HOCHSCHULE KONSTANZ TECHNIK, WIRTSCHAFT UND GESTALTUNG

Fakultät Informatik

Ausarbeitung im Fach Genetische Algorithmen

Prof. Dr. Wilhelm Erben

Sommersemester 2008

von

Max Nagl

nagl@fh-konstanz.de

Andreas Hofmann

ahofmann@htwg-konstanz.de

Inhaltsverzeichnis

1 Einleitung

Diese Ausarbeitung beschäftigt sich mit einer Variante des Problem des Handlungsreisenden, auch bekannt als Traveling Salesman Problem (TSP). Dabei geht es ursprünglich darum, für eine gegebene Menge von Städten die kürzeste Route zu finden, die durch alle Städte verläuft und wieder in am Ausgangspunkt endet. Die von uns betrachte Variante unterscheidet sich vom Original nur insofern, als dass der Startpunkt nicht identisch mit dem Endpunkt sein muss, wir also nicht unbedingt eine zyklische Route erhalten.

Da es sich um ein np-schweres Problem handelt, kann ein effizienter konventioneller Algorithmus nicht ohne weiteres gefunden werden. Wir betrachten ausschließlich die Lösung des TSP mittels genetischer Algorithmen. Zu diesem Zwecke kommt Matlab mit GEATbx, einer speziellen Toolbox für genetische und evolutionäre Algorithmen [Pohlheim, 2006], zum Einsatz.

Konkret arbeiten wir in dieser Ausarbeitung zunächst mit einem Datensatz, der 29 bayrische Städte und deren jeweilige Entfernungen zueinander enthält. Für diese Städte ist die optimale Route gesucht.

Wir werden zunächst ein vorhandenes Skript auf das Problem loslassen und dessen Ergebnisse und Parameter beobachten. Dann werden wir nacheinander Populationsgrößen, Unterpopulationsgrößen, Migrationsparameter sowie Selektions-, Rekombinations- und Mutationsverfahren variieren, um die jeweils optimalen Werte zu ermitteln. Im Anschluss erfolgt noch ein letztes Finetuning und Test des optimalen Algorithmus' auf einen neuen Datensatz.

Diese Ausarbeitung entstand ursprünglich mit der Aufgabenstellung von 2007 [Erben, 2007]. Da diese nicht mehr aktuell ist, haben wir die Beschreibung des Dateiformats (Kapitel 2) und die Beschreibung und Implementierung des Rekombinationsverfahren Order Crossover (Abschnitt 7.2) aus der aktuellen Aufgabenstellung [Erben, 2008a] hinzugenommen.

2 Dateiformat

Beschreiben Sie das in `bays29.tsp` verwendete Datenformat. [Erben, 2008a]

Der Dateiname `bays29.tsp` steht für `29` Städte in Bayern (`bay`) mit Entfernungen als Straßendistanz (`s`). Die hier nicht behandelte Datei `bayg29.tsp` enthält die Entfernungen in Luftlinie.

Die Datei enthält die für die Problemstellung benötigten Daten im Klartextformat. Im ersten Teil sind pro Zeile jeweils ein Schlüssel-Wert-Paar gegeben, getrennt durch einen Doppelpunkt und ein Leerzeichen. Hier sind der Name der Datei/Problemstellung (bays29), der Problemtyp (TSP), ein beschreibender Kommentar, die Anzahl der Dimensionen (29), und Formatierungsinformationen für die Matrix mit den Kantengewichten angegeben.

Daraufhin folgt der durch eine Zeile mit dem Inhalt `EDGE_WEIGHT_SECTION` eingeleiteter Bereich, in dem die 29x29-Matrix mit den Kantengewichten direkt in Textform abgebildet wird. Dieser Abschnitt existiert nur, weil der Parameter `EDGE_WEIGHT_TYPE` im ersten Abschnitt auf `EXPLICIT` gesetzt wurde. Das Format dieses Abschnitts wird gesteuert über den Parameter `EDGE_WEIGHT_FORMAT` (hier `FULL_MATRIX`, möglich wäre auch `UPPER_ROW`).

Im Anschluss folgt noch die `DISPLAY_DATA_SECTION`, wo die Koordinaten der Städte angegeben sind, welche hier nur für die grafische Darstellung benötigt werden. Wäre im ersten Teil der Parameter `EDGE_WEIGHT_TYPE` auf `EUC_2D` statt auf `EXPLICIT` gestellt, würden diese Koordinaten zur euklidischen Berechnung der Entfernungen/Gewichte verwendet werden.

3 Erstaufrufe von demotsplib.m

> Führen Sie einige Erstaufrufe von `demotsplib.m`, angewandt auf `bays29.tsp`, durch. Versuchen Sie herauszufinden, wie gut oder schlecht die Ergebnisse dieser Testläufe sind. Welche Parameter und Optionen sind voreingestellt? [Erben, 2007]

Die voreingestellten Einstellungen können Tabelle 3.1 entnommen werden. Sie werden teilweis in `demotsplib.m` definiert und teilweise aus `tbx3perm.m` (Teil von GEATbx) eingelesen.

Parameter	Wert
Anzahl Unterpopulationen	6
Populationsgröße	50
Migrationstopologie	0
Migrationsrate	0.1
Migrationsintervall	20
Selektionsverfahren	Default value (selsus)
Selektions-GenerationGap	0.95
Rekombinationsverfahren	Default value (recdis)
Recombination.Rate	1
Mutationsverfahren	'recgp'
Mutationsrate	Default value (1)
Mutation.Range	[1.0, 0.5, 0.3, 0.3, 1.0, 0.5]
Textausgabe	Jede 5. Generation
Diagramausgabe	Jede 10. Generation
Wegdarstellung	Jede 10. Generation
Abbruchbedingung	Nach 400 Generationen

Tabelle 3.1: Voreinstellungen in `demotsplib.m`

In seiner ursprünglichen Form läuft das Skript einmal durch und gibt seine Ergebnisse aus. Um mehrere Durchläufe laufen zu lassen und den Durchschnitt zu berechnen legen wir folgenden Wrapper um das Skript:

```
1 fid = fopen('original.csv', 'w');
2 fprintf(fid,'Best, Time\n');
3 fclose(fid);
4
5 for durchlauf = 1:10
6     tic;
```

```
7
8       % Hier das ursprüngliche Skript
9
10      fid = fopen('original.csv', 'a+');
11      fprintf(fid,'%4d,%4.4fs\n', GeaOpt.Run.BestObjectiveValue, toc);
12      fclose(fid);
13  end
```

Desweiteren wurde `TSPLIB_FILENAME = '';` durch `TSPLIB_FILENAME = 'bays29'` ersetzt, damit der Dateiname nicht von Hand ausgewählt werden muss.

