HOCHSCHULE KONSTANZ TECHNIK, WIRTSCHAFT UND GESTALTUNG
UNIVERSITY OF APPLIED SCIENCES

Genetische Algorithmen, SS 07

Traveling Salesman Problem

André Erb Jan Tammen

28. Juni 2007

Prof. Dr. Erben, Fakultät Informatik, HTWG Konstanz

Inhaltsverzeichnis

1. Einleitung

1.1. GEATbx und TSPLIB

Bei der GEATbx[1] handelt es sich um eine von Hartmut Pohlheim erstellte Erweiterung für die Mathematiksoftware Matlab. Mit Hilfe der GEATbx lassen sich auf einfache Weise genetische und evolutionäre Algorithmen verwenden.

Enthalten ist mit der TSBLIB[2] außerdem eine Sammlung von beispielhaften Instanzen des Traveling Salesman Problems. In der vorliegenden Arbeit soll anhand von zwei Beispieldatensätzen der TSBLIB ein genetischer Algorithmus konfiguriert und mit verschiedenen Parametern getestet werden.

[1] `http://www.geatbx.com`

[2] `http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/`

2. Aufgaben

2.1. Aufgabe a

Nach Durchführung einiger Testläufe mit dem Demoskript `demotsplib.m` und dem TSP-Problem `bays29.tsp` zeigte sich, dass die Ergebnisse i.A. noch recht weit vom Optimalwert 2020 (vgl. [Rei94, Appendix]) entfernt waren.

Das Beispielskript ist standardmäßig mit den folgenden Parametern konfiguriert:

- `NumberSubpopulation`: Anzahl der Subpopulationen, 6
- `NumberIndividuals`: Anzahl der Individuen pro Population, 50
- `Selection.GenerationGap`: Anzahl der Individuen pro Population, 0.95
- `Termination.Method`: Art des Abbruchkriteriums, 1 (entspricht dem Erreichen der maximalen Anzahl von Generationen)
- `Termination.MaxGen`: Anzahl der Generationen, nach denen der Algorithmus terminieren soll, 400

Durch die Wahl des genetischen Algorithmus' `tbx3perm` sind implizit weiterhin die folgenden Einstellungen gegeben:

- `VariableFormat`: Format der Variablen und Konvertierung zwischen diesem und in dem intern verwendeten Format, 5 (entspricht einem nicht festgelegten Format, demzufolge findet auch keine Konvertierung statt).
- `Recombination.Name`: Verwendetes Rekombinationsverfahren, `recpm` (entspricht Partial Matching Recombination)
- `Mutation.Name`: Verwendetes Mutationsverfahren, `mutswap` (entspricht Swap Mutation)

2.2. Aufgabe b

2.2.1. Architektur des Testsystems

Für die Durchführung der Tests wurde zunächst das der GEATbx beiliegende Skript **demotsplib.m** als Ausgangsbasis benutzt. Alle für die Testläufe zu variierenden Parameter lassen sich im Kopf des Skriptes **tsp.m** in Variablenform festlegen.

Für die Durchführung der Aufgabe c) waren beispielsweise die Anzahl der Subpopulationen und der Individuen pro Population in verschiedenen Konstellationen zu testen. Zusätzlich dazu lassen wir für jede Variablenkombination eine definierbare Anzahl von Testläufen laufen, um später Mittelwert und Standardabweichung berechnen zu können. Listing 2.1 zeigt den beispielhaften Aufbau des Testskripts für Aufgabe c). Für Aufgabe d) wurde eine modifizierte Version des in 2.1 beschriebenen Skriptes eingesetzt. Der Pseudocode zu der genannten Modifikation wird in Listing 2.2 aufgeführt. Der komplette Code der Skripte 2.1 und 2.2 ist im Anhang aufgeführt.

```
1  Subpopulationen ←—(1, 5, 8, 12)
2  Individuen ←—(15, 30, 45, 60)
3  AnzahlTestlaeufe ←—20
4
5  foreach anzahlSubpopulationen in Subpopulationen do
6    foreach anzahlIndividuen in Individuen do
7      for <AnzahlTestlaeufe> do
8        Fuehre genetischen Algorithmus aus
9        Berechne Minimal-/Maximalwert
10     end for
11     Ergebnisausgabe Logfile
12   end foreach
13 end foreach
```

Listing 2.1: Pseudocode Testskript Teil B

```
1  Subpopulationen ←—12;
2  Individuen      ←—60;
3  AnzahlTestlaeufe ←—20;
4
5  SelektionsMethoden ←—(selsus, selrws, seltour)
6
7  foreach SelektionsMethode in SelektionsMethoden do
8    for <AnzahlTestlaeufe> do
9        Fuehre genetischen Algorithmus aus
```

```
10      Berechne Minimal-/Maximalwert
11    end for
12      Ergebnisausgabe Logfile
13  end foreach
```

Listing 2.2: Pseudocode Testskript Teil D

Zur späteren Auswertung der Ergebnisse werden alle relevanten Daten in ein Logfile protokolliert. Abbruchkriterium für die genetische Optimierung ist hier das Erreichen der Maximalgeneration 100.

2.2.1.1. Eingesetzte Hard- und Software

Die Tests der Aufgabe c) wurden auf einem Pentium M mit 1.6 GHz unter Verwendung von Matlab 5.3 und GEATbx Version 3.8 durchgeführt.

2.3. Aufgabe c

2.3.1. Migration

Bei der Verwendung mehrerer Subpopulationen entwickeln sich diese zunächst unabhängig voneinander. Um nun aber auch global von den lokal in den Subpopulationen entstandenen Individuen profitieren zu können, wird nach einiger Zeit Information zwischen den einzelnen Populationen ausgetauscht - dies geschieht durch den Austausch einzelner Individuen. Diesen Vorgang nennt man *Migration*. Nach dem Austausch von Individuen entwickeln sich die Populationen wieder eine Zeit lang unabhängig voneinander, bis der Austauschvorgang von neuem beginnt [Poh05].

Der Parameter `Topology` legt fest, welche Topologie bei der Migration verwendet wird. Unter Topologie ist in diesem Zusammenhang zu verstehen, *welche* Unterpopulationen Individuen untereinander austauschen. Dieser Parameter kann in der GEATbx die folgenden drei Werte annehmen:

- 0: Vollständige Netzstruktur ohne Einschränkungen, d.h. Individuen können zwischen *allen* Subopulationen ausgetauscht werden.
- 2: Ringstruktur, d.h. die Subpopulationen werden ringförmig angeordnet und Individuen können nur in die im Ring folgende Subpopulation migrieren.
- 1: Nachbarschaftsstruktur, ähnlich zur Ringstruktur. Jedoch können Individuen in beiden Richtungen zwischen den Nachbarn transferiert werden.

Der Parameter `Rate` bestimmt, welcher Anteil jeder Population bei der Migration migriert werden soll. Dabei können Werte im Bereich von $[0, 1]$ angegeben werden. Die Werte repräsentieren prozentuale Angaben; so sorgt also z.B. das Setzen des Parameters auf den Wert 0.20 dafür, dass 20% der Individuen bei der Migration ausgetauscht werden. Mittels des Parameters `Interval` lässt sich einstellen, wie oft (d.h. nach welcher Anzahl von Generationen) eine Migration stattfinden soll.

2.3.2. Testläufe mit variierenden Populationsgrößen

Für die Durchführung dieser Testreihe wurde das bereits beschriebene Skript verwendet; variiert wurden die folgenden Parameter:

Subpopulationen: 1, 5, 8 und 12.

Individuen pro Population: 15, 30, 45, 60.

Für die 16 möglichen Kombinationen wurden jeweils 20 Testläufe durchgeführt und am Ende der Mittelwert berechnet. Insgesamt umfasste die Testreihe demnach 320 Testläufe. Die Anzahl der Generationen war hierbei stets auf 100 fixiert.

Die Ergebnisse der Testläufe sind in Tabelle 2.1 aufgeführt.

2.3.2.1. Interpretation der Ergebnisse

Wie man an den Testergebnissen sieht, erhöht sich die Güte des Resultats sowohl mit der Anzahl der *Subpopulationen* als auch der *Individuen pro Population*. Innerhalb der Testläufe mit gleicher Individuenanzahl verbessert sich der Mittelwert deutlich mit steigender Subpopulationsanzahl. Und auch zwischen den Testfällen mit steigender Individuenanzahl ist erkennbar, dass die Ergebnismittelwerte besser sind. Abbildung 2.1 zeigt eine grafische Auswertung der Testergebnisse.

