**Diplomarbeit**

# Implementation und Optimierung evolutionärer Programme mit Anwendung auf Tourenplanung mit Zeitfenstern

Oliver Hein
Kaiserhofstr. 18
60313 Frankfurt
Tel.: 069 - 20967

Betreuer:
Prof. Dr. M. Schmidt-Schauß
Lehrstuhl für Künstliche Intelligenz und Softwaretechnologie

Fachbereich Informatik
Johann-Wolfgang-Goethe-Universität
Frankfurt

**Ehrenwörtliche Erklärung**

Hiermit versichere ich, daß ich die vorliegende Arbeit selbstständig verfaßt habe und keine anderen Quellen oder Hilfsmittel als die in dieser Arbeit angegebenen verwendet habe.

Oliver Hein

Frankfurt, den 1. September 1994

**Inhaltsverzeichnis**


1. Einleitung und Ziel der Arbeit ..... 1

2. Tourenplanung mit Zeitfenstern ..... 2
   - 2.1 Einführung in die Tourenplanung ..... 2
   - 2.2 Mathematische Formulierung der Tourenplanung ..... 3
   - 2.3 Komplexität der Tourenplanung ..... 5
   - 2.4 Heuristiken zur Lösung des Tourenproblems ..... 7
     - 2.4.1 Das Sweep Verfahren ..... 8
     - 2.4.2 Die Savings Heuristik ..... 9
     - 2.4.3 Die Insertion Heuristik ..... 9

3. Grundlagen genetischer Algorithmen ..... 12
   - 3.1 Idee genetischer Algorithmen ..... 12
   - 3.2 Die genetischen Operatoren ..... 14
     - 3.2.1 Selektion ..... 14
     - 3.2.2 Rekombination ..... 14
     - 3.2.3 Mutation ..... 15
   - 3.3 Das Schematakonzept ..... 16
     - 3.3.1 Grundlagen des Schematakonzeptes ..... 16
     - 3.3.2 Das Schematakonzept und Selektion ..... 17
     - 3.3.3 Das Schematakonzept und Rekombination ..... 18
     - 3.3.4 Das Schematakonzept und Mutation ..... 19
     - 3.3.5 Das Schematatheorem und die Building Block Hypothese ..... 20

4. Funktion und Arbeitsweise von Genitor ..... 21
   - 4.1 Unterschiede zwischen Genitor und herkömmlichen genetischen Algorithmen ..... 21

4.2 Programmentwicklung unter Genitor ..... 22

5. Anwendung genetischer Algorithmen auf das Problem der Tourenplanung ..... 25
5.1 Codierung ..... 25
5.2 Permutations-Operatoren ..... 26
5.2.1 Der PMX-Operator ..... 27
5.2.2 Der Order1-Operator ..... 28
5.2.3 Der Order2-Operator ..... 28
5.2.4 Der Position-Operator ..... 29
5.2.5 Der Cycle-Operator ..... 29
5.2.6 Der Edge-Recombination-Operator ..... 30
5.3 Mutationsoperatoren ..... 32

6. Genetischer Algorithmus von Blanton / Wainwright zur Lösung des Tourenproblems ..... 33
6.1 Problemcodierung ..... 33
6.2 Neue Familie von Cross Over-Operatoren ..... 34
6.2.1 Merge Cross #1 Operator ..... 34
6.2.2 Merge Cross #2 Operator ..... 35

7. Genetische Algorithmen und lokale Suche ..... 37
7.1 Hybride genetische Algorithmen ..... 37
7.2 Genetic Sectoring Heuristic ..... 37

8. Eigener hybrider Algorithmus ..... 39

9. Implementation der Algorithmen .....
9.1 Die Heuristiken .....
9.2 Die klassischen genetischen Algorithmen .....
9.3 Die hybriden Algorithmen .....

10. Experimentelle Auswertung .....
10.1 Problemgenerierung.....
10.2 Performancemessung genetischer Algorithmen.....
10.3 Experiment 1: Vergleich der Cross Over-Operatoren .....
10.4 Experiment 2: Einsatz von Mutation .....
10.5 Experiment 3: Nutzen hybrider Techniken.....
10.6 Experiment 4: Vergleich zwischen Heuristik und genetischem Algorithmus mit verschiedenen Parametern .....
10.7 Bewertung der Experimente.....

11. Weitere Anwendungsgebiete für den entwickelten Ansatz .....

12. Zusammenfassung.....

Anhänge

Anhang 1 Programmlisting Insertion Heuristik.....
Anhang 2 Programmlisting zur Problemerzeugung.....
Anhang 3 Programmlisting Genitor Hauptprogramm .....
Anhang 4 Programmlisting Genitor Zielfunktion .....
Anhang 5 Programmlisting Merge Cross #1 Operator.....
Anhang 6 Programmlisting Merge Cross #2 Operator.....
Anhang 7 Codierung eines Beispielproblems.....
Anhang 8 Ausgabe der Insertion Heuristik.....
Anhang 9 Ausgabe von Genitor.....
Anhang 10 Bildschirmausgabe von Genitor.....

Literaturverzeichnis.....

# 1. Einleitung und Ziel der Arbeit

Die Tourenplannung ist ein in der Praxis häufig anzutreffendes Problem. Nicht erst seit Fluglinien mit vielen hundert Flugzeugen jeden Tag vor die Aufgabe gestellt sind, ihre knappen Ressourcen möglichst effizient einzusetzen, sondern auch alltägliche Probleme wie die Tourenplanung für Briefträger, Speditionen, Versandhäuser etc. erfordern eine effiziente Lösung.
Dagegen steht, daß die Tourenplanung NP-vollständig ist, und damit für große Probleme nur Heuristiken in Frage kommen. Die entstehenden Lösungen liegen meist weit vom globalem Optimum, eine weitere Optimierung wäre damit sinnvoll.
Die genetischen Algorithmen sind eine sehr allgemeine und robuste Optimierungsmethode und bieten sich für besonders schwere Probleme an. Sie finden bereits in einigen Bereichen Anwendung und es erscheint sinnvoll, genetische Algorithmen auf Tourenplanung mit Zeitfenstern anzuwenden, und die Ergebnissse mit Heuristiken zu vergleichen.
Es ist zu erwarten, daß genetische Algorithmen eine höhere Laufzeit als Heuristiken haben werden, entscheidend wäre allerdings eine verbesserte Lösung, weil selbst eine Verbesserung von 10 - 20% bereits eine deutliche Einsparung an Ressourcen wie Kapital, Rohstoffe und Arbeitszeit bedeuten würde.
Es ist Ziel der vorliegenden Arbeit, diese Frage zu untersuchen und Lösungsansätze und Vorschläge zu unterbreiten, um unter Umständen abschließend den Einsatz von genetischen Algorithmen für Tourenplanung mit Zeitfenstern zu empfehlen.
Die Arbeit gliedert sich im wesentlichen in drei Teile: Im ersten Teil wird das Problem der Tourenplanung mit Zeitfenstern eingeführt und typische Heuristiken zur Lösung vorgestellt.
Im zweiten Teil der Arbeit werden die genetischen Algorithmen beschrieben und ihre Funktionsweise erläutert. Außerdem wird das Entwicklungsystem Genitor, mit dem diese Untersuchung durchgeführt worden ist, vorgestellt.
Im dritten Teil werden dann die genetische Algorithmen auf die Tourenplanung angewendet. Die experimentellen Ergebnisse werden aufgezeigt und interpretiert. Abschließend sollen die Erkenntnisse dieser Untersuchung zusammengefaßt und bewertet werden.

# 2. Tourenplanung mit Zeitfenstern

## 2.1 Einführung in die Tourenplaung

Die Tourenplanung beschäftigt sich mit der Zuteilung einer Flotte von Fahrzeugen auf eine Menge geographisch verteilter Kunden. Die Fahrzeuge sind zentral in einem Depot stationiert, haben eine feste Kapazität und müssen nach Beendigung ihrer Route wieder in das Depot zurückkehren. Die Kunden beanspruchen eine bestimmte Transportkapazität und müssen in einem bestimmten Zeitrahmen (Time Window) beliefert werden.
Das Problem besteht darin Routenpläne für die Fahrzeugflotte so zu entwicklen, daß alle Kunden besucht werden und dabei möglichst kurze Routen entstehen.
Es wird davon ausgegangen, daß alle Fahrzeuge dieselbe Kapazität haben (homogene Fahrzeugflotte) und die Kunden nur innerhalb ihrer Zeitrahmen beliefert werden können (harte Time Windows).

Es gibt eine große Anzahl von praktischen Anwendungen für die Tourenplanung, wie z.B.:

- Flugroutenplanung
- Einsatzplanung für Speditionsunternehmen
- Postversand

Durch eine geschickte Routenplanung können wertvolle Ressourcen gespart und die wirtschaftliche Effizienz gesteigert werden. Die Einsparung einer Flugroute zum Beispiel vermindert die Umweltverschmutzung, spart Treibstoff, Personal- und Kapitalkosten etc. und läßt ein Unternehmen konkurrenzfähiger werden. Es ist daher verständlich, daß der Lösung des Tourenproblems großes Interesse entgegengebracht wird. Die Untersuchung von Bodin enthielt 1983 schon 700 Referenzen zu Tourenproblemuntersuchungen.[1]
Leider gibt es noch keine Patentrezepte für die Lösung von Tourenproblemen. Man stützt sich auf Verfahren, die gute Lösungen erbringen, aber nicht garantieren, die beste Lösung gefunden zu haben. Exakte Verfahren können nur für eine kleine

[1] L. Bodin, B. Golden, A. Assad, M. Ball: "Routing and Scheduling of Vehicles and Crews: The State of the Art"

Anzahl von Kunden durchgeführt werden und haben daher bisher keine praktische Relevanz.[2]

## 2.2 Mathematische Formulierung der Tourenplanung

Im folgenden wird eine mathematisch exakte Formulierung der Tourenplanung vorgenommen, die das Problem genau beschreibt und Hinweise auf die Komplexität des Problems liefert.[3]

$P = P^+ \cup P^-$, $P^+ = \{ i^+, i \in P \}$, $P^- = \{ i^- , i \in P \}$. P sei eine Menge von Kunden, so daß jeder Kunde $i \in P$ eine Transportkapazität $d_i$ von Knoten $i^+$ nach Knoten $i^-$ anfordert.

$[ a_i, b_i ]$ mit $i \in P$ sei das Zeitfenster für Kunde i.

$D$ sei die Fahrzeugkapazität. Alle Fahrzeuge haben die Kapazität D.

$I \subset ( P^+ \cup P^- ) \times (P^+ \cup P^-)$. I sei eine Menge von Kanten, so daß wenn $(i, j) \in I$ ein Fahrzeug von Kunde i zu Kunde j fahren kann.

$t_{ij}$ ist der Zeitaufwand, der benötigt wird, um von Kunde i zu Kunde j zu gelangen.

$c_{ij}$ sind die Transportkosten, um von Kunde i zu Kunde j zu gelangen.

Die Kante (i,j) ist nur definiert, wenn $a_i + t_{ij} \leq b_j$ mit $i, j \in P$ und $d_i + d_j \leq D$, also wenn die Restriktionen bezüglich Zeitrahmen der Kunden und Fahrzeugkapazität eingehalten werden.

$G = (N, A)$ sei ein Graph. N enthält die Knoten aus P und das Depot $\{0\}$. A sei eine Menge von Kanten mit $A \in I \cup ( \{0\} \times P^+ ) \cup ( P^- \times \{0\} )$.

$V$ sei die Menge der Fahrzeuge mit $v \in V$ als Fahrzeugindex.

---

[2] N. Christofides, A. Mingozzi, P. Toth: "Exact Algorithms for the Vehicle Routing Problem, based on Spanning Tree and Shortest Path Relaxation"
[3] M. Solomon: "Time Window Constrained Routing and Scheduling Problems: A Survey"

$X_{ij}^{v}$ sei eine binäre Variable, die gleich 1 ist, wenn Fahrzeug v von i nach j fährt, und die sonst Null ist.

$T_i$ sei eine Zeitvariable, und $Y_i$ sei eine Ladevariable.

**Formeln:**

(1) $$Min \quad \sum_{v \in V} \sum_{(i,j) \in A} c_{ij} X_{ij}^{v}$$

(2) $$\sum_{v \in V} \sum_{j \in N} X_{ij}^{v} = 1, \qquad i \in P^{+}$$

(3) $$\sum_{j \in N} X_{ij}^{v} - \sum_{j \in N} X_{ij}^{v} = 0 \qquad i \in P, v \in V$$

(4) $$\sum_{j \in N} X_{i^{+}j}^{v} = \sum_{j \in N} X_{ji^{+}}^{v} = 0, \qquad i \in P, v \in V$$

(5) $$T_{i^{+}} + t_{i^{+}i^{-}} \le T_{i^{-}}, \qquad i \in P$$

(6) $$X_{ij}^{v} = 1 \Rightarrow T_i + t_{ij} \le T_j, \qquad (i,j) \in I, v \in V$$

(7) $$a_i \le T_i \le b_i, \qquad i \in P$$

(8) $$X_{ij}^{v} = 1 \Rightarrow Y_i + d_i \le Y_j, \qquad (i,j) \in I, v \in V$$

(9) $$0 \le Y_i \le D, \qquad i \in P^{-}$$

(10) $$X_{ij}^{v} \; ist \; binär, \qquad (i,j) \in A, v \in V$$

Bedingungen (1)-(3) beschreiben die Minimierung eines Flußproblems für mehrere Fahrzeuge.
Bedingung (4) garantiert, daß das gleiche Fahrzeug $i^{+}$ und $i^{-}$ besucht.
Bedingung (5) fordert, daß $i^{+}$ vor $i^{-}$ besucht wird.
Bedingungen (6) und (8) sorgen dafür, daß innerhalb einer Route die Lade- und Zeitrestriktionen nicht verletzt werden.
Bedingungen (7) und (9) enthalten die Lade- und Zeitrestriktionen.

## 2.3 Komplexität der Tourenplanung

Es wird im folgenden bewiesen, daß die Tourenplanung mit Zeitfenstern NP-hart ist. Der Beweis erfolgt für das Traveling Salesman-Problem mit Zeitfenstern und läßt sich einfach auf Tourenprobleme mit einer Flotte von Fahrzeugen übertragen. Sollte das Erzeugen einer einzigen Route bereits NP-hart sein, gilt dies dann in besonderem Maße für n Routen.

**Theorem:** Das Problem, eine ausführbare Route für das Traveling Salesman Problem mit Zeitfenstern zu finden, ist NP-vollständig.[4]

**Beweis:**

Der Beweis geht vom 3-Partition-Problem aus, von dem man weiß, daß es NP-vollständig ist.[5]

**3-Partition:**

Instanz:

Eine endliche Menge A mit 3m Elementen, eine Schranke $B \in Z^+$ und eine "Größe" $s(a) \in Z^+ \; \forall \; a \in A$, mit

$$\sum_{a \in A} s(a) = mB.$$

Frage:

Kann A in m disjunktive Mengen $S_1, .., S_m$ zerlegt werden, so daß für $1 \leq i \leq m$ gilt:

[4] Savelsbergh, M.W.P.: "Local search in routing problems with time windows", S. 9
[5] Garey, M.S., Johnson, D.S.: "Computers and Intractability: A Guide to the Theory of NP-Completeness".

$$\sum_{a \in S_i} s(a) = B \quad ?$$

(Jedes $S_i$ muß also genau 3 Elemente von A enthalten)

Sei A = { $a_1$, ..., $a_{3m}$ }, es wird folgende Instanz des Traveling Salesman Problems mit Zeitfenstern konstruiert.

Es gibt 4m-1 Knoten mit den folgenden Zeitfenstern:

$1 \le i \le m$ : [ 0, mB]
$3m+1 \le i \le 4m-1$ : [(i-3m)B , (i-3m)B].

Die Distanzmatrix T = ($t_{ij}$) gibt die Entfernung zwischen den Knoten an und ist folgendermaßen definiert:

$$t_{i,j} = \begin{cases} s(a_i) & 1 \le i \le 3m; \quad 1 \le j \le 4m-1 \\ 0 & 3m+1 \le i \le 4m-1; \quad 1 \le j \le 4m-1 \end{cases}$$

Es wird angenommen, daß $a_i + a_j \ge a_k$ für alle Tripel i, j, k ist. Dies impliziert, daß für das Traveling Salesman Problem mit Zeitfenstern die Dreiecksungleichung gilt.

Man kann jetzt folgern, daß es nur eine Lösung für das Traveling Salesman Problem mit Zeitfenstern gibt, wenn es eine Partition der Menge A in 3m disjunkte Untermengen der Größe B gibt.

Damit ist bewiesen, daß das Traveling Salesman Problem mit Zeitfenstern NP-vollständig ist. Da das Tourenproblem mit Zeitfenstern mindestens ebenso schwer ist, kann es als NP-hart eingestuft werden. Der Einsatz von Heuristiken wird durch dieses Ergebnis gerechtfertigt.[6]

[6] Savelsbergh, M.W.P.: "Local search in routing problems with time windows", S. 10

## 2.4 Heuristiken zur Lösung des Tourenproblems

Die Kapitel 2.1 und 2.3 haben gezeigt, daß das Tourenproblem NP-vollständig ist und sich daher derzeit nicht mit vertretbarem wirtschaftlichem Aufwand exakt lösen läßt. Zur Lösung des Tourenproblems müssen also heuristische Verfahren herangezogen werden, das heißt Verfahren, die versuchen, mittels einer vernünftig erscheinenden Strategie, die aus Erfahrungswerten oder Plausibilitätsüberlegungen zusammengesetzt sein kann, eine "gute", aber meist nicht optimale Problemlösung herbeizuführen.

Die Lösung eines Tourenproblems beinhaltet die Lösung von zwei Teilproblemen:

- Zuordnungsproblem von Kunden zu einer Tour (Clusterung)
- Routingproblem für jede Tour (Bestimmung der kürzesten Rundreise für jeden Cluster)

Die heuristischen Verfahren lassen sich anhand dieser Teilprobleme klassifizieren:[7]

| Heuristische Verfahren für Standardprobleme | |
|---|---|
| Sukzessivverfahren | Parallelverfahren |

[7] Domschke, W.: "Logistik: ..", S. 138

| 1. Route first cluster second | 2. Cluster first route second | 3. Konstruktions-verfahren | 4. Verbesserungs-verfahren |
|---|---|---|---|
| vorwiegend für kantenorientierte Verfahren | vorwiegend für knotenorietierte Verfahren | Savingsverfahren | 2- oder 3-optimale Verfahren |

**Gruppe 1: (Route first - cluster second - Verfahren)**
Bei dieser Vorgehensweise wird erst die kürzeste Tour ohne Berücksichtigung von Zeit- und Platzrestriktionen mit allen Kunden ermittelt. Danach wird unter Berücksichtigung der Restriktionen diese "giant tour" in Teilstücke zerlegt.

**Gruppe 2: (Cluster first - route second)**
Hierbei werden zunächst Mengen von Kunden gebildet, die unter Berücksichtigung der Restriktionen in einer Tour bedient werden können und danach für jede Menge die kürzeste Route ermittelt. Das *Sweep-Verfahren* ist eines der bekanntesten Vertreter aus dieser Gruppe und wird später erläutert.

**Gruppe 3: (Konstruktionsverfahren)**
Tour und Route werden parallel zueinander entwickelt. Das Savings-Verfahren arbeitet nach diesem Prinzip und wird später vorgestellt.

**Gruppe 4: (Verbesserungsverfahren)**
Verbesserungsverfahren gehen von einer gegebenen Tour aus und verbessern sie im Hinblick auf die zu verfolgende Zielsetzung. Diese Verfahren werden hier nicht weiter behandelt.