Das Skript wird somit zehnmal ausgeführt und das beste Ergebnis jedes Durchlaufs zusammen mit der Dauer in der Datei `original.csv` abgespeichert. Diese Ergebnisse können Tabelle 3.2 entnommen werden. Die Werte sind schon recht gut, nur die Laufzeit ist inakzeptabel lang.

Versuch	Best	Time [s]
1	2026	137,40
2	2157	141,96
3	2074	135,85
4	2144	135,01
5	2080	142,15
6	2172	141,48
7	2146	136,35
8	2156	135,62
9	2034	133,57
10	2199	139,14
Schnitt	2119	137,85

Tabelle 3.2: Ergebnisse aus 10 Testdurchläufen von `demotsplib.m`

Von einem einzelnen Durchlauf wurde mit Hilfe der Funktion „Profile Code“ in Matlab eine Auflistung erzeugt die darstellt welche Funktion wie oft aufgerufen wurde und wie lange die Verarbeitung dauert.

In Abbildung 3.1 zeigt sich, dass ein Großteil der Zeit für Rekombination (`recombin` und `recgn`) verwendet wird. Auf etwaige Optimierungen im Bereich Rekombination kommen wir in Abschnitt 7 zurück. Ein weiterer großer Anteil der Laufzeit wird für die grafische Darstellung(resplot) verwendet. Um dies genauer zu untersuchen wurde für die Darstellung ein weiterer Test gefahren.

Um festzustellen wie viel Rechenzeit für die Darstellung benötigt wurde, wurde bei dem selben Skript einfach Optionen für die Text- sowie Diagramm- und Wegausgabe auf NaN (Not a Number) gesetzt und nochmal durchlaufen lassen.

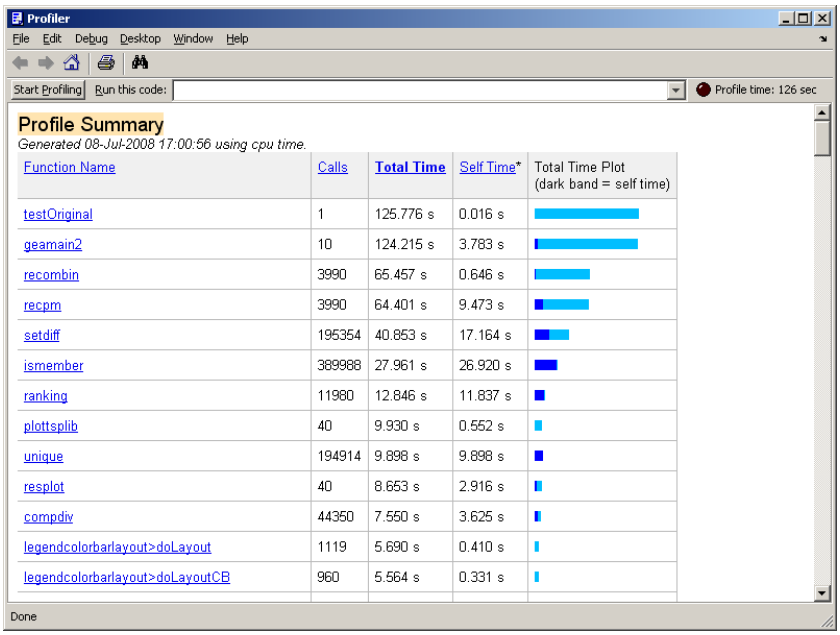

Abbildung 3.1: Laufzeit-Profil von `demotsplib.m`

Versuch	Best	Time [s]
1	2135	117,94
2	2074	115,47
3	2357	114,84
4	2145	116,90
5	2165	111,01
6	2106	116,25
7	2188	111,83
8	2155	107,06
9	2091	113,60
10	2159	116,21
Schnitt	2158	114,11

Tabelle 3.3: Testwerte des Originalskripts ohne Ausgabe

Wenn man die Tabellen 3.2 und 3.3 vergleicht, erkennt man dass die grafische Ausgabe circa 20% der Laufzeit ausmacht. Es ist zu erwarten, dass dieser Anteil in zukünftigen Testserien drastisch ansteigt, da die Laufzeit der Ausgabe konstant bleibt, während sich die Laufzeit des Algorithmus' durch unsere Optimierungen zunehmend verbessern wird. Der leicht schlechtere Durschnittswert in Tabelle 3.3 ist höchstwahrscheinlich Zufall. Bei allen weiteren Tests wurde kleine Ausgabe verwendet, da diese bei automatisierten Durchläufen ohnehin keinen Sinn ergibt und die Ergebnisse nur verzerren würde.

4 Automatisierung der weiteren Testläufe

> In einigen der nachfolgenden Aufgabenteile sollen einzelne Parameter ceteris paribus in von Ihnen festzulegenden Schrittweiten verändert werden und jeweils anschließend einige Testläufe erfolgen. Sorgen Sie dafür, dass dies weitgehend programmgesteuert geschieht und dass die einzelnen Testergebnisse in für statistische Auswertungen geeigneter Form festgehalten werden. Beschreiben Sie die Gesamtarchitektur Ihres Testsystems, d. h. die von Ihnen hierfür verwendeten Dateien und Scripts. [Erben, 2007]

Die Architektur unseres Systems besteht aus mehreren Schichten. Die einfachste Art alle Testserien nacheinander durchzuführen besteht im Ausführen des Skripts `TestAll.m`. Dieses ruft für jeden zu variierenden Parameter eine eigene Testfunktion auf.

Diese Testfunktionen können auch separat direkt aus der Konsole aufgerufen werden. Die Funktion `TestPopSize` zum Beispiel testet die Populationsgrößen. In jeder Testfunktion werden mehrere Tests gefahren. Dabei wird der beste, der schlechteste sowie der Durchschnittswert berechnet und abgespeichert. Bei der Zeitmessung wird lediglich der Durchschnittswert abgespeichert.

Die unterste Schicht bilden Funktionen, die den Aufruf mehrerer Tests automatisieren und die GEATbx-Optionen für das ursprüngliche Skript parametrisieren.

4.1 Das Skript TestAll.m

In diesem Skript werden alle Testfunktionen nacheinander aufgerufen. Es dient lediglich als Abkürzung um den gesamten im Rahmen dieser Ausarbeitung erarbeiteten Testablauf zu wiederholen.