Das beste Testergebnis von 2115 km (in der Tabelle fett markiert) erhielten wir erwartungsgemäß bei den jeweiligen Maximalwerten der variierten Parameter.

2.4. Aufgabe d

2.4.1. Roulette Wheel und Stochastic Universal Sampling

Bei dem von 'geatbx' angebotenem Verfahren 'selsus' handelt es sich um das 'Stochastic Universal Sampling'. Sowohl das 'selsus' wie auch das 'Roulette Wheel'-Verfahren,

Nr.	Subpop.	Indiv.	Mittelwert $\bar{x}$	Standardabw. σ_x	Minimalwert in Lauf r, Generation g	Maximalwert in Lauf r, Generation g
1	1	15	3093,30	215,63	2793, $r = 10$, $g = 95$	3715, $r = 20$, $g = 95$
2	5	15	2693,55	128,33	2443, $r = 12$, $g = 97$	2889, $r = 11$, $g = 94$
3	8	15	2644,65	136,63	2361, $r = 15$, $g = 94$	2904, $r = 16$, $g = 99$
4	12	15	2521,25	91,48	2269, $r = 16$, $g = 91$	2662, $r = 14$, $g = 97$
5	1	30	3024,40	243,43	2719, $r = 5$, $g = 89$	3567, $r = 18$, $g = 100$
6	5	30	2464,70	119,89	2253, $r = 2$, $g = 88$	2857, $r = 7$, $g = 97$
7	8	30	2448,15	112,17	2192, $r = 3$, $g = 99$	2622, $r = 7$, $g = 87$
8	12	30	2416,70	103,81	2209, $r = 5$, $g = 96$	2564, $r = 9$, $g = 95$
9	1	45	2838,25	213,74	2493, $r = 6$, $g = 91$	3153, $r = 17$, $g = 83$
10	5	45	2378,00	117,79	2173, $r = 15$, $g = 100$	2632, $r = 3$, $g = 99$
11	8	45	2347,50	105,99	2189, $r = 11$, $g = 99$	2610, $r = 14$, $g = 96$
12	12	45	2309,85	98,98	2168, $r = 9$, $g = 99$	2497, $r = 17$, $g = 90$
13	1	60	2747,60	157,29	2395, $r = 16$, $g = 100$	3071, $r = 8$, $g = 99$
14	5	60	2386,35	101,48	2157, $r = 19$, $g = 96$	2558, $r = 8$, $g = 98$
15	8	60	2320,75	105,88	2143, $r = 4$, $g = 97$	2540, $r = 18$, $g = 99$
16	12	60	**2285,85**	88,91	**2115**, $r = 20$, $g = 90$	2422, $r = 8$, $g = 96$

Tabelle 2.1.: Ergebnisse der Testreihe

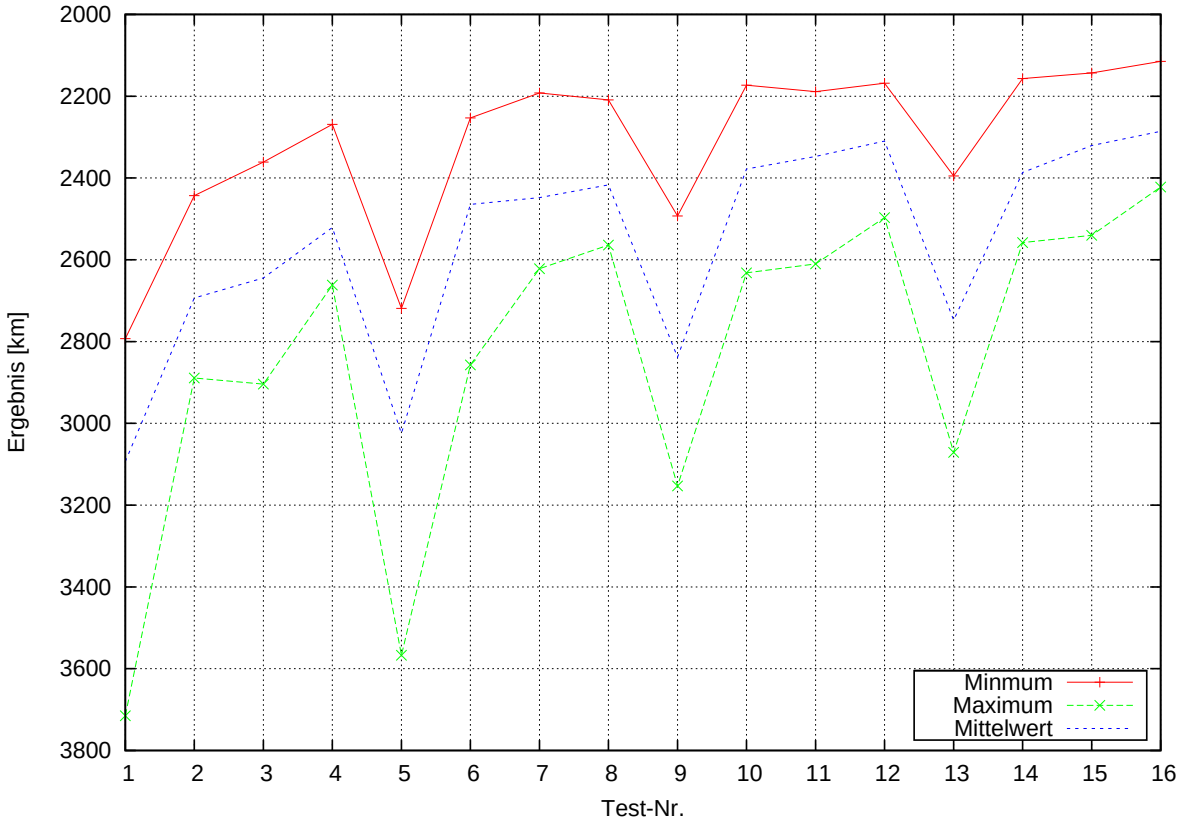

Abbildung 2.1.: Grafische Darstellung der Testreihe: Minimal-, Maximal- und Mittelwert

dienen nach [Poh05] zur Auswahl der Paarungs-Populationen. Dabei erhalten die zu auszuwählenden Individuen einen Bereich zugeteilt, der in seiner Größe relativ zum Fitnesswert und somit auch zur Auswahlwahrscheinlichkeit eines Individuums steht. Beide Verfahren suchen nun per Zufall positionierter Zeiger, die gewünschte Anzahl an Individuen aus. Der Unterschied zwischen dem 'Roulette-Wheel' und dem 'selsus'-Verfahren besteht nun im Ziehungszeitpunkt. Während beim 'Roulette-Wheel'-Verfahren jedes Individuum in einem eigenen Durchlauf unabhängig von den anderen gezogen wird, werden beim 'Stochastic Universal Sampling' alle Individuen auf einmal gezogen. Durch diese Vorgehensweise sind die Auswahl-Wahrscheinlichkeiten voneinander, und damit von der jeweiligen Ziehung abhängig, da es sich um Ziehen ohne Zurücklegen handelt.

Abbildung 2.2 nach [ZT03] verdeutlicht die oben beschriebenen Aspekte. Es werden pro Selektionsverfahren je 4 Individuen, Q_1 bis Q_4 ausgewählt. Beim 'Roulette-Wheel'-Verfahren finden dabei 4 Ziehungen statt, während beim 'selsus'-Verfahren 1 Ziehung durchgeführt wird.

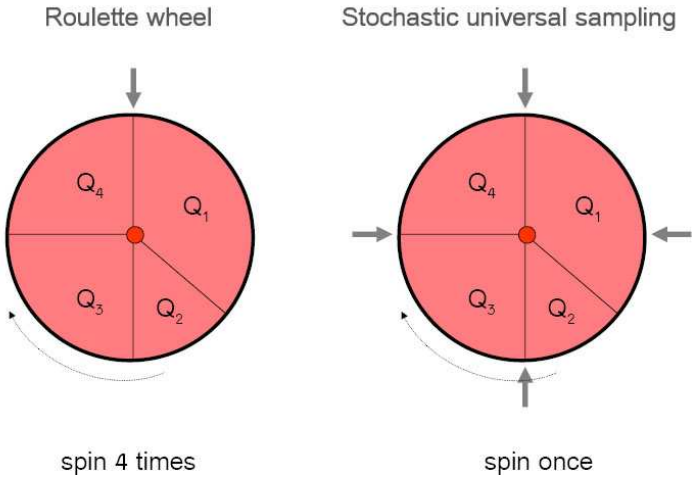

Abbildung 2.2.: Selektionsverf. 'Roulette-Wheel' und 'Stochastic Universal Sampling'.