### 2.4.1 Das Sweep-Verfahren

Das Sweep-Verfahren gehört zur Gruppe 2 (cluster first - route second) und wurde 1974 von Gillet und Miller entwickelt.
Ausgehend von einem euklidischen Koordinatensystem muß der Standort des Depots im Ursprung liegen und die Position der Kunden in Form von Koordinaten

($x_i$ , $y_i$) angegeben werden. Die Entfernung $d_{ij}$ zwischen Kunde i und Kunde j wird dann durch

$$d_{ij} := \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

bestimmt. Die Kunden werden dann nach aufsteigendem Polarwinkel φ gegen den Urzeigersinn sortiert und von 1 bis n numeriert.
Es folgt eine Iteration von 1 bis n, indem die Kunden unter Beachtung der Restriktionen in der Reihenfolge i, i+1, ..., n, 1, ..., i-1 in die einzelnen Touren aufgenommen werden.

Anschließend werden die einzelnen Touren noch mit einem beliebigen Verfahren, wie z.B. 2-opt und 3-opt optimiert. Der beste Gesamtplan von den n Iterationen ist das Ergebnis.

### 2.4.2 Das Savings-Verfahren

Das Savingsverfahren gehört zu Gruppe 3 (Konstruktionsverfahren) und wurde 1964 von Clarke und Wright vorgestellt. Es ist das in der Praxis am meisten diskutierte und eingesetzte Verfahren. Es gehört zu den *Parallelverfahren*, weil die Bildung und Veränderung von Touren und Routen parallel zueinander erfolgt.[8]
Das Verfahren wird durch n Pendeltouren initialisiert, das bedeutet es gibt n Touren der Form [0,i,0].
Anschließend wird versucht, Touren unter Beachtung der Restriktionen zusammenzulegen, indem der Anfang bzw. das Ende der jeweiligen Tour miteinander verknüpft wird. Die Vereinigung zweier Touren ergibt ein *Saving* (eine Ersparnis) in Höhe von

$$s_{ij} := d_{0i} + d_{0j} - d_{ij}$$

$s_{ij}$ ist um so größer, je näher i und j beieinander liegen und je weiter sie vom Depot entfernt sind. Es werden immer diejenigen Touren miteinander verknüpft, welche die größte Ersparnis liefern.[9]

---

[8] Domschke, W.: "Logistik: ..", S. 143
[9] Domschke, W.: "Logistik: ..", S. 144

### 2.4.3 Die Insertion Heuristik

Die Insertion Heuristik gehört zu den sequentiellen Verfahren und wurde von Mole und Jameson 1976 entwickelt. Solomon[10] hat in seiner Veröffentlichung von 1987 nachgewiesen, daß die Insertion Heuristik den beiden erst genannten klassischen Verfahren überlegen ist, sie soll aus diesem Grund als Qualitätsmaßstab für die genetischen Algorithmen dienen und hier auch näher erläutert werden. Ziel wird es sein, mit Hilfe der genetischen Algorithmen die Ergebnisse der Insertion Heuristik noch zu übertreffen.

Ausgehend von einem Initialisierungskriterium wird eine erste Route mit einem Kunden besetzt, und anhand von zwei weiteren Kriterien wird nach jeder Iteration ein neuer Kunde zu der Tour hinzugefügt, bis eine neue Tour eröffnet werden muß. Das Verfahren terminiert, wenn alle Kunden zugeteilt sind.

Als *Initialisierungskriterium* stehen zwei Kriterien zur Auswahl, die je nach Problemart eingesetzt werden können. Es kann als erstes der weiteste noch nicht zugeteilte Kunde eingefügt werden, oder ein noch nicht eingefügter Kunde, der das früheste Zeitfenster hat.

Es sei $(i_0, i_1, i_2, ..., i_m)$ die aktuelle Route mit $i_0 = i_m = 0$ ( = Depot).[11]
Für jeden noch nicht zugeteilten Kunden u wird nach Kriterium 1 die beste Einfügestelle in der Tour bestimmt.

*Kriterium 1:* $C_1 ( i(u), u, j(u)) = \min [ C_1( i_{p-1}, u, i_p)], \; p = 1, ..., m$

Als nächstes wird unter den noch einfügbaren Kunden der beste anhand des Kriteriums 2 ermittelt.

*Kriterium 2:* $C_2 ( i(u^*), u^*, j(u^*)) = \text{optimum} [ C_2 (i(u), u, j(u))]$

Kunde u* wird dann zwischen i (u*) und j(u*) eingefügt. Gibt es keine einfügbaren Kunden wird eine neue Route gestartet, sind alle Kunden verteilt terminiert das Verfahren.

---

[10] Solomon, M.: "Algorithms for the Vehicle Routing ...", S. 254
[11] Solomon, M.: "Algorithms for the Vehicle Routing ...", S. 254

Solomon[12] präsentiert drei konkrete Umsetzungen der oben genannten Kriterien wobei sich das erste als überlegen herausgestellt hat. Es wird daher im folgenden nur die erste Umsetzung genauer erläutert.

$$C_{11}(i,u,j) = d_{iu} + d_{uj} - \mu d_{ij}, \mu \geq 0$$
$$C_{12}(i,u,j) = b_{j_u} - b_j$$

$b_{j_u}$ ist die neue Servicezeit bei Kunden j wenn Kunde u eingefügt worden ist.

$$C_1(i,u,j) = \alpha_1 C_{11}(i,u,j) + \alpha_2 C_{12}(i,u,j)$$
$$\alpha_1 + \alpha_2 = 1, \alpha_1 \geq 0, \alpha_2 \geq 0$$
$$C_2(i,u,j) = \lambda d_{0u} - C_1(i,u,j), \lambda \geq 0$$

Diese Art von Strategie bevorzugt den Kunden, dessen gewichtete Zeit-Distanz-Kombination, die benötigt wird, um ihn in eine Tour aufzunehmen, minimal ist. Die Gewichtung kann individuell nach Problemart durch $\alpha$, $\mu$ und $\lambda$ vorgenommen werden.
Die Implementation dieser Heuristik in der Programmiersprache C ist im Anhang einzusehen.

[12] Solomon, M.: "Algorithms for the Vehicle Routing ...", S. 257

# 3. Grundlagen genetischer Algorithmen

In diesem Kapitel sollen die Grundlagen der genetischen Algorithmen eingeführt werden. Es werden zunächst die grundlegenden Definitionen und Anschauungen vermittelt, was genetische Algorithmen sind und wie sie funktionieren. Es wird dabei auf die wesentlichen Bestandteile eines genetischen Algorithmus eingegangen und Zusammenhänge beispielhaft erläutert. Anschließend soll eine theoretische Fundierung den Stoff untermauern.

## 3.1 Idee genetischer Algorithmen

Es gibt eine große Klasse interessanter Probleme für die bisher noch keine ausreichend schnellen Algorithmen gefunden worden sind. Viele dieser Probleme sind Optimierungsprobleme und sind in verschiedenen Anwendungen zu finden. Für einige dieser besonders harten Optimierungsprobleme, lassen sich probalistische Algorithmen finden, die zwar nicht die optimale Lösung garantieren, aber durch wiederholte Anwendung den Fehler beliebig klein werden lassen.[13]
Genetische Algorithmen sind stochastische Algorithmen, deren Suchmethode sich nach einem natürlichen Phänomen, nämlich das der genetischen Vererbung und der natürlichen Auswahl richtet.
Die Beschäftigung mit genetischen Algorithmen bringt es mit sich, daß viele Begriffe aus der Biologie Anwendung finden. Man spricht von Individuen oder Genotypen, auch als Chromosome bezeichnet, und meint damit eine potentielle Lösung des Optimierungsproblems.
Ein Evolutionsprozeß auf einer Population von Chromosomen entspricht der Suche innerhalb des Lösungsraumes. Bei der Suche entsteht der Zielkonflikt zwischen der Weiterentwicklung einer guten Lösung und der möglichst weiträumigen Erforschung des Lösungsraumes. Die Weiterentwicklung einer guten Lösung verspricht auf weitere noch bessere Lösungen zu stoßen und damit von dem Wissen innerhalb der Population zu profitieren, birgt aber die Gefahr in sich, zu früh in einem möglicherweise lokalem Optimum hängen zu bleiben und somit das globale Optimum nicht mehr finden zu können.[14] Die Genetischen Algorithmen

[13] Michalewicz, Z.: "Genetic Algorithms + Data Structures = Evolution Programs", S. 13
[14] Booker L. : "Improving Search in Genetic Algorithms", S. 1

bewegen sich dabei auf einem scheinbar fast optimalen Weg zwischen diesen beiden Methoden.[15]

Ausgehend von einer unter Umständen zufällig gewählten Anzahl von Chromosomen, der Population, wird durch die Mechanismen von Selektion, Rekombination und Mutation die gewünschte Bewertungsfunktion optimiert. Die Population gewinnt durch diesen Ausleseprozeß an Qualität, was die Wahrscheinlichkeit für gute Lösungen, mit jeder Iteration erhöht.
Es ist dazu notwendig, einerseits das Problem in einem Chromosom, meist ein binärer String, zu codieren, und andererseits eine Bewertungsfunktion zu haben, die einem Chromosom einen Qualitätswert, die sogenannte Fitness, zuweist. Die Selektion operiert auf der Population, indem aus zwei besonders guten Individuen ein neues entsteht, das über eine möglichst noch bessere Fitness verfügt. Die Mutation dagegen verändert einzelne Chromosome an ausgewählten Stellen zufällig, um dadurch noch unentdeckte Ecken des Suchraumes zu erforschen.

Die Struktur eines genetischen Algorithmus läßt sich folgendermaßen darstellen:

```
Procedure  Genetic_Algorithm
    Begin
        Current_Generation := 0;
        Initialisiere_Population ( Pop (Current_Generation));
        Bewerte_Population ( Pop (Current_Generation));
        While ( not Current_Generation = Max_Generation) do
            Current_Generation := Current_Generation + 1;
            Select_Parents (Pop (Current_Generation -1);
            Crossing_over (Parents);
            Mutation (Child);
            Bewerte (Child);
            Insert (Child , Pop (Current_Generation));
        End;
    End;
```

Abb. 1: Struktur genetischer Algorithmen

Goldberg[16] stellt vier wesentliche Unterschiede von genetischen Algorithmen zu traditionellen Methoden fest:

[15] Holland, J. : "Adaption in Natural and Artificial Systems"

- Genetische Algorithmen benutzen Codierungen von Lösungsmengen, nicht die Lösungsmengen selbst.
- Genetische Algorithmen suchen ausgehend von einer Menge von möglichen Lösungen und nicht von einem einzigen möglichen Lösungungspunkt aus.
- Genetische Algorithmen benötigen nur die Zielfunktion zur Lösungsfindung, andere Informationen werden nicht gebraucht.
- Genetische Algorithmen benutzen probalistische Übergangsregeln und sind daher nicht-deterministisch.

## 3.2 Die genetischen Operatoren

Ausgehend von der vorgestellten Struktur sollen jetzt die genetischen Operatoren näher erläutert werden.

### 3.2.1 Selektion

Bei der Selektion werden Chromosome zufällig ausgewählt, wobei Chromosome mit höherer Fitness auch eine höhere Wahrscheinlichkeit erhalten, ausgewählt zu werden. Die Selektion ist eine künstliche Anwendung des Darwinschen Gesetzes vom Überleben der best angepaßten Individuen. Die Selektion kann auf unterschiedliche Weise implementiert werden. Eine Methode ist die Auswahl durch ein gewichtetes Roulette. Die Größe der Roulettefelder für die Individuen entsprechen ihrer Fitness. Ein gutes Individuum wird dadurch häufiger ausgewählt als ein weniger gutes.[17]

### 3.2.2 Rekombination

[16] Goldberg, D.E. : "Genetic Algorithms in Search, Optimization, and Machine Learning", S. 7
[17] Goldberg, D.E. : "Genetic Algorithms in Search, Optimization, and Machine Learning", S. 11

Nach der Selektion erfolgt die Rekombination. Es werden zwei Individuen zufällig ausgewählt und eine Zufallszahl, der Rekombinationspunkt, generiert. Die beiden Individuen tauschen ab diesem Punkt ihre nachfolgendenen Zeichen (Allele).

**Beispiel:**

10010101 1. Individuum
01001010 2. Individuum
o.B.d.A. sei der Rekombinationspunkt = 3

100|01010 1. Individuum nach Rekombination
010|10101 2. Individuum nach Rekombination

### 3.2.3 Mutation

Durch die zumeist zufällige Initialisierung der Anfangspopulation kann unter Umständen ein wichtiger Teil des Suchraums ausgegrenzt und das globale Optimum unter Umständen gar nicht durch Selektion und Rekombination erreichbar sein. Die Mutation bringt ein neues Individuum in die Population, indem zufällig an einer Position des Repräsentationsstrings eine Veränderung vorgenommen wird. Es wird dabei keine Rücksicht auf die Fitness genommen. Sollte ein gutes Individuum dabei entstehen, wird sich dieses auch bei der Selektion in der nächsten Generation behaupten und kann unter Umständen neue Teile des Suchraumes erschließen. Schlechte Individuen werden von der Selektion weniger berücksichtigt und verschwinden wieder.

**Beispiel:**

1001**0**100010 Individuum
Zufallszahl = 5

1001**1**100010 mutiertes Individuum

## 3.3 Das Schemakonzept

### 3.3.1 Grundlagen des Schemakonzeptes

Das theoretische Fundament für genetische Algorithmen liefert die binäre Kodierung von Lösungen zusammen mit Schemata, die es erlauben, Ähnlichkeiten zwischen Chromosomen festzustellen.
Es wird ein besonderes Symbol * eingeführt, das stellvertretend für alle Symbole aus dem Kodierungsalphabet steht. Das Schema ( * * 1 1 ) steht zum Beispiel für alle Strings der Form 0011, 0111, 1011, 1111. Das Schema ( 1 1 0 1 ) repräsentiert nur den String 1101, und das Schema ( * * * * ) repräsentiert alle Strings der Länge 4.
Die Anzahl der * Symbole determiniert die Anzahl der Strings, die durch das Schema repräsentiert werden.

Es gibt zwei wichtige Eigenschaften von Schemata:[18]

- Die *order* gibt die Anzahl der festen Positionen an[19], z.B.
  *order* ( * * * 1 0 * * 1 1 0 1 ) = 6
  Die *order* zeigt damit an, wie spezialisiert ein Chromosom ist.

- Die *definierte Länge* eines Schemas gibt die Distanz zwischen der ersten und letzten festen Position an, z.B.
  $\delta$ ( * * * 1 0 * * 1 1 0 1 ) = 11 - 4 = 7
  Die definierte Länge gibt über die Kompaktheit an Information in Chromosomen Auskunft.

[18] Michalewicz, Z.: "Genetic Algorithms + Data Structures = Evolution Programs", S. 44
[19] im Fall der binären Kodierung, die Anzahl der Nullen und Einsen

Sei $\xi$ ( S , t ) die Anzahl der Strings in einer Population zu einem Zeitpunkt t, die durch das Schema S repräsentiert werden.
Sei *eval* ( S , t ) die durchschnittliche Fitness der durch S zum Zeitpunkt t repräsentierten Strings.

$$eval(S,t) = \frac{\sum_{j=1}^{p} eval(v_{i_j})}{p} \qquad mit\ \Theta_{i_1},\ldots,v_{i_p}\ \tau werden\ durch\ S\ repräsentiert$$

$$und\ \ p = \xi(S,t)$$

### 3.3.2 Das Schematakonzept und die Selektion

Während der Selektion werden Chromosome je nach Wahrscheinlichkeit für die nächste Generation ausgewählt. Die Wahrscheinlichkeit, für String $v_i$ ausgewählt zu werden, beträgt

$$prob_i = \frac{eval(v_i)}{F(t)} \quad mit\ \ F(t) = \sum_{i=1}^{popsize} eval(v_i).$$

F(t) gibt die Gesamtfitness der Gesamtpopulation an.
Die Wahrscheinlichkeit für einen String, der durch Schema S repräsentiert wird, beträgt demnach

$$prob_S = \frac{eval(S,t)}{F(t)}$$

Man kann nun die Anzahl der Strings, die durch Schema S in Generation t + 1 repräsentiert werden, ausdrücken durch:

$$\xi(S,t+1) = \xi(S,t) * popsize * \frac{eval(S,t)}{F(t)}.$$

Die Anzahl der Strings, die durch Schema S repräsentiert werden, wachsen im Verhältnis zur Fitness des Schemas durch die Fitness der Population. Die Anzahl der Strings, die durch ein überdurchschnittlich gutes Schema repräsentiert werden, wächst dadurch schneller, als die Anzahl der Strings, die nur durch ein unterdurchschnittlich gutes Schema repräsentiert werden.[20]

[20] Michalewicz, Z.: "Genetic Algorithms + Data Structures = Evolution Programs", S. 46

Der Langzeiteffekt ergibt sich, wenn man annimmt, daß Schema S mit ε % über der durchschnittlichen Fitness der Population liegt.

$$\xi ( S , t ) = \xi ( S , 0 ) * ( 1 + \varepsilon )^t$$

Ein überdurchschnittliches Schema erhält in den nächsten Generationen also exponentiell viele Strings.

### 3.3.3 Das Schematakonzept und Rekombination

Die Selektion führt keine neuen Punkte im Suchraum ein, es wird dadurch notwendig, die Auswirkungen der Rekombination auf das Schema zu untersuchen.
Die Wahrscheinlichkeit, daß bei der Rekombination ein Schema S zerstört wird, beträgt

$$p_d = \frac{\delta(S)}{m-1}.$$

Der Rekombinationspunkt liegt gleichverteilt zwischen 1 und m. Eine hohe definierte Länge impliziert also eine hohe Wahrscheinlichkeit, daß das Schema zerstört wird. Die Wahrscheinlichkeit, daß ein Schema überlebt, ist $p_s = 1 - p_d$.
Da nicht jedes Chromosom einer Reproduktion unterliegt, wird eine Auswahlwahrscheinlichkeit $p_c$ angenommen. Die Überlebenswahrscheinlichkeit für Schema S beträgt dann

$$p_s(S) = 1 - p_c * \frac{\delta(S)}{m-1} .$$

Die Effekte aus Selektion und Rekombination zusammengenommen ergeben dann

$$\xi(S,t+1) = \xi(S,t) * popsize * \frac{eval(S,t)}{F(t)} * \left[1 - p_c * \frac{\delta(S)}{m-1}\right].$$

Die durch Selektion erweiterte Formel gibt die zu erwartende Anzahl von Strings an, die in der nächsten Generation von Schema S repräsentiert werden. Die Anzahl hängt von der relativen Fitness und der definierten Länge von Schema S ab. Die

überdurchschnittlichen Schemata mit kurzer definierter Länge vereinigen immer noch exponentiell viele Strings in den nächsten Generationen auf sich.[21]

### 3.3.4 Das Schematakonzept und Mutation

Der Mutationsoperator verändert mit Wahrscheinlichkeit $p_m$ eine Position innerhalb des Strings. Ein Schema überlebt die Mutation, wenn alle festen Positionen nicht mutiert werden. Die Funktion *order* gibt die Anzahl der Bits an, die nicht mutiert werden dürfen, ohne das Schema zu zerstören. Die Überlebenswahrscheinlichkeit des Schema S nach Mutation ist dann

$$p_s(S) = (1 - p_m)^{order(S)} \quad mit\ p_m << 1 \quad ist\ p_s(S) \approx 1 - order(S) * p_m$$

Wird der Effekt der Mutation zu Selektion und Mutation hinzugefügt, ergibt sich folgende Formel:

$$\xi(S, t+1) = \xi(S,t) * popsize * \frac{eval(S,t)}{F(t)} * \left[1 - p_c * \frac{\delta(S)}{m-1} - order(S) * p_m\right]$$

Die Anzahl der repräsentierten Strings in der nächsten Generation hängt also von der relativen Fitness von Schema S, der *definierten Länge* und der *order* ab. Die Anzahl wächst exponentiell für Schemata mit überdurchschnittlicher Fitness, kleiner *definierten Länge* und kleiner *order*.

