

Lösung zur Aufgabe DOPI
Informaticup 2008

DOPI-Löser

Manuel Holtgrewe, Daniel Karch, Jochen Seidel

Januar 2008

Zusammenfassung

Wir bearbeiten beim Informaticup 2008 die Aufgabe „DOPI“.

Eine zentrale Erkenntnis für die Lösung der Aufgaben ist eine Reduktion von DOPI auf das Problem, in einem vollständigen Graphen einen minimalen Hamilton-Pfad zu finden.

Für eine exakte Lösung schlagen wir einen Branch-and-Bound Ansatz vor. Zudem beschreiben wir, wie man das Problem mit Integer Linear Programming lösen kann.

Die erste Heuristik ist ein genetischer Algorithmus. Die zweite Heuristik basiert auf der Lin-Kernighan Heuristik für TSP, für größere Robustheit sorgt die Metaheuristik Iterierte Lokale Suche.

Zudem schrieben wir eine graphische Benutzeroberfläche für Mac Os X.

Inhaltsverzeichnis

1 Einführung	5
1.1 Grundlegendes	5
2 Anmerkungen zu DOP und DOPI	5
2.1 Definition der Probleme	5
2.2 Eigenschaften der Hamming-Distanz	6
2.3 Reduktion auf das gewichtete Hamilton-Pfad Problem	6
2.4 Symmetriebetrachtungen	7
2.5 Obere und untere Schranken	8
3 Konstruktive Heuristiken	8
3.1 Zufällige Lösung	8
3.2 Die Greedy Heuristik	9
4 Exakte Lösungen	9
4.1 Branch-and-Bound Algorithmus	9
4.2 Formulierung als ILP	12
5 Iterierte Lokale Suche	13
5.1 Auf den Schultern von Giganten — Ähnlichkeit zu TSP	13
5.2 Lokale Suche	14
5.3 k -opt Züge	15
5.4 Der Lin-Kernigham Algorithmus für WHP	16
5.5 Effiziente Repräsentation von Permutationen	17
5.5.1 Repräsentation als Array	17
5.5.2 Repräsentation als Segment-Baum	18
5.6 Eine Metaheuristik zur Verbesserung der Robustheit	19
5.7 Möglichkeiten zur Parallelisierung	19
6 Genetischer Algorithmus	20
6.1 Ablauf des Algorithmus	20
6.2 Parameter und Möglichkeiten zur Optimierung	21
6.3 Möglichkeiten zur Parallelisierung	22
7 Implementierung und Entwurfsentscheidungen	23
7.1 Historie zur Heuristik-Auswahl	23
7.2 Struktur der Implementierung	23
7.3 Vorgehen bei Implementierung und Tuning	25
7.4 Zusätzliche Werkzeuge	25
8 Die graphische Benutzeroberfläche	25

9	Anleitung zum Benutzen und Installieren der Programme	26
9.1	Branch-And-Bound	26
9.2	Integer Linear Program Generator	26
9.3	Iterierte Lokale Suche	27
9.4	Genetischer Algorithmus	28
9.5	Graphische Benutzeroberfläche	28
10	Ausblick	29
A	Kompilieren unter Ubuntu Linux	30
B	Kompilieren unter Mac Os X	30

1 Einführung

Zum diesjährigen Informatocup betrachten wir das *Data Ordering Problem with Inversion*, kurz *DOPI*.

Diese Ausarbeitung ist wie folgt aufgebaut:

In Abschnitt 2 beschreiben wir ein paar theoretische Erkenntnisse, die später das Schreiben von Lösern einfacher machen.

Abschnitt 3 erklärt die konstruktiven Heuristiken, die als Ausgangspunkt für die verbessernden Heuristiken in Abschnitt 6 und Abschnitt 5 benutzt werden. Abschnitt 4 beschreibt unsere beiden Ansätze für exakte Löser. Unsere Implementierungs- und Entwurfsentscheidung dokumentieren wir in Abschnitt 7.

Eine Anleitung zum Benutzen und Installieren der Programme findet sich in Abschnitt 9. Die graphische Benutzeroberfläche ist in Abschnitt 8 beschrieben, Abschnitt 10 gibt einen Ausblick auf noch offene Probleme.

1.1 Grundlegendes

Wir schreiben eine *Reihung* von k Objekten x_0, \dots, x_k als $\langle x_0, \dots, x_k \rangle$.

Ein *Graph* G ist ein Paar (V, E) aus Knotenmenge V und Kantenmenge $E \subseteq V \times V$. Wir betrachten ungerichtete Graphen und identifizieren die zwei Kanten (u, v) und (v, u) , $u, v \in V$.

In einem Graphen G ist ein *Pfad* P der Länge k eine Reihung von Knoten $\langle v_0, \dots, v_k \rangle$ (mit $v_i \in V$). Knoten können in einem Pfad doppelt vorkommen. Alternativ kann ein Pfad auch durch Kanten repräsentiert werden:

$$P = \langle v_0, \dots, v_k \rangle = \langle (v_0, v_1), \dots, (v_{k-1}, v_k) \rangle.$$

Die XOR-Verknüpfung von zwei Binärwörtern x und y bezeichnen wir mit $x \oplus y$.

2 Anmerkungen zu DOP und DOPI

2.1 Definition der Probleme

Zur Erinnerung noch einmal die Definition von DOP und DOPI.

Definition 2.1 (Data Ordering Problem, DOP): Das DOP ist wie folgt definiert: Es sind n Binärwörter mit jeweils k Bits über einen Bus zu übertragen. Die Wörter können in beliebiger Reihenfolge gesendet werden. Die Zahl der Transitionen zwischen zwei Wörtern w_1 und w_2 ist ihre Hamming-Distanz¹ $h(w_1, w_2)$. Gesucht ist eine Permutation der Wörter, so dass die Summe der Transitionen von aufeinander folgenden Wörtern minimal wird.

Eine DOP Instanz \mathcal{I} ist eine Folge von Binärwörtern $\langle w_0, \dots, w_{n-1} \rangle$. ◇

¹Die Hamming-Distanz ist die Zahl der nicht übereinstimmenden Zeichen.

Definition 2.2 (Data Ordering Problem with Inversion, DOPI): Das DOPI ist eine Erweiterung des DOP: Zusätzlich zu dem Bus gibt es noch eine Leitung, die angibt ob ein Wort invertiert oder nicht übertragen wird. Wechsel im Invertierungsbit zählen bei der Zahl der Transitionen mit. \diamond

Tatsache 2.3: DOP und DOPI sind \mathcal{NP} -vollständig, siehe etwa [13]. \diamond

2.2 Eigenschaften der Hamming-Distanz

Tatsache 2.4: Seien w_1 und w_2 Binärwörter, zu einem Binärwort w sei \bar{w} das invertierte Binärwort. Es sei $h : \{0,1\}^n \times \{0,1\}^n \rightarrow \mathbb{N}_0$ die Hamming-Distanz. Durch einfaches Nachrechnen sieht man:

$$h(w_1, w_2) = h(\bar{w}_1, \bar{w}_2) \quad (1)$$

$$h(w_1, \bar{w}_2) = h(\bar{w}_1, w_2). \quad (2)$$

Definition 2.5: Wir erweitern die Hamming-Distanz auf mehrere Wörter:

$$h(w_1, \dots, w_n) = h(w_1, \dots, w_{n-1}) + h(w_{n-1}, w_n). \quad (3)$$

Für eine Folge $W = \langle w_1, \dots, w_n \rangle$ von Wörtern ist $h(W)$ also die Anzahl der Transitionen zwischen den Wörtern. \diamond

Tatsache 2.6: Gilt $h(w_i, w_{i+1}) > h(w_i, \bar{w}_{i+1})$, so gilt wegen (2) auch

$$h(w_1, \dots, w_i, w_{i+1}, \dots, w_n) > h(w_1, \dots, w_i, \bar{w}_{i+1}, \dots, \bar{w}_n). \quad (4)$$

Tatsache 2.7: Für die Hamming-Distanz gilt die Dreiecksungleichung. Für alle Binärworte x, y und z gilt:

$$h(x, y) + h(y, z) \geq h(x, z)$$

2.3 Reduktion auf das gewichtete Hamilton-Pfad Problem

Korollar 2.8: Es sei $\mathcal{I} = \langle w_0, \dots, w_{n-1} \rangle$ eine Folge von Binärworten. Aus (1) und (2) ergibt sich aus \mathcal{I} eine Folge der optimalen Invertierungen.

Für das erste Wort entscheidet man sich beliebig zwischen „invertiert“ und „nicht invertiert“. Ist für die Wörter w_0 bis w_i der Invertierungsstatus festgelegt dann ergibt sich der Invertierungsstatus für $i + 1$ durch die lokal beste Wahl:

Sei x das Wort w_i mit optimalem Invertierungsstatus. Dann wird w_{i+1} genau dann invertiert wenn $h(x, \bar{w}_{i+1}) < h(x, w_{i+1})$.

Also kann für zwei benachbarte Wörter durch Festlegen des Invertierungsbits des zweiten Wortes immer die kleinste Zahl der Transitionen erzwungen werden. \diamond

Damit kann DOPI auf das Problem zurückgeführt werden, im Graphen G mit Gewichtsfunktion w einen Hamilton-Pfad minimalen Gewichtes zu finden.

Definition 2.9 (Gewichteter Hamilton-Pfad, WHP): Es sei V eine Knotenmenge und $G = (V, E)$ ein vollständiger Graph. Also ist $E = V \times V$. Es sei weiter $w : V \times V \rightarrow \mathbb{N}$ die Abstandsfunktion. Weiter fordern wir, dass $w(a, b) = w(b, a)$, also dass die Abstandsfunktion symmetrisch ist und die Dreiecksungleichung erfüllt.

Ein *Gewichteter Hamilton-Pfad* ist nun ein Pfad durch den Graphen, der alle Knoten genau einmal besucht. Das Gewicht des Pfades ist die Summe der Gewichte aller benutzten Kanten.