4.2 Die Funktion TestCase

Parameter	Beschreibung
Filename	Der Dateinamen der Datei mit den Entfernungsdaten
TestName	Die Bezeichnung des aktuellen Tests
Generations	max. Anzahl an Generationen
Minutes	max. Anzahl an Minuten
PopSize	Die Populationsgröße
Subpop	Anzahl der Unterpopulationen
Migrationtopology	Die Migrationstopologie
Migrationrate	Die Migrationsrate
Migrationintervall	Das Migrationsintervall
Selection	Das Selektionsverfahren
Recombination	Das Rekombinationsverfahren
Swap	Rate von mutswap
Move	Rate von mutmove
Invert	Rate von mutinvert

Tabelle 4.1: Parameter der Funktion `TestCase`

Diese Funktion ist eine Abwandlung des Originalskripts. Die Optionen des Genetischen Algorithmus' wurden parameterisiert, damit man sie beim Funktionsaufruf übergeben kann. Textuelle sowie grafische Ausgaben wurden deaktiviert.

Die Parameter der Funktion können Tabelle 4.1 entnommen werden.

Rückgabewert ist eine Struktur, welche die benötigte Zeit, den besten Wert, sowie den Testnamen enthält.

4.3 Die Funktion TestCases

Parameter	Beschreibung
Filename	Der Dateinamen der Datei mit den Entfernungsdaten
TestName	Die Bezeichnung des aktuellen Tests
TestNumber	Anzahl der Tests
Generations	max. Anzahl an Generationen
Minutes	max. Anzahl an Minuten
PopSize	Die Populationsgröße
Subpop	Anzahl der Unterpopulationen
Migrationtopology	Die Migrationstopologie
Migrationrate	Die Migrationsrate
Migrationintervall	Das Migrationsintervall
Selection	Das Selektionsverfahren
Recombination	Das Rekombinationsverfahren
Swap	Rate von mutswap
Move	Rate von mutmove
Invert	Rate von mutinvert

Tabelle 4.2: Parameter der Funktion `TestCases`

Diese Funktion ruft `TestNumber`-mal hintereinander die Funktion `TestCase` auf und errechnet aus den Ergebnissen der einzelnen Tests den besten, schlechtesten und den Durchschnittswert des *Best Objective Value*. Außerdem wird auch die Durchschnittslaufzeit berechnet.

Wie Tabelle 4.2 entnommen werden kann, sind die Parameter mit denen der Funktion `TestCase` identisch, mit Ausnahme des zusätzlichen Parameters `TestNumber`.

Der Rückgabewert ist eine Struktur, welche den besten Wert, den schlechtesten Wert und den Durchschnittswert beinhaltet, sowie die durchschnittliche Laufzeit und den Testnamen.

4.4 Die Funktion printResult

Parameter	Beschreibung
Filename	Der Dateinamen der Datei in der das Ergebnis gespeichert werden soll
Result	Der Rückgabewert von TestCases

Tabelle 4.3: Parameter der Funktion `PrintResult`

Die Funktion `printResult` gibt als erstes die Werte aus Result auf der Konsole aus und hängt die Werte anschließend in einer Zeile an die Datei mit dem übergebenen Dateinamen an. Die Werte werden durch Kommas getrennt, damit sie schnell in ein Tabellenkalkulationsprogramm eingelesen werden können.

Diese Funktion besitzt keinen Rückgabewert.

4.5 Die TestFunktionen

Es gibt mehrere Testfunktionen, einmal Testfunktionen für die einzelnen Parameter Variationen, z.B. `testPopSize.m`, `testSelection.m` oder `testRecomb.m`. Außerdem gibt es noch `testFinal.m`, die unseren finalen Abschlusstest durchführt.

Der Aufbau aller Testfunktionen ist ähnlich (Beispielcode hier aus `TestPopSize`):

1. Zu Beginn werden die Parameter, welche bei allen Tests innerhalb dieser Testserie identisch sind.

```
1 Number = 10;
```

```
Generations = 100;
Subpop = 6;
Migrationtopology = 0;
Migrationrate = 0.1;
Migrationintervall = 20;
Selection = 'selsus';
Recombination = 'recgp';
Swap = 1;
Move = 1;
Invert = 1;
```

2. Dann wird die Datei zur Speicherung der Ergebnisse vorbereitet:

```
filename = 'popsize.csv';
fid = fopen(filename, 'w');
fprintf(fid, 'PopSize,Best,Average,Worst,Average Time\t\n');
fclose(fid);
```

3. Dann wird der eigentliche Test durchgeführt. Nach jeden Testcase wird das Ergebnis abgespeichert.

```
for i = [2,3,5,7,10,15,20,25,30,40,50,60,75,80,100,125,150]
    Result = TestCases( 'bays29', num2str(i), Number, Generations,
                        1000000, i, Subpop, Migrationtopology,
                        Migrationrate, Migrationintervall,
                            Selection,
                        Recombination, Swap, Move, Invert);
    PrintResult(filename, Result);
end
```

   In diesem Beispiel wurden etliche Kombinationen mit unterschiedlich vielen Populationsgrößen durchgeführt. In den meisten Fällen führen wir zwei getrennte Testserien durch, einmal mit Begrenzung der Laufzeit, einmal mit Begrenzung der Generationen. Das Beispiel oben ist die Anzahl der Generationen durch den Parameter `Generations` begrenzt, für Zeit ist allerdings 1000000 eingetragen, was faktisch unbegrenzt bedeutet. Es wäre sicherlich schöner, an dieser Stelle „NaN“ statt einer sehr hoch gewählten Zahl anzugeben, dies wird von Matlab jedoch nicht akzeptiert.

Die Ausgabedateien werden vor jedem Schreiben geöffnet und danach sofort wieder geschlossen. Dies ermöglicht eine Einsicht in die Daten während der Versuch läuft, was gelegentlich praktisch ist.

5 Populationsgrößen, Unterpopulationen und Migration

Führen Sie einige Testläufe mit unterschiedlichen Populationsgrößen durch. Geben Sie zunächst nur 1 Population vor; variieren Sie dann aber auch die Anzahl der *Unterpopulationen*. Machen Sie sich dabei mit dem Begriff *Migration* vertraut: Beschreiben Sie das zu Grunde liegende Modell und die durch den Parameter `Migration.Topology` zu steuernden Varianten. Welche Rolle spielen die Parameter `Migration.Rate` und `Migration.Interval`? [Erben, 2007]

5.1 Populationsgröße

Für diesen Test lassen wir alle Parameter außer der Populationsgröße konstant. Die genauen Parameterwerte können Tabelle 5.1 entnommen werden. Wir variieren die Populationsgröße in mit zunehmend größerer Schrittweite von 2 bis 150.

Tabelle 5.2 enthält die Ergebnisse der Testserie mit einer Begrenzung der Generationen auf 100, Tabelle 5.3 enthält die Ergebnisse der Testserie mit einer Begrenzung der Zeit auf 30 Sekunden.