[21] Michalewicz, Z.: "Genetic Algorithms + Data Structures = Evolution Programs", S. 49

### 3.3.5 Schematheorem und Building Block Hypothese

**Schema Theorem:**[22]

> Schemata mit kurzer *definierter Länge* und geringer *order* bei überdurchschnittlicher Fitness repräsentieren exponentiell viele Strings von Generation zu Generation.

**Building Block Hypothese:**[23]

> Ein genetischer Algorithmus strebt durch die Kombination von kurzen Schemata mit geringer *order* und hoher Fitness, genannt **Building Blocks**, möglichst optimale Lösungen an.

Die Building Block Hypothese wurde zwar schon durch eine große Anzahl von praktischen Experimenten bestätigt, eine abschließende und umfassende Theorie konnte aber bisher noch nicht gefunden werden.

[22] Michalewicz, Z.: "Genetic Algorithms + Data Structures = Evolution Programs", S. 51
[23] Michalewicz, Z.: "Genetic Algorithms + Data Structures = Evolution Programs", S. 51

# 4. Funktion und Arbeitsweise von Genitor

## 4.1 Unterschiede zwischen Genitor und herkömmlichen genetischen Algorithmen

Genitor steht für GENetic ImplemenTOR und unterscheidet sich im wesentlichen in zwei Punkten von herkömmlichen genetischen Algorithmen.[24] Zum einen verwendet Genitor zur Selektion statt der relativen Fitness den Rang eines Individuums, und zum anderen erfolgt der Übergang von einer Generation zur nächsten durch Erzeugung nur eines neuen Individuums. Eltern und Nachkommen existieren dadurch gleichzeitig. Schemata mit hoher definierter Länge werden dadurch von Genitor nicht in der Weise benachteiligt wie bei herkömmlichen genetischen Algorithmen.

Die Verwendung des Ranges anstatt der Fitness eines Individuums macht für die Selektion keinen Unterschied und hat den Vorteil, daß nicht nach jeder Generation die relative Fitness der gesamten Population neu berechnet werden muß. Das neue Individuum muß nur nach Rang in die Population eingeordnet werden.

Es gibt jedoch noch weitere Vorteile, die die Überlegenheit von Genitor in bezug auf viele Anwendungen ausmachen.

Bei der genetischen Suche können zwei Probleme auftauchen, die die Performance erheblich beeinflußen können. Es kann durch Superindividuen zu einer verfrühten Konvergenz kommen und dadurch nur ein lokales Optimum gefunden werden. Zum anderen kann die Suche durch fehlenden Selektiven Druck uneffektiv werden.[25]

John Greffenstette versuchte mit seinen Programmpaket *Genesis*[26] dieses Problem mit Hilfe von "scaling windows" zu lösen. Die Parameter wie Selektions- und Mutationswahrscheinlichkeit oder Populationsgröße werden je nach Fortschritt der Suche modifiziert

Diese Vorgehensweise erscheint nicht konsistent mit der Theorie genetischer Algorithmen und wird durch die Einführung des Ranges vermieden.

Durch den Rang kann es keine Superindividuen geben, da es nicht zu hohen relativen Fitnessunterschieden kommt. Der selektive Druck bleibt durch den individuellen Generationsübergang erhalten.

---

[24] Whitley, D. : "The Genitor Algorithm and .... ", S. 118
[25] Whitley, D. : "The Genitor Algorithm and .... ", S. 117
[26] Grefenstette J. : "A user´s guide to GENESIS", Tech. Report CS-84-11, Vanderbuilt University

Durch die Entwicklung von Genitor wurde es möglich, die Anzahl der Parameter für die genetische Suche zu reduzieren. Die Populationsvielfalt und der genetische Druck kann mit Genitor direkt kontrolliert werden.[27]

## 4.2 Programmentwicklung unter Genitor

Genitor wurde 1990 von Darrel L. Whitley an der Colorado State University entwickelt. Es handelt sich um ein modular aufgebautes und dadurch einfach zu bedienendes Programmpaket.[28] Es sind Kenntnisse in Unix und C notwendig, da zur Implementation der individuellen Zielfunktionen C-Routinen modifiziert werden müssen. Auch die Entwicklung neuer genetischer Operatoren erfolgt in C.
Im Paket ist ein Manual enthalten, das die Anpassung an den jeweiligen Rechner erleichert. Eine Reihe von Beispielproblemen wie z.B. Traveling Salesman steht fertig implementiert zur Verfügung und ist sofort lauffähig. Auch eine Reihe von genetischen Operatoren wie Edge Recombination sind im Lieferumfang enthalten.
Genitor bietet drei verschiedene Datenstrukturen zur Codierung der Anwendung :

- BIN
- FLOAT
- INT

Für die Tourenprobleme werden in erster Linie Integercodierungen benutzt.
Die Implementierung einer Anwendung besteht aus der Zusammenstelltung der notwendigen Routinen aus der Genitor Bibliothek. So werden die Routinen zur Populationsverwaltung, für Zufallszahlen und Selektion, sowie die verschiedenen genetischen Operatoren in das Hauptprogramm importiert, es ist meistens nur die Kenntnis der Schnittstellen, nicht aber die Kenntnis der Routine notwendig.
Das Hauptprogramm gliedert sich in drei Teile: Zuerst die Initialiserung der Variablen, dann die Iterationsschleife über die Anzahl der Generationen und zuletzt die Ausgabe des Ergebnisses.[29]

---

[27] Whitley, D. : "The Genitor Algorithm and .... ", S. 117
[28] Genitor ist im Internet kostenlos unter ftp 129.82.102.183 zu bekommen.
[29] Ein Beispiel für ein typisches Hauptprogramm befindet sich im Anhang

Für den Aufruf von Genitor werden einige Parameter benötigt, die im folgenden beschrieben werden.

| | |
|---|---|
| Poolsize | 120 |
| Length_String | 25 |
| Number_Trials | 25000 |
| StatusInterval | 1500 |
| RandomSeed | 12345678 |
| Bias | 1.9000000 |
| MutateRate | 0.0000000 |
| InitPool | dump.pool |
| FinalPool | final.pool |

Die Parameter werden z.B. in dem Textfile Test.config abgespeichert und Genitor mit *GenitorINT -c Test.config* aufgerufen. GenitorINT zeigt, daß die Datenstruktur Integer verwendet wird.

*Poolsize* bestimmt die Größe der Population, z.B. 120 Individuen.

*Length_String* gibt die Länge eines Chromosoms an, bei einem Traveling Salesman Problem z.B. die Anzahl der zu besuchenden Städte.

*Number_Trials* ist die Anzahl der Generationen, die berechnet werden sollen.

*StatusIntervall* zeigt an, wie oft ein Zwischenbericht über den Zustand der Population ausgegeben werden soll, z.B. alle 1500 Generationen. Der Bericht sieht folgerndermaßen aus:

10000 | Bst: 21.43000 Wst: 17.53400 Mean: 19.87600 Avg: 19.99301

Es sind bereits 10000 Generationen erfolgt, das beste Individuum hat einen Wert von 21.43, das schlechteste 17.534.

*RandomSeed* initalisiert den Zufallsgenerator.

*Bias* gibt das Wahrscheinlichkeitsgefälle zwischen dem besten und schlechtesten Individuum an. Ein hoher Wert bevorzugt die guten Individuen und birgt die Gefahr von zu früher Konvergenz in sich. Ein niedriger Wert führt zu vermindertem genetischen Druck und verlangsamt die Suche.

*MutateRate* zeigt die Mutationswahrscheinlichkeit an.

Mit *InitPool* kann man mit einer bestimmten Anfangspopulation starten. Dies ist der Fall, wenn man eine abgebrochene Iteration fortsetzen, oder nicht mit einer Zufallspopulation starten möchte.

Mit *FinalPool* kann man die letzte Generation unter angegebenem Namen abspeichern.

# 5. Anwendung genetischer Algorithmen auf das Tourenproblem

## 5.1 Codierung

Ein Teilproblem der Tourenplanung ist es, die Kunden möglichst geschickt auf die Fahrzeuge zu verteilen. Es muß dadurch auch eine möglichst einfache und für genetische Algorithmen gut geeignete Art der Codierung des Problems gefunden werden.
Zur Lösung des bekannten Traveling Salesman Problems haben sich ganzzahlige Vektoren bewährt:

( 3 , 5 , 2 , 1 , 4 )

Der Vektor zeigt, daß zuerst Stadt 3, dann die Städte 5, 2, 1 und zuletzt Stadt 4 besucht werden. Diese Art der Codierung hat den Vorteil, daß keine Städte doppelt vorkommen können und somit keine ungültigen Lösungen entstehen können. Es müssen lediglich Permutationen des Vektors ausprobiert werden. Eine binäre Codierung wäre dagegen mit Nachteilen verbunden.
Das Tourenproblem mit einer Anzahl von Fahrzeugen erfordert eine noch weiter gehende Codierung, da nicht nur die Tour eines Handelsreisender behandelt werden muß, sondern die Touren einer ganzen Flotte von Fahrzeugen gleichzeitig optimiert werden soll.
Es entsteht ein sogennantes Partitionsproblem, für das es bereits einige Lösungsansätze gibt.
Es werden im folgenden drei Lösungsansätze vorgestellt.[30]

Die erste Methode benötigt zur Aufteilung einen Vektor von ganzen Zahlen. Die i-te Zahl ist die Nummer der Gruppe von Objekt i.

Beispiel: {AE} {CD} {BF} wird repräsentiert durch
( 1 , 3 , 2 , 2 , 1 , 3 )

Die zweite Methode benötigt einen Gruppenseparator zur Trennung der Partitionen.

[30] Jones, R., Beltramo M.: "Solving Partitioning Problems with Genetic Algorithms", S. 442

Beispiel: {AE} {CD} {BF} wird repräsentiert durch
( 1 , 5 , 7 , 3 , 4 , 8 , 2 , 6 )

Die Sieben und die Acht stehen dabei für die Gruppenseparatoren.

Die dritte Methode verwendet eine Greedy Heuristik zur Aufteilung. Die Greedy Heuristik wird zur Decodierung der Permutation verwendet. Dabei werden die k Gruppen mit den ersten k Objekten initialisiert und danach die folgenden Objekte der Gruppe so zugeteilt, daß die Zielfunktion optimiert wird.

Beispiel: { **A C D** B E F } wird partiniert in
{ **A** F } { **C** B } { **D** E } ( Zielfunktion willkürlich )

Methode Zwei hat den Nachteil, daß nicht alle Permutationen eine sinnvolle Partition ergeben. Zum Beispiel könnte es vorkommen, daß zwei Gruppenseparatoren nebeneinander stehen und es weniger als k Gruppen gibt. Es muß daher immer noch einen Kontrollmechanismus geben, der ungültige Lösungen ausfiltert.
Bei Methode Eins sind viele Crossover Operatoren nicht möglich. Methode Drei hat sich als die leistungsfähigste herausgestellt. Es sind keine ungültigen Lösungen möglich und alle Crossover-Operatoren können ohne Modifikation angewendet werden.[31]
Bei den später folgenden Experimenten wird die Greedy Heuristik zur Codierung der Tourenprobleme verwendet.

## 5.2 Permutationsoperatoren

Im folgenden werden die für Tourenprobleme wichtigsten und relevanten Crossover Operatoren vorgestellt. Es werden nur Operatoren vorgestellt, die für Traveling Salesman Probleme und damit auch für Tourenplanungen geeignet sind, d.h. es entstehen nur gültige Nachkommen aus der Anwendung der Operatoren.

[31] Jones, R., Beltramo M.: "Solving Partitioning Problems with Genetic Algorithms", S. 445

Binäre Operatoren werden nicht vorgestellt und sind auch aufgrund der Problemcodierung nicht brauchbar.

In Experimenten sollen die Operatoren später auf ihre Tauglichkeit für Tourenprobleme hin untersucht werden.

### 5.2.1 Der PMX-Operator (Partially-Mapped Cross Over)

Die Idee beim PMX-Operator[32] besteht darin, die relative Anordnung der zu besuchenden Kunden beim resultierenden Chromosom möglichst beizubehalten. Dies führt dazu, daß hochwertige Individuen die gute relative Ordnung an ihre Nachkommen weitergeben können.
Dazu wird zunächst ein Teilstring aus einem Elternteil kopiert. Der Rest wird aus dem anderen Elternteil nach relativer Position aufgefüllt.

Beispiel:

1. Elternteil: ( 1 2 3 | 4 5 6 7 | 8 9 )
2. Elternteil ( 4 5 2 | 1 8 7 6 | 9 3 )

Der Teilstring 4-5-6-7 wird aus dem 1. Elternteil in in das resultierende Chromosom kopiert.

Kind ( - - - | 4 5 6 7 | - - )

Aus dem 2. Elternteil werden alle Werte direkt übernommen, die nicht im Teilstring vertreten sind.

Kind ( - - 2 | 4 5 6 7 | 9 3 )

Die fehlenden Stellen werden aus dem 1. Elternteil ergänzt und behalten auch hier ihre relative Position zueinander.

Kind ( 1 8 2 | 4 5 6 7 | 9 3 )

Das entstehende Chromosom ist wieder ein gültiger Lösungsvektor.

---

[32] Oliver, Smith, Holland: "A Study Of Permutation Crossover Operators On The Traveling Salesman Problem"

### 5.2.2 Der Order Cross Over-Operator

Ähnlich wie der PMX-Operator funktioniert der Order Cross Over-Operator[33]. Zunächst wird wieder ein Teilstring des 1. Elternteils direkt übernommen, und dann werden die restlichen Stellen, der relativen Ordnung, nach aus dem 2. Elternteil kopiert.

Beispiel:

| | |
|---|---|
| 1. Elternteil | ( 1 2 3 \| 4 5 6 7 \| 8 9 ) |
| 2. Elternteil | ( 4 5 2 \| 1 8 7 6 \| 9 3 ) |
| Kind | ( - - - \| 4 5 6 7 \| - - - ) |

Das Auffüllen der restlichen Stellen unterscheidet sich zum PMX-Operator dadurch, daß ab dem zweiten Strich beim zweiten Elternteil, die Zahlen ab der ersten Position beim resultierenden Chromosom eingeordnet werden, wenn sie nicht im bereits kopierten Teilsstring vorkommen. Die Folge der Zahlen wäre dann 9-3-4-5-2-1-8-7-6.

| | |
|---|---|
| Kind | ( 2 1 8 \| 4 5 6 7 \| 93 ) |

Auch bei diesem Operator wird versucht, bei hochwertigen Individuen die relative Position der Kunden möglichst beizubehalten.

### 5.2.3 Der Order Based Cross Over-Operator

Der Order Based Cross Over-Operator[34] versucht auf andere Weise die relative Ordnung beizubehalten.
Zuerst werden einige Stellen zufällig bestimmt.

---

[33] L. Davis: "Applying Adaptive Algorithms To Epistatic Domains"
[34] Syswerda, G.: "Schedule Optimization Using Genetic Algorithms"

Beispiel:

1. Elternteil ( 1 2 **3** **4** 5 **6** 7 8 **9** )
2. Elternteil ( 4 1 2 8 7 6 9 3 5 )

Die ausgewählten Positionen befinden sich im zweiten Elternteil an den Stellen 1, 6, 7, 8. Diese Positionen werden nun mit der Reihenfolge aus dem 1. Elternteil besetzt, also 3-4-6-9. Es entsteht wieder ein gültiges resultierendes Chromosom, nur die relativen Reihenfolgen der beiden Elternteile wurden ineinander vermischt.

Kind ( **3** 1 2 8 7 **4** **6** **9** 5 )

### 5.2.4 Der Position Based Cross Over-Operator[35]

Ähnlich wie beim Order Based Cross Over-Operator werden zunächst einige zufällige Stellen aus dem ersten Elternteil ausgewählt und dann aus dem zweiten Elternteil aufgefüllt.

Beispiel:

1. Elternteil ( 1 **2** 3 **4** 5 6 **7** **8** 9 )
2. Elternteil ( 4 5 2 1 8 7 6 9 3 )

Kind ( - 2 - 4 - - 7 8 - )

Das Auffüllen der restlichen Positionen verläuft anders als beim Order Based Cross Over-Operator. Die restlichen Stellen werden einfach von links nach rechts aus dem 2. Elternteil kopiert, wenn sie nicht schon bei den zufällig ausgewählten Zahlen vertreten waren.

Kind ( 5 2 1 4 6 9 7 8 3 )

### 5.2.5 Der Cycle-Operator

[35] Starkweather, T., et al.: "A Comparison of Genetic Sequencing Operators", S. 72

Grundgedanke beim Cycle Cross Over-Operator[36] ist es, daß innerhalb des neuen Lösungsstrings jede Stadt auf eine der beiden Positionen gesetzt wird, die sie schon innerhalb eines Elternteils belegt hat.

Beispiel:

| | |
|---|---|
| 1. Elternteil | ( 1 2 3 4 5 6 7 8 9 ) |
| 2. Elternteil | ( 4 1 2 8 7 6 9 3 5 ) |

Zu Beginn wird ein zufälliger Wert aus einem beliebigen Elternteil übernommen. Zum Beispiel die dritte Position vom 2. Elternteil.

| | |
|---|---|
| Kind | ( - - 2 - - - - - - ) |

Innerhalb des anderen Elternteils steht an dritter Position die 3. Die Positionen sollen möglichst erhalten bleiben, also wird die 3 entweder an Position 3 oder Position 8 geschrieben. Da Position 3 besetzt ist, wird Position 8 gewählt. Als nächstes wird die Acht von Elternteil 1 untersucht und in gleicher Weise verfahren. Das Verfahren endet, wenn alle Positionen besetzt sind und ein neues Individuum entstanden ist.

| | |
|---|---|
| Kind | ( 4 1 2 8 5 6 7 3 9 ) |

### 5.2.6 Der Edge Recombination-Operator

Der Edge Recombination-Operator[37] unterscheidet sich von den bisher genannten Operatoren darin, daß nicht mehr versucht wird, die relative Ordnung weiter zu vererben.
Vielmehr sollen die Verbindungen (Kanten) zwischen Kunden erhalten bleiben. Dadurch erzeugt der Edge Recombination-Operator Individuen in denen fast nur Kanten vorkommen, die auch in mindestens einem der Elternteile vorkommen.