Eine DOPI-Instanz $\mathcal{I} = (G, w)$ ist ein Graph mit einer Gewichtsfunktion für die Kanten. \diamond

Anmerkung 2.10: In einem vollständigen Graphen ist jede Permutation der Knoten ein gültiger Hamilton-Pfad. \diamond

Satz 2.11 (Reduktion von DOPI auf WHP): Es sei $\mathcal{I} = \langle w_0, \dots, w_{n-1} \rangle$ eine DOPI-Instanz. Betrachte WHP-Instanz $\mathcal{J} = (G, w)$ mit

$$\begin{aligned} V &= \{0, \dots, n-1\} \\ E &= V \times V \\ w(i, j) &= \min \{h(w_i, w_j), h(\overline{w}_i, w_j)\}. \end{aligned}$$

Nun gilt: Eine optimale Lösung \mathcal{S} der WHP-Instanz \mathcal{I} (eine Permutation) impliziert eine Lösung der DOPI-Instanz \mathcal{J} nach der Konstruktion aus Korollar 2.8. \diamond

Im Wesentlichen ist diese Reduktion in [13] schon beschrieben. Allerdings wird sie dort nicht benutzt, um eine starke Heuristik für WHP zu finden. Stattdessen wird analog zum TSP eine Approximation mit asymptotisch konstantem Faktor zu finden.

Anmerkung 2.12: Offensichtlich kann die Konstruktion in Satz 2.11 in Zeit $\mathcal{O}(n^2)$ durchgeführt werden. Dies ist optimal Schranke, da w Platz $\Theta(n^2)$ benötigt und berechnet werden muss. \diamond

2.4 Symmetriebetrachtungen

Für Hamilton-Pfade in gewichteten Graphen sieht man folgende einfache Eigenschaften schnell ein.

Tatsache 2.13: Das Gewicht einer Lösung ändert sich durch Spiegeln nicht. \diamond

Tatsache 2.14: Tatsache 2.13 gilt auch für Teile der Lösung.
Betrachte den Hamilton-Pfad

$$p = \langle i_0, \dots, i_j, i_{j+1}, \dots, i_k, i_{k+1}, \dots, i_{n-1} \rangle.$$

Spiegelt man den Teilpfad von $j+1$ bis j dann ändert sich das Gewicht der Teilpfade $\langle i_1, \dots, i_j \rangle$, $\langle i_{j+1}, \dots, i_k \rangle$ und $\langle i_{k+1}, \dots, i_{n-1} \rangle$ nicht. Unterschiede gibt es nur an den Schnittpunkten zwischen j und $j+1$ sowie zwischen k und $k+1$. \diamond

2.5 Obere und untere Schranken

Computers are useless. They can only give you answers.
— Pablo Picasso

Aus Anmerkung 2.10 folgt, dass es trivial ist, eine obere Schranke für das Gewicht eines WHP zu finden: Jeder mögliche Pfad p durch alle Knoten ist Lösung und damit sein Gewicht $w(p)$ auch eine obere Schranke.

Interessanter ist es, untere Schranken zu betrachten. Vor allem, um die Qualität einer Lösung abzuschätzen. In [10] und [13] wird wie folgt eine untere Schranke gewonnen:

Lemma 2.15 (Untere Schranke für WHP): Betrachte den minimalen spannenden Baum (MST) von G . Das Gewicht dieses MST ist dann eine untere Schranke für das Gewicht des Hamilton-Pfades. \diamond

Beweis (von Lemma 2.15): Sei T ein MST von G mit Gewichtsfunktion w .

Angenommen es gibt einen Hamilton-Pfad P , der ein echt kleineres Gewicht hat als T . Dann ist P auch ein spannender Baum im Widerspruch zur Annahme. \blacksquare

Tatsache 2.16 (Eine weniger scharfe untere Schranke): Wähle für alle Knoten die lokal minimalen Kanten (also für den Knoten u die Kante $\{u, v\}$, mit dem kleinsten Gewicht). Die Menge der lokal minimalen Kanten sei M . Dann ist $W = \sum_{e \in M} w(e)$ eine untere Schranke für das Gewicht des Hamilton-Pfades.

Dies ist offensichtlich der Fall, da W eine untere Schranke für das Gewicht eines MST von G ist. \diamond

Tatsache 2.17 (Untere Schranke durch Lösen des LPs): In Abschnitt 4.2 geben wir eine ILP-Formulierung zur exakten Lösung des Problems an. Die Lösung des zugrundeliegenden linearen Programms (ohne Ganzzahligkeitsbedingung) stellt dann eine untere Schranke für WHP und somit auch für DOPI dar. \diamond

3 Konstruktive Heuristiken

3.1 Zufällige Lösung

*Anyone who considers arithmetical methods
of producing random numbers is, of course,
in a state of sin.*
— Johann von Neumann

Die einfachste Heuristik, um eine Startlösung für andere Algorithmen zu erzeugen ist es, eine zufällige Lösung zu erzeugen. Eine Folge von n Worten kann in $\mathcal{O}(n)$ zufällig permutiert werden (siehe etwa [8]).

3.2 Die Greedy Heuristik

faciente cupidine vires²
— Ovid

Eine einfache Heuristik nach dem Greedy-Verfahren arbeitet so:

1. Beginne mit einer leeren Wort-Folge.
2. Nehme ein zufälliges Wort hinzu.
3. Wähle unter allen verbleibenden Worten, jeweils invertiert und nicht invertiert, jenes aus, das den kleinsten Abstand zum Ende oder Anfang der Folge hat. Gibt es mehrere dieser Wörter, dann ziehe gleichverteilt eines von ihnen. Füge es an den Anfang bzw. Ende der Folge an.
4. Wenn es noch weitere Wörter gibt, dann gehe zu Schritt 3.

Der Algorithmus läuft in $\mathcal{O}(n^2)$. Man kann den Algorithmus auch n mal laufen lassen und im zweiten Schritt jeweils das 1. bis n . Wort wählen. Es ergibt sich ein Algorithmus mit einer Laufzeit von $\mathcal{O}(n^3)$, der unter Umständen eine bessere Lösung produziert.

4 Exakte Lösungen

4.1 Branch-and-Bound Algorithmus

*If I find 10,000 ways something won't work, I haven't failed.
I am not discouraged, because every wrong attempt discarded is another step forward.*
— Thomas A. Edison

Der erste exakte Löser basiert auf dem *Branch-and-Bound* Verfahren:

Am Anfang wird eine obere Schranke w^* für das Gewicht eines Hamilton-Pfades ausgerechnet. Im einfachsten Fall ist dies das Gewicht eines beliebigen Pfades (etwa des Pfades $\langle 0, \dots, n-1 \rangle$). Zusätzlich speichern wir noch P^* , den bis jetzt besten Pfad.

Es wird eine erschöpfende Suche durchgeführt, in der das Gewicht jedes möglichen Hamilton-Pfades berechnet wird. Dabei wird systematisch für alle Indizes i von 0 bis $n-1$ der Wert des i -ten Knotens festgelegt.

Wir berechnen jeweils das Gewicht w des bis jetzt festgelegten Pfades bis zum i -ten Knoten. Außerdem wird eine untere Schranke ℓ für den Pfad durch die noch zu besuchenden Knoten ausgerechnet. Ist nun $w + \ell \geq u$, so kann für die bis hier festgelegten Knoten kein Pfad gefunden werden, dessen Gewicht kleiner ist als das des besten bekannten Pfades. Wird ein Pfad P mit n Knoten berechnet, und ist $w < w^*$ dann werden w^* und P^* mit w und P aktualisiert.

²Die Gier macht stark.

Algorithmus 4.1 (Branch-and-Bound Algorithmus)

Die Prozedur BRANCH-AND-BOUND führt lediglich ein paar Initialisierungen durch. Dann ruft sie die eigentliche Löser-Prozedur BRANCH-AND-BOUND-RECURSE auf. Der beste Pfad wird in P^* , sein Gewicht in w^* gespeichert.

BRANCH-AND-BOUND

- ▷ Initialisiere w^* und P^* .
- 1 $P^* \leftarrow \langle 0, \dots, n-1 \rangle$
- 2 $w^* \leftarrow$ Gewicht von P^*
 - ▷ Starte erschöpfende Suche.
- 3 BRANCH-AND-BOUND-RECURSE($\langle \rangle, 0, \{0, \dots, n-1\}, 0$)

BRANCH-AND-BOUND-RECURSE(P, i, M, w)

- ▷ P ist der aktuelle Pfad, i der aktuelle Index.
- ▷ M ist die Menge der noch nicht platzierten Knoten.
- ▷ w ist das momentane Gewicht von P .
- 1 **if** $i > 0$ ▷ Berechnen des neuen Gewichtes.
- 2 **then** $w \leftarrow w + d(P[i-1], P[i])$ ▷ $d(\cdot, \cdot)$ ist die Abstandsfunktion des Graphen.
- 3 **if** $w + \text{LOWER-BOUND}(M) \geq w^*$ ▷ „Bounding“.
- 4 **then return**
- 5 **if** $i = n - 1$ ▷ Wir haben einen vollständigen Hamilton-Pfad.
- 6 **then** $\{u\} \leftarrow M$
- 7 $P[i] \leftarrow u$
- 8 **if** $w < w^*$
- 9 **then** $w^* \leftarrow w$
- 10 $P^* \leftarrow P$
- 11 **return**
- 12 **for** $u \in M$ ▷ „Branching“.
- 13 **do** $P[i] \leftarrow u$
- 14 BRANCH-AND-BOUND-RECURSE($P, i + 1, M \setminus u, w$) ◇

Damit Algorithmus 4.1 vollständig beschrieben ist, müssen wir noch eine Implementierung von LOWER-BOUND(M) angeben. Dafür stehen generell die in Abschnitt 2.5 aufgeführten Schranken zur Verfügung.

Die Schranke über den MST ist schärfer als die, die nur lokal minimale Kanten benutzt. Allerdings ist es zu teuer, für jede benutzte untere Schranke den MST neu zu berechnen. Einen MST beim das Hinzufügen und Löschen von Knoten zu aktualisieren ist auch nicht schnell genug.

In unserer Implementierung verwenden wir daher die zweite Möglichkeit. Algorithmus 4.2 beschreibt die Prozeduren, die diese untere Schranke effizient verwalten.

Abbildung 1 zeigt ein Beispiel, wie sich die Untere Schranke $w + \ell$ zusammensetzt. Links ist der bis jetzt schon gefundene Pfad eingezeichnet, sein Gewicht ist w . Rechts sind die lokal minimalen Kanten für M eingezeichnet. Diese formen einen Wald.