Parameter	Wert
Filename	'bays29'
TestNumber	10
Subpop	6
Generations	100
Minutes = 0.5;	
Migrationtopology	0
Migrationrate	0.1
Migrationintervall	20
Selection	'selsus'
Recombination	'recgp'
Swap	1
Move	1
Invert	1

Tabelle 5.1: Verwendete feste Parameter beim Test der Populationsgrößen

PopSize	Best	Average	Worst	Average Time	PopSize/Time	Average*Time
2	4247	4559	4814	1,3	1,59	5751
3	3932	4322	4636	1,8	1,68	7730
5	3741	4117	4424	2,9	1,73	11891
7	3541	4010	4276	4,0	1,74	16125
10	3504	3917	4276	5,2	1,91	20504
15	3441	3741	4098	9,1	1,66	33867
20	3207	3546	3895	11,7	1,7	41602
25	3009	3428	3840	13,8	1,81	47477
30	3008	3365	3708	18,1	1,66	60840
40	2938	3186	3494	25,2	1,59	80280
50	2778	3173	3572	29,8	1,68	94675
60	2634	3190	3430	37,7	1,59	120235
75	2840	3079	3381	47,1	1,59	144912
80	2740	3048	3340	52,5	1,52	160150
100	2444	2864	3146	63,0	1,59	180540
125	2615	2844	3046	77,1	1,62	219194
150	2488	2752	2991	93,4	1,61	256949

Tabelle 5.2: Tests mit verschiedenen Populationsgrößen mit Generationenbeschränkung

PopSize	Best	Average	Worst	Average Time	Average Generations
2	3840	4095	4318	30,1	2488
3	3375	3797	4140	30,1	1846
5	3708	3925	4199	30,1	1169
7	3684	3924	4104	30,1	844
10	3438	3759	4029	30,1	649
15	3272	3676	4081	30,1	389
20	3227	3515	3916	30,1	303
25	3466	3642	3934	30,1	249
30	3048	3487	3966	30,1	195
40	3027	3303	3640	30,2	131
50	2613	3131	3400	30,2	104
60	2798	3179	3530	30,2	80
75	2748	3061	3559	30,3	60
80	2792	3091	3301	30,3	54
100	2722	2970	3258	30,4	41
125	2828	3143	3368	30,4	30
150	2766	3118	3426	30,6	25

Tabelle 5.3: Tests mit verschiedenen Populationsgrößen mit Zeitbeschränkung

In beiden Fällen sieht man, dass sich der durchschnittliche Wert mit steigender Populationsgröße verbessert. Die Laufzeit erhöht sich (beim auf 100 Generationen beschränkten Test) ebenfalls mit steigender Populationsgröße.

Um genauere Aussagen über das Verhältnis von Laufzeit und Ergebnisqualität treffen zu können, wurden in Tabelle 5.2 zwei zusätzliche Spalten eingefügt. Die sechste Spalte zeigt die Populationsgröße geteilt durch die Zeit. Die siebte Spalte zeigt den Durchschnittswert mal die durchschnittlich benötigte Zeit. Der Wert in der sechsten Spalte ist bis auf ein paar Ausreißer sehr konstant. Doppelte Populationsgröße bedeutet damit doppelten Zeitaufwand. Der Wert in der siebten Spalte steigt allerdings mit steigender Populationsgröße. Das bedeutet, dass eine Verdoppelung der Dauer nicht gleichzeitig eine Halbierung des Wertes nach sich zieht.

Schränkt man die Laufzeit ein, flacht die Kurve ab und verschlechtert sich sogar wieder. In Tabelle 5.3 scheint eine Populationsgröße von 100 die besten Ergebnisse zu liefern, deshalb arbeiten wir mit dieser weiter. In Abbildung 5.1 sieht man sehr gut, dass dieser Wert den niedrigsten Average-Wert und Worst-Wert und den zweitniedrigsten Best-Wert liefert.

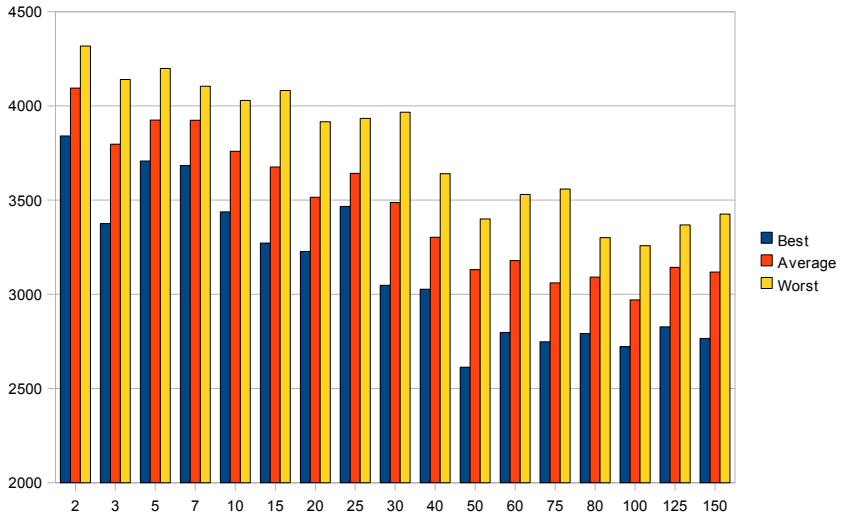

Abbildung 5.1: Einfluss der Populationensgröße (Laufzeit beschränkt)

5.2 Unterpopulationen

Wir testen für die Gesamtpopulationsgrößen 50, 100 und 200 jeweils mit den Subpopulationsgrößen 1 bis 5, 7, 10, 15, 20 und 25. Wie vorher fahren wir zwei getrennte Testserien. Tabelle 5.4 enthält die Ergebnisse mit einer Generationenbeschränkung von 100, Tabelle 5.5 die Ergebnisse mit einer Laufzeitbeschränkung von 30 Sekunden.