Beispiel:

---

[36] Oliver, Smith, Holland: "A Study Of Permutation Cross Over Operators On The Traveling Salesman Problem"
[37] Whiltey, D., Starkweather, T., Fuquay, D.: "Scheduling Problems and Traveling Salesmen: The Genetic Edge Recombination Operator"

1. Elternteil ( 1 2 3 4 5 6 )
2. Elternteil ( 2 4 3 1 5 6 )

Es wird eine "edge-map" angelegt, die die Verbindungen zwischen den Kunden angibt.

| Stadt | besitzt Kanten zu den Städten | Priorität |
|---|---|---|
| 1 | 2,6,3,5 | 4 |
| 2 | 1,3,4,6 | 4 |
| 3 | 2,4,1 | 3 |
| 4 | 3,5,2 | 3 |
| 5 | 4,6,1 | 3 |
| 6 | 1,5,2 | 3 |

Der Kunde mit den wenigsten benutzten Kanten erhält die höchste Priorität. Die Startkunden sind durch die beiden Elternteile vorgegeben, nämlich 1 oder 2, beide haben die gleiche Priorität, daher entscheidet der Zufall. Sei Kunde 2 angenommen und die "edge-map" aktualisiert, d.h. Kunde 2 wird gestrichen.

| Stadt | besitzt Kanten zu den Städten | Priorität |
|---|---|---|
| 1 | 6,3,5 | 3 |
| 2 | 1,3,4,6 | - |
| 3 | 4,1 | 2 |
| 4 | 3,5 | 2 |
| 5 | 4,6,1 | 3 |
| 6 | 1,5 | 2 |

Kunde 2 besitzt Kanten zu den Kunden 1, 3, 4, 6. Unter diesen möglichen Kandidaten wird der Kunde mit der niedrigsten Priorität ausgewählt. Der Zufall entscheidet unter den Kunden 3, 4 und 6. Des Verfahren terminiert, wenn alle Kunden aufgeteilt sind.

Möglicher Nachkomme:

Kind ( 2 3 4 5 1 6 )

## 5.3 Mutationsoperatoren

Mutationsoperatoren haben in Bezug auf genetische Algorithmen traditionell eine untergeordnete Rolle gespielt. Der zufällige Einfluß des Mutationsoperators kann eine störende Wirkung entfalten und sollte mit Vorsicht eingesetzt werden.
Eine sinnvolle Anwendung ergibt sich allerdings, wenn eine Population zu früh konvergiert. Der *adaptive Mutationsoperator* kann dieser unerwünschten Entwicklung dadurch entgegenwirken, daß bei ähnlichen Individuen, d.h. Individuen mit vielen identischen Stellen, die Mutationsrate höher ist als bei verschiedenen Individuen.
Außerdem besteht die Möglichkeit, daß durch eine besonders große Population, der Nutzen der Mutation stark sinkt, da schon ein hohes Maß an Zufall bei der Initialisierung der Population stattgefunden hat. Ein solches Phänomem ist in Kapitel 10 behandelt worden.

Beispiel:

Mutationsrate = 20 %
1. Elternteil: ( 1 2 3 4 5 6 7 8 9 10 )
2. Elternteil ( 1 2 3 4 5 10 9 8 7 6 )
Grad der Übereinstimmung = 50 %
$\Rightarrow$ Mutationsrate = 10 %

Mutationsrate = 20 %
1. Elternteil: ( 1 2 3 4 5 6 7 8 9 10 )
2. Elternteil ( 1 2 3 4 5 6 7 8 9 10 )
Grad der Übereinstimmung = 100 %
$\Rightarrow$ Mutationsrate = 20 %

Je höher die unerwünschte Konvergenz ist, desto mehr Stellen des Individuum werden durch den Mutationsoperator verändert. Es besteht dadurch die Möglichkeit, noch unentdeckte Stellen des Suchraumes zu überprüfen, und Konvergenz noch etwas hinauszuschieben.

# 6. Genetischer Algorithmus von Blanton /Wainwright zur Lösung des Touren-problems

## 6.1 Problemcodierung

Blanton/Wainwright[38] haben 1993 das Tourenproblem mit Zeit- und Kapazitäts-restriktionen untersucht und festgestellt, daß die herkömmlichen Cross Over-Operatoren für Probleme mit mehreren Restriktionen nur eingeschränkt einsetzbar sind. Sie entwickelten neben einer neuer Familie von Operatoren außerdem eine geschickte Art, Tourenprobleme dieser Art zu codieren.
Ein Chromosom besteht wie beim Traveling Salesman Problem aus einem Integervektor, der durch die Greedy-Heuristik, wie in 5.1 beschrieben wurde, partitioniert wird.
Die zu minimierende Zielfunktion liefert für einen Lösungsvektor die Zahl der bisher noch nicht besuchten Kunden. Sind alle Kunden besucht, wird von der insgesamt zurückgelegten Strecke eine große ganze Zahl ( z.B. 100000 ) subtrahiert und damit einem Lösungsvektor eine hohe Qualität innerhalb der Population gegeben. Lösungen, die alle Kunden berücksichtigen, werden sich aufgrund des genetischen Drucks schnell ausbreiten und Lösungen, die Kunden außer Acht gelassen haben, schnell verdrängen.
Nimmt die zurückgelegte Strecke durch eine bessere Route ab, so wird diese Lösung durch eine kleinere negative Zahl bevorzugt. Der kleinste Funktionswert liefert damit die beste Lösung.
Diese Art der Codierung erfordet Cross Over-Operatoren, die auf diese Art der Codierung eingehen und ein Individuum nicht nur nach der relativen Position der Kunden untereinander beurteilen, wie dies bei den Operatoren in Kapitel 5.2 der Fall war.

[38] Blanton, Joe L. Jr., Wainwright, Roger L.: "Multiple Vehicle Routing with Time and Capacity Constraints using Genetic Algorithms", S. 452

## 6.2 Neue Familie von Cross Over-Operatoren

Die beiden Cross Over-Operatoren Merge Cross #1 (MX1) und Merge Cross #2 (MX2) wurden speziell für Optimierungsprobleme mit mehreren Restriktionen wie Zeit und Kapazität entworfen und unterscheiden sich in einem wesentlichen Punkt von den herkömmlichen Traveling Salesman Cross Over-Operatoren.[39]
Das Konzept der relativen Position innerhalb eines Chromosoms ist für mehrfache Restriktionen nicht sehr gut geeignet, deshalb wurde ein sogenannter *globaler Reihenfolgevektor* von Blanton/Wainwright eingeführt, der beim Crossing Over für die Auwahl der Gene entscheident ist.
Damit die Merge Operatoren effektiv eingesetzt werden können, muß eine gewünschte Reihenfolge zwischen den Genen existieren, die beim Crossing Over berücksichtigt werden kann. Bei Tourenproblemen könnte dies zum Beispiel das Zeitfenster sein, das Kunden (Genen) eine zeitliche Reihenfolge vorschreibt. Kunde mit frühem Zeitfenster sollten natürlich vor Kunden mit spätem Zeitfenster besucht werden und daher auch früher im Chromosom auftauchen.

### 6.2.1 Merge Cross #1 Operator

Beim Merge Cross #1 Operator werden die Gene ihrer Reihenfolge nach, gemäß eines globalen Reihenfolgevektors, auf das resultierende Individuum übertragen. Damit keine doppelten Gene innerhalb eines Chromosoms vorkommen können, werden die ausgwählten Gene beim jeweils anderen Chromosom vertauscht.

Beispiel:

Chromosom A ( **2** 5 6 1 0 7 3 8 4 9 )
Chromosom B ( **4** 1 6 9 3 8 2 0 5 7 )

Reihenfolge nach Zeitfenster:
Reihenfolge ( 0 1 2 3 4 5 6 7 8 9 ) (globaler Reihenfolgevektor)

Das erste Gen von Chromosom A ist in der Reihenfolge vor dem ersten Gen von Chromosom B. Das erste Gen von Chromosom A ( 2 ) wird daher ausgewählt und

[39] Blanton, Joe L. Jr., Wainwright, Roger L.: "Multiple Vehicle Routing with Time and Capacity Constraints using Genetic Algorithms", S. 454

das erste Gen von Chromosom B gegen das ausgewählte ( 2 ) innerhalb von Chromosom B ausgetauscht damit die Lösung gültig bleibt.

| | |
|---|---|
| Chromosom A | ( **2** 5 6 1 0 7 3 8 4 9 ) |
| Chromosom B | ( **2** 1 6 9 3 8 **4** 0 5 7 ) |
| Kind AB | ( 2 - - - - - - - - - ) |

Bei den restlichen Genen wird genauso verfahren, was zu folgendem Ergebnis führt.

| | |
|---|---|
| Kind AB | ( 2 1 6 5 0 7 3 4 8 9 ) |

### 6.2.2 Merge Cross #2 Operator

Der Merge Cross #2 Operator[40] funktioniert wie Merge Sort. Die beiden Vektoren werden anhand des globalen Reihenfolgevektors miteinander verschmolzen.

Beispiel:

| | |
|---|---|
| Chromosom A | ( **2** 5 6 1 0 7 3 8 4 9 ) |
| Chromosom B | ( **4** 1 6 9 3 8 2 0 5 7 ) |

| | |
|---|---|
| Reihenfolge nach Zeitfenster: | |
| Reihenfolge | ( 0 1 2 3 4 5 6 7 8 9 ) ( globaler Reihenfolgevektor) |

Wieder wird das erste Gen von Chromosom A als erstes ausgewählt, dann aber bei Chromosom A wie bei Chromosom B einfach gestrichen.
Als nächstes werden bei A wie bei B wieder die ersten Gene verglichen (im Beispiel 5 und 4). Die 4 wird ausgewählt und verfahren wie beschrieben.

| | |
|---|---|
| Chromosom A | ( X **5** 6 1 0 7 3 8 4 9 ) |
| Chromosom B | ( **4** 1 6 9 3 8 X 0 5 7 ) |
| Kind AB | ( 2 - - - - - - - - - ) |

Sind alle Gene behandelt terminiert das Verfahren.

---

[40] Blanton, Joe L. Jr., Wainwright, Roger L.: "Multiple Vehicle Routing with Time and Capacity Constraints using Genetic Algorithms", S. 454

Kind AB ( 2 4 1 5 6 0 7 3 8 9 )

Die Merge Cross Operatoren haben sich im Experiment von Blanton/Wainwright gegenüber den herkömmlichen Operatoren wie Cycle, Position und Edge Recombination als überlegen herausgestellt und durchweg bessere Ergebnisse erbracht. Eine Überlegenheit von einem der beiden Merge Cross Operatoren konnte nicht festgestellt werden. Die Ergebnisse waren je nach Problem verschieden.[41]

Die Merge Cross Operatoren sollen später in den Vergleich von genetischen Algorithmen mit traditionellen Heuristiken einfließen.

[41] Blanton, Joe L. Jr., Wainwright, Roger L.: "Multiple Vehicle Routing with Time and Capacity Constraints using Genetic Algorithms", S. 454

# 7. Genetische Algorithmen und lokale Suche

## 7.1. Hybride genetische Algorithmen

Zur Verbesserung der Perfomance genetischer Algorithmen empfahl L. Davis[42], die leistungsfähigen Teile einer Heuristik mit dem genetischen Algorithmus zu verschmelzen und damit einen leistungsfähigeren *hybriden Algorithmus* zu erzeugen. Tatsächlich deuten Untersuchungen darauf hin, daß die Schwächen genetischer Algorithmen teilweise erheblich durch den Einsatz von Heuristiken vermindert werden konnten.
In einer Untersuchung von Thangiah[43] wurden genetischen Algorithmen und Heuristiken gemeinsam in einer Art Arbeitsteilung eingesetzt. Das Verfahren funktioniert nach einer *cluster-first route-second*-Methode und hat sich gegenüber einer reinen Heuristik als überlegen herausgestellt.
Der genetische Algorithmus übernimmt das Clustering der Kunden, d.h. die Einteilung der Kunden auf Routen, während die Heuristik die einzelnen Routen nach der Wegstrecke optimiert.

## 7.2 Genetic Sectoring Heuristic (GSH)

Der genetische Algorithmus benutzt Pseudo Polar Koordinatenwinkel $C_1$, .. , $C_n$ zur Einteilung der n Kunden. Die Kunden werden in k cluster für k Fahrzeuge eingeteilt, und die genetische Population wird mit k zufällig erzeugten Winkeln $S_1 = 0$ , ... , $S_k$ initialsiert.
Kunde $C_i$ ( i = 1 , ... , n ) wird zu Cluster $A_j$ ( j = 1, ... , k ) gerechnet, wenn gilt:

$$S_j < C_i \leq S_{j+1}$$

Der Fitness-Wert eines Chromosoms wird aus der Wegstrecke der Fahrzeuge bei Einteilung gemäß der Clusterung errechnet.
Die Wegstrecke selber wird durch eine beliebige Heuristik wie z.B. die *cheapest insertion method* optimiert.

---

[42] Davis, L.: "Handbook of Genetic Algorithms"
[43] Thangiah, S.R., et al.: "Vehicle Routing with Time Deadlines using Genetic and Local Algorithms"

Die Ergebnisse zeigen[44], daß die hybride Methode der rein heuristischen Methode im Schnitt überlegen ist, eine Hybridisierung eines genetischen Algorithmus für Tourenprobleme erscheint daher ratsam.

[44] Thangiah, S.R., et al.: "Vehicle Routing with Time Deadlines using Genetic and Local Algorithms", S. 510

# 8. Eigener hybrider Algorithmus

# 9. Implementation der Algorithmen

## 9.1 Die Heuristik

## 9.2 Die genetischen Algorithmen

## 9.3 Der hybride Algorithmus

# 10. Experimentelle Auswertung

## 10.1 Problemgenerierung

Für den Vergleich der verschiedenen Optimierungsverfahren werden eine Reihe von Beispielproblemen für die Tourenplanung benötigt, die Aufschluß geben können, welche Verfahren wann eingesetzt werden sollten. Es ist dabei besonders darauf zu achten, daß sich die Problematiken der gewählten Beispiele möglichst unterscheiden, um Vorteile und Nachteile der einzelnen Verfahren besser sichtbar machen zu können. Beispielsweise kann ein Tourenproblem sehr restriktiv formuliert werden, indem sehr kleine und eng beieinanderliegende Zeitfenster gewählt werden. Andererseits kann die Anzahl der Kunden und damit die Größe des Problems stark variiert werden.
Da es in der Literatur noch keine Standardprobleme für die Tourenplanung mit Zeitfenstern gibt, ist es notwendig, neue zu erzeugen. Im Anhang ist ein vom Verfasser eigens für diesen Zweck entwickltes Programm zur Problemerzeugung dargestellt. Es erlaubt, unterschiedliche Problemschwerpunkte zu setzten.

Das Programm benötigt folgende Angaben:

- Anzahl der Kunden
- Anzahl der LKW
- Kapazität der LKW
- Maßzahl für die Streuung der Zeitfenster
- Maßzahl für die Breite des Zeitfensters
- Maximale Kapazität pro Kunde

Das Programm erzeugt alle für den Optimierungsprozeß benötigte Daten und speichert diese in einem Textfile ab.[45]
Die Koordinaten eines Kunden werden zufällig in einem XY-Koordinatensystem von 0 bis 100 erzeugt. Die Kapazität eines Kunden bestimmt sich aus einem vom Benuzter angebenen Prozentsatzes, mal der Kapazität der Fahrzeuge und mal einer Zufallszahl zwischen 0.1 und 1.

[45] Im Anhang befindet sich ein codiertes Beispielproblem, das mit dem Problemgenerator erzeugt wurde.

Kapazität eines Kunden : $C_K = C_F * Anteil * random(0.1, 1)$

Der Beginn eines Zeitfensters eines Kunden hängt von der Distanz des Kunden zum Depot, einer Maßzahl für die Streuung der Zeitfenster und einer Zufallszahl ab.

Beginn eines Zeitfensters:

$$TW_B = dist\ (Depot, Kunde) + (Konstante * Maßzahl_S * random\ (0.1, 1))$$

Das Ende eines Zeitfensters hängt von der Kapazität eines Kunden, der Maßzahl für die Breite des Zeitfensters und einer Zufallszahl ab.

Ende eines Zeitfensters:

$$TW_E = TW_B + C_K + (\ Maßzahl_B * C_K * random\ (0.1, 1))$$

Auf diese Weise wird garantiert, daß nur lösbare Probleme erzeugt werden, und die einzelnen Probleme unterschiedliche Anforderungen in Bezug auf die Restriktionen wie Zeitfenster und Kapazität stellen.

Um die Experimentelle Auswertung überschaubar zu halten, werden im folgenden drei Problemgrößen mit zwei unterschiedlichen Ausprägungen gewählt. Insgesamt werden also jeweils sechs unterschiedliche Probleme ausgewertet. Die Problemgrößen sind 15, 30, und 60 Kunden, die Ausprägungen beziehen sich auf die Kapazität der Fahrzeuge und die Art der Zeitfenster.

## 10.2 Performance-Messung genetischer Algorithmen

Um die Performance genetischer Algorithmen besser beurteilen und vergleichen zu können, werden zwei Kritieren angewendet, die die durchschnittliche Fitness aller Individuen einer Population und die durchschnittliche Fitness der besten Individuen über die Generationen angeben.[46] Man kann dadurch feststellen, wie sich eine Population entwickelt und wie schnell die Suche nach dem Optimum konvergiert.

Das erste Kriterium ist die sogenannte *Online-Performance* on-perf (t,s) :

$$on-perf(t) = \frac{1}{t} \times \sum_{i=1}^{t} \mu(i).$$

t gibt die Anzahl der erzeugten Individuen und μ(i) den Wert der Zielfunktion für Individuum i an. Die Online-Performance gibt damit die durchschnittliche Fitness einer Population an.

Das zweite Kriterium ist die sogennante *Offline-Performance* off-perf(t) :

$$off-perf(t) = \frac{1}{t} \times \sum_{i=1}^{t} \mu^{*}(i).$$

t gibt wieder die Anzahl der erzeugten Individuen an und μ*(i) den Wert der Zielfunktion für das beste Individuum aus der Population. Die Offline-Performance mißt damit den durchschnittlichen Fitnesswert der besten Individuen nach t erzeugten Individuen und zeigt die Konvergenz der Suche nach dem Optimum an.

Für die folgende experimentellen Auswertung sind diese beiden Kriterien ungeeignet, da es in erster Linie nur um die beste erzeugte Lösung im Vergleich zu einer Heuristik geht. Es wird daher nur das beste erzeugte Individuum betrachtet und zum Performancevergleich herangezogen.

[46] de Jong, K.A.: "An analysis of the behavior of a class of genetic adaptive systems"

## 10.3 Experiment 1: Vergleich der Cross Over-Operatoren

Im ersten Experiment sollen die Cross Over-Operatoren auf ihre Leistungsfähigkeit im Hinblick auf das Tourenproblem untersucht werden.
Es sollen 6 Probleme mit unterschiedlichen Anforderungen in Bezug auf Anzahl der Kunden und Grad der Restriktionen von den genetischen Algorithmen gelöst werden. Die Parameter werden je nach Härte des Problems angepaßt bleiben aber zwischen den Operatoren weitgehend gleich. Nur die Merge Cross-Operatoren wurden aufgrund ihrer Spezialisierung mit anderen Parametern getestet und haben ein ungewöhnlich gutes Konvergenzverhalten gezeigt.
Die Cross Over-Operatoren wurden bereits in Kapitel 5.2 und 6.2 vorgestellt. Sie verfügen alle über unterschiedliche Strategien und sollten auch zu unterschiedlichen Ergebnissen führen. Die besten Cross Over-Operatoren sollen später bei dem Vergleich mit der Heuristik zum Einsatz gebracht werden.
Bei Experiment 1 wird keine Mutation angewendet. Erst in Experiment 2 soll die Mutation auf ihre Leistungsfähigkeit hin untersucht werden.