Nachfolgende Abbildung nach [Poh05] zeigt die von [Poh05] beschriebene Implementierung von 'selsus'. Dabei werden die Zeiger, welche die Indiv. auswählen im gleichen Abstand $1/n$ angeordnet sind. n bezeichnet dabei die Anzahl der auszuwählenden Individuen. Die Zufallszahl entscheidet nun über den Anfangspunkt der Zeigeranordnung, welche alle Individuen gleichzeitig wählt. In der Ziehung des zeigten Beispiels nach [Poh05], ergeben sich hierdurch die Individuen 1, 2, 3, 4, 6, 8.

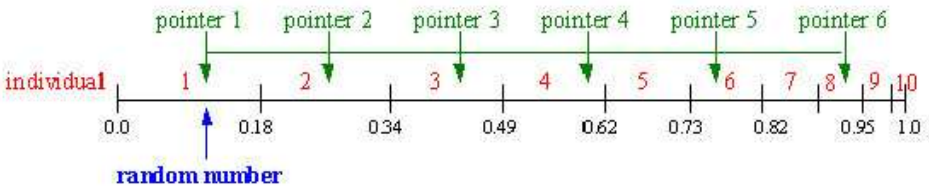

Abbildung 2.3.: Beispielziehung mit Hilfe des 'Stochastic Universal Sampling'.

2.4.2. 'Tournament'-Selektion

Bei der 'Tournament'-Selektion lässt man nach [Poh05] zufällig ausgewählte Individuen gegeneinander antreten. Der jeweils fitteste der 'Tournament'-Gruppe wird schließlich selektiert. Dieser Vorgang wird so oft wiederholt, bis die gewünschte Anzahl an zu selektierenden Individuen erreicht ist.

Folgende Abbildung nach [ZT03] verdeutlicht dies. Dabei werden aus einer Subpopulation zufällig drei Indiv. ausgewählt, die gegeneinander antreten. Das Individuum mit dem höchsten Fitnesswert (hier 7), wird schließlich in den 'Mating-Pool' aufgenommen.

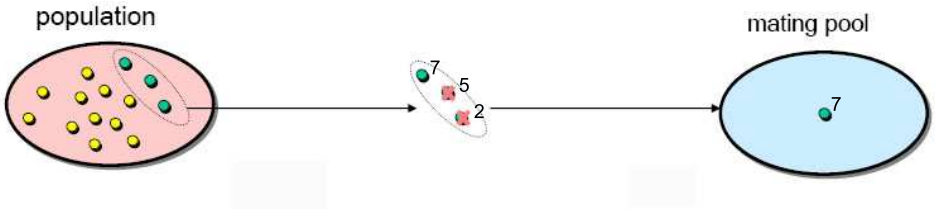

Abbildung 2.4.: Ziehung mit Hilfe der 'Tournament'-Selektion

2.4.3. Testläufe mit variierenden Selektions-Methoden

Für die Durchführung der Testreihen in Aufgabe d), wurde die in 2.2.1 beschriebene Modifikation des Testskriptes benutzt.

Selektions-Methoden: selsus, selrws, seltour

Für jeden der 3 Selektionsarten wurden 20 Testläufe gefahren, womit sich eine Gesamtgröße von 60 Testläufen ergibt. Die in Aufgabe c) bereits durchgeführte Testreihe von 12 Subpopulationen, 60 Individuen und der Selektionsart 'selsus', wurde wiederholt.

Die Ergebnisse ähneln dabei den in Tabelle 2.1 gezeigten Werten. So beträgt der Mittelwert aller wiederholten Testläufe 2225.70, während er in Aufgabe c) bei 2285.85 lag. Vergleiche hierzu die Ergebnisse aus Tabelle 2.2, der Zeile 1, mit den Ergebnissen aus Tabelle 2.1, Testlauf Nr. 16. Die vollständigen Testläufe, zu allen Selektionsarten, sind dem Anhang in Form der entsprechenden Logging-Dateien zu entnehmen.

Selektion	Subpop.	Indiv.	Mittelwert $\bar{x}$	Standardabw. σ_x	Minimalwert in Lauf r, Generation g	Maximalwert in Lauf r, Generation g
selsus	12	60	2225.70	119.76	**2028**, $r = 1$, $g = 92$	2445, $r = 7$, $g = 100$
selrws	12	60	2234.45	113.72	2090, $r = 5$, $g = 99$	2479, $r = 18$, $g = 100$
seltour	12	60	**2163.90**	96.21	2036, $r = 1$, $g = 93$	2422, $r = 16$, $g = 94$

Tabelle 2.2.: Ergebnisse aus 60 Testläufen, mit den Selektionsverfahren 'selsus', 'selrws' und 'seltour'.

2.4.3.1. Interpretation der Ergebnisse

Die Wiederholungstestläufe, die mit Hilfe der Selektionsart 'selsus' erzielt wurden, ähneln den Ergebnissen aus Kapitel 2.3.2. Mit 2225.70, konnte sogar ein leicht verbesserter Mittelwert erzielt werden. Allerdings weichen dabei die einzelnen Werte mit einer Standardabweichung von 119.76 km, auch mehr vom Mittelwert ab, als in den Testläufen der Aufgabe c). Bei fest vorgegebenen Parameterwerten von 12 Subpopulationen und 60 Individuen, liefert das 'Roulette-Wheel'-Selektionsverfahren leicht schlechtere Ergebnisse als mit den ersten 20 Testläufen des 'selsus'-Selektionsverfahrens. Das beste Ergebnis lieferte die 'selsus'-Selektion mit einem minimalen Wert von 2028 Kilometern. Im Mittel aller Werte lieferte jedoch die 'Tournament'-Selektion, mit 2163.90 Kilometern den besten Wert. Die einzelnen Werte zeigen dabei mit einer Standardabweichung von 96.21 km, eine verhältnismäßig geringe Streuung um den Mittelwert. Das schlechteste Ergebnis der Selektionsart 'seltour', liegt mit 2422 ebenfalls jeweils unter den Maximalwerten der beiden ersten Selektionsarten, wie in Tabelle 2.2 gezeigt.

Somit konnte durch Ersetzen der Selektionsart 'selsus' durch das 'Roulette-Wheel'-Verfahren, keine signifikante Änderung der Ergebnisse erzielt werden. Der Minimal- sowie der Mittelwert erreichten ein leicht schlechteres Gesamtergebnis, welches keine genaueren Rückschlüsse zulässt. Durch Verwenden der 'Tournament'-Selektionsart ergab sich optisch eine leichte Verbesserung des Mittelwerts, welche jedoch prozententual ebenfalls nicht ins Gewicht fällt.

Jedoch ergeben sich innerhalb des Verlaufs einer Testreihe gewisse Unterschiede. Abbildung 2.5 zeigt hierzu den Verlauf der kumulierten Mittelwerte, für jeden der drei Testreihen.