PopSize	Subpop	Total	Best	Average	Worst	Average Time
50	1	50	2640	3057	3496	6,0
25	2	50	2624	2806	3070	5,1
16	3	50	2626	2895	3178	4,9
12	4	50	3656	4066	4378	5,8
10	5	50	3776	4088	4385	5,0
7	7	50	3383	4001	4258	4,9
5	10	50	3779	4109	4402	5,1
3	15	50	3810	4161	4538	4,4
2	20	50	4138	4417	4635	3,6
2	25	50	4048	4358	4596	4,3
100	1	100	2678	2962	3200	12,7
50	2	100	2274	2668	2928	10,9
33	3	100	2478	2571	2825	10,2
25	4	100	3243	3979	4244	11,4
20	5	100	3322	3762	4081	10,4
14	7	100	3337	3662	4123	9,1
10	10	100	3401	3768	4148	9,2
6	15	100	3857	4061	4239	7,4
5	20	100	3719	4014	4260	8,8
4	25	100	3787	4034	4302	6,7
200	1	200	2443	2785	3055	26,1
100	2	200	2288	2523	2762	22,6
66	3	200	2244	2413	2770	22,0
50	4	200	3393	3711	4028	23,4
40	5	200	3199	3627	3944	21,4
28	7	200	3000	3522	3829	19,7
20	10	200	2992	3283	3564	20,4
13	15	200	3314	3666	3903	19,9
10	20	200	3486	3677	3803	17,4
8	25	200	3389	3651	3932	15,7

Tabelle 5.4: Tests mit verschiedenen Anzahlen an Subpopulationen mit Generationenbeschränkung

PopSize	Subpop	Total	Best	Average	Worst	Average Time	Average Generations
50	1	50	2253	2515	2821	30,1	613
25	2	50	2099	2264	2441	30,1	768
16	3	50	2058	2250	2584	30,1	755
12	4	50	3419	3702	4036	30,1	561
10	5	50	3413	3813	4163	30,1	664
7	7	50	3421	3775	3972	30,1	660
5	10	50	3376	3583	4315	30,1	644
3	15	50	3416	3664	4046	30,1	743
2	20	50	3913	4142	4275	30,1	882
2	25	50	3574	4082	4300	30,1	725
100	1	100	2315	2557	2803	30,1	262
50	2	100	2081	2325	2532	30,1	305
33	3	100	2028	2249	2436	30,1	337
25	4	100	3241	3760	4187	30,1	269
20	5	100	3363	3867	4143	30,1	312
14	7	100	3364	3715	4077	30,1	345
10	10	100	3245	3517	3743	30,1	340
6	15	100	3470	3705	3993	30,1	413
5	20	100	2804	3479	3836	30,1	345
4	25	100	3071	3444	4058	30,1	457
200	1	200	2297	2576	2898	30,2	118
100	2	200	2211	2567	2811	30,2	133
66	3	200	2071	2270	2495	30,2	146
50	4	200	3118	3532	3930	30,1	128
40	5	200	3035	3371	3614	30,1	145
28	7	200	2791	3330	3875	30,2	160
20	10	200	3213	3529	3825	30,1	151
13	15	200	3134	3437	3945	30,1	150
10	20	200	3092	3372	3689	30,1	179
8	25	200	3270	3569	3826	30,1	197

Tabelle 5.5: Tests mit verschiedenen Anzahlen an Subpopulationen mit Zeitbeschränkung

Abbildung 5.2 stellt die Ergebnisse der laufzeitbeschränkten Testserie grafisch dar. Hier ist eindeutig zu erkennen, dass drei Subpopulationen unabhängig von der Populationsgröße das beste Ergebnis liefern. Dies gilt ebenfalls für die Testserie mit Generationenbeschränkung.

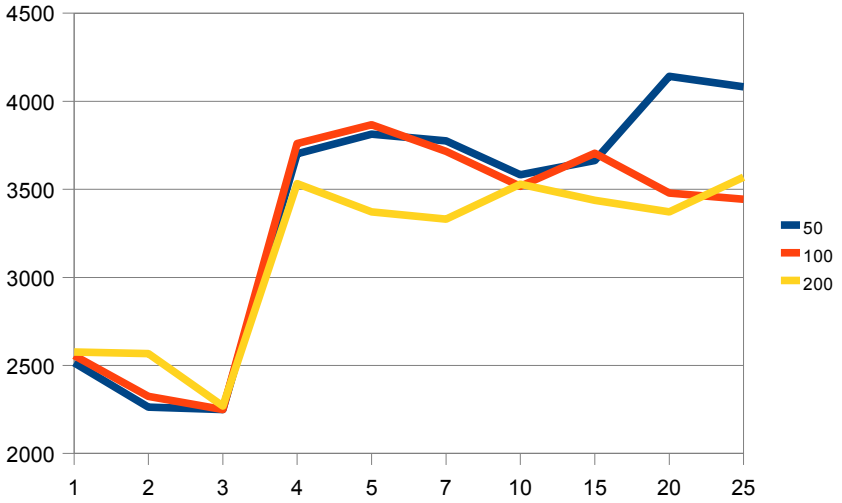

Abbildung 5.2: Einfluss der Subpopulationen bei einer Gesamtpopulation von 50, 100 oder 200 (Laufzeit beschränkt)

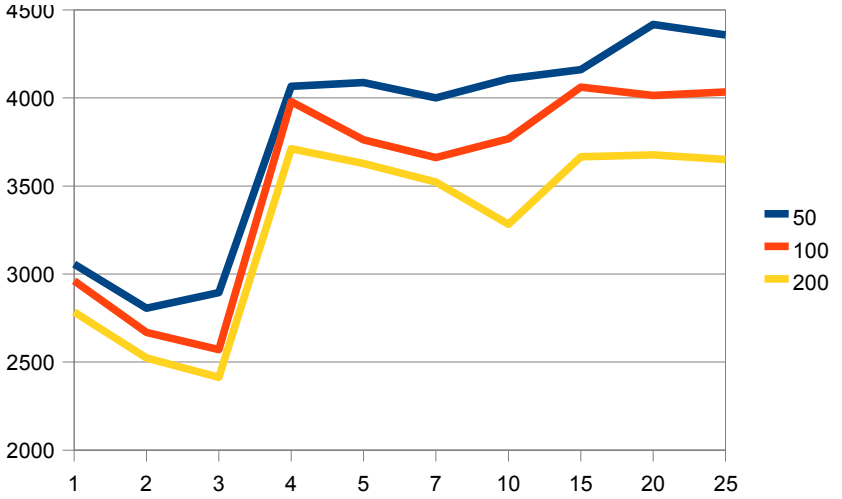

Abbildung 5.3: Einfluss der Subpopulationen bei einer Gesamtpopulaton von 50, 100 oder 200 (Generationen beschränkt)

5.3 Migration

Migration ist ein Mechanismus zum Austausch von Elemente zwischen Subpopulationen. Eine Migration erfolgt, indem aus jeder Subpopulation ein Teil der Individuen entfernt und in einen *Migrationspool* zwischengespeichert wird. Aus diesem werden die Elemente dann wieder auf die verschiedenen Subpopulationen verteilt.