Zunächst die Ergebnisse der 6 Probleme:

**Problem 1:** (15 Kunden, hoher Grad an Restriktionen)

| Operator | Bestes Individuum | Parameter |
|---|---|---|
| Position Based | 418[47] | Pool=100, Gen.=2000[48] |
| Order | 414 | Pool=100, Gen.=2000 |
| Order Based | 402 | Pool=100, Gen.=2000 |
| Cycle | 404 | Pool=100, Gen.=2000 |
| PMX | 402 | Pool=100, Gen.=2000 |
| Edge Recombination | 432 | Pool=100, Gen.=2000 |
| Merge Cross #1 | 430 | Pool=100, **Gen.=300** |

[47] Zurückgelegte Wegstrecke
[48] Pool=100 bedeutet, daß die Population 100 Individuen enthält. Gen.=2000 bedeutet, daß 2000 Generationen berechnet werden. Sofern nicht anders angegeben wird ein Bias von 1,9 verwendet.

| | | |
|---|---|---|
| Merge Cross #2 | 459 | Pool=100, **Gen.=300** |

**Problem 2:** (15 Kunden, niedriger Grad an Restriktionen)

| **Operator** | **Bestes Individuum** | **Parameter** |
|---|---|---|
| Position Based | 306 | Pool=100, Gen.=2000 |
| Order | 314 | Pool=100, Gen.=2000 |
| Order Based | 314 | Pool=100, Gen.=2000 |
| Cycle | 306 | Pool=100, Gen.=2000 |
| PMX | 306 | Pool=100, Gen.=2000 |
| Edge Recombination | 330 | Pool=100, Gen.=2000 |
| Merge Cross #1 | 330 | Pool=100, **Gen.=300** |
| Merge Cross #2 | 336 | Pool=100, **Gen.=300** |

**Problem 3:** (30 Kunden, hoher Grad an Restriktionen)

| **Operator** | **Bestes Individuum** | **Parameter** |
|---|---|---|
| Position Based | 699 | Pool=300, Gen.=6000 |
| Order | 750 | Pool=300, Gen.=6000 |
| Order Based | 693 | Pool=300, Gen.=6000 |
| Cycle | 718 | Pool=300, Gen.=6000 |
| PMX | 698 | Pool=300, Gen.=6000 |
| Edge Recombination | 707 | Pool=300, Gen.=6000 |
| Merge Cross #1 | 739 | Pool=300, **Gen.=500** |
| Merge Cross #2 | 756 | Pool=300, **Gen.=500** |

**Problem 4:** (30 Kunden, niedriger Grad an Restriktionen)

| Operator | Bestes Individuum | Parameter |
|---|---|---|
| Position Based | 600 | Pool=300, Gen.=6000 |
| Order | 1 Kunde nicht besucht | Pool=300, Gen.=6000 |
| Order Based | 609 | Pool=300, Gen.=6000 |
| Cycle | 657 | Pool=300, Gen.=6000 |
| PMX | 620 | Pool=300, Gen.=6000 |
| Edge Recombination | 2 Kunden nicht besucht | Pool=300, Gen.=6000 |
| Merge Cross #1 | 644 | Pool=300, **Gen.=500** |
| Merge Cross #2 | 693 | Pool=300, **Gen.=500** |

**Problem 5:** (60 Kunden, hoher Grad an Restriktionen)

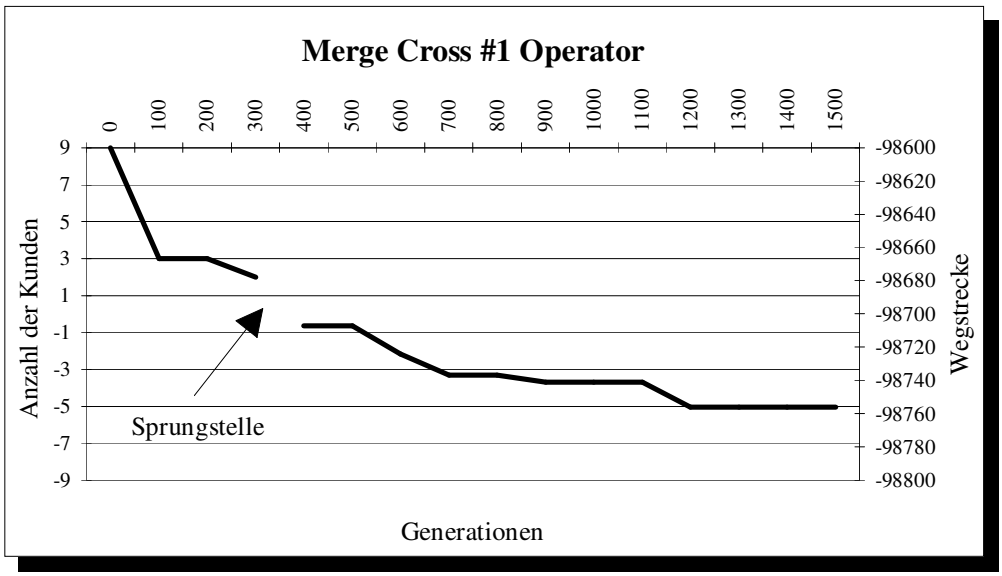
| Operator | Bestes Individuum | Parameter |
|---|---|---|
| Position Based | 1201 | Pool=1000, Gen.=25000 |
| Order | 5 Kunden nicht besucht | Pool=1000, Gen.=25000 |
| Order Based | 1203 | Pool=1000, Gen.=25000 |
| Cycle | 1214 | Pool=1000, Gen.=25000 |
| PMX | 1258 | Pool=1000, Gen.=25000 |
| Edge Recombination | 7 Kunden nicht besucht | Pool=1000, Gen.=25000 |
| Merge Cross #1 | 1251 | Pool=1000, **Gen.=1500** |
| Merge Cross #2 | 1261 | Pool=1000, **Gen.=1500** |

**Problem 6:** (60 Kunden, niedriger Grad an Restriktionen)

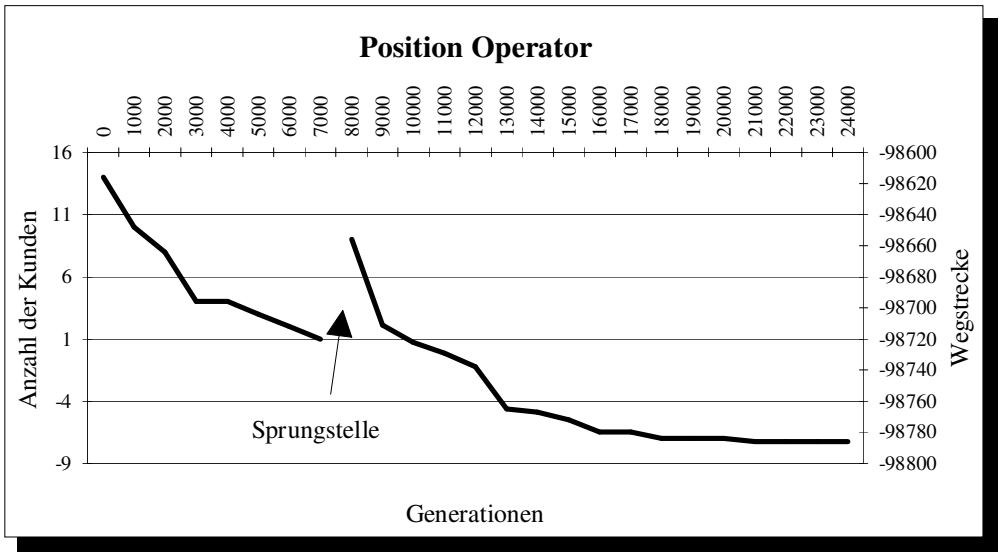
| Operator | Bestes Individuum | Parameter |
|---|---|---|
| Position Based | 1072 | Pool=1000, Gen.=25000 |
| Order | 11 Kunden nicht besucht | Pool=1000, Gen.=25000 |
| Order Based | 1088 | Pool=1000, Gen.=25000 |
| Cycle | 1127 | Pool=1000, Gen.=25000 |
| PMX | 1151 | Pool=1000, Gen.=25000 |
| Edge Recombination | 14 Kunden nicht besucht | Pool=1000, Gen.=25000 |
| Merge Cross #1 | 1192 | Pool=1000, **Gen.=1500** |
| Merge Cross #2 | 1223 | Pool=1000, **Gen.=1500** |

Besonders augenfällig ist das außergewöhnlich gute Konvergenzverhalten der *Merge Cross-Operatoren*. Es wurden durchweg gute Ergebnisse mit einem sehr niedrigen Rechenaufwand erzielt. Die Spezialisierung der Merge Cross-Operatoren zahlt sich gegenüber den Standard Cross Over-Operatoren aus.
Weiterhin fallen die Operatoren *Position* und *Cycle* durch sehr gute Ergebnisse auf. Der Aufwand ist zwar höher als bei den Merge Cross-Operatoren, die Ergebnisse aber bis zu 20% besser.
Die Operatoren *Edge Recombination* und *Order* scheinen dagegen wenig für Tourenprobleme geeignet. Teilweise konnten nicht alle Kunden besucht werden. Die Operatoren werden später für den Vergleich mir der Heuristik nicht mehr berücksichtigt.

In den folgenden Schaubildern ist das Konvergenzverhalten der besten Cross Over-Operatoren dargestellt. Die sehr gute Leistung der Merge Cross-Operatoren wird dadurch verdeutlicht.




Der Merge Cross #1 -Operator findet für das 60 Kundenproblem bereits nach 400 Generationen eine Lösung bei der alle Kunden besucht werden können. In den weiteren 1000 Generationen wird die Wegstrecke noch weiter optimiert.
Die Sprungstelle kommt durch die Besonderheit der Zielfunktion zustande. Sobald alle Kunden besucht werden können, wird eine Konstante (100000) subtrahiert, um diesen Lösungen eine hohe Wahrscheinlichkeit bei der Selektion zu gewähren.




Der Position-Operator benötigt eine wesentlich größere Anzahl von Generationen, die Lösung ist aber etwas besser (etwa 10%) als beim Merge Cross-Operator.

## 10.4 Experiment 2: Einsatz von Mutation

Im Experiment 2 soll untersucht werden, ob der Einsatz von Mutation einen Performancebeitrag liefern kann. Die Parameter wie Größe der Population und Anzahl der Generationen wurde beibehalten, um einen echten Vergleich zu ermöglichen. Anstelle der Parameter wird die prozentuale Veränderung zu den Ergebnissen ohne Mutation dargestellt. Die Mutationswahrscheinlichkeit beträgt bei allen Versuchen 20%.
Die Operatoren *Edge Recombination* und *Order* wurden aufgrund ihrer schlechten Leistung in Experiment 1 nicht mehr untersucht.

Zunächst die Ergebnisse:

**Problem 1:** (15 Kunden, hoher Grad an Restriktionen)

| Operator | Bestes Individuum | Veränderung |
|---|---|---|
| Position Based | 418 | 0% |
| Order Based | 404 | -0,5% |
| Cycle | 402 | +0,5% |
| PMX | 402 | 0% |

| | | |
|---|---|---|
| Merge Cross #1 | 430 | 0% |
| Merge Cross #2 | 475 | -3,5% |

**Problem 2:** (15 Kunden, niedriger Grad an Restriktionen)

| Operator | Bestes Individuum | Veränderung |
|---|---|---|
| Position Based | 306 | 0% |
| Order Based | 306 | +2,5% |
| Cycle | 322 | +0,5% |
| PMX | 306 | 0% |
| Merge Cross #1 | 330 | 0% |
| Merge Cross #2 | 336 | 0% |

**Problem 3:** (30 Kunden, hoher Grad an Restriktionen)

| Operator | Bestes Individuum | Veränderung |
|---|---|---|
| Position Based | 695 | -0,7% |
| Order Based | 698 | +0,7% |
| Cycle | 695 | -3,2% |
| PMX | 694 | -0,6% |
| Merge Cross #1 | 744 | +0,7% |
| Merge Cross #2 | 774 | +2,4% |

**Problem 4:** (30 Kunden, niedriger Grad an Restriktionen)

| Operator | Bestes Individuum | Veränderung |
|---|---|---|
| Position Based | 600 | 0% |
| Order Based | 625 | +2,6% |
| Cycle | 729 | +10,1% |
| PMX | 626 | +1% |
| Merge Cross #1 | 650 | +1% |

| | | |
|---|---|---|
| Merge Cross #2 | 659 | -5% |

**Problem 5:** (60 Kunden, hoher Grad an Restriktionen)

| Operator | Bestes Individuum | Veränderung |
|---|---|---|
| Position Based | 1201 | 0% |
| Order Based | 1208 | +0,4% |
| Cycle | 1204 | -0,8% |
| PMX | 1307 | +3,9% |
| Merge Cross #1 | 1247 | -0,3% |
| Merge Cross #2 | 1262 | 0% |

**Problem 6:** (60 Kunden, niedriger Grad an Restriktionen)

| Operator | Bestes Individuum | Veränderung |
|---|---|---|
| Position Based | 1077 | -0,5% |
| Order Based | 1089 | 0% |
| Cycle | 1087 | -3,5% |
| PMX | 2 Kunden nicht besucht | |
| Merge Cross #1 | 1155 | -3,1% |
| Merge Cross #2 | 1223 | 0% |

Die Ergebnisse lassen keine eindeutigen Aussagen zu. Die Veränderungen liegen im positiven wie im negativen Bereich. Die absolute Veränderung ist allerdings gering, so daß der Nutzen von Mutation für Tourenprobleme in Frage gestellt werden darf. Auch das Konvergenzverhalten unterschied sich nicht zu Experiment 1.

Über die Gründe kann an dieser Stelle nur spekuliert werden. Die relativ große Population könnte den Nutzen der Mutation einschränken, da bei der zufälligen

Initialisierung schon ein größerer Teil des Suchraumes abgedeckt ist als bei kleinen Populationen.
Aufgrund der Ergebnisse von Experiment 2 wird für die weitere Untersuchung davon ausgegangen, daß die Mutation keinen entscheiden Beitrag zur Lösung leisten kann. Die Mutation wird bei dem abschließenden Vergleich von genetischen Algorithmen mit der Heuristik auch nicht zum Einsatz kommen.

## 10.5 Experiment 3: Nutzen hybrider Techniken

In Experiment 3 soll der Nutzen der vom Autor vorgeschlagenen hybriden Methode untersucht werden. Die Versuchsumstände wie die Wahl der Parameter unterscheiden sich nicht von Experiment 1 und 2. Es soll hier aber noch auf das Konvergenzverhalten der Algorithmen eingegangen werden.
Die Cross Over-Operatoren Merge #1 und Merge #2 nehmen nicht am Experiment 3 teil, da sie schon auf das Tourenproblem zugeschnitten sind. Es soll in erster Linie gezeigt werden, ob die Leistung der klassischen Operatoren wie Cycle und Position nicht durch hybride Techniken gesteigert werden können.
Es wird die Veränderung zu Experiment 1 (ohne Mutation) und die benötigte Anzahl der Generationen angeben.

Zuerst wieder die Testergebnisse:

**Problem 1:** (15 Kunden, hoher Grad an Restriktionen)

| Operator | Bestes Individuum | Veränderung |
|---|---|---|
| Position Based | 418 | 0% |
| Order Based | 404 | -0,5% |
| Cycle | 404 | 0% |
| PMX | 402 | 0% |

**Problem 2:** (15 Kunden, niedriger Grad an Restriktionen)

| Operator | Bestes Individuum | Veränderung |
|---|---|---|
| Position Based | 306 | 0% |
| Order Based | 306 | -2,5% |
| Cycle | 326 | +6,1% |

| | | |
|---|---|---|
| PMX | 306 | 0% |

**Problem 3:** (30 Kunden, hoher Grad an Restriktionen)

| Operator | Bestes Individuum | Veränderung |
|---|---|---|
| Position Based | 709 | +1,1% |
| Order Based | 696 | +0,4% |
| Cycle | 691 | -3,8% |
| PMX | 695 | -0,4% |

**Problem 4:** (30 Kunden, niedriger Grad an Restriktionen)

| Operator | Bestes Individuum | Veränderung |
|---|---|---|
| Position Based | 633 | +5,5% |
| Order Based | 642 | +5,4% |
| Cycle | 633 | -3,6% |
| PMX | 617 | -0,48% |

**Problem 5:** (60 Kunden, hoher Grad an Restriktionen)

| Operator | Bestes Individuum | Veränderung |
|---|---|---|
| Position Based | 1197 | -0,33% |
| Order Based | 1200 | -0,25% |
| Cycle | 1221 | +0,58% |
| PMX | 1223 | -2,8% |

**Problem 6:** (60 Kunden, niedriger Grad an Restriktionen)

| Operator | Bestes Individuum | Veränderung | Generationen |
|---|---|---|---|
| Position Based | 1098 | +2,4% | |
| Order Based | 1096 | +0,7% | |
| Cycle | 1087 | -3,5% | |
| PMX | 1167 | 0,14% | |

Der Einsatz der hybriden Methode hat auf den ersten Blick keine nennenswerte Veränderung herbeigeführt. Die Ergebnisse sind abgesehen von ein paar Ausreißern nicht sehr verändert. Betrachtet man jedoch die Anzahl der Generationen, die benötigt wurden bis zur Konvergenz der Population, so stellt man fest, daß das Verfahren jetzt erheblich schneller konvergiert. Die Suche ist zielstrebiger und damit schneller geworden, ohne das Ergebnis sehr zu beeinflußen. Die klassischen Cross Over Operatoren sind durch den Einsatz von hybriden Techniken schneller geworden. Die Differenz in der Konvergenzgeschwindigkeit von den klassischen Cross Over-Operatoren zu den neuen Operatoren Merge #1 und Merge #2 hat sich deutlich verringert, die Qualität der Lösung bleibt unverändert besser.
Der Einsatz hybrider Methoden scheint sich durch Experiment 3 zu rechtfertigen. Sie werden aus diesem Grund auch bei Experiment 4 zum Einsatz kommen.

## 10.6 Experiment 4: Vergleich Heuristik - genetischer Algorithmus

Im abschließenden Experiment 4 sollen die genetischen Algorithmen mit der Insertion Heuristik verglichen werden. Es wurde zwei genetische Algorithmen zum Vergleich ausgewählt. Der Position Operator in Verbindung mit der hybriden Methode hat in bezug auf Konvergenzverhalten und Qualität der Lösung mit am besten abgeschnitten. Der Merge #1 Operator wurde als ein für das Tourenproblem speziell entwickelter Operator wegen seinem äußerst guten Konvergenzverhalten ausgewählt. Die Parameter wurden aus Experiment 1 und 3 übernommen. Mutation wurde nicht verwendet.
Die Insertion Heuristik ist mit Parametern für Zeit- bzw. Distanzkriterien ausgestattet. Es wurden zwei Konfigurationen ausgewählt, die für die Beispielprobleme die besten Ergebnisse geliefert haben.
Die Spalte "Veränderung" bezieht sich immer auf die Veränderung zum besten heuristischen Ergebnis.