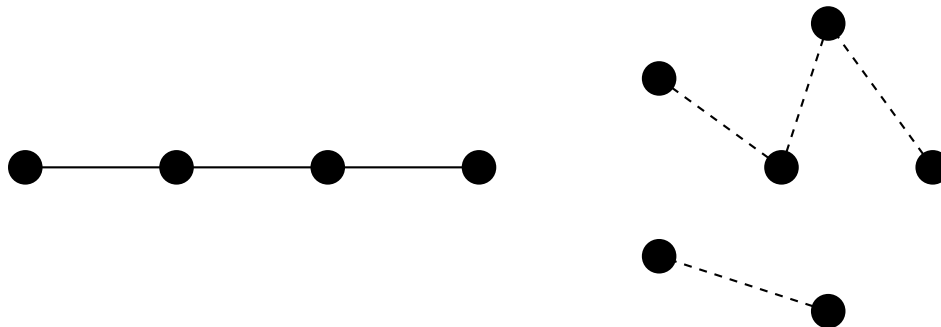


Abbildung 1: Beispiel für das Berechnen einer unteren Schranke. Die Kanten für das Gewicht w sind durchgezogen, die für das Gewicht ℓ sind gestrichelt.

Algorithmus 4.2 (Schnelle untere Schranken für Algorithmus 4.1)

Prozedur LOWER-BOUND-INIT initialisiert die benötigten Datenstrukturen. Wir identifizieren wie gehabt die Kanten (u, v) und (v, u) . Wir gehen davon aus, dass die Kanten durchnummeriert sind – etwa von 0 bis $\frac{n(n-1)}{2} - 1$.

LOWER-BOUND-INIT

- 1 $\ell \leftarrow 0$ ▷ Die untere Schranke
- 2 $timesUsed \leftarrow \langle 0, \dots, 0 \rangle$ ▷ n Einträge
- 3 $smallest \leftarrow$ Array, indiziert durch Kanten
- 4 **for** $u \in \{0, \dots, n-1\}$
- 5 **do** $smallest[u] \leftarrow$ lokale kleinste Kante des Knotens u
- 6 **for** $u \in \{0, \dots, n-1\}$
- 7 **do** LOWER-BOUND-ADD-NODE(u)

LOWER-BOUND-ADD-NODE(u)

- 1 $timesUsed[smallest[u]] \leftarrow timesUsed[smallest[u]] + 1$
- 2 **if** $timesUsed[smallest[u]] = 1$
- 3 **then** $\ell \leftarrow \ell + d(smallest[u])$

LOWER-BOUND-REMOVE-NODE(u)

- 1 $timesUsed[smallest[u]] \leftarrow timesUsed[smallest[u]] - 1$
- 2 **if** $timesUsed[smallest[u]] = 0$
- 3 **then** $\ell \leftarrow \ell - d(smallest[u])$ ◇

Die Prozedur BRANCH-AND-BOUND kann die Prozeduren aus Algorithmus 4.2 wie folgt verwenden: Am Anfang wird einmal LOWER-BOUND-INIT aufgerufen und die Datenstrukturen werden initialisiert. Vor jedem rekursiven Aufruf von BRANCH-AND-BOUND-RECURSE wird LOWER-BOUND-REMOVE-NODE(u) aufgerufen und nach dem Aufruf LOWER-BOUND-ADD-NODE(u).

Der Aufruf von LOWER-BOUND(u) in Zeile wird einfach durch das Auslesen von ℓ umgesetzt.

Anmerkung 4.3: Die erste obere Schranke P^* kann etwa mit dem genetischen Algorithmus oder der iterierten lokalen Suche gefunden werden. Unsere Implementierung verwendet ILS. \diamond

Anmerkung 4.4: Weiterhin kann zur Berechnung der unteren Schranke noch die kleinstmögliche Kante, die den schon festgelegten Pfad mit dem Restwald verbindet, hinzugezogen werden, falls sie nicht ohnehin schon lokal minimale Kante eines Waldknotens war. Dies ist im exakten Löser ebenfalls implementiert. \diamond

4.2 Formulierung als ILP

When in doubt, use brute force.
— Ken Thompson

Durch die Reduktion in Satz 2.11 und die Ähnlichkeit zum Problem des Handelsreisenden (TSP) fällt auch direkt eine weitere Möglichkeit ab, DOPI exakt zu lösen.

Das folgende *Integer Linear Programming* Problem erlaubt es, das TSP elegant zu formulieren (siehe etwa [12]). Soweit nicht anders angegeben sind i und j aus $\{0, \dots, n-1\}$.

$$\text{minimiere } \sum_{i,j} w(i,j)x_{i,j}$$

unter den Nebenbedingungen

$$\begin{aligned} x_{i,j} &\geq 0 && \forall i,j \\ x_{i,j} &\leq 1 && \forall i,j \\ \sum_{j \neq i} x_{i,j} &= 1 && \forall i \\ \sum_{j \neq i} x_{j,i} &= 1 && \forall i \\ u_i &\geq 0 && \forall i \\ u_i &\leq n-1 && \forall i \\ u_i - u_j + nx_{i,j} &\leq n-1 && \forall i \in \{0, \dots, n-1\}, j \in \{1, \dots, n-1\} \end{aligned}$$

Dabei ist in einer Lösung $x_{i,j}$ gleich 1 genau dann, wenn j auf i in der Permutation folgt.

Wir können daraus nun ein Integer Linear Program gewinnen, indem wir die Zielfunktion ersetzen durch:

$$\text{minimiere } \sum_{i,j} w(i,j)(x_{i,j} - y_{i,j})$$

Und die folgenden Nebenbedingungen hinzufügen:

$$\begin{aligned} y_{i,j} &\geq 0 && \forall i,j \\ y_{i,j} &\leq 1 && \forall i,j \\ \sum_{i,j} y_{i,j} &= 1 && \forall i,j \\ x_{i,j} - y_{i,j} &\geq 0 && \forall i,j \end{aligned}$$

Dies bewirkt, dass genau eine beliebige Kante aus der TSP-Tour entfernt wird. Durch die letzte Zeile wird ferner sichergestellt, dass diese Kante auch in der TSP-Tour war.

Mit dem kommerziellen ILP-Löser CPLEX konnten wir zufällige DOPI-Instanzen für $k = 1000$ bis etwa $n = 30$ in wenigen Sekunden lösen. Das freie ILP-Löser Paket GLPK konnte „nur“ zufällige DOPI-Instanzen mit $k = 1000$ bis etwa $n = 20$ „schnell“ lösen. Dies ist aber immer noch wesentlich schneller als der Branch-and-Bound Algorithmus.

Abschnitt 9.2 beschreibt die Benutzung für den exakten Löser, der auf dem Erzeugen und Lösen eines ILPs basiert. In die GUI haben wir diese Variante nicht integriert.

Anmerkung 4.5: Ist für das Problem eine obere oder untere Schranke bekannt, kann dies die Lösung des ILPs noch weiter beschleunigen. \diamond

5 Iterierte Lokale Suche

Lather, rinse and repeat.
— *Shampoo bottle inscription.*

In Abschnitt 5.1 beschreiben wir zunächst, wie wir die Ähnlichkeit des WHP zum Problem des Handelsreisenden ausnutzen: Wir können existierende, starke Heuristiken ausnutzen. In den Abschnitten 5.2 und 5.3 beschreiben wir kurz die lokale Suche und die von uns verwendeten lokalen Operatoren. Abschnitt 5.4 beschreibt dann unsere Variante des Lin-Kernigham Algorithmus und Abschnitt 5.5 beschreibt, wie wir Lösungen repräsentieren. Abschnitt 5.6 erklärt dann, wie wir die Metaheuristik Iterierte Lokale Suche verwenden, um das Verfahren robuster zu machen.

5.1 Auf den Schultern von Giganten — Ähnlichkeit zu TSP

Satz 2.11 beschreibt eine einfache Reduktion von DOPI auf WHP und wieder zurück. Neben der Tatsache, dass wir nur noch eine Tabelle von Abständen betrachten müssen, gibt es noch einen weiteren Vorteil bei dieser Reduktion: WHP ist sehr ähnlich zum Problem des Handelsreisenden.

Definition 5.1 (Problem des Handelsreisenden, TSP): Gegeben ist eine Menge V von n Städten und eine Abstandsfunktion $w : V \times V \rightarrow \mathbb{R}$. Gesucht ist nun eine Rundreise $T = \langle v_0, v_1, \dots, v_{n-1}, v_0 \rangle$, so dass ihre Länge l minimal wird.

Dabei ist

$$l = w(v_{n-1}, v_0) + \sum_{i=0}^{n-2} w(v_i, v_{i+1}),$$

also die Summe der zurückgelegten Abstände.

Für uns ist das *symmetrische TSP mit Dreiecksungleichung* interessant, dabei ist die *Gewichtsfunktion* w symmetrisch und erfüllt die Dreiecksungleichung (siehe auch Tatsache 2.7). Wir können uns für den Wertebereich der Abstandsfunktion auf \mathbb{N} beschränken. \diamond

Aufgrund dieser starken Ähnlichkeit vermuten wir, dass aus einer starken Heuristik für WHP eine starke Heuristik für TSP abgeleitet werden kann und umgekehrt.

Das TSP ist eines der am besten untersuchten \mathcal{NP} -vollständigen Probleme. Daher entschieden wir, nicht zu versuchen eine revolutionäre, neue Heuristik für WHP und damit für DOPI zu finden. In wenigen Wochen würden wir die Arbeit von Experten über Jahrzehnte nicht übertreffen.

Besonders interessant ist in unserem Falle die existierende Literatur über Heuristiken zum TSP. Die momentan stärkste Heuristik für das TSP ist Helsgauns Modifikation (LKH) der *Lin-Kernigham Heuristik (LK)* (siehe [9, 5, 6, 7]).

Unsere ILS-Heuristik basiert auch auf der ursprünglichen Lin-Kernigham Heuristik von 1973. Wir haben LK in C++ von Grund auf neu implementiert. Dabei haben wir einige Optimierungen einfließen lassen, die 1973 noch nicht bekannt waren. Diese Optimierungen hat auch Helsgaun verwandt und sie sind auch in [4] beschrieben.