Der Parameter `Migration.Interval` legt das *Migrationsintervall* fest. Dieses bestimmt, in welchem zeitlichen Abstand (bzw. nach wie vielen Generationen) Migrationen durchgeführt werden.[Pohlheim, 2006]

Der Parameter `Migration.Rate` legt die *Migrationsrate* fest. Diese bestimmt den prozentualen Anteil der Elemente einer Subpopulation, der bei einer Migration ausgetauscht wird. [Pohlheim, 2006]

Der Parameter `Migration.Topology` legt die *Migrationstopologie* fest. Diese bestimmt, in welche Subpopulationen die in den Migrationspool emigrierten Elemente wieder immigrieren dürfen. GEATbx bietet drei grundlegende Möglichkeiten. [Pohlheim, 2006]

Complete Net Topology Es gibt keine Einschränkungen, die Elemente können in beliebige Subpopulationen immigrieren. Dies entspricht bei GEATbx dem Parameterwert 0.

1-D Neighbourhood Topology Die Subpopulationen werden in einer eindimensionalen Nachbarschaft angeordnet, also einer Linie. Elemente dürfen nur in Subpopulationen immigrieren, die mit ihrer ursprünglichen Subpopulation benachbart sind. Für innere Subpopulationen gibt es also je zwei potentielle Zielsubpopulationen, für die beiden Randsubpopulationen jeweils nur eine. Diese Topologie entspricht bei GEATbx dem Parameterwert 1. GEATbx scheint keine zweidimensionale Nachbarschaftstopoplogie, bei der jede Subpopulation vier Nachbarn hätte, anzubieten. [Pohlheim, 2006]

1-D Ring Topology Die Ringtopologie entspricht der eindimensionalen Nachbarschaftstopologie, nur dass die Linie sich jetzt an den Enden berührt einen Kreis bildet. Die erste Subpopulation ist also Nachbar der letzten Subpopulation und umgekehrt. Damit gibt es für jede Subpopulation zwei potentielle Migrationsziele. Diese Topologie entspricht bei GEATbx dem Parameterwert 2.

Wir haben auch eine Testserie für die verschiedenen Migrationsparameter durchgeführt. Tabelle 5.6 enthält die verwendeten festen Parameterwerte, Tabelle 5.7 die Ergebnisse.

Wir variieren jeweils nacheinander die Migrationstopologie (0, 1, 2), die Migrationsrate (0.01, 0.05 ,0.1 ,0.15 ,0.2 ,0.5) und das Migrationsintervall (2, 5, 10, 15, 20, 25, 35, 50, 75). Da die Migrationseinstellungen sich nur sehr begrenzt auf die Ausführungsdauer auswirken, wird hier auf den zeitbegrenzten Test verzichtet.

Bei der Migrationtopologie scheint der Wert 2 (1-D Ring Topology) am besten zu sein. Bei Migrationsrate ist es nicht so klar, aber der Wert 0,15 scheint recht gut. Beim

Parameter	Wert
Filename	'bays29'
TestNumber	10
PopSize = 100	
Subpop	3
Generations	100
Selection	'selsus'
Recombination	'recgp'
Swap	1
Move	1
Invert	1

Tabelle 5.6: Verwendete feste Parameter beim Test der Migrationsparameter

Migrationtopology	Migrationrate	Migrationintervall	Best	Average	Worst	Average Time
0	0,10	20	2258	2429	2722	35,2
1	0,10	20	2169	2406	2792	35,1
2	0,10	20	2182	2308	2460	35,0
0	0,01	20	2196	2471	2957	34,6
0	0,05	20	2205	2372	2833	34,9
0	0,10	20	2264	2382	2485	34,0
0	0,15	20	2099	2388	2577	35,2
0	0,20	20	2161	2347	2593	34,6
0	0,50	20	2099	2343	2592	34,6
0	0,10	2	2073	2312	2704	29,7
0	0,10	5	2221	2359	2530	32,9
0	0,10	10	2250	2392	2741	35,3
0	0,10	15	2115	2332	2571	35,0
0	0,10	20	2232	2404	2576	35,0
0	0,10	25	2075	2319	2747	35,6
0	0,10	35	2076	2391	2538	34,6
0	0,10	50	2142	2466	2789	35,3
0	0,10	75	2162	2402	2690	34,4

Tabelle 5.7: Tests mit verschiedenen Migrationsoptionen

Migrationintervall gibt es ebenfalls keinen klaren Favoriten, wir verwenden in Zukunft den Wert 35.

5.4 Erneute Bestimmung der Populationsgröße

Parameter	Wert
Filename	'bays29'
TestNumber	10
Subpop	3
Generations	100
Minutes = 0.5	
Migrationtopology	2
Migrationrate	0.15
Migrationintervall	35
Selection	'selsus'
Recombination	'recgp'
Swap	1
Move	1
Invert	1

Tabelle 5.8: Verwendete feste Parameter beim zweiten Test der Populationsgrößen

Nachdem wir nun die anderen Parameter optimiert haben, führen wir noch einmal den Test auf die Populationsgröße durch. Wir variieren die Populationsgröße dabei um den alten Wert 100 herum, von 80 bis 120 mit Schrittweite 5.

PopSize	Best	Average	Worst	Average Time
80	2192	2408	2578	27,5
90	2305	2384	2543	30,8
95	2214	2407	2755	32,7
100	2260	2399	2685	35,2
105	2184	2452	2708	36,1
110	2184	2412	2691	38,6
120	2091	2270	2453	44,8

Tabelle 5.9: Tests mit verschiedenen Populationsgrößen mit Generationenbeschränkung

PopSize	Best	Average	Worst	Average Time	Average Generations
80	2255	2463	2687	30,2	105
90	2238	2448	2672	30,2	93
95	2175	2343	2464	30,2	83
100	2196	2472	2720	30,3	77
105	2316	2461	2679	30,2	81
110	2378	2539	2797	30,4	73
120	2402	2500	2592	30,3	65

Tabelle 5.10: Tests mit verschiedenen Populationsgrößen mit Zeitbeschränkung

Die Tabellen 5.9 und 5.10 enthalten wie immer jeweils die Ergebnisse mit Generationenbeschränkung und mit Laufzeitbeschränkung. Auf Grund der Werte in Tabelle 5.10 arbeiten wir in zukünftigen Testserien mit der Populationsgröße 95 statt 100 weiter.