Zunächst die Ergebnisse:

**Problem 1:** (15 Kunden, hoher Grad an Restriktionen)

| Operator | Lösung | Veränderung |
|---|---|---|
| Insertion Heuristik #1 | 514 | |
| Insertion Heuristik #2 | | |
| Merge #1 | 430 | -16,34% |
| Position (hybrid) | 418 | -18,68% |

**Problem 2:** (15 Kunden, niedriger Grad an Restriktionen)

| Operator | Lösung | Veränderung |
|---|---|---|
| Insertion Heuristik #1 | 419 | |
| Insertion Heuristik #2 | | |
| Merge #1 | 330 | -21,24% |
| Position (hybrid) | 306 | -26,97% |

**Problem 3:** (30 Kunden, hoher Grad an Restriktionen)

| Operator | Lösung | Veränderung |
|---|---|---|
| Insertion Heuristik #1 | 883 | |
| Insertion Heuristik #2 | | |
| Merge #1 | 739 | -16,31% |
| Position (hybrid) | 709 | -19,71% |

**Problem 4:** (30 Kunden, niedriger Grad an Restriktionen)

| Operator | Lösung | Veränderung |
|---|---|---|
| Insertion Heuristik #1 | 826 | |
| Insertion Heuristik #2 | | |
| Merge #1 | 644 | -22,03% |

| | | |
|---|---|---|
| Position (hybrid) | 633 | -23,37% |

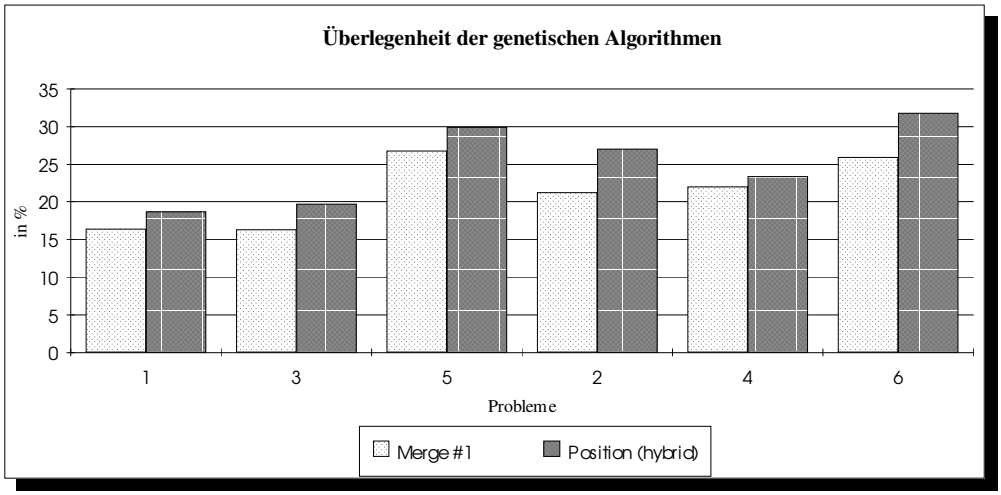
**Problem 5:** (60 Kunden, hoher Grad an Restriktionen)

| Operator | Lösung | Veränderung |
|---|---|---|
| Insertion Heuristik #1 | 1707 | |
| Insertion Heuristik #2 | | |
| Merge #1 | 1251 | -26,71% |
| Position (hybrid) | 1197 | -29,88% |

**Problem 6:** (60 Kunden, niedriger Grad an Restriktionen)

| Operator | Lösung | Veränderung |
|---|---|---|
| Insertion Heuristik #1 | 1608 | |
| Insertion Heuristik #2 | | |
| Merge #1 | 1192 | -25,87% |
| Position (hybrid) | 1098 | -31,72% |

Die Ergebnisse lassen keine Zweifel, die genetischen Algorithmen haben die Insertion Heuristik bei allen Beispielproblemen geschlagen. Die Ergebnisverbesserung beträgt zwischen 16 und 31%.
Es scheint eine Tendenz dahin zu bestehen, daß bei zunehmender Größe des Problems auch die Überlegenheit der genetischen Algorithmen steigt. Weiter ist festzustellen, daß bei einem niedrigem Grad an Restriktionen die Überlegenheit auch deutlicher wird. Zusammengenommen scheinen genetische Algorithmen für große Probleme mit niedrigem Grad an Restriktion besonders gut geeignet. Selbst aber bei kleinen Problemen mit hohem Grad an Restriktionen haben die genetischen Algorithmen noch bessere Ergebnisse gezeigt.
Im folgenden Schaubild lassen sich diese Zusammenhänge noch einmal verdeutlichen.




Die Probleme sind nach Grad der Restriktionen und Anzahl der Kunden geordnet. Die steigende Tendenz bei den Problemen 1, 3 und 5 (die restriktiven Probleme) und bei den Problemen 2, 4 und 6 (die weniger restriktiven Probleme) ist deutlich zu erkennen. Auch der Niveauunterschied zwischen Gruppe 1, 3, 5 und Gruppe 2, 4, 6 ist feststellbar.

## 10.7 Bewertung der Experimente

# 11. Weitere Anwendungen für den entwickelten Ansatz

# 12. Zusammenfassung

# Anhang

## Anhang 1:

Implementation der Insertion Heuristik:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>

#define max 70              /* Maximale Anzahl von Kunden */

int   n;                    /* Anzahl der Kunden */
int   k,c;                  /* Anzahl der LKW und Kapazitaet */
int   Kor[max+1][5];        /* Koordinaten */
int   b[max+1];             /* Beginn des Service eines Kunden (innerhalb time window) */
int   W[max+1][max+1];      /* Wartezeiten auf Routen */
int   route[max][max];      /* Routen */
int   besuch[max+1];        /* Kunde schon besucht ? */
int   PF[max+1];            /* Push Forward */
char *name;                 /* Problemfilename */

int mu     = 1;             /* Parameter */
int alpha1 = 0;
int alpha2 = 1;
int lamda  = 1;

int erg[max+1];
int akt    = 0;             /* aktuelle Anzahl der Routen initialisieren */



int dis(i, j)                /* Berechnet Entfernung zwischen Koordinaten */
  int i,j;
  {
  double a;
  a = sqrt( (double)( ((Kor[i][0]-Kor[j][0])*(Kor[i][0]-Kor[j][0])) +
                  ((Kor[i][1]-Kor[j][1])*(Kor[i][1]-Kor[j][1])) )    );
  return(a);
  };


int vecsum(vec)  /* Quersumme des Vektors zeigt ob alle Kunden geroutet */
  int vec[max];
  {
  int i,j;
  j=0;
  for (i=1;i<=n;i++)
    j=j+vec[i];
  return(j);
  };
```

```
void init()             /* Initialisiert eine Route */
  {
  int i,l,m;
  l=0;
  m=0;
  if (akt < k)          /* Noch LKWs frei ? */
    {
    for (i=1;i<=n;i++)
      if (besuch[i] == 0) /* Kunde noch nicht besucht ? */
        if (m < dis(0,i))
          {
          m = dis(0,i);
          l = i;
          };
    route[akt][1] = l;          /* Kunde mit der groessten Entfernung zum */
    b[l] = Kor[l][3];           /* Servicezeit gleich Startzeit des Windows */
    besuch[l] = 1;              /* Kunde wird als besucht markiert */
    W[akt][l] = b[l] - dis(0,l); /* Wartezeit; */
    }
  else
    {
    printf ("Zuviele LKWs benoetigt !\n");
    exit(1);
    };
  };


int c11 (i, u, j)
  int i,u,j;
  {
  int t;
  t = dis(i,u) + dis(u,j) - mu * dis(i,j);
  return(t);
  };  /* Unterkriterium 1 */


int c12 (i, u, j)
  int i,u,j;
  {
  int t;
  t = b[j] - b[i] + b[u];
  return(t);
  };  /* Unterkriterium 2 */


int c1 (i, u, j)
  int i,u,j;
  {
  int t;
  t = alpha1 * c11(i,u,j) + alpha2 * c12(i,u,j);
  return(t);
  };  /* Kriterium 1 */
```

```
int c2 (i, u, j)
  int i,u,j;
  {
  int t;
  t = lamda * dis(0,u) - c1(i,u,j);
  return(t);
  };  /* Kriterium 2 */


int feasible (u,  p)            /* Überprüft, ob Kunde u an Position p der Tour einfügbar ist */
  int u,p;
  {
  int flag1 = 1;
  int flag2 = 1;
  int i,t;

  /* 1. Teil von Lemma 1.1   Servicezeit muss im Time Window liegen */

  t = b[route[akt][p-1]] + dis(route[akt][p-1],u) +  /* moegliche Servicezeit ausrechnen */
      Kor[route[akt][p-1]] [2];


  if (t <= Kor[u][3])  /* Servicezeit kleiner Time window ? */
    t = Kor[u][3];

  if ( t + Kor[route[akt][p]] [2] > Kor[u][4] )
    {
    flag1 = 0;
    };

  /* 2. Teil von Lemma 1.1   Der Push Forward darf die weitere Route
     nicht stoeren */

  t = t + Kor[u][2] + dis(u,route[akt][p]);     /* neue Servicezeit */
                                                /* fuer alten Kunden */

  PF[route[akt][p]] = t - b[route[akt][p]];

  for (i=p+1;i<=n-1;i++)
    if (route[akt][i] != 0)  /* Ende erreicht ? */
      {
      PF[route[akt][i]] = PF[route[akt][i-1]] - W[akt][i];
      if (PF[route[akt][i]] < 0)
        PF[route[akt][i]] = 0;
      };

  /* Push Forward ist Zeitverschiebung durch Einfuegung */

  if (PF[route[akt][i]] < 0)
    PF[route[akt][i]] = 0;


  for (i=p;i<=n;i++)
    if (route[akt][i] != 0)
      {
      if (b[route[akt][i]] + PF[route[akt][i]] +
```

```
        Kor[route[akt][i]] [2] > Kor[route[akt][i]] [4])
       flag2 = 0;
      };



  if ((flag1==1) && (flag2==1))  /* Beide Teile erfuellt ? */
    {
    return(1);
    }
  else
    {
    return(0);
    };
  };  /* feasible prueft ob Kunde u verfuegbar fuer Position p */


void ausgabe()
  {
  int   i,j,a;
  int   l,g,wl,wg;
  FILE *fo;

  strcat(name, ".erg");
  fo = fopen (name, "w");

  printf  (    "***  Ausgabe der Routen ***\n\n");
  fprintf (fo, "***  Ausgabe der Routen ***\n\n");

  g=0;
  wg=0;
  for (i=0;i<=akt;i++)
    {
    printf  (    "%d. Route : \n",i+1);
    fprintf (fo, "%d. Route : \n",i+1);
    j = 1;
    l = 0;
    wl= 0;
    while (route[i][j] != 0)
      {
      l = l + dis(route[i][j-1],route[i][j]);
      wl= wl+ W[i][route[i][j]];
      printf  (    "Kunde : %d \tStrecke %d  ", route[i][j], dis(route[i][j-1],route[i][j]));
      printf  (    "\tUhrzeit : %d\n", b[route[i][j]]);
      fprintf (fo, "Kunde : %d \tStrecke %d  ", route[i][j], dis(route[i][j-1],route[i][j]));
      fprintf (fo, "\tUhrzeit : %d\n", b[route[i][j]]);
      j = j + 1;
      };
    l = l + dis(route[i][j-1],0);
    g = g + l;
    wg=wg + wl;
    printf  (    "Laenge der Route %d : %d\n\n", i+1, l);
    fprintf (fo, "Laenge der Route %d : %d\n\n", i+1, l);
    };
  printf  (    "Gesamtlaenge aller Routen : %d\n\n", g);
  fprintf (fo, "Gesamtlaenge aller Routen : %d\n\n", g);
  if (vecsum(besuch) != n)
    {
    printf  (    "Anzahl der nicht besuchten Kunden : %d\n", n-vecsum(besuch));
    fprintf (fo, "Anzahl der nicht besuchten Kunden : %d\n", n-vecsum(besuch));
```

```
    };
  fclose(fo);
  };




void einfuegen(a, j) /* Fuegt Kunden a an Position j ein */
  int a,j;
  {
  int i,t1,t2;

  t1 = route[akt][j];
  for (i=j;i<=n-1;i++)
    {
    t2 = route[akt][i+1];
    route[akt][i+1] = t1;
    t1 = t2;
    };
  route[akt][j] = a;

  for (i=j;i<=n-1;i++)  /* Alle Servicezeiten und Wartezeiten neu berechnen */
    if (route[akt][i] != 0)  /* Ende erreicht ? */
      {
      b[route[akt][i]] = b[route[akt][i-1]] + dis(route[akt][i-1],route[akt][i]) +
                         Kor[route[akt][i-1]] [2];
      /* Servicezeit ist Servicezeit vom Vorgaenger plus ausladen und Weg */

      W[akt][route[akt][i]] = Kor[route[akt][i]][3] - b[route[akt][i]];

      if (W[akt][route[akt][i]] < 0)
        W[akt][route[akt][i]] = 0;

      if (b[route[akt][i]] < Kor[route[akt][i]] [3]) /* Servicezeit durch Warten korrigieren ? */
        b[route[akt][i]] = Kor[route[akt][i]] [3];
      };

  };


void einlesen()
  {
  FILE *fp;
  int   i,a;

  fp = fopen(name, "r");

  if(fp == NULL)
    {
    printf("File not found !\n");
    exit(1);
    }
  else
    {
    fscanf(fp,"%d",&n);        /* Anzahl der Kunden */
    fscanf(fp,"%d %d",&k,&c);  /* Anzahl der LKW und Kapazitaet */
    fscanf(fp,"%d %d %d %d", &Kor[0][0], &Kor[0][1], &Kor[0][3], &Kor[0][4]);
    Kor[0][2]=0;
    for (i=1;i<=n;i++)
```

```
    if (a=fscanf(fp,"%d %d %d %d %d", &Kor[i][0], &Kor[i][1],
                    &Kor[i][3], &Kor[i][4], &Kor[i][2]) != 5)
      {
          printf("\nFehler in Koordinatenfile ! %d %d\n",a,i);
          exit(1);
      };
   };
 fclose(fp);
 };


void routen()

{
int a,i,j;

for (i=0;i<=akt;i++)
  {
  a=1;
  printf("Route %d : ",i);
  while (route[i][a] != 0)
    {
    printf("%d",route[i][a]);
    a=a+1;
    };
  printf("\n");
  };
};


void main(argc,argv)

int   argc;
char *argv[];

  {
  int i,j,h,a;
  int flag=0;

  for (i=1;i<max;i++)       /* Besucht Array initialisieren */
    besuch[i]=0;
  for (i=0;i<n;i++)         /* Routen Array initialisieren */
    for (j=0;j<n;j++)
      route[i][j] = 0;
  for (i=0;i<max;i++)
    PF[i] = 0;

  printf("*** Insertion Heuristics ***\n\n");

  name = argv[1];
  einlesen(); /* Liest Problemdaten ein. */
  init();     /* Initialisieren der ersten Route. */

  while (vecsum(besuch) != n)  /* alle Kunden geroutet ? */

    {
    /* Kriterium 1 berechnen */

    for (i=1;i<=n;i++)   /* Fuer alle Kunden */
      {
      if (besuch[i] == 0)  /* Kunde noch nicht besucht ? */
```

```
    {
    h = 100000;
    a = 0;
    j = 0;
    while ( (route[akt][j] != 0) || (j==0) )  /* aktuelles Routenende erreicht ? */
      {
      j = j + 1;
      if (feasible(i,j) == 1)  /* Kunde verfuegbar ? */




          if (h > c1(route[akt][j-1], i, route[akt][j]))
            {
        flag=1;  /* Es gibt noch Kunden, die eingefuegt werden koennen */
        a = j;
            h = c1(route[akt][j-1], i, route[akt][j]);
            };
        };
    };
  if (flag==1)
    erg[i] = a;  /* Beste Position fuer Kunden */
  else
       erg[i] = 0;  /* Kunde ist nicht zuteilbar auf dieser Route */
  flag = 0;
  };

 /* Kriterium 2 berechnen */

 if (vecsum(erg) != 0)  /* gibt es noch zuteilbare Kunden fuer Route ? */
   {
   h=100000;
   a=0;
   for (i=1;i<=n;i++)  /* Fuer alle Kunden */
     {
     if (erg[i] != 0)  /* Kunde verfuegbar ? */
       if (h > c2(route[akt][erg[i]-1], i, route[akt][erg[i]]))
         {
         a = i;
         h = c2(route[akt][erg[i]-1], i, route[akt][erg[i]]);
         };
          };
   besuch[a] = 1;            /* Kunde als besucht kennzeichnen */
   einfuegen(a,erg[a]);    /* Kunde in Route einfuegen */

   }
 else  /* neue Route initialisieren */
   {
   akt = akt + 1;
   init();
   };

 };

ausgabe();   /* alle Routen ausgeben */

};
```

**Anhang 2:**

Implementation des Problemgenerators:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>


int       n,k,C,KC,twc,i,F,g;
int       Kor[100][5];
float     a,b;
char      name[20];
char      temp[4];
FILE      *out;


double dist(x1,y1,x2,y2)

int x1,y1,x2,y2;

  {
  return (sqrt((double)((x1-x2)*(x1-x2)) + ((y1-y2)*(y1-y2))));
  };


void main()

  {
  printf("\n***** VRP-Generator *****\n\n");
  printf("Anzahl der Kunden : ");
  scanf ("%d",&n);
  printf("Anzahl der LKW : ");
  scanf ("%d",&k);
  printf("Kapazitaet der LKW : ");
  scanf ("%d",&C);
  printf("Streuung der Time Windows (1-5) : ");
  scanf ("%d",&F);
  printf("Maximale Kapazitaet pro Kunde in Prozent von LKW : ");
  scanf ("%d",&KC);
  printf("Verhaeltnis Time Window zu Kapazitaet (1-5) :  ");
  scanf ("%d",&twc);
```

```
printf ("Name des Problems : ");
scanf ("%s", name);


for (i=0;i<=n;i++)    /* Koordinaten bestimmen */
  {
  a=rand();
  a=a/65535;
  Kor[i][0]=(int)(a*100);
  b=rand();
  b=b/65535;
  Kor[i][1]=(int)(b*100);
  };


for (i=1;i<=n;i++)    /* Kapazitaeten bestimmen */
  {
  a=rand();
  a=a/65535;
  Kor[i][4]=(int)((a*C*KC/100) + 1);
  };



for (i=1;i<=n;i++)    /* Time Window Anfang bestimmen */
  {
  a=rand();
  a=a/65535;
  Kor[i][2]=(int)((a*100*F)+dist(Kor[0][0],Kor[0][1],Kor[i][0],Kor[i][1]));
  };


for (i=1;i<=n;i++)    /* Time Window Ende bestimmen */
  {
  a=rand();
  a=a/65535;
  Kor[i][3]=(int)((a*Kor[i][4]*twc) + Kor[i][2] + Kor[i][4]);
  };

g=Kor[1][3]+dist(Kor[0][0],Kor[0][1],Kor[1][0],Kor[1][1]);
for (i=2;i<=n;i++)          /* Depot Time Window setzen */
  if(g<(Kor[i][3]+dist(Kor[0][0],Kor[0][1],Kor[i][0],Kor[i][1])))
    g=Kor[i][3]+dist(Kor[0][0],Kor[0][1],Kor[i][0],Kor[i][1]);
Kor[0][2]=0;
Kor[0][3]=g;

out=fopen(name,"w");

fprintf(out,"%d\n",n);
fprintf(out,"%d  %d\n",k,C);
fprintf(out,"%d  %d  %d  %d\n",Kor[0][0],Kor[0][1],Kor[0][2],Kor[0][3]);
for(i=1;i<=n;i++)
  fprintf(out,"%d  %d  %d  %d  %d\n",Kor[i][0],Kor[i][1],Kor[i][2],Kor[i][3],Kor[i][4]);

fclose(out);

};
```

**Anhang 3:**

Hauptprogramm Genitor:

Es wurden einige Modifikationen gegenüber dem Original vorgenommen.

```
/************************************************************/
/*                                                          */
/*  Copyright (c) 1990                                      */
/*  Darrell L. Whitley                                      */
/*  Computer Science Department                             */
/*  Colorado State University                               */
/*                                                          */
/*  Permission is hereby granted to copy all or any part of */
/*  this program for free distribution.   The author's name */
/*  and this copyright notice must be included in any copy. */
/*                                                          */
/************************************************************/
/****************************************************************************
                                                    main_tsp.c
 ****************************************************************************

 This sample main() driver function implementation is designed for use with
 the Traveling Salesman Problem (TSP).  The TSP is a classic optimization
 problem defined as follows:  given a fixed set of cities, in what relative
 sequence must the cities be visited such that the shortest route is taken
 AND such that each city is visited once and only once?

 Rather than manipulating the traditional strings of 1s and 0s, Genitor's
 algorithm for the TSP manipulates symbolic data (actually, integers
 representing 'cities' in the tour).  Further, a special operator developed
 at Colorado State University, the "edge recombination" operator, can be
 used to genetically recombine the cities.

 ****************************************************************************

 This file provides an example of a genetic algorithm implementation using
 the ga_ modules's subroutines provided with the Genitor package.  For most
```

```
 experiments, you should be able to USE THE MODULES WITHOUT ALTERATIONS.
 However, your MAIN() AND EVAL() PROCEDURES CAN DIFFER considerably depending
 on your application.  Further, once you are familiar with the coding
 conventions of the Genitor package, you may wish to
 CREATE YOUR OWN GENETIC OPERATORS to add to the library of operators
 distributed with Genitor.  The hope is that you will find the basic Genitor
 modules useful in whatever form of experiment you undertake.

 While, this main() procedure is provided only as an example, it does display
 conventions to which you must adhere in order to use the modular Genitor
 environment.  To find out more about coding with the Genitor package, you
 must read the documentation included with the package before beginning.
 ************************************************************************/



#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include "ga_random.h"
#include "gene.h"
#include "ga_global.h"
#include "ga_params.h"
#include "ga_pool.h"
#include "ga_selection.h"
#include "ga_status.h"
#include "ga_signals.h"
#include "op_cycle.h"


/*****************************************
 Declare evaluation function which tells you
 how "good" a particular gene's solution is.
 NOTE: the input parameters for the eval
      function should always be a gene and
            the gene's length
 *****************************************/

#include "eval_tsp.h"

int
main (argc, argv)
int   argc;
char *argv[];
{
 int             i,j,o,p,q,r;
 GENEPTR         mom, dad, child, a;
 CITY_DATA       *city_table;
 FILE            *fp,*fq;
 int             TiWi[100][3];      /* Kunden Time Windows und Kapazitaeten */
 int             **coord_array;
 char            na[80];


/**********************
 Setup signal handlers.
 **********************/
 setup_signal();
```

```
/************************************************************
 Set the global parameters according to command line arguments.
 ************************************************************/
 argc--;
 argv++;
 parse_command_line (argc, argv);

/**********************
 Print Parameter Values
 **********************/
 fprintf (stdout, "\n");
 print_params(stdout);
 fprintf (stdout, "\n");

 strcpy(na, NodeFile);
 strcat(na,".dat");
 fq = fopen (na, "w");


/*******************************
 Seed the Random Number Generator
 *******************************/
 srand(RandomSeed);

/*****************************************************
 Allocate a genetic pool referenced by the global, Pool
 *****************************************************/
 if ( !(Pool = get_pool(PoolSize, StringLength)) )
        fatal_error(NULL);


/*****************************************************
 Read in a description of the points to be toured and
 create a representation of the distance between them.
 *****************************************************/

 make_dist_array (NodeFile, StringLength);

 init_pool ( SeedPool, Pool, 0, Pool->size, tsp_eval);

 sort_pool (Pool);

 mom    = get_gene (Pool->string_length);
 dad    = get_gene (Pool->string_length);
 child  = get_gene (Pool->string_length);

 city_table = get_city_table (Pool->string_length);

/**********
 Optimize !
 **********/

 for (/* CurrentGeneration already set :
            either intialized to 0 in its declaration OR
            initialized by a restart of a previous experiment */;

            CurrentGeneration < NumberTrials;
            CurrentGeneration++)
        {
        /*********************************
         Choose two genes for reproduction.
```

```
            **********************************/

            get_parents(mom, dad, Pool, linear, SelectionBias);

          if ( bitgen () )
            cycle (mom->string, dad->string, child->string,
                Pool->string_length, city_table);
          else
            cycle (dad->string, mom->string, child->string,
                Pool->string_length, city_table);

            mutation (child->string, Pool->string_length, MutateRate);

            if (CurrentGeneration > 4000)
              if ( bitgen () )
             hybrid (child->string, Pool->string_length, TiWi);




           /*************************
            So, kid, how good are you?
            *************************/

        child->worth = tsp_eval (child->string, Pool->string_length,0,NodeFile,TiWi);

     /*******************************************************
            Insert new gene into population according to its worth
            *******************************************************/
            insert_gene (child, Pool);

     /*********************************************************************
            If the StatusInterval parameter was set and this is the appropriate
            time, print the population best, worst, mean, and average to stdout
            *********************************************************************/
            if (StatusInterval && !(CurrentGeneration % StatusInterval))
              {
              show_progress (stdout, Pool, CurrentGeneration);
              fprintf (fq, "%5d , %5.0f\n", CurrentGeneration, Pool->data[0].worth);
              };

     /*****************************************************************
            If the DumpInterval parameter was set and this is the appropriate
            time, save the population and key parameters to disk for later
            reference (or to restart execution later.
            *****************************************************************/
        if (DumpInterval && !(CurrentGeneration % DumpInterval))
         dump_status(Pool, DumpBase);

            }

/*****************
 Summarize Results
 *****************/
 {
  final_pool (FinalPool, Pool, CurrentGeneration);
  fprintf (stdout, "\n");
  print_pool (stdout, Pool, 0, 1);
  tsp_eval (Pool->data[0].string, Pool->string_length, 1, NodeFile);
  fclose(fq);
```

```
 }
}
```

**Anhang 4:**

Zielfunktion für Tourenproblem mit n Kunden k Fahrzeugen und Zeitfenstern:

Es wurden erhebliche Änderungen gegenüber dem Original von Darrel L. Whitley vorgenommen, das Copyright muß aber trotzdem angegeben werden.

```
/************************************************************/
/*                                                          */
/*  Copyright (c) 1990                                      */
/*  Darrell L. Whitley                                      */
/*  Computer Science Department                             */
/*  Colorado State University                               */
/*                                                          */
/*  Permission is hereby granted to copy all or any part of */
/*  this program for free distribution.   The author's name */
/*  and this copyright notice must be included in any copy. */
/*                                                          */
/************************************************************/

#include <stdio.h>
#include <math.h>
#include <string.h>
#include "gene.h"
#include "ga_random.h"
#include "op_cycle.h"

#define DIFFSQ(x,y)  ((x-y)*(x-y))

**DistArray;            /* Distanzmatrix */
**CoordArray;           /* Koordinatenmatrix */
int dx,dy;              /* Depotkoordinaten */
int ta,te;              /* Depot time window */
int k,C;                /* Anzahl der LKW und maximale Kapazitaeten der LKW */
```

```
float
twert (tour, num_points)    /* Berechnet Routenlaenge */
int tour[];
int num_points;
{
 int    i;
 float sum;

 /* Distanz vom Depot zum ersten Kunden addieren */
 sum = (int)((sqrt((double)(DIFFSQ(dx, CoordArray[tour[0]][0]))
                          + (DIFFSQ(dy, CoordArray[tour[0]][1])))) + 0.5;

 /* Distanz vom letzten Kunden zum Depot addieren */
 sum = sum+(int)((sqrt((double)(DIFFSQ(dx, CoordArray[tour[num_points-1]][0]))
                            + (DIFFSQ(dy, CoordArray[tour[num_points-1]][1])))) + .5;




 /* Distanz zwischen den Kunden addieren */
 for (i=0; i<num_points-1; i++)
          sum = sum + (float) DistArray[tour[i]][tour[i+1]];

 return (sum);
}

float
tsp_eval (tour, num_points, aus, NodeFile, TiWi)      /* Evaluationsfunktion */

int  tour[];                                  /* Permutation */
int  num_points;                              /* Laenge der Permutation (Anzahl der Kunden) */
int  aus;                                     /* Flag fuer Ausgabe */
char NodeFile[];
int  TiWi[100][3];

{
  float sum;
  int   i,t,o,p,l,f,ff,best,a,anz;            /* Indizes und Flag */
  int   plan[100][100];
  int   lang[100];
  int   kap[100];
  float    vor,nach,diff;                     /* Temporaere Variablen */
  FILE  *fo;

  for (i=0;i<k;i++)                           /* Initialisierung */
    {
    plan[i][0] = tour[i];
    lang[i]    = 1;
    kap[i]     = 0;
    };

  anz=0;
  f=1;

  for (i=k;i<num_points;i++)                  /* Restliche Kunden zuteilen */
    {
```

```
  ff = 0;
  diff = 1000;
  for (l=0;l<k;l++)                          /* auf alle LKW */
    {
    a = plan[l][lang[l]-1];                  /* letzter Kunde auf Tour */
    if ((TiWi[a][0] + TiWi[a][2] + DistArray[a][tour[i]])
            < TiWi[tour[i]][0])              /* Time Window ok ? */
          if (kap[l]+TiWi[tour[i]][2] < C)   /* Kapazitaet ok ? */
            {
            ff = 1;                          /* Kunde einfuegbar */
            vor = twert(plan[l],lang[l]);
            plan[l][lang[l]] = tour[i];      /* Kunde temporaer einfuegen */
            lang[l] = lang[l] + 1;
            if (aus==1)
              {
              for(o=0;o<k;o++)
                {
                for(p=0;p<lang[o];p++)
                  printf("%d ",plan[o][p]);
                printf("\n");
                };
              printf("\n");
              scanf("%d",&t);




              printf("\n");
              }; */
            nach = twert(plan[l],lang[l]);
            lang[l] = lang[l] - 1;
            if ((nach-vor) < diff)
              {
              diff = nach-vor;
              best = l;
              };
            };
    if (l==(k-1) && ff)
          {
          lang[best] = lang[best] + 1;  /* Kunde einfuegen durch hochzaehlen */
          kap [best] = kap [best] + TiWi[tour[i]][2];
          }
    }; /* for l */

  if (ff==0)                                 /* Anzahl der unbesuchten Kunden */
    {
    anz = anz + 1;
    f=0;
    };
  }; /* for i */

if (f==1)                                    /* Alle Kunden besucht ? */
  {
  sum = 0.0;
  for (i=0;i<k;i++)                          /* Berechnung der Gesamtstreckenlaenge */
    sum = sum + twert(plan[i],lang[i]);
  sum = sum - 100000;                        /* Besonders guten Wert zuordnen */
  }
else
  sum = anz;
```

```
 if (aus==1)      /* Ausgabe des Ergebnisses */
   {
   strcat(NodeFile,".erg");
   fo = fopen (NodeFile, "w");
   sum = 0;
   printf (    "\n\n**** Ausgabe der Touren ****\n\n");
   fprintf(fo, "\n\n**** Ausgabe der Touren ****\n\n");
   for (i=0;i<k;i++)
     {
     sum = sum + twert(plan[i],lang[i]);
     printf (    "Laenge der %d. Tour :\t%10.2f\t\t",i+1,twert(plan[i],lang[i]));
     fprintf(fo, "Laenge der %d. Tour :\t%10.2f\t\t",i+1,twert(plan[i],lang[i]));
     for (l=0;l<lang[i];l++)
           {
           printf (    "%d ",plan[i][l]);
           fprintf(fo, "%d ",plan[i][l]);
           };
     printf (    "\n");
     fprintf(fo, "\n");
     };









   if (f==1)
     {
     printf (    "\nLaenge der Gesamttour : %10.2f\n", sum);
     fprintf(fo, "\nLaenge der Gesamttour : %10.2f\n", sum);
     }
   else
     {
     printf (    "\nEs wurden %d Kunden nicht besucht.\n",anz);
     printf (    "Laenge der Gesamttour : %10.2f\n\n", sum);
     fprintf(fo, "\nEs wurden %d Kunden nicht besucht.\n",anz);
     fprintf(fo, "Laenge der Gesamttour : %10.2f\n\n", sum);
     };
   fclose(fo);
   };

 return(sum);
};

#define ABS(x)      (x<0 ? 0-x : x)

/*****************************************************************************
 * FUNCTION: get_coord_array
 *
 * DESCRIBE: Allocates space for integer coordinate array.
 *
 * INPUT PARAMETERS: number of coordinates per point
 *                   number of points
 *
 * RETURN VALUE: pointer to coordinate array
 *****************************************************************************/
int **
get_coord_array(num_coords, num_points)
int             num_coords;
```

```
int             num_points;
{
 int **coords;
 int i;

 /*****************************************************
  malloc one extra point location so that nodes numbered
  1 - N can be indexed directly; 0 will not be used
  *****************************************************/

 if (!(coords = (int **) malloc ((num_points+1)*sizeof(int *))))
            fatal_error ("get_coord_array: Malloc failure (1).\n");

 for (i=0; i<=num_points; i++)
              if (!(coords[i] = (int *) malloc (num_coords*sizeof(int))))
                        fatal_error ("get_coord_array: Malloc failure (2).\n");

 return (coords);
}
```

```
/*****************************************************************************
 * FUNCTION: free_coord_array
 *
 * DESCRIBE: deallocates space for integer coordinate array
 *
 *           NOTE: use this function only when the integer coordinate
 *                 array was Allocated using get_coord_array
 *
 * INPUT PARAMETERS: coordinate array to free,
 *                   number of points
 *
 * RETURN VALUE: none
 *****************************************************************************/
 void
 free_coord_array(coords, num_points)
 int            *coords[];
 int             num_points;
 {
  int i;

  for (i=0; i<=num_points; i++)
              free (coords[i]);

  free (coords);
  }

/*****************************************************************************
 * FUNCTION: get_2D_dist_array
 *
 * DESCRIBE: Allocates space for 2D integer distance array
 *
 * INPUT PARAMETERS: number of points
```

```
 *
 * RETURN VALUE: pointer to distance array
 ******************************************************************************/
int **
get_2D_dist_array(num_points)
{
 int **dists;
 int i;

 /************************************************
  malloc one extra location so that nodes numbered
  1 - N can be indexed directly; 0 will not be used
  ************************************************/

 if (!(dists = (int **) malloc ((num_points+1)*sizeof(int *))))
            fatal_error ("get_2D_dist_array: Malloc failure (1).\n");


 for (i=0; i<=num_points; i++)
             if (!(dists[i] = (int *) malloc ((num_points+1)*sizeof(int))))
                        fatal_error ("get_2D_dist_array: Malloc failure (2).\n");

 return (dists);
}




/******************************************************************************
 * FUNCTION: free_2D_dist_array
 *
 * DESCRIBE: deallocates space for integer distance
 *
 *          NOTE: use this function only when the integer distance
 *               array was Allocated using get_2D_dist_array
 *
 * INPUT PARAMETERS: distance array to free,
 *                  number of points
 *
 * RETURN VALUE: none
 ******************************************************************************/
void
free_2D_dist_array(dists, num_points)
int            *dists[];
int             num_points;
{
 int i;

 for (i=0; i<=num_points; i++)
             free (dists[i]);

 free (dists);
}

/******************************************************************************
 * FUNCTION: read_coords
 *
 * DESCRIBE: Read x y coordinates (describing points in the tour) from input
```

```
*          stream. Coordinate values must be separated by white space.
*          The coordinates will be saved in the input xy array, with x
*          in the 0th element and y in the 1rst.
*
*          Points will be given an identification number which corresponds
*          to their file position.  Thus, the first x y coordinates
*          describe point 1, the second point 2, and so on.
*
*          NOTE: This function assumes that points are described by
*               integer coordinates in a 2D area.
*
* INPUT PARAMETERS: input stream
*                   xy array pointer
*                   number of points
*
* RETURN VALUE: 1 if successful, 0 upon failure
*******************************************************************************/

int
read_coords(fp, xy, num_points, TiWi)
FILE      *fp;
int       *xy[];
int       num_points;
int           TiWi[100][3];
{
 int i, dum;

 /* Liest Anzahl der Kunden ein. (Wird nicht benoetigt !)  */
 fscanf(fp, "%d", &dum);




 /* Anzahl der LKW und Kapazitaet der LKW einlesen */
 fscanf(fp, "%d %d", &k, &C);

 /* Depotkoordinaten und Depot Time Window einlesen */
 fscanf(fp, "%d %d %d %d", &dx, &dy, &ta, &te);

 for (i=1; i<=num_points; i++)
    if (fscanf(fp, "%d %d %d %d %d", &xy[i][0], &xy[i][1], &TiWi[i][0], &TiWi[i][1],
&TiWi[i][2]) != 5)
                        {
                         warning ("read_coords: Error reading input coordinate file.\n");
                         return (0);
                        }
 return (1);
 }

/*******************************************************************************
 * FUNCTION: calc_distances
 *
 * DESCRIBE: Uses the coordinates saved in the input xy array (x
 *          in the 0th element and y in the 1rst) to derive the
 *          distances between the points represented by the coordinates.
 *          The calculated distances are placed in the input distance
 *          array.
 *
 *          NOTE: This function assumes that points are described by
 *               integer coordinates in a 2D area.
```

```
 *
 *          NOTE: distances are rounded up to integers (13.3 = 14)
 *
 * INPUT PARAMETERS: xy array pointer
 *                   number of points
 *                   distance array pointer
 *
 * RETURN VALUE: none
 ************************************************************************/
void
calc_distances(xy, num_points, distances)
int       *xy[];
int        num_points;
int       *distances[];
{
 int i,j;
 int point1, point2;

 for (i=1; i<=num_points; i++)
    for (j=1; j<i; j++)
                        {
                         distances[i][j] = distances[j][i] =
                         (int) ( (sqrt ((double) (DIFFSQ(xy[i][0], xy[j][0])
                                  + (DIFFSQ(xy[i][1], xy[j][1])))) + .5
                           /* add .5 to round by truncation */
                         );
   }
}









/************************************************************************
 * FUNCTION: make_dist_array
 *
 * DESCRIBE: makes a 2 dimensional array of distances between points
 *
 * INPUT PARAMETERS: filename of list of coordinates of points
 *                   number of points
 *
 *
 * RETURN VALUE: none
 *
 * CALLS: get_coord_array
 *        read_coords
 *        get_2D_dist_array
 *        calc_distances
 ************************************************************************/
void
make_dist_array(coord_file, num_points, TiWi)
char            coord_file[];
int             num_points;
int                     TiWi[100][3];
{
 FILE *fp;
```

```
 if (fp = fopen(coord_file, "r"))
             {
              CoordArray        = get_coord_array(2, num_points);
              read_coords (fp, CoordArray, num_points, TiWi);
              DistArray = get_2D_dist_array (num_points);
              calc_distances(CoordArray, num_points, DistArray);
              fclose (fp);
             }
 else
            fatal_error ("Cannot read input city file");
}

void hybrid (tour, num_points, TiWi )

int tour[];
int num_points;
int TiWi[100][3];

{
int i,j,temp;

for (i=0;i<k;i++)
  for (j=k;j<num_points;j++)
    if (TiWi[tour[j]][0] < TiWi[tour[i]][0])
      {
      temp = tour[j];
      tour[j] = tour[i];
      tour[i] = temp;
      };
};




void mutation (tour, num_points, MutateRate )

int   tour[];
int   num_points;
float MutateRate;

{
int i,j,temp;


if (fracrand() < (MutateRate-1))
  {
  i=0;
  j=0;

  while (j==i)
    {
    i = randomain (0,num_points);
    j = randomain (0,num_points);
    };
  temp    = tour[i];
  tour[i] = tour[j];
  tour[j] = temp;
```

```
  };
};
```

