, 1995, 182–189

[Fla98] Flanagan, David: *Java in an Nutshell. Deutsche Ausgabe fuer Java 1.1*. second. Cambridge, Koeln, Paris : O'Reilly, 1998. – ISBN 3–89721–100–9

[GG81] Gabber, O. ; Galil, Z.: Explicit constructions of linear-sized superconcentrators. In: *Journal of Computer and System Sciences* 22 (1981), S. 407–420

[GH91] Grötschel, Martin ; Holland, Olaf: Solution of large-scale symmetric travelling salesman problems. In: *Math. Program.* 51 (1991), Nr. 2, S. 141–202. – ISSN 0025–5610

[gnu07] *The R Project for Statistical Computing*. http://www.r-project.org. Version: 2007, Abruf: 10. Juli 2007

[Gol76] Golden, Bruce L.: A Statistical Approach to the TSP. In: *Operations Research Center Working Paper;OR 052-76* 7 (1976), April. http://hdl.handle.net/1721.1/5264

[Hel00] Helsgaun, K.: An effective implementation of the Lin-Kernighan traveling salesman heuristic. In: *European J. Oper. Res.* 126 (2000), Nr. 1, 106–130. citeseer.ist.psu.edu/helsgaun00effective.html

[Heu03] Heun, Volker: *Grundlegende Algorithmen*. Vieweg Verlag, 2003

[HK70] Held, M. ; Karp, R. M.: The travelling salesman problem and minimum spanning trees. In: *Operations Research* 18 (1970), S. 1138–1162

[JM95] Johnson, David S. ; McGeoch, Lyle A.: *The Traveling Salesman Problem: A Case Study in Local Optimization*. Version: 1995. http://www.research.att.com/~dsj/papers/TSPchapter.pdf, Abruf: 31.01.2007

[JMR96] Johnson ; McGeoch ; Rothberg: Asymptotic Experimental Analysis for the Held-Karp Traveling Salesman Bound. In: *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1996

[JPY88] Johnson, David S. ; Papadimtriou, Christos H. ; Yannakakis, Mihalis: How easy is local search? In: *J. Comput. Syst. Sci.* 37 (1988), Nr. 1, S. 79–100. http://dx.doi.org/http://dx.doi.org/10.1016/0022-0000(88)90046-3. – DOI http://dx.doi.org/10.1016/0022–0000(88)90046–3. – ISSN 0022–0000

[KL70] Kernighan, B. W. ; Lin, S.: An Efficient Heuristic Procedure for Partitioning Graphs. In: *Bell Sys. Tech. J.* 49 (1970), Nr. 2, S. 291–308

[KT06] Kleinberg ; Tardos: *Algorithm Design*. Pearson Education Addison Wesley, 2006

[LK73] LIN, S. ; KERNIGHAN, B. W.: An Effective Heuristic Algorithm for the Traveling-Salesman Problem. In: *Operations Research* 21 (1973), mar, Nr. 2, S. 498–516. – ISSN 0030–364X

[LLRKS85] LAWLER, E.L. ; LCNSTRA, J.K. ; RINOOY KBAN, A.H.G. ; SHMOYS, D.B.: *The Traveling Salesman Problem.* J.Wiley & Sons Ltd, 1985

[Mot05] MOTOKI, Mitsuo: Test Instance Generation for MAX 2SAT. In: *CP*, 2005, S. 787–791

[Mot07] MOTOKI, Mitsuo: *Persönliche Konversation.* Email, Mai 2007

[Net99] NETO, David M.: *Efficient cluster compensation for lin-kernighan heuristics*, University of Toronto, Diss., 1999. – Adviser-Derek Corneil

[OM84] ONG, H. L. ; MOORE, J. B.: Worst-case analysis of two travelling salesman heuristics. In: *Operations Research Letters* 2 (1984), S. 273–277

[OR03] OSTERMAN, Colin ; REGO, César: The Satellite List and New Data Structures for Symmetric Traveling Salesman Problems / Hearin Center for Enterprise Science, University of Mississippi. 2003. – Forschungsbericht