6 Selektion

> Ersetzen Sie das Default-Verfahren `selsus` (Welches ist das?) durch die klassische *Roulette Wheel Selection*. Hat dies sichtbare Auswirkungen auf die Leistung Ihres Algorithmus? Oder ist vielleicht die *Tournament Selection* (`seltour`) geeigneter? [Erben, 2007]

Beim voreingestellten Selektionsverfahren `selsus` handelt es sich um das *Stochastic Universal Sampling*. Die drei Selektionsverfahren, mit denen wir uns beschäftigen werden, sind die folgenden:

selrws Roulette Wheel Selection

selsus Stochastic Universal Sampling

seltour Tournament Selection

Diese Verfahren werden alle in den Vorlesungsunterlagen beschrieben. [Erben, 2008b]

Parameter	Wert
Filename	'bays29'
TestNumber	10
PopSize = 95	
Subpop	3
Generations	100
Minutes = 0.5	
Migrationtopology	2
Migrationrate	0.15
Migrationintervall	35
Recombination	'recgp'
Swap	1
Move	1
Invert	1

Tabelle 6.1: Verwendete feste Parameter beim Test der Selektionsverfahren

Wir testen alle drei Selektionsverfahren. Die verwendeten Parameterwerte sind in Tabelle 6.1. Da die Selektioneinstellung sich nur sehr begrenzt auf die Ausführungsdauer auswirkt, wird hier auf den zeitbegrenzten Test verzichtet.

Selektion	Best	Average	Worst	Average Time
selsus	2063	2386	2537	34,43
selrws	2175	2387	2558	34,07
seltour	2107	2306	2434	30,2

Tabelle 6.2: Tests mit verschiedenen Selektioneinstellungen

Wie aus Tabelle 6.2 ersichtlich ist, besitzt `selsus` zwar den besseren Bestwert, `seltour` ist allerdings stabiler. Es wird mit `seltour` weitergetestet. In unserem Abschlusstest werden wir `selsus` aber noch einmal berücksichtigen.

7 Rekombination

> Ersetzen Sie den in `demotsplib` verwendeten Rekombinationsoperator `recgp` (*generalized position crossover*) durch `recpm` und vergleichen Sie die Resultate in einigen Testläufen. Welches Verfahren steckt hinter `recpm`? [Erben, 2007]

Hinter `recpm` verbirgt das aus der Vorlesung bekannte *Partially Matched Crossover* (PMX). Statt sofort einen Vergleich zwischen diesem und dem voreingestellten *Generalized Position Crossover* aufzustellen, implementieren wir zunächst die Verfahren *Cycle Crossover* (Aufgabenstellung 2007) und *Order Crossover* (Aufgabenstellung 2008) und vergleichen dann alle vier Verfahren miteinander.

7.1 reccycle

> Den Cycle Crossover (reccycle) gibt es leider nicht im Angebot der Toolbox. Implementieren Sie diesen daher selbst und führen Sie wiederum einige vergleichende Testläufe durch. [Erben, 2007]

Cycle Crossover wurde in der Vorlesung behandelt und wird deshalb hier nicht ausführlich beschrieben. Das Grundprinzip ist, dass durch die Position der Gene in den Eltern eine 1-zu-1-Abbildung definiert wird. Hat der erste Elternteil an der ersten Stelle eine 1 und der zweite eine 4, so wird 1 auf 4 abgebildet. Diese Abbildung, für alle Gene fortgeführt, bildet Zyklen. Die Rekombination wird so durchgeführt, dass bei den Kindern alle Gene, die zu einem Zyklus gehören, ausgetauscht werden. Dies ist in Abbildung 7.1 dargestellt.

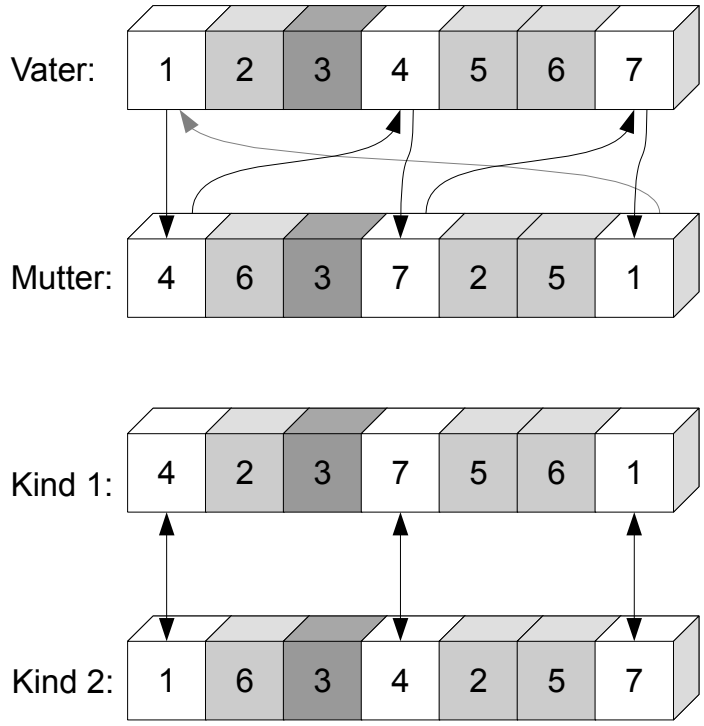

Abbildung 7.1: Das Rekombinationsverfahren *Cycle Crossover*

In Matlab haben wir diesen Algorithmus wie folgt implementiert:

```
function NewChrom = reccycle(OldChrom, RecRate)
  NewChrom = OldChrom;

    % Länge einer Zeile berechnen
    length = size(OldChrom,2);
    % Anzahl der zu vertauschenden Zeilen
    lines = (floor(size(OldChrom,2)/2)-1);

    for line = 1:lines
        if rand(1,1) > RecRate, continue, end;

        line1 = order(line*2-1);
        line2 = order(line*2);
        % Zwei Zeilen in chrom1 und chrom2 kopieren
        % (steigert die Effizienz)
        chrom1 = OldChrom(line1,:);
        chrom2 = OldChrom(line2,:);

```

```
19          % Wenn beide Zeilen gleich springe zur Nächsten
20          if chrom1 == chrom2, continue, end;
21
22          % Zufällige Startposition wählen, welche nicht bei beiden
                Zeilen
23          % den gleichen Wert enthält
24          spos = ceil(rand(1)*length);
25          while chrom1(spos) == chrom2(spos)
26              spos = ceil(rand(1)*length);
27          end
28          pos = spos;
29
30          % So lange das Ende des Zyklus nicht gefunden ist ...
31          while 1
32
33              % Vertausche die Werte an der aktuellen Position in
                    NewChrom
34              tmp = NewChrom(line1,pos);
35              NewChrom(line1,pos) = NewChrom(line2,pos);
36              NewChrom(line2,pos) = tmp;
37
38              % finde die nächste Position im Zyklus
39              pos = find(chrom1 == chrom2(pos));
40
41              % Breche ab wenn die aktuelle Position gleich der
42              % Anfangsposition ist
43              if pos == spos ,break,end;
44          end
45      end
```