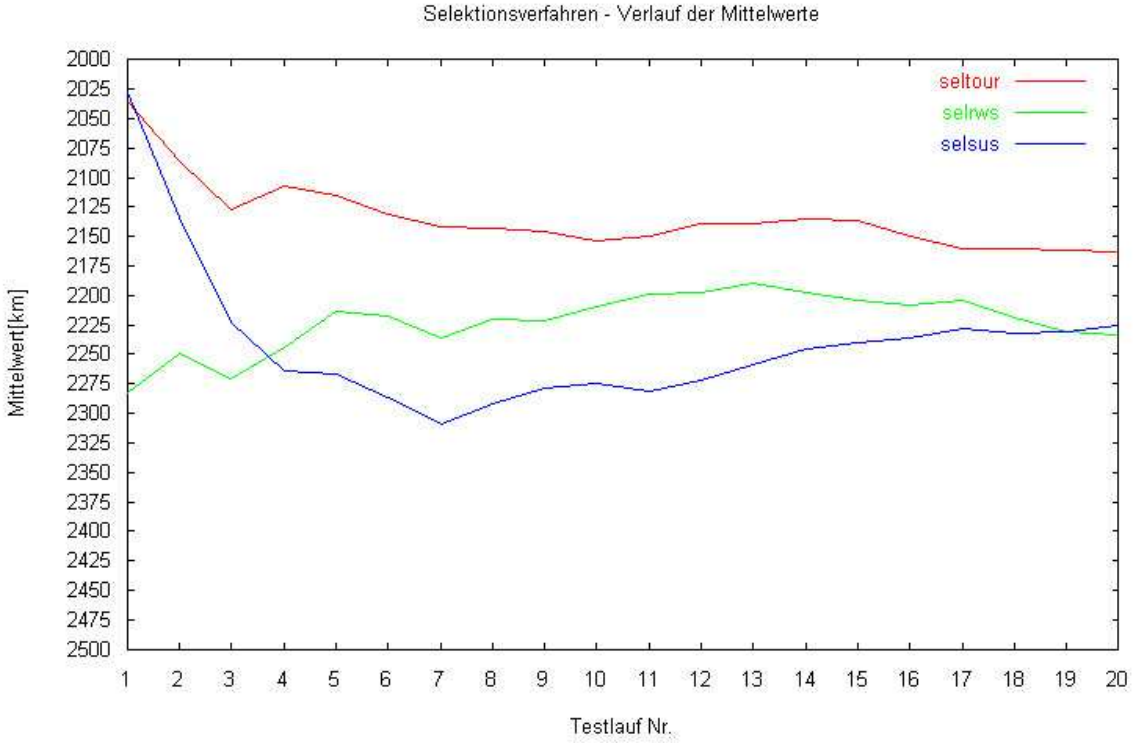

Abbildung 2.5.: Verlauf der Mittelwerte der Selektionsarten 'selsus', 'selrws' und 'seltour'.

Jedes der drei Verfahren zeigt eine Konvergenz der Mittelwerte gegen den Bereich 2150 bis 2250. Das 'selsus'-Verfahren erreicht früh seine besten Wert, fällt dann aber steil ab und kovergiert die restlichen Läufe gegen den Konvergenzbereich. Das Roulette-Wheel-Verfahren beginnt relativ schlecht und konvergiert schnell. Das Tournament-Verfahren erreicht ebenfalls zu Beginn seinen besten Wert und konvergiert auf etwas höherem Niveau als das 'selsus'-Verfahren gegen den Konvergenzbereich.

2.5. Aufgabe e

2.5.1. Rekombination

Im Allgemeinen wird versucht aus den Genen der Eltern, mit Hilfe von Rekombinationsmethoden neue Chromosomen bei den Nachfolgenerationen zu erzeugen. Nach [Erb07] können so, mit neuen Chromosomen auch neue Lösungen gefunden werden. Zuerst werden hierfür nach [Erb07] die im Mating-Pool befindlichen Chromosome mit einer vorbestimmten Wahrscheinlichkeit zur Paarung ausgewählt. 'Unter den ausgewählten Chromosomen werden zufällige Elternpaare (parents) gebildet', [Erb07]. Danach werden je nach Rekombinationsart, die zufällig ausgewählten Gene, unter den Eltern getauscht und auf die Nachkommen übertragen.

2.5.2. Rekombinationsoperator 'recpm'

Bei herkömmlichen Rekombinationsverfahren, wie bspw. dem 'One-Point'- oder auch 'Two-Point'-Crossover, kann es vorkommen, dass bestimmte Gene, innerhalb eines Chromosoms doppelt vorkommen, was in bestimmten Problemstellungen zu illegalen Lösungen führen würde. Gerade beim TSP-Problem, bei dem jeder Ort nur einmal besucht werden darf, können derartige Lösungen nicht akzeptiert werden. Das folgende Beispiel verdeutlicht das entstehen einer illegalen Lösung für das TSP-Problem, anhand des 'Two-Point'-Crossovers. Der dabei entstehende Nachkomme 'Kind1', trägt die Gene '1', '2' und '3' doppelt in sich.

Elter1-Chrom.: 1 2 3 | 4 5 6 | 7 8

Elter2-Chrom.: 4 6 7 | 3 1 2 | 5 8

Kind1-Chrom.: 1 2 3 3 1 2 7 8

Um derartigen Verdopplungen und illegalen Lösungen vorzubeugen, verwendet man das Rekombinationsverfahren 'recpm', welches für 'Partial Matching Recombination' steht. Die von Goldberg und Lingle vorgeschlagene Methode ist dabei besser bekannt als 'PMX', [Ber93]. Dabei werden ebenfalls wieder die Rekombinationsbereiche der beiden Elternchromosome ausgetauscht und den Kindern hinzugefügt. Die restlichen Positionen der Kindchromosome werden mit den jeweils eigenen Genen aufgefüllt, es sei denn, dass diese bereits in den rekombinierten Teilen auftauchen. Ist dies der Fall, so wird die Abbildungsfolge, die sich aus den Rekombinationsteilen ergibt, als Mapping benutzt, um die übrigen Gene entsprechend zu tauschen und dem Kind an entsprechender Stelle hinzuzufügen. Das Beispiel in Abb. 2.6 und Abb. 2.7 soll die genannten Aspekte verdeutlichen.

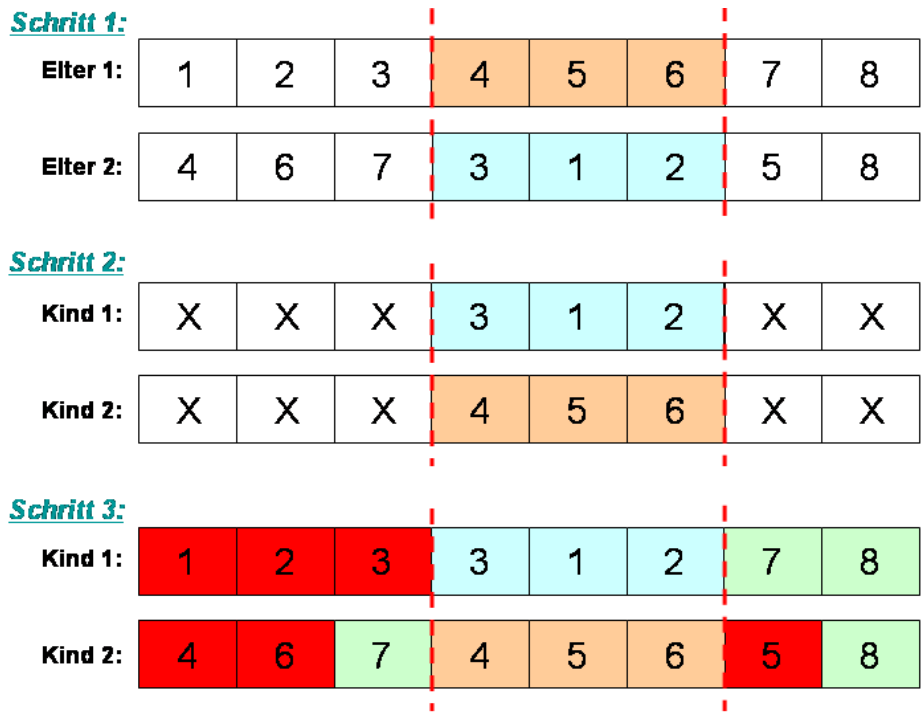

Abbildung 2.6.: Schritt 1-3 des 'Partial Recombination Crossover'

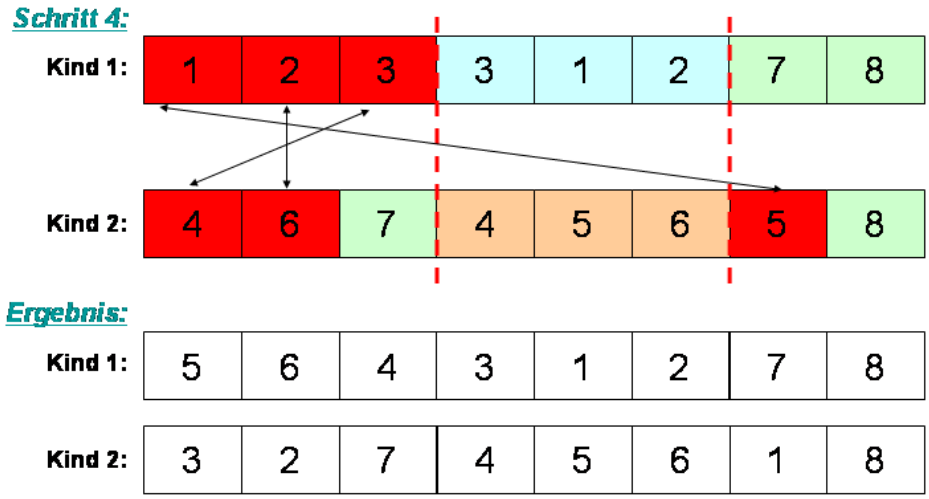

Abbildung 2.7.: Schritt 4 mit Ergebnis des 'PMX'

Im ersten Schritt wird der Rekombinationsbereich bestimmt. Die sich darin befindlichen Gene werden im 2. Schritt zwischen den Eltern ausgetauscht und den jeweiligen Kindern hinzugefügt. Die Gene im Rekombinationsteil legen das später verwendete 'Mapping' fest (hier 3<->4, 1<->5 und 2<->6). Im 3. Schritt werden die Kindchromosome mit den jeweils restlichen Genen der Eltern aufgefüllt. Die grün markierten Gene kennzeichnen dabei Werte, die sich ohne Konflikte (Verdopplungen) hinzufügen lassen. Die rot Markierten kennzeichnen die Gene, bei denen durch einfaches Hinzufügen ein illegaler Zustand erreicht würde. Diese müssen über das erwähnte Mapping, gegeneinander ausgetauscht werden. In diesem Fall wird bspw. die '1' von 'Kind1' gegen die '5' von 'Kind2' ausgetauscht. Dieser Vorgang wiederholt sich solange, bis ein legaler Zustand beider Chromosome erreicht ist.