**Anhang 5:**

Implementation des Merge Cross #1 Operators:

```
#include <stdio.h>
#include <malloc.h>
#include "ga_random.h"
#include "gene.h"
#include "op_mx1.h"


/*****************************************************************
 *  FUNCTION: merge1
 *
 *  DESCRIPTION: performs merge cross #1 operation
 *
 *  INPUT PARAMETERS:       two parent gene strings
 *                          space for child string
```

```
 *                                      the length of the two strings
 *                                      the time window
 *
 *
 *  RETURN VALUE:
 *
 ****************************************************************/

void merge1 (dad, mom, kid, length, TiWi)

int         dad[], mom[], kid[];
int         length;
int         TiWi[100][3];

{

int i,j;

for (i=0;i<length;i++)
  {
  if (TiWi[dad[i]][0] < TiWi[mom[i]][0])
    {
    kid[i] = dad[i];
    for (j=i;j<length;j++)
      if (mom[j] == dad[i])
        mom[j] = mom[i];
    mom[i]=dad[i];
    }
  else
    {
    kid[i] = mom[i];
    for (j=i;j<length;j++)
      if (dad[j] == mom[i])
        dad[j] = dad[i];
    dad[i]=mom[i];
    };
  };

};


/*****************************************************************************
 * FUNCTION: maketour
 *
 * DESCRIBE: Randomly generates a legal "traveling salesman" tour
 *          (i.e. where each point is visited only once.)
 *
 *          Essentially, this routine fills an array with all possible
 *          points on the tour and randomly chooses the 'next' city from
 *          this array.  When a city is chosen, the array is shortened
 *          and the procedure repeated.
 *
 * INPUT PARAMETERS: an empty tour, number of points per tour
 *
 * RETURN VALUE: none
 *
 * RESULT: fills input tour
 *****************************************************************************/
void
maketour(gene, num_points)
GENE_DATA gene[];
```

```
int      num_points;
{
static int          first_time = 1;
static GENE_DATAPTR temp;
int                 remainder;
int                 next, i;

if (first_time)
   {
    if (!(temp = (GENE_DATAPTR) malloc (num_points*sizeof(GENE_DATA))))
                fatal_error ("maketour: malloc failure\n");
    first_time = 0;
   }

for(i = 0; i < num_points; i++)
   temp[i] = (GENE_DATA) i+1;

remainder = num_points - 1;

for(i = 0; i < num_points; i++)
   {
    next = randomain(remainder, 0);
    gene[i] = temp[next];
    temp[next] = temp[remainder];
    remainder--;
   }
}
```

## Anhang 6:

Implementation des Merge Cross #2 Operators:

```
#include <stdio.h>
#include <malloc.h>
#include "ga_random.h"
#include "gene.h"
#include "op_mx2.h"


/*************************************************************
 *  FUNCTION: merge2
 *
 *  DESCRIPTION: performs merge cross #2 operation
 *
```

```
 *  INPUT PARAMETERS:        two parent gene strings
 *                           space for child string
 *                           the length of the two strings
 *                           the time window
 *
 *
 *  RETURN VALUE:
 *
 ***************************************************************/


void merge2 (dad, mom, kid, length, TiWi)

int        dad[], mom[], kid[];
int        length;
int        TiWi[100][3];


{
int i,j,x,y;

x=0;
y=0;

for (i=0;i<length;i++)
  {
  if (TiWi[dad[x]][0] <= TiWi[mom[y]][0])
    {
    kid[i] = dad[x];
    for (j=y;j<length;j++)
      if (mom[j] == dad[x])
        mom[j] = 0;
    while ((mom[y] == 0) && (y < length))
      y = y + 1;
    dad[x] = 0;
    while ((dad[x] == 0) && (x < length))
      x = x + 1;
    }
  else
    {
    kid[i] = mom[y];


    for (j=x;j<length;j++)
      if (dad[j] == mom[y])
        dad[j] = 0;
    while ((dad[x] == 0) && (x < length))
      x = x + 1;
    mom[y] = 0;
    while ((mom[y] == 0) && (y < length))
      y = y + 1;
    };
  };
};


/***************************************************************************
 * FUNCTION: maketour
 *
 * DESCRIBE: Randomly generates a legal "traveling salesman" tour
 *           (i.e. where each point is visited only once.)
```

```
 *
 *          Essentially, this routine fills an array with all possible
 *          points on the tour and randomly chooses the 'next' city from
 *          this array.  When a city is chosen, the array is shortened
 *          and the procedure repeated.
 *
 * INPUT PARAMETERS: an empty tour, number of points per tour
 *
 * RETURN VALUE: none
 *
 * RESULT: fills input tour
 ***************************************************************************/
void
maketour(gene, num_points)
GENE_DATA gene[];
int      num_points;
{
static int          first_time = 1;
static GENE_DATAPTR temp;
int                 remainder;
int                 next, i;

if (first_time)
   {
   if (!(temp = (GENE_DATAPTR) malloc (num_points*sizeof(GENE_DATA))))
              fatal_error ("maketour: malloc failure\n");
   first_time = 0;
   }

for(i = 0; i < num_points; i++)
  temp[i] = (GENE_DATA) i+1;

remainder = num_points - 1;

for(i = 0; i < num_points; i++)
   {
   next = randomain(remainder, 0);
   gene[i] = temp[next];
   temp[next] = temp[remainder];
   remainder--;
   }
}
```

## Anhang 7:

Beispielproblem:

| | |
|---|---|
| 20 | ; 20 Kunden |
| 8  100 | ; 8 Fahrzeuge mit einer Kapazität von 100 Einheiten |
| 25  8  0  213 | ; Depotkoordinaten = (25, 8), Zeitfenster Depot = 0, 213 |

```
15  26  106  119  10          ; 1. Kunden mit Koordinaten (15, 26),
                                 Zeitfenster = (106,119) und 10
                                 Kapazitätseinheiten
47  8  39  41  2              ; Kunden 2 bis 20
35  11  95  98  3
24  6  40  53  10
4  19  98  113  8
13  18  51  56  5
49  26  101  114  8
38  32  88  99  8
38  39  164  180  10
41  7  80  81  1
31  15  62  69  4
17  45  95  104  8
25  20  18  23  3
30  39  55  66  6
46  43  119  120  1
43  33  135  149  10
37  29  38  44  4
19  17  70  71  1
10  41  152  157  5
20  23  52  64  10
```

**Anhang 8:**

Ausgabe der Insertion Heuristik:

*** Ausgabe der Routen ***

1. Route :

| | | |
|---|---|---|
| Kunde : 4 | Strecke 2 | Uhrzeit : 40 |
| Kunde : 18 | Strecke 12 | Uhrzeit : 70 |
| Kunde : 15 | Strecke 37 | Uhrzeit : 119 |
| Kunde : 19 | Strecke 36 | Uhrzeit : 156 |

Laenge der Route 1 : 123

2. Route :
Kunde : 20 Strecke 15 Uhrzeit : 52
Kunde : 12 Strecke 22 Uhrzeit : 95
Kunde : 16 Strecke 28 Uhrzeit : 135
Kunde : 9 Strecke 7 Uhrzeit : 164
Laenge der Route 2 : 105

3. Route :
Kunde : 14 Strecke 31 Uhrzeit : 55
Kunde : 5 Strecke 32 Uhrzeit : 98
Kunde : 1 Strecke 13 Uhrzeit : 119
Laenge der Route 3 : 96

4. Route :
Kunde : 7 Strecke 30 Uhrzeit : 101
Laenge der Route 4 : 60

5. Route :
Kunde : 8 Strecke 27 Uhrzeit : 88
Laenge der Route 5 : 54

6. Route :
Kunde : 13 Strecke 12 Uhrzeit : 18
Kunde : 17 Strecke 15 Uhrzeit : 38
Kunde : 11 Strecke 15 Uhrzeit : 62
Kunde : 10 Strecke 12 Uhrzeit : 80
Laenge der Route 6 : 71

7. Route :
Kunde : 2 Strecke 22 Uhrzeit : 39
Laenge der Route 7 : 44

8. Route :
Kunde : 6 Strecke 15 Uhrzeit : 51
Laenge der Route 8 : 30

Gesamtlaenge aller Routen : 583

**Anhang 9:**

Ausgabe von Genitor:

**** Ausgabe der Touren ****

Laenge der 1. Tour : 51.00 11 23 13 3
Laenge der 2. Tour : 68.00 9 24 8
Laenge der 3. Tour : 62.00 21 10 25

Laenge der 4. Tour : 68.00 2 29 4
Laenge der 5. Tour : 146.00 15 16 7 17 12 22
Laenge der 6. Tour : 121.00 18 6 20 5 27 28
Laenge der 7. Tour : 92.00 14 30 19 26 1

Laenge der Gesamttour : 608.00

**Anhang 10:**

Bildschirmausgabe von Genitor:

PoolSize 1000

| | |
|---|---|
| LengthString | 30 |
| Trials | 25000 |
| GenerationCurrent | 0 |
| RandomSeed | 12345678 |
| Bias | 1.150000 |
| MutateRate | 1.100000 |
| OutfileDump | dump |
| StatusInterval | 1000 |
| NodeFile | K30S3V2.vrp |
| Experiments | 1 |

| | | | | |
|---|---|---|---|---|
| 0 | Bst: 11.000000 | Wst: 23.000000 | Mean: 20.000000 | Avg: 20.243000 |
| 1000 | Bst: 6.000000 | Wst: 18.000000 | Mean: 13.000000 | Avg: 13.173000 |
| 2000 | Bst: 6.000000 | Wst: 12.000000 | Mean: 11.000000 | Avg: 10.651000 |
| 3000 | Bst: 5.000000 | Wst: 11.000000 | Mean: 9.000000 | Avg: 9.127000 |
| 4000 | Bst: 4.000000 | Wst: 9.000000 | Mean: 8.000000 | Avg: 7.923000 |
| 5000 | Bst: 2.000000 | Wst: 8.000000 | Mean: 6.000000 | Avg: 6.209000 |
| 6000 | Bst: 1.000000 | Wst: 6.000000 | Mean: 5.000000 | Avg: 4.867000 |
| 7000 | Bst: -99236.000000 | Wst: 5.000000 | Mean: 4.000000 | Avg: -95.500999 |
| 8000 | Bst: -99236.000000 | Wst: 4.000000 | Mean: 3.000000 | Avg: -96.307999 |
| 9000 | Bst: -99360.000000 | Wst: 3.000000 | Mean: 2.000000 | Avg: -295.596008 |
| 10000 | Bst: -99360.000000 | Wst: 2.000000 | Mean: 2.000000 | Avg: -1189.650024 |
| 11000 | Bst: -99360.000000 | Wst: 2.000000 | Mean: 1.000000 | Avg: -2778.253906 |
| 12000 | Bst: -99360.000000 | Wst: 1.000000 | Mean: 1.000000 | Avg: -8734.313477 |
| 13000 | Bst: -99360.000000 | Wst: 1.000000 | Mean: 1.000000 | Avg: -21837.652344 |
| 14000 | Bst: -99366.000000 | Wst: 1.000000 | Mean: 1.000000 | Avg: -47051.609375 |
| 15000 | Bst: -99366.000000 | Wst: 1.000000 | Mean: -99264.000000 | Avg: -97083.859375 |
| 16000 | Bst: -99366.000000 | Wst: -99267.000000 | Mean: -99293.000000 | Avg: -99300.375000 |
| 17000 | Bst: -99373.000000 | Wst: -99297.000000 | Mean: -99320.000000 | Avg: -99324.078125 |
| 18000 | Bst: -99373.000000 | Wst: -99322.000000 | Mean: -99339.000000 | Avg: -99341.265625 |
| 19000 | Bst: -99374.000000 | Wst: -99340.000000 | Mean: -99356.000000 | Avg: -99355.062500 |
| 20000 | Bst: -99378.000000 | Wst: -99360.000000 | Mean: -99360.000000 | Avg: -99361.687500 |
| 21000 | Bst: -99378.000000 | Wst: -99360.000000 | Mean: -99366.000000 | Avg: -99364.406250 |
| 22000 | Bst: -99386.000000 | Wst: -99366.000000 | Mean: -99366.000000 | Avg: -99368.765625 |
| 23000 | Bst: -99386.000000 | Wst: -99366.000000 | Mean: -99373.000000 | Avg: -99373.351562 |
| 24000 | Bst: -99386.000000 | Wst: -99373.000000 | Mean: -99378.000000 | Avg: -99377.039062 |

Bestes Individuum :

11 18 9 23 15 21 2 10 16 6 29 24 14 30 19 26 17 7 20 25 5 13 1 8 27 12 4 22 28 3  -99386.000

## Literaturverzeichnis:

Blanton, Joe L. Jr., Wainwright, Roger L.: "Multiple Vehicle Routing with Time and Capacity Constraints using Genetic Algorithms", Proceedings of the Fifth International Conference on Genetic Algorithms, Morgan Kaufmann, 1993, S. 452ff.

Bodin, L., B. Golden, A. Assad and M. Ball: "Routing and Scheduling of Vehicles and Crews: The State of the Art", Comput. Opns. Res., 10, S. 62-212.

Booker L. : "Improving Search in Genetic Algorithms", in: "Genetic Algorithms and Simulated Annealing", hrsg. v. Davis, L.: Pitman, London; Morgan Kaufman Publ.; Los Altos, California; 1987, S. 61-72.

Christofides, N, Mingozzi, A., Toth, P.: "The Vehicle Routing Problem", Combinatorial Optimizations, John Wiley & Sons, New York, 1971, S. 315

Christofides, N, Mingozzi, A., Toth, P.: "Exact Algorithms for the Vehicle Routing Problem, Based on Spanning Tree and Shortest Path Relaxation", Mathematical Programming 20, North Holland Publishing, 1981, S. 255-282.

Davis, L.: "Genetic Algorithms and Simulated Annealing", Pitman, London, Morgan Kaufmann Publ., Los Altos, California, 1987.

Davis, L.: "Handbook of Genetic Algorithms", Van Nostrand Reinhold, New York, 1991.

Davis, L.: "Applying Adaptive Algorithms To Epistatic Domains", Proceedings of the International Joint Conference on Artificial Intelligence, 1985, S. 162-164.

de Jong, K.A.: "An analysis of the behavior of a class of genetic adaptive systems", Doctoral Dissertation, University of Michigan, 1975.

Desrochers, M., Desrosiers, J., Solomon, M.: "A new optimization Algorithm for the vehicle routing Problem with time windows", Operations Research, Vol. 40, N0. 2, March-April 1992.

Detering, Reinhard: "Unix Handbuch System V", 6. Auflage, Sybex-Verlag, Düsseldorf, 1989.

Domschke, W.: "Logistik: Rundreisen und Touren", 3. Auflage, Oldenburg-Verlag, München, Wien, 1989.

Gabbert, P.S., Brown, D.E., Huntley, C.L.: "A System for Learning Routes and Schedules with Genetic Algorithms", Proceedings of the Fourth International Conference on Genetic Algorithms, Morgan Kaufmann Publ., 1991, S. 430-436.

Garey, M.S., Johnson, D.S.: "Computers and Intractability: A Guide to the Theory of NP-Completeness", W.H. Freeman, San Francisco, 1979.

Goldberg, D.E. : "Genetic Algorithms in Search, Optimization, and Machine Learning", Addison-Wesley Publ. Co., 1989 reprinted.

Grefenstette, John J.: "A users´s guide to GENESIS", Technical Report No. CS-84-11, Nashville, Vanderbilt University, Department of Computer Science.

Harbison, S.P., Steel, G.L.: "C Ein Referenzhandbuch", 1. Auflage, Wolfram´s Fachverlag, Flensburg, 1991.

Hau, Matthias: "Implementation und Optimierung genetischer Algorithmen zur Lösung von Job Shop Scheduling-Problemen", Diplomarbeit, Universität Frankfurt, Fachbereich Informatik, 1.4.1993.

Heitkoetter, Joerg: "The Hitch-Hiker´s Guide to Evolutionary Computation", Department of Compter Science, University of Dortmund, Issue 1.10, 1993

Holland, J.H. : "Adaption in Natural and Artificial Systems", University of Michigan Press, Ann Arbor, 1975.

Jones, Donald R., Beltramo, Mark A.: "Solving Partitioning Problems with Genetic Algorithms", Proceedings of the Fourth International Conference on Genetic Algorithms, Morgan Kaufmann Pub., S. 442-449.

Juliff, Kate: "A Multi-chromosome Genetic Algorithm for Pallet Loading", Proceedings of the Fifth International Conference on Genetic Algorithms, Morgan Kaufmann, 1993, S. 467ff.

Kolen, A.W.J., et al.: "Vehicle Routing with Time Windows", Operations Research, Vol. 35, No. 2, March-April 1987.

McMahon, M.B., Fox, B.R.: "An Introduction to Genetic Operators for Sequencing Problems", Foundations of Genetic Algorithms, Morgan Kaufmann, 1991.

Michalewicz, Z.: "Genetic Algorithms + Data Structures = Evolution Programs", Springer Verlag, Berlin, 1992.

Nelson, M.D., et al.: "Implementation Techniques for the Vehicle Routing Problem", Computers & Operations Research, Vol. 12, No. 3, 1985, pp. 273-283.

Oliver, Smith, Holland: "A Study Of Permutation Crossover Operators On The Traveling Salesman Problem", Proceedings of the Second International Conference on Genetic Algorithms, 1987, S. 224-230.

Plum, Thomas: "Das C-Lernbuch", Carl Hanser Verlag, München, 1985.

Prosser, Patrick: "A Hybrid Genetic Algorithms for Pallet Loading", University of Strathclyde, Department of Computer Science,

Rawlings, Greg. J.E.: "Foundations Of Genetic Algorithms", Morgan Kaufmann Publ., Los Altos, California, 1991.

Savelsbergh, M.W.P.: "Local search in routing problems with time windows", Centre for Mathematics and Computer Science, Amsterdam, Report OS-R8409.

Savelsbergh, M.W.P.: "The vehicle routing problem with time windows: minimizing duration", Eindhoven University of Technology, COSOR Memorandum 91-03, January 1991.

Solomon, Marius M.: "Algorithms for the vehicle routing and scheduling problems with time window constraints", Operations Research, Vol. 35, No. 2, March-April 1987.

Solomon, Marius M., Desrosiers, J.: "Time Window Constrained Routing and Scheduling Problems: A Survey", Northeastern University, Mgmt. Science Dept., G-87-09, February 1987.

Starkweather, T., et al.: "A Comparison of Genetic Sequencing Operators", Proceedings of the Fourth International Conference on Genetic Algorithms, Morgan Kaufmann Publishers, 1991, S. 69-76.

Syswerda, G.: "Schedule Optimization Using Genetic Algorithms", Handbook of Genetic Algorithms, Van Nostrand Reinhold, New York, 1991, S. 332-349.

Syswerda, G.: "Schedule Optimization Using Genetic Algorithms", Handbook of Genetic Algorithms, Van Nostrand Reinhold, New York, 1991, S. 332-349.

Thangiah, S.R., et al.: "Vehicle Routing with Time Deadlines using Genetic and Local Algorithms", Proceedings of the Fifth International Conference on Genetic Algorithms, Morgan Kaufmann, 1993, S. 506 ff.

Whitley, Darrel : "The Genitor Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproduction Trials is Best", Proceedings of the Third International Conference on Genetic Algorithms, Morgan Kaufmann, 1989.

Whitley, Darrel and Starkweather, Timothy: "Genitor II: a distributed genetic algorithm", J. Expt. Theor. Artif. Intell. 2 (1990) S. 189-214.

Whiltey, D., Starkweather, T., Fuquay, D.: "Scheduling Problems and Traveling Salesmen: The Genetic Edge Recombination Operator", Proceedings of the Third International Conference on Genetic Algorithms, Morgan Kaufmann Publishers, 1989, S. 133-140.

Whiltey, D., Starkweather, T., Shaner, D.: "The Traveling Salesman and Sequencing Scheduling: Quality Solutions Using Genetic Edge Recombination", Handbook of Genetic Algorithms", Van Nostrand Reinhold, New York, 1991, S. 350-372.

Vignaux, G.A., Michalewicz, Z.: "Genetic Algorithms for the Transportation Problem", Methodologies for Intelligent Systems, 4, Elsevier Science Publishing Co., 1989.