[Pap92] PAPADIMITRIOU, Christos H.: The Complexity of the Lin-Kernighan Heuristic for the Traveling Salesman Problem. In: *SIAM Journal on Computing* 21 (1992), Nr. 3, 450-465. `http://dx.doi.org/10.1137/0221030`. – DOI 10.1137/0221030

[Pap94] PAPADIMITRIOU, Christos H.: *Computational Complexity.* Addison-Wesley, 1994

[Per94] PERTTUNEN, Jukka: On the Significance of the Initial Solution in Travelling Salesman Heuristics. In: *The Journal of the Operational Research Society* 45 (1994), oct, Nr. 10, 1131–1140. `http://links.jstor.org/sici?sici=0160-5682%28199410%2945%3A10%3C1131%3AOTSOTI%3E2.0.CO%3B2-2`. – ISSN 0160–5682

[PR87] PADBERG, M. W. ; RINALDI, G.: Optimization of a 532-city symmetric traveling salesman problem by branch and cut. In: *Operations Research Letters* 6 (1987), S. 1–7

[PR91] PADBERG, MANFRED ; RINALDI, GIOVANNI: A Branch-and-Cut Algorithm for the Resolution of Large-Scale Symmetric Traveling Salesman Problems. In: *SIAM Review* 33 (1991), mar, Nr. 1, 60–100. `http://links.jstor.org/sici?sici=0036-1445%28199103%2933%3A1%3C60%3AABAFTR%3E2.0.CO%3B2-T`. – ISSN 0036–1445

[PS82] PAPADIMITRIOU, Christos H. ; STEIGLITZ, Kenneth: *Combinatorial optimization: algorithms and complexity.* Upper Saddle River, NJ, USA : Prentice-Hall, Inc., 1982. – ISBN 0–13–152462–3

[Rei91] Reinelt, Gerhard: TSPLIB - A traveling salesman problem library. In: *INFORMS J. Comput.* 3 (1991), Nr. 4, 376-384. `http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/`, Abruf: 10. Juli 2007

[Rei94] Reinelt, G.: The Traveling Salesman: Computational Solutions for TSP Applications. In: *Lecture Notes in Computer Science* 840 (1994)

[SG76] Sahni, Sartaj ; Gonzalez, Teofilo: P-Complete Approximation Problems. In: *J. ACM* 23 (1976), Nr. 3, S. 555–565. `http://dx.doi.org/10.1145/321958.321975`. – DOI 10.1145/321958.321975. – ISSN 0004–5411

[SSFS02] Sanvicente-Sanchez, Hector ; Frausto-Solas, Juan: MPSA: A Methodology to Parallelize Simulated Annealing and Its Application to the Traveling Salesman Problem. In: *MICAI 2002: Advances in Artificial Intelligence : Second Mexican International Conference on Artificial Intelligence Merida, Yucatan, Mexico, April 22-26, 2002. Proceedings* (2002), 145–158. `http://www.springerlink.com/content/urk3gbwvumub0c7c`

[SY91] Schäffer, Alejandro A. ; Yannakakis, Mihalis: Simple local search problems that are hard to solve. In: *SIAM J. Comput.* 20 (1991), Nr. 1, S. 56–87. `http://dx.doi.org/http://dx.doi.org/10.1137/0220004`. – DOI http://dx.doi.org/10.1137/0220004. – ISSN 0097–5397

[Weg03] Wegener, Ingo: *Komplexitätstheorie*. Springer Verlag, 2003

[WF01] Walsh, A. ; Fonckowiak, J.: *Java*. MIT press, 2001

[Yam03] Yamamoto, Masaki: Instance generating algorithms for MAX2SAT with optimal solutions / Tokyo Institute of Technology, Dept. of Mathematical and Computing Sciences. Version: June 2003. `citeseer.ist.psu.edu/598178.html`. Meguro-ku Ookayama, Tokyo 152-8552, Japan, June 2003 (C-177). – Forschungsbericht