2.5.3. Cycle Crossover

Der *Cycle Crossover*-Operator stellt sicher, dass die aus zwei gültigen Elternchromosomen erzeugten Nachkommen wiederum gültige Individuen sind. Dazu wird festgelegt, dass jedes Gen im Nachkommen von der selben Stelle eines der Elternchromosome übernommen wird.

Der Algorithmus läuft in den folgenden Schritten ab, hier für die Erzeugung eines Kindes k aus zwei Eltern e_1 und e_2:

1. Bestimmen einer zufälligen Startposition s.
2. Übernehmen des Wertes an der Stelle s an die selbe Stelle von k: $k[s] = e_1[s]$.
3. Suchen der Position i in e_2, an welcher der soeben gefundene Wert $e_1[s]$ steht.
4. Übernehmen des Wertes aus e_1 nach k, welcher an der Stelle i steht: $k[i] = e_1[i]$.

Dieser Vorgang wird solange wiederholt, bis sich ein Kreis (*cycle*) gebildet hat, d.h. man zu einem bereits übernommenen Wert gelangt. Abschließend werden die verbleibenden Stellen unverändert aus dem zweiten Elternteil übernommen.

2.5.3.1. Implementierung in Matlab

Listing 2.3 zeigt unsere Implementierung des Cycle Crossover-Operators in Matlab. Dabei haben wir die Struktur der Funktion übernommen aus einer der GEATbx beiliegenden Operator-Funktionen, nämlich der `recmp.m`.

```
% RECombination Cycle
%
% Syntax:  NewChrom = reccycle(Chrom, RecRate)
%
% Input parameters:
%    Chrom     - Matrix containing the chromosomes of the old
%                population. Each row corresponds to one individual.
%    RecRate   - Probability of crossover ocurring between pairs
%                of individuals.
%
% Output parameter:
%    NewChrom - Matrix containing the chromosomes of the population
%                after mating, ready to be mutated and/or evaluated,
%                in the same format as Chrom.
%
% Basiert auf recmp.m der GEATbx.

% Authors:  Andre Erb, Jan Tammen
% History:  22.06.07      file created


function Children = recgp(Parents, RecRate);

    % Groesse der Population und Chromosomlaenge bestimmen
    [sizePop, sizeChrom] = size(Parents);

    % Keine Rekombination moeglich
    if sizeChrom < 2,
        Children = Parents;
        return;
    end

    % Wenn Rate nicht uebergeben wurde, Rate leer setzen
    if nargin < 2 | isnan(RecRate),
        RecRate = [];
    end

    % Wenn Rate leer ist, auf Default-Wert setzen
    if isempty(RecRate),
```

```
40          RecRate = 0.7;
41      end
42
43      % Anzahl der Paare und Indizes der zu rekombinierenden Chromosome ermitteln
44      numPairs = floor(sizePop/2);
45      indOfChromToCross = find(rand(1, numPairs) < RecRate);
46
47      ungeradeIndizes = 1:2:sizePop - 1;
48      geradeIndizes = 2:2:sizePop;
49
50      for chromIndex = indOfChromToCross
51          % Zwei Elternchromosomen bestimmen
52          parent1 = Parents(ungeradeIndizes(chromIndex), :);
53          parent2 = Parents(geradeIndizes(chromIndex), :);
54
55          % Alle nicht-gleichen Gene in den beiden Chromosomen finden,
56          % um Endlosschleifen zu vermeiden
57          indOfUnequalGenes = find(parent1 ~= parent2);
58
59          % Leere Kindchromosomen mit Laenge der Elternchromosomen anlegen
60          child1Chromosom = parent1;
61          child2Chromosom = parent2;
62
63          % Rekombinieren nur dann, wenn Chromosomen nicht komplett gleich sind
64          if size(indOfUnequalGenes, 2) > 0
65              % Eine zufaellige Startposition bestimmen
66              startPosition = ceil(size(indOfUnequalGenes, 2).*rand(1,1));
67
68              % Startwert ist der Wert an der Startposition des ersten Elternchromosoms
69              startValue = parent1(indOfUnequalGenes(startPosition));
70              cycle = indOfUnequalGenes(startPosition);
71
72              % Zyklus erstellen
73              while startValue ~= parent2(cycle(end)),
74                  % Position bestimmen, an der in Parent1 der Wert der selbe ist, wie der
                        vorherige des Parent2
75                  index = indOfUnequalGenes(find(parent1(indOfUnequalGenes) ==
                        parent2(cycle(end))));
76
77                  % Zyklus speichern
```

```
78                cycle = [cycle index];
79            end % while
80
81            % Gene an Positionen des Zyklus' zwischen den Kindern austauschen
82            temp = child2Chromosom;
83            child2Chromosom(cycle) = child1Chromosom(cycle);
84            child1Chromosom(cycle) = temp(cycle);
85
86        end % if
87
88        % Kindchromosomen in Ergebnismatrix speichern
89        Children(ungeradeIndizes(chromIndex), :) = child1Chromosom;
90        Children(geradeIndizes(chromIndex), :) = child2Chromosom;
91    end % for
92
93    % Falls Populationsgroesse ungerade, letztes Chromosom der Eltern einfach
          uebernehmen
94    if rem(sizePop, 2),
95        Children(sizePop, :) = Parents(sizePop, :);
96    end
97
98 % Ende
```

Listing 2.3: Implementierung Cycle Crossover in Matlab

2.5.4. Testläufe mit variierenden Rekombinations-Methoden

Um die verschiedenen Rekombinationsoperatoren zu testen, wurden jeweils 20 Testläufe mit den folgenden Einstellungen vorgenommen:

Anzahl Subpopulationen: 12

Anzahl Individuen: 60

Selektionsoperator: Tournament-Selektion, `seltour`

Mutationsoperator: Kombination aus `mutswap`, `mutmove` und `mutinvert`

Rekombinationsoperator: Variation zwischen `recgp`, `recpm` und dem eigenimplementierten `reccycle`

Tabelle 2.3 zeigt die Ergebnisse der Testläufe.

Rekombination	Mittelwert $\bar{x}$	Standardabw. σ_x	Minimalwert in Lauf r, Generation g	Maximalwert in Lauf r, Generation g	Mittelwert CPU-Zeit [s]
`recgp`	2201,50	106,00	2035, $r = 11$, $g = 100$	2353, $r = 1$, $g = 100$	131,31
`recpm`	**2195,45**	97,81	**2034**, $r = 1$, $g = 82$	2358, $r = 19$, $g = 98$	31,95
`reccycle`	2302,30	**89,05**	2147, $r = 6$, $g = 96$	2449, $r = 3$, $g = 98$	19,65

Tabelle 2.3.: Zusammengefasste Ergebnisse der Testreihe

2.5.4.1. Interpretation der Ergebnisse

Wie man an den Ergebnissen der Testreihe ablesen kann, scheint es zwischen den einzelnen Rekombinationsoperatoren keine signifikanten Unterschiede zu geben. Das beste Mittelwertergebnis dieser Testreihe mit ca. 2195 km wurde unter Verwendung des `recpm`-Verfahrens erreicht. Der von uns implementierte `reccycle`-Operator schnitt bei den Mittelwerten schlechter ab, jedoch erzielte er die geringsten Standardabweichung in der Testreihe. Was man allerdings festhalten muss, ist die Tatsache, dass `recgp` einen erheblich höheren CPU-Aufwand erfordert als der `reccycle`-Operator. Hier sind benötigte Ergebnis-Güte und Rechenzeit gegeneinander abzuwägen. Abbildung 2.8 zeigt nochmals den kompletten Verlauf der Testergebnisse in graphischer Form.