Unsere Heuristik ist jedoch nicht so ausgefeilt wie LKH. Dies geht auf Kosten der Lösungsqualität und Laufzeit. Wir haben aus zufällig erzeugten DOPI-Eingaben entsprechende Eingaben für LKH erzeugt. Aus der resultierenden Handelsreisenden-Tour entfernten wir die schwerste Kante. Die dabei entstehenden Hamilton-Pfade waren dabei leichter als die Ergebnisse unserer Heuristik. Allerdings war unsere Heuristik meist weniger als 1 % über dem Ergebnis von LKH.

Es ist noch anzumerken, dass der leichteste Hamilton-Pfad in einem vollständigen Graphen mit Dreiecksungleichung leichter sein kann als ein Hamilton-Pfad, der durch eine „TSP-Tour ohne schwerste Kante“ entsteht. Abbildung 2 zeigt hierfür ein Beispiel.

Man beachte, dass die Lösungen nur weniger als 1 % über der in 2.5 beschriebenen unteren Schranke liegen.

5.2 Lokale Suche

Lokale Suche ist ein Ansatz für Heuristiken zu kombinatorischen Optimierungsproblemen: Ausgehend von einer *gültigen Lösung*, wird diese schrittweise durch *lokale Operatoren* verbessert. Lokale Operatoren sind dabei Veränderungen der Lösung, die die Gültigkeit erhalten, die Lösung verbessern und schnell ausgeführt werden können.

Zu jeder Permutation $\pi : \{0, \dots, n-1\} \rightarrow \{0, \dots, n-1\}$ ist die Anordnung der Punkte zu einem Pfad $P_\pi = \langle v_{\pi(0)}, \dots, v_{\pi(n-1)} \rangle$ eine gültige Lösung.

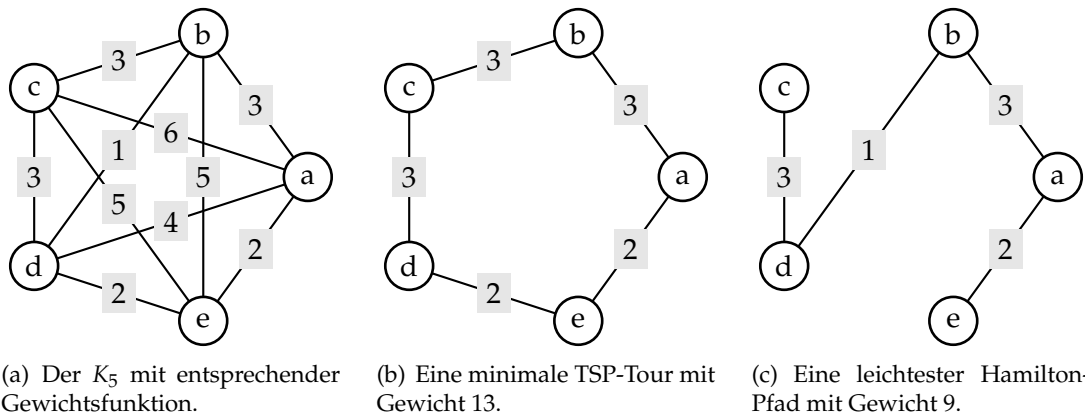


Abbildung 2: In diesem K_5 ist die Dreiecksungleichung erfüllt. Das Gewicht des leichtesten Hamilton-Pfades (9) ist jedoch kleiner als das der kleinsten TSP-Tour ohne schwerste Kante (10).

Ein Beispiel für einen lokalen Operator ist etwa das Vertauschen von zwei Elementen des Pfades:

Definition 5.2 (swap-Operator): Der lokale Operator $swap(i, j)$ verändert eine Lösung, indem er die Permutation π zur Permutation π' ändert mit:

$$\pi'(x) = \begin{cases} \pi(i) & \text{wenn } x = j \\ \pi(j) & \text{wenn } x = i \\ \pi(x) & \text{sonst.} \end{cases}$$

Die Elemente an Position i und j werden also vertauscht. ◇

5.3 k -opt Züge

Der $swap$ -Operator ist sehr einfach zu implementieren. Wird die Permutation als Array implementiert kann er in $\mathcal{O}(1)$ ablaufen. Allerdings liefert eine lokale Suche mit $swap$ -Operatoren keine sehr guten Lösungen.

Für TSP betrachtet man daher k -opt Züge (engl.: k -opt moves), was man auch für WHP machen kann. Die Definition für k -opt Züge ist nicht so allgemein, wie etwa in [5] aber für unsere Zwecke genügt sie.

Definition 5.3 ((Sequentieller) k -opt Zug): Für $k \geq 2$ besteht ein k -opt Zug darin, k Kanten aus dem Pfad zu entfernen und dafür k Kanten wieder einzufügen, so dass ein gültiger Hamilton-Pfad entsteht.

Ein (sequentieller) k -opt Zug besteht aus einer Sequenz S von Kanten:

$$S = \langle x_1, y_1, x_2, y_2, \dots, x_k, y_k \rangle$$

Wir fordern außerdem noch, dass x_i und y_i jeweils einen Endpunkt gemeinsam haben. Also muss gelten: $x_i = (t_{2i-1}, t_{2i})$, $y_i = (t_{2i}, t_{2i+1})$ und $x_{i+1} = (t_{2i+1}, t_{2i+2})$. Weiter muss S „geschlossen“ sein: $y_k = (t_{2r}, t_1)$.

Die Kanten x_i werden gelöscht, die Kanten y_i werden eingefügt. Nach dem Einfügen und dem Löschen muss wieder ein gültiger Pfad entstehen. \diamond

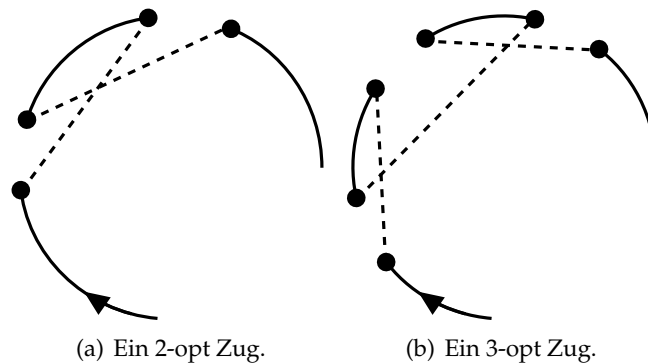


Abbildung 3: Beispiele für 2-opt und 3-opt Züge.

Abbildung 3 zeigt einen 2-opt und einen 3-opt Zug. Die gestrichelten Kanten sind eingefügte Kanten, fehlende Kreissegmente sind die gelöschten Kanten. Der 3-opt Zug ist nur einer von zwei möglichen.

Dass ein gültiger Pfad entsteht kann durch „richtiges“ Wählen der einzufügenden Kante sichergestellt werden. Zum Beispiel muss verhindert werden, dass ein Kreis entsteht.

Basierend auf k -opt Zügen kann ein Merkmal zur Qualität einer Lösung formuliert werden:

Definition 5.4 (λ -Optimalität): Eine WHP-Lösung ist λ -optimal, wenn es kein k gibt mit $k \leq \lambda$ so dass ein verbessernder k -opt Zug existiert. \diamond

Man kann λ -optimale Lösungen erzeugen, indem man wiederholt für jedes $k \leq \lambda$ den besten k -opt Zug ausführt. Dies tut man, bis es keinen solchen Zug mehr gibt und eine λ -optimale Lösung ist gefunden.

Das Problem ist nur, dass im Vorhinein nicht klar ist, wie λ zu wählen ist und eine naive Suche für ein fest gewähltes λ eine Zeit von $\mathcal{O}(n^\lambda)$ benötigt. Die Idee von Lin und Kernigham war nun einen Greedy-Algorithmus mit variablem λ .

5.4 Der Lin-Kernigham Algorithmus für WHP

Man baut die Menge S aus Definition 5.3 iterativ auf. Man entfernt abwechselnd immer eine Kante x_i und fügt eine Kante y_{i+1} ein. Dabei heißt im j -ten Schritt

$$g_j = w(x_1) - w(y_1) + w(x_2) - w(y_2) + \dots + w(x_j) - w(y_j)$$

der *Gewinn*.

Unsere Variante des Lin-Kernigham Algorithmus ist wie folgt:

- Wähle eine Kante x zum Entfernen und hänge sie an S an.
- Wähle eine Kante y zum Einfügen und hänge sie an S an.
- Wenn g_j nicht positiv ist, verwerfe die aktuelle Lösung und fange mit einer neuen Startkante an.
- Wenn der Pfad geschlossen werden kann und der Gewinn immer noch positiv ist dann tue dies, der Gewinn ist die Reduktion im Gewicht des aktuellen Pfades.
- Andernfalls, gehe zum ersten Schritt.

Der Algorithmus wird abgebrochen, wenn S eine bestimmte Größe überschreitet, etwa 50. Eine gefundene Lösung ist damit 50-optimal. Die Komplexität für einen Schritt ist $\mathcal{O}(\lambda \cdot n)$.

5.5 Effiziente Repräsentation von Permutationen

Die Arbeit [4] beschreibt effiziente Datenstrukturen für TSP-Heuristiken. Dabei ist die Repräsentation der momentanen Handelsreisenden-Tour zentral. Die Tour kann effizient als Permutation von $\{0, \dots, n-1\}$ gespeichert werden. Ein gewichteter Hamilton-Pfad ist auch eine Permutation, also können diese Datenstrukturen auch zur Repräsentation eines solchen Pfades genutzt werden.

Für den Lin-Kernigham Algorithmus muss die Datenstruktur für die Permutation folgende Operatoren zur Verfügung stellen:

- $Prev(i), Next(i)$ — Gebe zu dem Wert i den vorherigen/nächsten Wert in der Permutation zurück. In einem Pfad $P = \langle \pi(0), \dots, \pi(n-1) \rangle$ ist also zu $i = \pi(j)$ ist $\pi(j+1)$ gesucht.
- $Between(i, j, k)$ — Überprüft, ob der Wert j zwischen i und k liegt, also ob der Pfad die Form $\langle \dots, i, \dots, j, \dots, k, \dots \rangle$ oder $\langle \dots, k, \dots, j, \dots, i, \dots \rangle$ hat.
- $Flip(i, j)$ — Drehe den Teilpfad $\langle i, \dots, j \rangle$ bzw. $\langle j, \dots, i \rangle$ um (ohne Einschränkung der Allgemeinheit sei das erste der Fall). Dies entspricht dem Entfernen der Kanten $(Prev(i), i)$ und $(j, Next(j))$ aus dem Pfad und dem gleichzeitigen Hinzufügen von $(Prev(i), j)$ und $(Next(j), j)$.