2.6. Aufgabe f

2.6.1. Mutation

Im Allgemeinen wird versucht mit Hilfe von Mutationsmechanismen aus einem bestehenden Individuum, ein Individuum mit neuen Eigenschaften zu erzeugen. Dies geschieht nach [Erb07] indem aus dem Genpool aller Individuen, zufällig ein Gen ausgwählt wird, das über einen bestimmten Mutations-Algorithmus verändert wird. Über Mutation wird oft versucht neue Lösungen zu erzeugen, welche alleine durch die Regeln der Selektion und Rekombination oft nicht erreicht werden könnten. So wird nach [BKTA] durch Mutation 'eine frühzeitige Konvergenz eines Algorithmus verhindert'. Im Folgenden werden hierzu die unterschiedlichen Mutationsoperatoren 'mutswap ', 'mutmove' und 'mutinvert' genauer betrachtet.

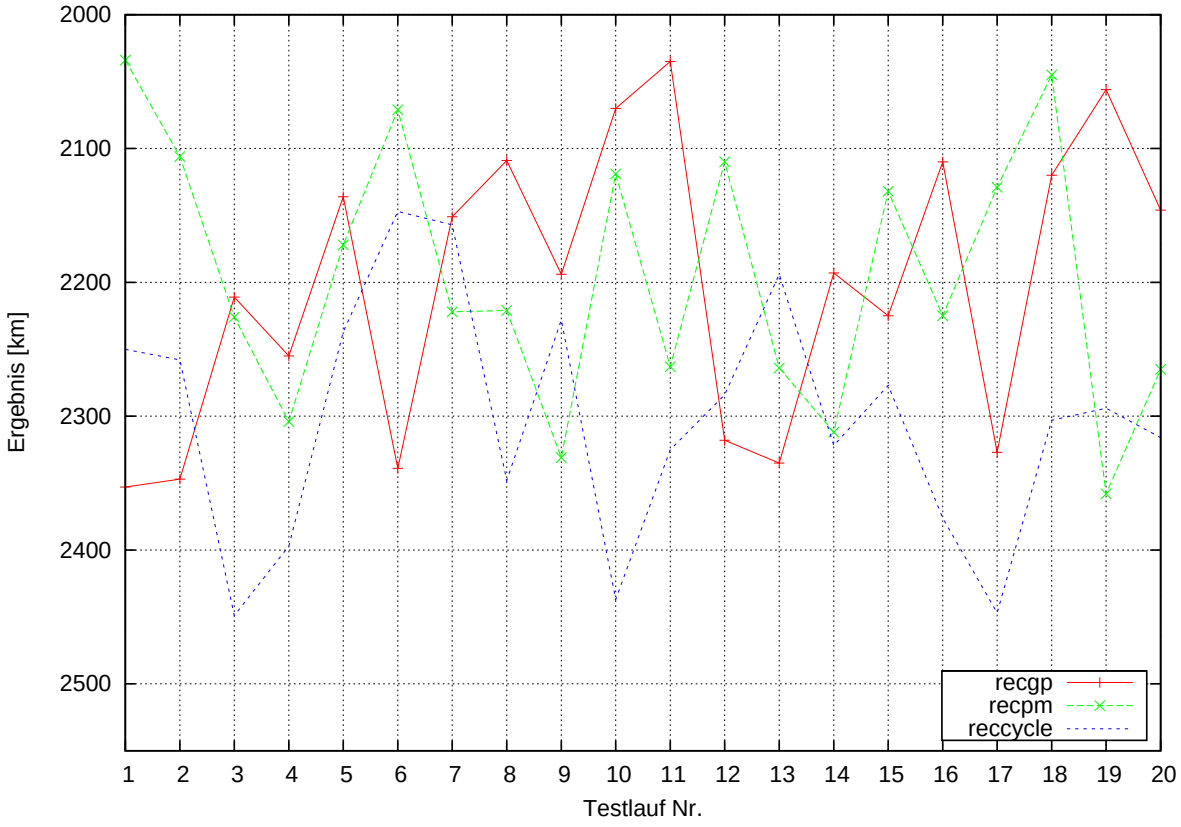

Abbildung 2.8.: Grafische Darstellung der Testreihe mit verschiedenen Rekombinationsoperatoren

2.6.2. Mutationsoperator 'mutswap'

Die Mutationsart 'mutswap' ist ein einfacher Algortihmus, bei dem nach [Poh05] 'eine zufällig ausgwählte Variable eines Indivduums, mit einer anderen Variable des selben Individuums vertauscht wird'. Unter einer Variablen ist ein bestimmtes Gen eines Chromosoms zu verstehen. Die folgende Abbildung 2.9 veranschaulicht dies. Hierbei werden die Positionen der Variabeln mit den Werten 7 und 3 vertauscht.

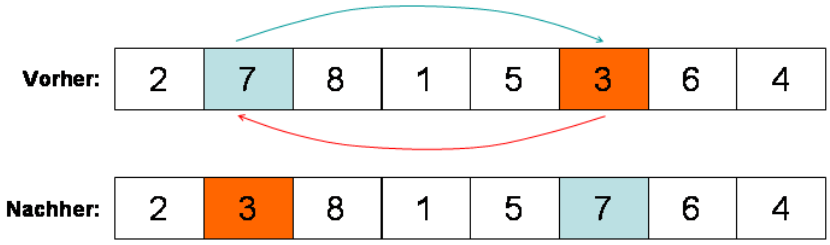

Abbildung 2.9.: Mutation mit 'mutswap'-Operator

2.6.3. Mutationsoperator 'mutmove'

Bei der 'mutmove'-Methode handelt es sich um eine Mutation, bei der die Position i einer Variabeln innerhalb eines Chromosoms an eine neue Position k verschoben wird. Alle Variabeln mit Positionen ab $i+1$ bis k rutschen während der Verschiebung um eine Position nach vorne. Abbildung 2.10 verdeutlicht dies. Hierbei wird die Variable mit dem Wert 7 um vier Positionen nach rechts verschoben. Dabei rutschen die Variabeln mit den Werten 8, 1, 5 und 3 um je eine Position nach vorne.

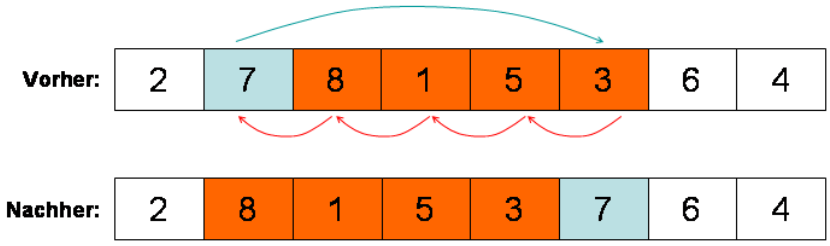

Abbildung 2.10.: Mutation mit 'mutmove'-Operator

2.6.4. Mutationsoperator 'mutinvert'

Bei Verwendung des 'mutinvert'-Operators wird nach [Poh05] 'jede Variablen-Position innerhalb zweier Positionen invertiert'. D. h. es wird zuerst die Sequenz oder Gruppe von Variabeln ermittelt, die zwischen zwei zufälligen Positionen liegen. Die Positionen dieser Sequenz-Gruppe werden anschließend invertiert. Abbildung 2.11 zeigt wie aus der Sequenz 8, 1, 5, 3, die Sequenz 3, 5, 1, 8 entsteht.

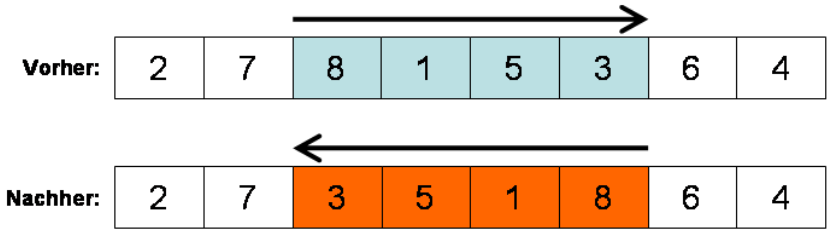

Abbildung 2.11.: Mutation mit 'mutinvert'-Operator

2.6.5. Einsatz aller Mutationsoperatoren

Wie am Anfang des Kapitels 2.6.1 beschrieben, ist das Ziel bei der Mutation, neue Eigenschaften bei Individuen zu erzeugen. Würde man nun 1 Mutationsart einsetzen, so könnte es vorkommen, dass ein bereits ein oder mehrfach mutiertes Individuum, wieder in seinen Ausgangszustand vor der ersten Mutation versetzt wird. Des Weiteren könnte es vorkommen, dass eine bereits bestehende Veränderung im Erbgut der Nackommen ebenfalls wieder zurückmutiert werden würde. D.h. man würde durch erneutes Anwenden quasi alle Mutationsschritte rückgängig machen. So wäre es möglich, dass bei 'mutswap' exakt die gleichen Variabeln zurückgetauscht werden würden. Achtmaliges Anwenden der Operation 'mutmove', würde eine Variable bspw. wieder an ihre Ausgangsposition zurückversetzen. Auch bei 'mutinvert' wäre es möglich, dass eine bereits invertierte Sequenz zurück invertiert wird.

Abbildung 2.12 illustriert am Beispiel von 'mutswap' das unerwünschte Ergebnis nach mehrfachen Anwenden derselben Mutation.

Aus diesem Grund mischt man die angesprochenen Mutationsverfahren innerhalb eines Testlaufs. Dadurch minimiert sich die Wahrscheinlichkeit, mit derselben Mutationsart, eine bereits erreichte und gewünschte Veränderung des Erbgutes wieder versehentlich rückgängig zu machen.

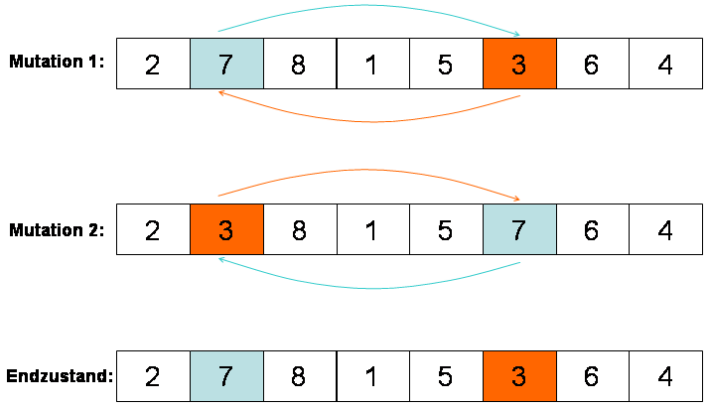

Abbildung 2.12.: Unerwünschter Zustand nach zweimaligen Anwenden von 'mutswap'.

2.6.6. Testläufe mit variierenden Mutationsraten

Für die Durchführung der Testreihen in Aufgabe f), wurde eine angepasste Version des 2.2.1 beschriebenen Testskriptes benutzt.

Mutations-Verfahren: 'mutswap', 'mutmove', 'mutinvert' (kombiniert)

Mut.-Raten: 0.001, 0.005, 0.01, 0.05, 0.08, 0.1, 0.2, 0.3, 0.4, 0.5, 1, 3, 5, 10, 15, 20, 25

Es wurden 16 Testreihen, mit jeweils 20 Testläufen durchgeführt. Jeder der Testläufe wurde mit einer Kombination der drei Muationsarten durchgeführt. Zwischen den einzelnen Testläufen wurde die Muationsrate variiert. Die Ergebnisse aller Testläufe werden hierzu in Tabelle 2.4 dargestellt.

2.6.6.1. Interpretation der Ergebnisse

Tendenziell zeigt sich durch eine Erhöhung der Mutationsrate eine eher negative Auswirkung auf die einzelnen Testergebnisse. So führen die hohen Mutationsraten ab '1', zu einer deutlichen Verschlechterung der Werte. Auch die Ergebnisse zwischen den Raten '0.1' und '0.3', verschlechtern sich zusehens mit der Erhöhung. Die besten Ergebnisse konnten mit Mutationsraten zwischen '0.005' bis '0.05', sowie '0.4' bis '1' erzielt werden. Der minimalste Wert von 2026 Kilometern, wurde dabei mit der Mutationsrate von '0.5' erreicht. Das durschnittlich beste Ergebnis wurde mit der Mutationsrate von '0.01' und einem Mittelwert von 2138.45 erzielt. Die Standardabweichung von 58.19, weist dabei ebenfalls den niedrigsten Wert aller Testläufe auf.

Mut.-Rate	Subpop.	Indiv.	Mittelwert $\bar{x}$	Standardabw. σ_x	Minimalwert in Lauf r, Generation g	Maximalwert in Lauf r, Generation g
0.001	12	60	2387.95	110.29	2217 , $r = 8$, $g = 100$	2611 , $r = 18$, $g = 92$
0.005	12	60	2170.20	80.31	2062 , $r = 11$, $g = 97$	2422 , $r = 16$, $g = 94$
0.01	12	60	**2138.45**	**58.19**	2059 , $r = 7$, $g = 87$	**2249** , $r = 4$, $g = 99$
0.05	12	60	2223.45	101.40	2084 , $r = 1$, $g = 97$	2423 , $r = 18$, $g = 94$
0.08	12	60	2379.30	156.63	2121 , $r = 16$, $g = 96$	2633 , $r = 14$, $g = 88$
0.1	12	60	2456.35	124.13	2257 , $r = 16$, $g = 95$	2742 , $r = 7$, $g = 83$
0.2	12	60	2772.90	136.93	2613 , $r = 8$, $g = 99$	3072 , $r = 14$, $g = 74$
0.3	12	60	3029.95	131.85	2834 , $r = 15$, $g = 84$	3372 , $r = 2$, $g = 72$
0.4	12	60	2159.30	69.12	2047 , $r = 11$, $g = 100$	2278 , $r = 13$, $g = 100$
0.5	12	60	2171.20	93.45	**2026** , $r = 8$, $g = 92$	2339 , $r = 16$, $g = 93$
1	12	60	2217.30	89.41	2065 , $r = 3$, $g = 86$	2403 , $r = 2$, $g = 95$
3	12	60	2449.45	110.05	2229 , $r = 13$, $g = 100$	2606 , $r = 2$, $g = 90$
5	12	60	2677.10	138.78	2424 , $r = 1$, $g = 93$	2898 , $r = 20$, $g = 72$
10	12	60	3061.95	172.52	2769 , $r = 13$, $g = 92$	3334 , $r = 12$, $g = 90$
15	12	60	3287.40	161.55	2997 , $r = 17$, $g = 100$	3582 , $r = 13$, $g = 95$
20	12	60	3395.55	110.47	3151 , $r = 20$, $g = 53$	3600 , $r = 5$, $g = 95$

Tabelle 2.4.: Ergebnisse aus 16 Testreihen a 20 Testläufe, mit kombinierten Mutationsverfahren und unterschiedlichen Mutationsraten.

2.7. Aufgabe g

2.7.1. Optimale Parameter

Durch die in dieser Arbeit durchgeführten Testläufe ergeben sich für uns die optimalen Parameter wie folgt:

Anzahl Generationen: 100

Anzahl Subpopulationen: 12

Anzahl Individuen: 60

Selektionsoperator: Tournament Selection, `seltour`

Rekombinationsoperator: Recombination Partial Matching, `recpm`

Mutationsoperator: Kombination aus `mutswap`, `mutmove` und `mutinvert`

2.7.2. Testläufe mit variierenden TSP-Problemen

Abschließend führen wir jeweils 20 Testläufe mit den o.g. Parametern und drei verschiedenen TSP-Problemen durch; die Ergebnisse sind in Tabelle 2.5 aufgezeigt.