5.5.1 Repräsentation als Array

Die einfachste Repräsentation einer Permutation von $\{0, \dots, n-1\}$ ist die als Array der Länge n . Der Wert von $\pi(i)$ wird an Position i im Array gespeichert. Außerdem hält man noch die Umkehrung der Permutation vor, in der die Position j von dem Wert $i = \pi(j)$ in einem zweiten Array an Position i gespeichert wird.

Index	0	1	2	3	4	5
π	3	5	2	1	4	0
π^{-1}	5	3	2	0	4	1

Abbildung 4: Repräsentation einer Permutation bzw. eines Hamilton-Pfades als Array.

Abbildung 4 zeigt ein Beispiel für die Repräsentation einer Permutation als Array.

Mit der Array-Repräsentation können die Operatoren *Prev*, *Next* und *Between* so implementiert werden, dass sie in $\mathcal{O}(1)$ laufen. Der *Flip*-Operator benötigt jedoch Zeit proportional in der Länge des umzudrehenden Intervalls, also im schlimmsten Fall $\Omega(n)$.

Unsere LK-Heuristik führt jedoch für jeden k -opt Zug k *Flip*-Operationen aus, die gegebenenfalls verworfen werden, wenn der Pfad nicht geschlossen werden kann. Es stellt sich also die Frage nach einer effizienteren Datenstruktur für den *Flip*-Operator.

5.5.2 Repräsentation als Segment-Baum

Wenn man den LK-Algorithmus betrachtet dann stellt man fest, dass er den Pfad in einem gewissen Sinne gar nicht so stark verändert. Es werden Teilpfade umgedreht und verschoben, aber wenn man das größte k auf einen Wert, etwa 50, begrenzt dann entstehen höchstens 51 solcher Teilpfade.

Die Teilpfade können einfach durch den ersten und letzten Index in der Permutation vor dem Laufen des Algorithmus beschrieben werden. Zusätzlich kann man einfach noch speichern, ob der Teilpfad in der ursprünglichen oder in umgedrehter Richtung auftritt. Das Ergebnis eines Durchlaufes kann also als Sequenz von Teilpfaden und Richtungsinformation beschrieben werden.

Dies motiviert die Datenstruktur *Segment-Baum*:

Ein Segment ist ein Tripel (s, t, r) , das einen Teilpfad beschreibt. s und t sind Anfangs- und Endindex des Intervalls. r ist das Bit, das angibt ob der Teilpfad in umgekehrter oder normaler Richtung durchlaufen wird.

Die Datenstruktur besteht nun aus einer doppelt verketteten Liste von Segmenten, die den veränderten Pfad repräsentiert. Zusätzlich existiert noch ein Suchbaum, der zu jedem Anfangs-Index eines Segments auf das Listenelement verweist, in dem das Segment für den Teilpfad gespeichert ist.

Zu einem beliebigen Index in der Permutation kann das zugehörige Segment dann mit einer *locate*-Operation des Suchbaums gefunden werden (liefert den nächsten kleineren Wert).

Abbildung 5 zeigt ein Beispiel für einen Segment-Baum.

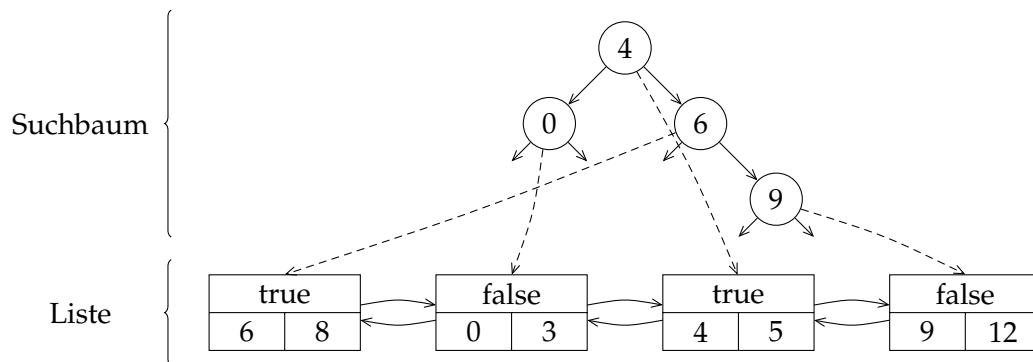


Abbildung 5: Beispiel für einen Segment-Baum. Der Baum repräsentiert die Folge $\langle 8, 7, 6, 0, 1, 2, 3, 5, 4, 9, 10, 11, 12 \rangle$. Die Kreise sind Knoten im Suchbaum. Durchgezogene Linien sind Nachfolger-/Vorgänger-Zeiger. Gestrichelte Linien sind Datenzeiger. In der Liste ist im oberen Teil das „umgedreht“ Bit. Im linken Teil ist der Intervall-Anfang, im rechten Teil das Intervall-Ende.

5.6 Eine Metaheuristik zur Verbesserung der Robustheit

Ein Problem von lokaler Suche ist, dass man lokale Minima findet und nicht aus ihnen hinaus kann. *Iterierte Lokale Suche* (siehe etwa [11]) ist eine Metaheuristik, die darauf abzielt, aus lokalen Optima hinauszufinden:

Die momentane Lösung wird *perturbiert*, also zufällig so „gestört“, dass sie gültig bleibt, aber gegebenenfalls einen schlechteren Wert hat. Auf die perturbierte Lösung wird dann wieder lokale Suche – in unserem Fall LK – angewandt. Wir perturbieren die Lösung mit dem *swap*-Operator.

Durch die Iterierter Lokale Suche mit Perturbation wurde unser LK-Algorithmus robuster, die Lösungsqualität wurde besser. Allerdings geht dies zu Kosten der Laufzeit.

Eine Alternative zur Perturbation ist es mit einer zufällige Lösung neu anzufangen. Wir vermuten, dass dabei auf größeren Instanzen (etwa 1'000 Knoten) die Laufzeit schlechter wird, ohne dass es viel bessere Ergebnisse gibt.

5.7 Möglichkeiten zur Parallelisierung

Wir gehen in diesem Abschnitt davon aus, dass ein Thread pro Prozessorkern erzeugt wird.

Unsere Variante der Lin-Kernigham Heuristik ist recht schnell. Mit einem Parameter *max moves per word* von 300 erzeugt das Programm auf zufällig erzeugten Instanzen mit Parametern $n = 1000, k = 1000$ in unter einer Minute zum Beispiel eine Lösung von 449'269. Verdoppelt man *max moves per word* so verbessert sich die Lösung auf der gleichen Instanz nicht stärker sondern nur auf 449'237 und das Programm braucht etwa doppelt so lange.

Für sehr große n kann etwa das Suchen der nächsten zu entfernenden Kante parallelisiert werden. Jeder der p Threads bekommt dann $\frac{n}{p}$ Knoten zugewiesen, für die er den potentiellen Gewinn ermittelt.

Die Verwendung der Metaheuristik ILS eröffnet eine weitere Möglichkeit zur Parallelisierung: Das Programm verbringt schon für kleinere Eingaben von $n = 1000$ viel Zeit damit, die Lösung zu perturbieren. Sobald der Algorithmus in einem lokalen Optimum landet können nun mehrere Threads gleichzeitig nach einer Perturbation, die aus dem lokalen Optimum herausführt.

Lösungen werden durch eine Permutation und einen Segment-Tree repräsentiert. Sie brauchen mit 32 Bit *unsigned* Werten etwa $8 \times n + \mathcal{O}(1)$ Byte Speicher. Für $n = 1000$ sind dies gut 8 Kilobyte. Auf einem Sun UltraSPARC T2 mit 8 Kernen und insgesamt 64 Hardware-Threads ergibt sich ein Speicherverbrauch von gut 512 Kilobyte. Damit bleiben noch knapp 3,5 MB Cache für die Adjazenzmatrix. Repräsentiert man Abstände mit `short` Werten, so passt sie für $n = 1000$ noch vollständig in den Cache.

Für größere n kann man noch überlegen, nur die untere oder obere Hälfte der symmetrischen Adjazenzmatrix zu speichern. Allerdings ist fraglich, ob durch Fehlvorhersagen von Sprüngen dadurch nicht zu viel Performance verloren geht.

6 Genetischer Algorithmus

It is the nature of all greatness not to be exact.
— Edmund Burke

Wir haben einen genetischen Algorithmus basierend auf dem Ansatz in von Logofatu et al. in [10] implementiert. Dieser Algorithmus basiert auf Ideen der natürlichen Selektion. Wir bezeichnen eine gültige Lösung für das WHP-Problem, also eine Permutation der Knotenindizes, als ein *Individuum*. Die Menge der zu einem Zeitpunkt t bestehenden Individuen nennen wir eine *Population* P_t . Die Qualität einer Lösung nennen wir die *Fitness* des entsprechenden Individuums.

6.1 Ablauf des Algorithmus

Der Algorithmus läuft im Groben wie folgt ab: Eine Population P_t wird in einem Zeitschritt durch *Mutation* und *Kreuzung* vergrößert, es entsteht so die Population P'_t . Danach werden die besten Individuen aus P'_t ausgewählt und in eine neue Population P_{t+1} überführt. Man hofft, so die Gesamtfitness der Population schrittweise steigern zu können. Durch die Mutation wird außerdem versucht zu verhindern, dass der Algorithmus sich in lokalen Optima verfängt. Das Verfahren wird abgebrochen, wenn seit einer vorher festgelegten Anzahl von Generationen keine Verbesserung der Fitness mehr eingetreten ist. Die Operatoren zur Mutation und Kreuzung funktionieren wie im Folgenden beschrieben:

Mutation. Ein Individuum wird mit einer Wahrscheinlichkeit, die proportional zu seiner Fitness ist, aus der Population ausgewählt (*Rouletterad-Selektion*). Wir nennen dieses Individuum *Elter*. Auf das Elter wird der *Simple Cycle Inversion Mutation*-Operator angewendet, um ein von ihm verschiedenes Kind zu erzeugen. Der Operator wählt zufällig ein Indexpaar $(i, j) \in \{1, \dots, k\}^2$ aus. Ein Kind wird nun erzeugt, indem die Sequenz der Indizes des Elters zwischen i und j umgekehrt wird, wie im untenstehenden Bild skizziert. Die restlichen Positionen werden übernommen. Falls im zufällig gewählten Indexpaar $i > j$ sein sollte, so stelle man sich die Permutation als Ring vor, der entsteht, indem Anfang und Ende der Permutation verbunden werden. Die Positionen werden dann wie gehabt zwischen i und j getauscht und der Ring wieder aufgetrennt.

Da für das Umkehren einer Sequenz von der Länge $l := |j - i|$ insgesamt $\lfloor \frac{l}{2} \rfloor$ *swap*-Operationen benötigt werden, benötigt der SCIM-Operator $\mathcal{O}(l)$ Zeit.

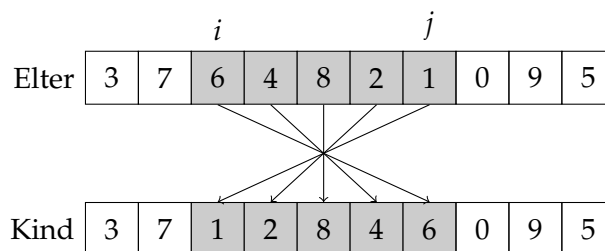


Abbildung 6: Die Vorgehensweise des SCIM-Operators

Kreuzung. Zur Kreuzung eines Elternpaares wird der *Cycle OX*-Operator (*COX*, siehe [3]) verwendet. Es werden wiederum per *Rouletterad-Selektion* zwei Individuen aus der Population ausgewählt, die *Mutter* und der *Vater*. Wie bei der Mutation wird wieder ein Paar (i, j) von Indizes bestimmt. Das Kind übernimmt nun die Sequenz zwischen den Indizes i und j vom Vater. Die freien Positionen werden aufgefüllt, indem von der Mutter die noch nicht verwendeten Indizes unter Beachtung der relativen Reihenfolge übernommen werden, siehe Abb. 7. Der Fall, dass i größer gewählt wird als j , wird behandelt wie gerade bei der Mutation beschrieben. Da zur Erzeugung des Kindes jeder Index der Permutation betrachtet wird, benötigt eine Kreuzung mit dem *COX*-Operator $\mathcal{O}(k)$ Zeit.

6.2 Parameter und Möglichkeiten zur Optimierung

Der Ablauf des genetischen Algorithmus ist von einigen Parametern abhängig. Es muss festgelegt werden, welcher Anteil einer Population jeweils zur Mutation und zur Kreuzung ausgewählt wird. Weiterhin muss die Größe der Population gewählt werden. Wählt man die Population zu klein, so nimmt die Heterogenität der Individuen ab. Das heißt, dass neue Generationen sich nur schwach von ihren Vorfahren unterschei-

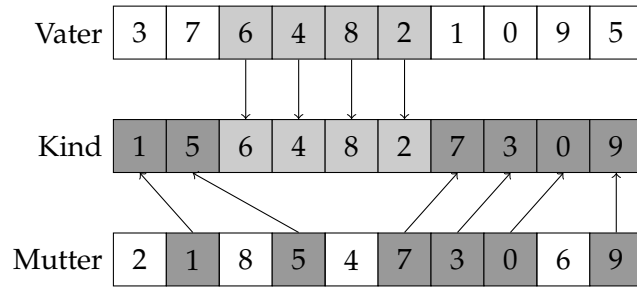


Abbildung 7: Die Vorgehensweise des COX-Operators

den und so leichter in ein lokales Optimum laufen, aus dem sie nicht mehr entfliehen können. Wählt man die Population zu groß, so geht das auf Kosten der Laufzeit.

Diese Parameter lassen sich über die Kommandozeile angeben. Gibt man keine Parameter an, so wird als Größe der Population (*popsiz*e) das Fünffache der Anzahl der Eingabewörter gewählt. Pro Zeitschritt werden $0.5 * popsiz$ e Individuen zur Kreuzung und $0.2 * popsiz$ e Individuen zur Mutation ausgewählt (jeweils mit Zurücklegen).

Eine weitere Möglichkeit, das Ergebnis des genetischen Algorithmus zu beeinflussen, liegt in der Wahl der Startpopulation. Setzt man die Startpopulation lediglich aus zufälligen Lösungen zusammen, so wird nicht in annehmbarer Zeit eine gute Lösung gefunden. Man sollte also einige Lösungen einfließen lassen, die schon eine gewisse Güte aufweisen. Sind allerdings zu viele davon in der Startpopulation vorhanden, so setzen sie sich zu schnell durch und die Lösungen konvergieren zu früh. Wir haben uns dafür entschieden, für eine Instanz mit n Wörtern mit n Individuen zu beginnen, die der Greedy-Lösung entsprechen sowie mit $4n$ zufälligen Lösungen.

6.3 Möglichkeiten zur Parallelisierung

Zwei Möglichkeiten zur Parallelisierung des genetischen Algorithmus ergeben sich analog zur Evolution in der Biologie:

Einerseits können innerhalb einer Population mehrere Elternpaare gleichzeitig Kinder zeugen. An einer Stelle weicht der genetische Algorithmus aber von der Situation in der Natur ab: Ein Individuum kann gleichzeitig von mehreren Threads gelesen werden und damit auch zur selben Zeit mehrere Kinder zeugen.

Andererseits könnten mehrere Populationen betrachtet werden, die sich gleichzeitig unabhängig voneinander entwickeln und sich zu bestimmten Zeitschritten miteinander vermischen. Hierbei finden sich in der Literatur verschiedene Methoden, etwa zufälliges Verteilen oder ein Übernehmen von starken Individuen in mehrere Populationen. Hier ist noch zu beachten, dass man starke Individuen zwar in viele Populationen übernehmen will aber auch die Heterogenität innerhalb einer und zwischen mehreren Populationen gewährleisten.

Eine dritte Möglichkeit bestünde darin, die Operatoren zu parallelisieren und etwa mehrere Prozessoren gleichzeitig an der Mutation eines einzelnen Individuums arbeiten zu lassen. Das lohnt sich aber nur für große Individuen, also lange Pfade.

7 Implementierung und Entwurfsentscheidungen

7.1 Historie zur Heuristik-Auswahl

Wir beschlossen am Anfang einen exakten Löser mit Branch-And-Bound zu implementieren.

Zunächst war uns nicht klar, welcher heuristische Ansatz der Beste sein würde. Wir versuchten erst übliche Heuristiken wie Ant Colony Optimization, Hill-Climbing mit einfacher lokaler Suche für DOPI zu implementieren.

Literaturrecherchen zu DOPI führten zu [13, 10] und damit der Greedy-Heuristik und einem genetischen Algorithmus.

An einem Punkt wurde uns dann klar, dass eine Reduktion auf WHP möglich war und sich daher das Studieren von Heuristiken für TSP anbot. Die Heuristik von Lin-Kernigham ist die erfolgreichste für praktische TSP-Instanzen.

DOPI-Instanzen ergeben zwar WHP-Instanzen, die nicht wirklich praktischen WHP-Instanzen entsprechen würden aber wir hofften trotzdem auf gute Ergebnisse. Unter anderem ist der Maximale Abstand zwischen zwei Worten durch $\frac{k}{2}$ beschränkt, zum anderen war in der Aufgabenstellung keine Vorgabe, etwa eine Verteilung über Wörtern gegeben.

Wir landeten somit also bei der Wahl, Lin-Kernigham und einen genetischen Algorithmus zu implementieren. Weiterhin fiel mit der Reduktion auf WHP noch die Möglichkeit, ein ILP zu verwenden und eine Vereinfachung des Branch-And-Bound Löser ab.

7.2 Struktur der Implementierung

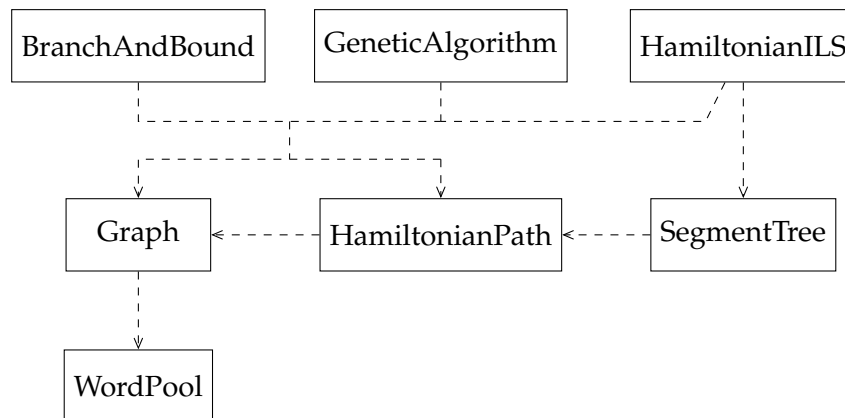


Abbildung 8: Klassendiagramm der Implementierung.

Abbildung 8 zeigt ein Klassendiagramm mit den wesentlichen Klassen der Implementierung.

Die Klasse WordPool. WordPool repräsentiert eine geordnete Sequenz von Binärwörtern. Die Wörter können als Zeichenketten mit den Buchstaben „0“ und „1“ gesetzt werden. WordPool repräsentiert die Buchstaben-Bitfolgen in echte Bitfolgen. So wird aus der Zeichenkette, die abwechselnd aus Blöcken von acht „00“ und acht „1“ besteht die Zahl 0xff00ff00 (in Hexadezimaldarstellung).

Dies ermöglicht auch ein sehr effizientes Berechnen von Transitionen zwischen Wörtern: Sind x und y Arrays von Maschinenwörtern, so ist die Zahl der gesetzten Bits in $x \oplus y$ die Zahl der Transitionen zwischen x und y . Ein schnelles Verfahren zum Zählen der gesetzten Bits, das auf Tabellen basiert findet sich z.B. in [1].

Das Invertieren von Wörtern geht in dieser Darstellung auch schnell: Zu einem 32-Bit *unsigned* Wert x ist das Inverse $x \oplus 0xffffffff$.