TSP-Bsp.	Sub-pop.	Indiv.	Mittelwert $\bar{x}$	Standardabw. σ_x	Minimalwert in Lauf r, Generation g	Maximalwert in Lauf r, Generation g
`bays29`	12	60	2140,65	87,39	2028, $r = 13$, $g = 96$	2375, $r = 6$, $g = 95$
`bayg29`	12	60	1692,90	39,54	1610, $r = 16$, $g = 89$	1746, $r = 13$, $g = 97$
`berlin52`	12	60	11218,55	660,23	9854, $r = 20$, $g = 100$	12326, $r = 2$, $g = 99$
`berlin52`[1]	12	60	8262,10	278,77	7786, $r = 18$, $g = 362$	8931, $r = 7$, $g = 394$
`berlin52`[1]	20	100	**8110,45**	**193,89**	**7734**, $r = 1$, $g = 336$	8424, $r = 8$, $g = 330$

[1] Anzahl der Generationen: 400

Tabelle 2.5.: Ergebnisse der Testreihe

2.7.2.1. Interpretation der Ergebnisse

Der erreichte Wert für `bays29` erschien uns mit 2028 nahe genug am Idealwert 2020. Für `bayg29` konnte sogar der bestbekannte Wert von 1610 (vgl. [Rei94, Appendix]) erreicht werden. Für diese beiden TSP-Beispiele scheinen unsere gefunden Einstellungen demnach sehr gut zu funktionieren. Im Gegensatz dazu verfehlten wir bei dem Problem mit 52 Knoten den Optimalwert 7542 um Längen. Daher versuchten wir, durch Heraufsetzen der Generationen hier noch Verbesserungen zu erlangen, was uns in Maßen auch gelang. Durch Einsatz von 400 Generationen wurden die Ergebnisse deutlich verbessert, wie die Tabelle zeigt.

Durch die Erhöhung der Subpopulationen sowie der Anzahl der Individuen im letzten Testlauf konnten weitere leichte Verbesserungen erreicht werden. So ergibt sich der von uns erreichte Minimalwert zu 7734.

A. Quellcode Testskript

```
%%% Autoren: Andre Erb, Jan Tammen
%%% Erstellt: 2007-05-25

% Fuer Aufgabe c), je 20 Testlaeufe, insgesamt: 4 * 4 * 20 = 320
subpopulations = [1, 5, 8, 12];
individuals = [15, 30, 45, 60];
numTestRuns = 20;
numSubpopRuns = 4;
numIndivRuns = 4;

% Default-Parameter holen
GeaOpt = geaoptset(tbx3perm);

% Subpopulationen
for indiv = 1 : numSubpopRuns

   % Individuen pro Subpopulation
   for subpop = 1 : numIndivRuns

      % Werte initialisieren
      min = 1000000; % Minimalwert
      minNumGen = 0; % Generation, in der Minimalwert auftrat
      minNumRun = 0; % Testlauf, in dem Minimalwert auftrat
      max = 0;       % Maximalwert
      maxNumGen = 0; % Generation, in der Maximalwert auftrat
      maxNumRun = 0; % Testlauf, in dem Maximalwert auftrat

      % Alle Ergebnisse in einem Array ablegen, damit spaeter Mittelwert/Std.abw.
      % berechnet werden koennen
      results = zeros(numTestRuns, 1);
      index = 1;    % Index des Ergebnis-Arrays

```

```
33      % Schleife fuer die Testlaeufe
34      for run = 1 : numTestRuns
35         disp(sprintf('Individuen: %d, Subpop: %d, Testlauf: %d', individuals(
               indiv), subpopulations(subpop), run));
36
37         % Setzen der Parameter
38         GeaOpt = geaoptset( GeaOpt ...
39          % Anzahl der Subpopulationen
40            , 'NumberSubpopulation',     subpopulations(subpop) ...
41            % Anzahl der Individuen pro Subpopulation
42            , 'NumberIndividuals',       individuals(indiv) ...
43            , 'Output.TextInterval',     0 ...
44            ...
45            , 'Output.GrafikInterval',   0 ...
46            , 'Output.GrafikMethod',     [111111] ...
47            , 'Output.GrafikStyle',      [514143] ...
48            ...
49            , 'Output.StatePlotInterval', 0       ...
50            , 'Output.StatePlotFunction', 'plottsplib' ...
51            ...
52            % Ende nach 100 Generationen
53            , 'Termination.Method',      [1] ...
54            , 'Termination.MaxGen',      100 ...
55            ...
56            % Selektionsverfahren: selsus, selrws, seltour
57            , 'Selection.Name',          'selsus' ...
58            % Rekombination: recgp, recpm, reccycle
59            , 'Recombination.Name',      'recgp' ...
60             % Mutationsrate
61            , 'Mutation.Rate',           0.9     ...
62            % Mutationsverfahren
63            , 'Mutation.Name',       {'mutswap', 'mutmove', 'mutinvert'} ...
64            );
65
66         % Speichern der Ergebnisse
67         GeaOpt = geaoptset( GeaOpt ...
68            , 'Output.SaveTextInterval', 20 ...
69            , 'Output.SaveTextFilename', 'tsp.log' ...
70            );
71
```

```
72          % Auswaehlen der Zielfunktion
73          GeaOpt = geaoptset( GeaOpt , 'System.ObjFunFilename', 'objtsplib');
                objfun = [];
74
75          % TSP-Daten laden
76          global TSPLIB_FILENAME;
77          global TSPLIB_NAME;
78          TSPLIB_FILENAME = 'bays29'; TSPLIB_NAME = '';
79          tsp_readlib(TSPLIB_FILENAME);
80          GeaOpt = geaoptset( GeaOpt , 'System.ObjFunAddPara', {TSPLIB_NAME});
81
82          % Ober- und Untergrenzen der Zielfunktion holen
83          VLUB = geaobjpara(GeaOpt.System.ObjFunFilename, 1, GeaOpt.System.
                ObjFunAddPara)
84          GeaOpt = geaoptset( GeaOpt , 'System.ObjFunVarBounds', VLUB); VLUB =
                [];
85
86          % Aufruf der Haupt-Funktion
87          [xnew, GeaOpt] = geamain2(objfun, GeaOpt, VLUB, []);
88
89          BestResult = GeaOpt.Run.BestObjectiveValue;
90          PosBest = GeaOpt.Run.PosBest;
91
92          % Min- und Max-Wert bestimmen
93          if BestResult < min
94             min = BestResult;
95             minNumGen = PosBest(1);
96             minNumRun = run;
97          end
98          if BestResult > max
99             max = BestResult;
100            maxNumGen = PosBest(1);
101            maxNumRun = run;
102         end
103
104         % Speichern des Ergebnisses und Inkrementieren des Array-Indizes
105         results(index) = BestResult;
106         index = index+1;
107
108     end
```

```
109
110       % Ausgabe der Ergebnisse
111       disp(results);
112       disp(sprintf('Mittelwert von %g Runs: %.2f', numTestRuns, mean(results)))
              ;
113       disp(sprintf('Standardabweichung von %g Runs: %.2f', numTestRuns, std(
              results)));
114       disp(sprintf('Min: %g (Run: %g, Generation: %g), Max: %g (Run: %g,
              Generation: %g)', ...
115           min, minNumRun, minNumGen, max, maxNumRun, maxNumGen));
116       logToLogfile(GeaOpt.Output.SaveTextFilename, ...
117           sprintf('Ergebnisse: %s\nMin: %g (Run: %g, Generation: %g), Max: %g (
                  Run: %g, Generation: %g)\nMittelwert von %g Runs: %.2f\
                  nStandardabweichung: %.2f', ...
118           sprintf('%g ', results), min, minNumRun, minNumGen, max, maxNumRun,
                  maxNumGen, ...
119           numTestRuns, mean(results), std(results)));
120     end
121
122 end
```

Quellen

[Ber93] Bergmann, Andreas: *Adaptive Steuerung von Strategieparametern bei genetischen Algorithmen*. Universität Bonn. http://www.informatik.uni-bonn.de/III/forschung/publikationen/tr/. Version: 1993

[BKTA] Büning, Prof. Dr. H. K. ; Kramer, Oliver ; Ting, Chuan-Kang ; Aslan, Daniel: *Genetische Algorithmen - Ausarbeitung im Rahmen des Seminars „Evolutionäre Algorithmen“*. http://wwwcs.uni-paderborn.de/cs/ag-klbue/de/courses/ws04/ea/students/ga_report.pdf

[Erb07] Erben, Prof. Dr. W.: *Genetische Algorithmen - Klassische Genetische Algorithmen*. Skriptum HTWG Konstanz, SS 2007

[Poh05] Pohlheim, Hartmut: *GEATbx: Genetic and Evolutionary Algorithm Toolbox for use with MATLAB Documentation*. http://www.geatbx.com/docu/index.html. Version: 2005

[Rei94] Reinelt, Gerhard: *Lecture Notes in Computer Science*. Bd. 840: *The Traveling Salesman, Computational Solutions for TSP Applications*. Springer, 1994 http://link.springer.de/link/service/series/0558/tocs/t0840.htm. – ISBN 3–540–58334–3

[ZT03] Zitzler, Eckart ; Thiele, Lothar: *Evolutionary Methods in Multi-Objective Optimization*. http://www.tik.ee.ethz.ch/~thiele/NUS/EMO1.pdf. Version: 2003