WordPool kapselt damit eine DOPI-Instanz und die schnelle „low level“ Implementierung von Inversion und Berechnung von Transitionen.

Die Klasse Graph. Graph repräsentiert einen vollständigen, ungerichteten Graphen mit Ganzzahlen als Kantengewichten. Ein Graph wird mit einem WordPool als Parameter initialisiert und führt die Reduktion aus Satz 2.11 durch. Die Gewichte der Kanten werden in einer Adjazenzmatrix gespeichert (siehe etwa [2]).

Eine WHP-Instanz wird also durch die Klasse gekapselt.

Die Klasse HamiltonianPath. HamiltonianPath repräsentiert einen Hamilton-Pfad als Permutation in einem Array. Neben der Permutation π wird auch noch ihr Inverses π^{-1} und das Gesamtgewicht des Pfades gespeichert.

Die Klasse stellt die Operatoren *Flip* und *Swap* zur Verfügung. Außerdem erlaubt diese Klasse das Ausgeben des Pfades im verlangten Format, also die Umkehrung der Reduktion aus Satz 2.11.

Ein HamiltonianPath-Objekt ist also eine gültige Lösung für eine WHP-Instanz und erlaubt die Ausgabe als DOPI-Lösung.

Die Klasse SegmentTree. SegmentTree implementiert einen Segment-Baum (siehe Abschnitt 5.5.2).

Objekte dieser Klasse kapseln Veränderungen auf HamiltonPath Objekten und erlauben damit eine schnelle Implementierung der LK-Heuristik. Die Veränderungen können dann in dem HamiltonianPath Objekt permanent gespeichert werden.

Die Klassen BranchAndBound, GeneticAlgorithm und HamiltonianILS. Diese Klassen implementieren den exakten Branch-And-Bound Löser aus Abschnitt 4.1 und die beiden Heuristiken aus den Abschnitten 6 und 5.

Sie kapseln im Wesentlichen die Algorithmen und halten die nötigen Datenstrukturen vor.

7.3 Vorgehen bei Implementierung und Tuning

Nachdem die grundsätzliche Struktur der Algorithmen festgelegt war, implementierten wir sie erst einmal auf naheliegende Weise.

Dann instrumentierten wir unseren Code mit Statistik-Zählern und Stoppuhren. Auf diese Weise war es möglich, das Verhalten der Algorithmen zu betrachten und Flaschenhälse zu finden. An einzelnen Stellen wurde dann optimiert.

Am Anfang war uns klar, dass die Abstandsfunktion oft angerufen würde. Daher berechneten wir diese (noch naiv auf Basis von Zeichenketten) vor (in der Klasse, die jetzt Graph heißt). Es war jedoch nicht klar, wie lange dies praktisch dauern würde. Wir stellten fest, dass es auf größeren Instanzen eine erhebliche Zeit in Anspruch nahm. Daher stellten wir die Repräsentation der Wörter auf „echte Bitfolgen“ – wie in Abschnitt 7.2 beschrieben – um.

Zum Feintuning der Algorithmen erlaubten wir das Setzen der Parameter (etwa die Populationsgröße beim genetischen Algorithmus) der Algorithmen über die Kommandozeile. Die erlaubte das einfache Einstellen der Parameter ohne neu kompilieren zu müssen.

7.4 Zusätzliche Werkzeuge

Wir schrieben auch eine Reihe von kleinen Werkzeugen, um die Entwicklung zu vereinfachen.

Erstellen von Instanzen. Die ersten Test-Instanzen erzeugten wir noch von Hand. Wir schrieben dann allerdings ein kleines Werkzeug *tools/rand.pl*, das das zufällig Instanzen erstellte.

Prüfen von Ergebnisse. Zum Überprüfen unserer Ergebnisse schrieben wir auch ein Werkzeug (*tools/checker.py*). Hinzu kam noch ein Werkzeug zum Ausführen eines Lösers und dem Prüfen des Ergebnisses (*tools/checked-run.sh*).

Vergleich mit Helsgauns LK-Heuristik. Zum Vergleich mit Helsgauns LK-Heuristik erstellten wir ein Werkzeug, das eine DOPI-Eingabe in eine Eingabe für Helsgauns Programm LKH erzeugte. Ein anderes Werkzeug las die Lösung von LKH ein, löschte die schwerste Kante der TSP-Tour und gab dann das Gewicht des Hamiltonian Pfad aus.

8 Die graphische Benutzeroberfläche

Wir haben die graphische Benutzeroberfläche (GUI) für Mac Os X mit Cocoa/Objective-C++ geschrieben.

Da die Anforderungen sehr einfach waren ist auch die Programmstruktur einfach. Sie entspricht im Wesentlichen dessen, was intuitiv aus dem „best practice“ folgt:

Ein *AppController* kontrolliert die Anwendung und GUI. Die Klasse *SolverState* kapselt den Zugriff auf die C++ Löser-Klassen. *AppState* kapselt den Zustand der GUI. Die Klassen *SolutionView* und *TransmissionView* kapseln die Elemente zum Anzeigen der Grafiken.

Die Bedienungsanleitung ist in Abschnitt 9.5.

9 Anleitung zum Benutzen und Installieren der Programme

Unsere Abgabe besteht aus einem Tarball *dopi.tar.gz*. Dieses kann auf Linux und Mac Os X mit *tar xzf dopi.tar.gz* entpackt werden.

Im Folgenden beziehen sich die Pfade immer relativ zum Ordner *dopi*, der erzeugt wird wenn das Archiv entpackt wird.

9.1 Branch-And-Bound

Installation. Wir haben das Programm *exact* in Binärform für Mac Os X 10.5 und Ubuntu Linux intrepid im Verzeichnis *bin/osx* bzw. *bin/linux* beigelegt. Es ist statisch gelinkt und sollte ohne weiteres lauffähig sein.

Um das Programm auf Mac Os X neu zu übersetzen können die folgenden Befehle verwendet werden. Hinweise zum Compilieren unter Linux finden sich in Anhang A.

```
cd dopi
cd src/optimized
make exact
```

Bedienung. Wie gefordert erwartet das Programm das Problem über die Standardeingabe (*stdin*). Die Ausgabe geschieht über die Standardausgabe (*stdout*):

```
./exact < Eingabe.txt > Ausgabe.txt
```

9.2 Integer Linear Program Generator

Der ILP Generator funktioniert – wenn die nötige Software (s.u.) installiert ist direkt nach dem Entpackten aus dem Archiv.

Als Zugabe haben wir noch unseren Generator für ILPs (siehe Abschnitt 4.2) mitgeschickt. Damit dieser Löser funktioniert, müssen folgende Softwarepakete installiert sein. Die Versionsangaben sind jeweils die von uns getesteten Versionen, nicht all zu alte sollten aber auch funktionieren.

- Python \geq 2.5. Sollte unter Linux als Paket der Distribution zur Verfügung stellen. Unter Mac Os X kann man es leicht mit MacPorts installieren.
- Die Bash Shell.

- GLPK \geq 4.31. Das Programm *glpsol* muss sich im *\$PATH* befinden. Sollte wie Python durch die Distribution oder MacPorts installierbar sein.

Das Skript *glpsol4dopi.sh* erzeugt zuerst mit dem Text aus der Standardeingabe eine temporäre Datei mit der DOPI-Eingabe. Dann erzeugt es mit dem Python-Skript *dopi2cpxlp.py* aus der DOPI-Datei eine Textdatei mit einem ILP. Das ILP wird nun mit *glpsol* gelöst und die Ausgabe in eine weitere temporäre Datei geschrieben. *glpsol2dopisol.py* erzeugt dann aus der temporären Lösungs-Datei und der ursprünglichen Datei die geforderte Ausgabe aus der Aufgabenstellung.

Aufrufen des ILP-basierten Lölers. Der folgende Befehl muss im Verzeichnis *tools* ausgeführt werden, sonst werden die Programme nicht gefunden.

```
./glpsol4dopi.sh < test.dopi
```

Die Ausgabe sollte sein (die folgenden Zeilen sind rechts am „...“ abgeschnitten):

```
466
411
S1111001110011111100010010011001110110000111110010101100011...
S100101100011011110100000111101010001010100011001011110011...
I1100110110111010111100110010010111001111000010010011110000...
S0100111100111010001011000010010011101000011110100111110011...
S0000101110111100101110001011101001000101101110110000111111...
I0000011001110000011101111010101011100000011000100110010011...
S1110000110100000101110110010001101101000101001100111010011...
I1101111111100101100110101010001000010100000101101011001000...
I1101100000100010001011001011111000000000100101111110010000...
S0000101101001000011111110100001100101011001101110101101100...
```

9.3 Iterierte Lokale Suche

Installation. Wir haben das Programm *ils* in Binärform für Mac Os X 10.5 und Ubuntu Linux intrepid im Verzeichnis *bin/osx* bzw. *bin/linux* beigelegt. Es ist statisch gelinkt und sollte ohne weiteres lauffähig sein.

Um das Programm auf Mac Os X neu zu übersetzen können die folgenden Befehle verwendet werden. Hinweise zum Compilieren unter Linux finden sich in Anhang A.

```
cd dopi
cd src/optimized
make ils
```

Bedienung. Wie gefordert erwartet das Programm das Problem über die Standardeingabe (*stdin*). Die Ausgabe geschieht über die Standardausgabe (*stdout*):

```
./ils < Eingabe.txt > Ausgabe.txt
```

Mehr Optionen lassen sich über die Kommandozeilenoption `--help` anzeigen lassen.

9.4 Genetischer Algorithmus

Installation. Wie auch *ils* ist das Programm *genetic* für Mac Os X und Ubuntu Linux intrepid im Verzeichnis *bin/osx* bzw. *bin/linux*.

Das optionale Neu-Übersetzen unter Mac Os X geht mit folgenden Befehlen. Hinweise zum Compilieren unter Linux finden sich in Anhang A.

```
cd dopi
cd src/optimized
make genetic
```

Bedienung. Wie gefordert erwartet das Programm das Problem über die Standardeingabe (*stdin*). Die Ausgabe geschieht über die Standardausgabe (*stdout*):

```
./genetic < Eingabe.txt > Ausgabe.txt
```

Mehr Optionen lassen sich über die Kommandozeilenoption `--help` anzeigen lassen.

9.5 Graphische Benutzeroberfläche

Die Datei *DopiUI.dmg* enthält das Mac Os X Programm *DopiUI.app*. Die DMG-Datei kann unter Mac Os X geöffnet werden. Das Programm *DopiUI.app* kann direkt aus der DMG-Datei heraus gestartet werden. „Installiert“ wird es durch das Kopieren auf die Festplatte des Rechners.

Das Programm ist nur unter Mac Os X 10.5 und höher lauffähig.

Die Benutzung der Oberfläche sollte selbstserklärend sein:

- Mit dem „Wähle...“ Knopf kann eine zu ladende DOPI-Eingabedatei ausgewählt werden.
- Der Knopf „Starte Löser...“ führt die mit den Checkboxes markierten Löser auf der Eingabedatei aus. Der exakte Löser ist in der Standardeinstellung nicht aktiviert, damit zu große Eingaben nicht unbeabsichtigt mit ihm bearbeitet werden.

Die beiden Schaltflächen „Vergleich“ und „Übertragung“ erlauben das Umschalten der Visualisierung.

Unten im Fenster wird zusätzlich noch die Zahl der Transitionen in der Eingabe sowie eine mit der MST-Methode berechneten untere Schranke angezeigt.

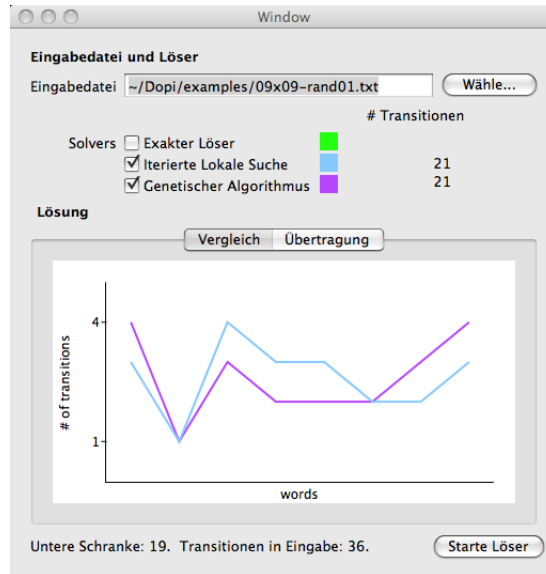


Abbildung 9: Die Oberfläche des Programms.

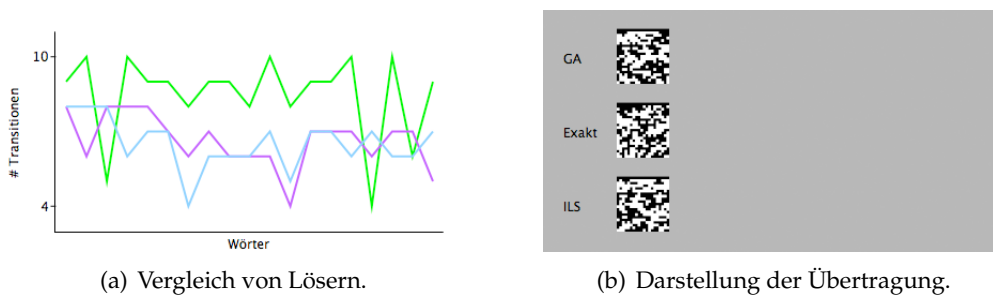


Abbildung 10: Beispiele für Visualisierungen.

10 Ausblick

Die Visualisierungsart „Vergleich“ (Abb. 10(a)) zeichnet ein Kurvendiagramm. Auf der y-Achse ist die Zahl der Transitionen an einer Stellen (zwischen zwei Wörtern) abgetragen. Auf der x-Achse ist abgetragen, an welcher Stelle in der Lösung die Transitionen stattfinden.

Die Visualisierungsart „Übertragung“ (Abb. 10(b)) zeichnet die Wörter in der Reihenfolge, in der sie übertragen werden. Jede Spalte von Kästchen entspricht einem Wort. Das Inversionsbit ist das oberste, dann folgen die Bits der Eingabe mit aufsteigendem Index. Die Wörter sind von links nach rechts angeordnet, wie sie übertragen werden. Ein schwarzes Kästchen entspricht einer „1“, ein weißes Kästchen einer „0“.

Bei dieser Darstellung erkennt man jedoch für zu große k wegen der Skalierung nicht mehr viel.

Unsere Löser für DOPI sind natürlich noch nicht der Weisheit letzter Schluss:
Zum einem wäre es interessant, weitere Heuristiken aus der Arbeit Helsgauns aus-
zuprobieren.

Es wäre auch interessant, neben dem genetischen Algorithmus und Lin-Kernigham
weitere Heuristiken zu betrachten. Hier bieten sich zunächst allgemeinen Heuristiken
wie „Simulated Annealing“ an. Man könnte jedoch auch versuchen, spezielle, erfolg-
reiche Heuristiken für TSP an WHP anzupassen.

Eine Parallelisierung der Heuristiken und des Branch-and-Bound Löser sollte unter
Verwendung von OpenMP mit wenig Aufwand möglich sein.

A Kompilieren unter Ubuntu Linux

Die Anleitung bezieht sich auf Ubuntu *intrepid*.

Benötigte Pakete Für das Kompilieren müssen die Pakete *libargtable2-0* und *gcc* (letz-
teres in Version ≥ 4.3) installiert sein.

Das *libargtable2-0* ist im Paketbereich *universe*. In der Ubuntu-Dokumentation sollte
stehen, wie man diesen Bereich aktiviert.

Dann können die Pakete nachinstalliert werden:

```
sudo apt-get install gcc  
sudo apt-get install libargtable2-0
```

Kompilieren der Programme Das Kompilieren der Pakete geschieht dann mit folgen-
den Befehlen: *dopi* ist das Verzeichnis aus dem Tarball Archiv *dopi.tar.gz*.

```
cd dopi  
cd optimized  
CXXFLAGS=-DGREATER_4_3_0 make
```

Dies kompiliert die Programme *exact*, *ils* und *genetic*.

Außerdem werden noch die Programme *lower* und *greedy* erstellt, die eine untere
Schranke über den MST und eine Greedy-Heuristik implementieren.

B Kompilieren unter Mac Os X

MacPorts. Zum Kompilieren der Programme werden ein paar Pakete benötigt, die
nicht mit Mac Os X mitgeliefert werden. Der einfacheste Weg diese nachzuinstallieren
ist mit MacPorts: <http://www.macports.org/install.php>. Unter dieser URL fin-
det sich auch eine Installationsanleitung.

Ein aktueller GCC. Ist MacPorts installiert dann kann ein aktueller GCC installiert werden. Folgender Kommandozeilenbefehl installiert GCC 4.3:

```
sudo port install gcc43
```

Nun muss noch der installierte GCC als Standard-Compiler eingerichtet werden. Dazu kann folgende Zeile an die Datei *.bashrc* angehängt werden. Alternativ kann sie auch zu Beginn jeder Terminal-Sitzung eingegeben werden.

```
export CXX=g++-mp-4.3
```

Die Bibliothek argtable. Für die Programme *ils* und *genetic* muss die Bibliothek *argtable* installiert sein.

```
sudo port install argtable
```

Kompilieren der Programme Nun können die Programme kompiliert werden. *dopi* ist das Verzeichnis aus dem Tarball Archiv *dopi.tar.gz*.

```
cd dopi
cd optimized
make
```

Dies kompiliert die Programme *exact*, *ils* und *genetic*.

Außerdem werden noch die Programme *lower* und *greedy* erstellt, die eine untere Schranke über den MST und eine Greedy-Heuristik implementieren.

Literatur

- [1] ANDERSON, Sean E.: *Bit Twiddling Hacks*. <http://www-graphics.stanford.edu/~seander/bithacks.html>, Mai 2008
- [2] CORMEN, Thomas H. ; LEISERSON, Charles E. ; RIVEST, Ronald L. ; STEIN, Clifford: *Introduction to Algorithms, Second Edition*. The MIT Press, 2001. – ISBN 0262531968
- [3] DAVIS, L.: Applying adaptive algorithms to epistatic domains. In: *Proceedings, International Joint Conference on Artificial Intelligence* (1985)
- [4] FREDMAN, M. L. ; JOHNSON, D. S. ; MCGEOCH, L. A. ; OSTHEIMER, G.: Data Structures for Traveling Salesmen. In: *Journal Of Algorithms* 18 (1995), S. 432–479
- [5] HELSGAUN, Keld: An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic / Roskilde University. 1998 (81). – DATALOGISKE SKRIFTER (Writings on Computer Science),
- [6] HELSGAUN, Keld: An effective implementation of the lin-kernighan traveling salesman heuristic. In: *European Journal of Operational Research* 126 (2000), S. 106–130
- [7] HELSGAUN, Keld: An Effective Implementation of K-opt Moves for the Lin-Kernighan TSP Heuristic / Roskilde University. 2006 (109). – DATALOGISKE SKRIFTER (Writings on Computer Science)
- [8] KNUTH, Donald E.: *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1997. – ISBN 0201896842
- [9] LIN, Shen ; KERNIGHAN, Brian W.: An Effective Heuristic Algorithm for the Travelling-Salesman Problem. In: *Operations Research* 21 (1973), S. 498–516
- [10] LOGOFATU, Doina ; DRECHSLER, Rolf: Efficient Evolutionary Approaches for the Data Ordering Problem with Inversion. In: *Applications of Evolutionary Computing* (2006), S. 320–331
- [11] LOURENÇO, H. R. ; MARTIN, O.C. ; STÜTZLE, T.: Iterated Local Search. In: GLOVER, F. (Hrsg.) ; KOCHENBERGER, G. (Hrsg.): *Handbook of Metaheuristics*, Kluwer Academic Publishers, 2003, S. 321–353
- [12] MILTERSEN, Peter B.: *MILP, ILP and TSP, Course notes for Search and Optimization, Spring 2004*. <http://www.daimi.au.dk/dSoegOpt/ilp.pdf>, March 2004
- [13] MURGAI, Rajeev ; FUJITA, Masahiro ; OLIVEIRA, Arlindo: Using complementation and resequencing to minimize transitions. In: *DAC '98: Proceedings of the 35th annual conference on Design automation*. New York, NY, USA : ACM, 1998. – ISBN 0-89791-964-5, S. 694–697