7.2 Order Crossover

Neben dem in der Vorlesung erwähnten *Partially matched Crossover* (`recpm`) gibt es für das TSP unter anderem auch das so genannte *Order Crossover* (OX) von L. Davis. Leider gehört dieses Rekombinationsverfahren nicht zum Angebot der Toolbox. Implementieren Sie es daher selbst (Dateiname `recoxNAME.m`), nachdem Sie sich kundig gemacht haben, welcher Algorithmus dahinter steckt. Führen Sie wiederum einige vergleichende Testläufe durch. [Erben, 2008a]

Das Order-Crossover-Verfahren ist von der Grundidee wesentlich einfacher als das Cycle-Crossover-Verfahren. Hier wird einfach ein zufällig gewählter Bereich der Eltern direkt in die Kinder übernommen. Die danach noch nicht in einem Kind vorhandenen Gene werden aus dem jeweils anderen Elternteil in der Reihenfolge übernommen, in der sie dort auftreten. Die wird in Abbildung 7.2 dargestellt. Der genaue Verfahrensablauf zur Erzeugung eines Kindes ist folgender [Chern, 2005]:

- Auswahl eines zufälligen Unterbereichs eines Elternteils
- Kopieren dieses an die selbe Stelle im unfertigen Kind
- Für alle Gene im zweiten Elternteil (von vorne)
  - Wenn Gen bereits im Kind, überspringe
  - Wenn Gen nicht im Kind, kopiere an die erste freie Stelle im Kind

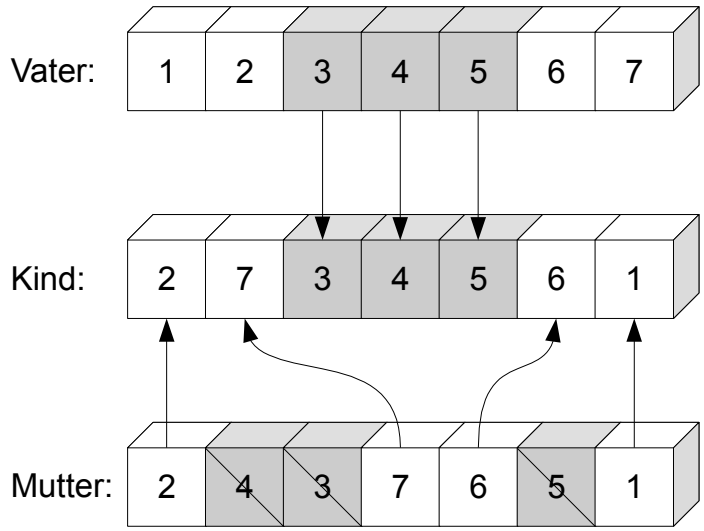

Abbildung 7.2: Das Order-Crossover-Verfahren

In Matlab haben wir diesen Algorithmus wie folgt implementiert:

```
1 function NewChrom = recox(OldChrom, RecRate)
2   NewChrom = OldChrom;
3
4     % Länge einer Zeile berechnen
5     length = size(OldChrom,2);
6     % Anzahl der zu vertauschenden Zeilen
7     lines = (floor(size(OldChrom,1)/2));
```

```
8
9     for line = 1:lines
10        if rand(1,1) > RecRate, continue, end;
11
12        line1 = line*2-1;
13        line2 = line*2;
14        % Zwei Zeilen in chrom1 und chrom2 kopieren
15        % (steigert die Effizienz)
16        chrom1 = OldChrom(line1,:);
17          chrom2 = OldChrom(line2,:);
18
19        % Wenn beide Zeilen gleich springe zur Nächsten
20        if chrom1 == chrom2, continue, end;
21
22        % Zufällige Start- und Endposition wählen
23        spos = ceil(rand(1)*length);
24        maxlength = length - spos;
25        endpos = ceil(spos + rand(1)*maxlength);
26
27        % Speichere die zu vertauschenden Werte in t1 und t2 und
              schreibe
28        % sie an die neue Position in NewChrom
29        t1 = chrom1(spos:endpos);
30        t2 = chrom2(spos:endpos);
31        NewChrom(line2,spos:endpos) = t1;
32        NewChrom(line1,spos:endpos) = t2;
33
34        % Bilde die Differenz
35        [dif1,dif1index] = setdiff(chrom1,t2);
36        [dif2,dif2index] = setdiff(chrom2,t1);
37
38        % Finde die ursprüngliche Reihenfolge der Differenz
39        % (leider Vertauscht Matlab die Reihenfolge bei setdiff
40        [tmp,dif1order] = sort(dif1index);
41        [tmp,dif2order] = sort(dif2index);
42
43        % Ersetze die Restlichen Werte in NewChrom durch die anderen
44        % restlichen Werte
45        ipos = 1;
46        for i = [1:(spos-1),(endpos+1):length]
47            NewChrom(line2,i) = dif2(dif2order(ipos));
48            NewChrom(line1,i) = dif1(dif1order(ipos));
```

```
49              ipos = ipos + 1;
50          end
51      end
```

7.3 Vergleich der Rekombinationsverfahren

Beide unserer selbst implementierten Rekombinationsverfahren sind signifikant besser als die bereits vorhandenen. Tabelle 7.2 zeigt, dass `reccycle` in Sachen Geschwindigkeit die anderen Verfahren um Längen schlägt. Bei vorgegebener Generationenanzahl ist `reccycle` am schnellster, aber `recox` liefert die besseren Werte. Beschränken wir die Laufzeit, so liefert `reccycle` bessere Werte, da es in der selben Zeit etwa dreimal so viele Generationen schafft wie recox. Dies liegt darin begründet, dass `recox` sehr häufig die Matlab-Funktion `setdiff` aufruft, welche wiederum `ismember` und `unique` aufruft. Bei diesen Aufrufen müssen viele Vergleiche durchgeführt werden, was das Verfahren ausbremst.

Parameter	Beschreibung
Filename	'bays29'
TestNumber	10
PopSize = 95	
Subpop	3
Generations	100
Minutes = 0.5	
Migrationtopology	2
Migrationrate	0.15
Migrationintervall	35
Selection	'seltour'
Swap	1
Move	1
Invert	1

Tabelle 7.1: Verwendete feste Parameter beim Test der Rekombinationsverfahren

Recombination	Best	Average	Worst	Average Time	Average Generations
recgp	2300	2438	2674	33,58	100
recpm	2254	2544	2855	8,5	100
reccycle	2113	2385	2654	3,21	100
recox	2046	2185	2348	8,01	100
recgp	2097	2427	2634	30,17	89
recpm	2028	2138	2351	30,1	387,5
reccycle	2026	2061	2125	30,08	1078,6
recox	2020	2089	2175	30,1	377,2

Tabelle 7.2: Tests mit verschiedenen Rekombinationsverfahren

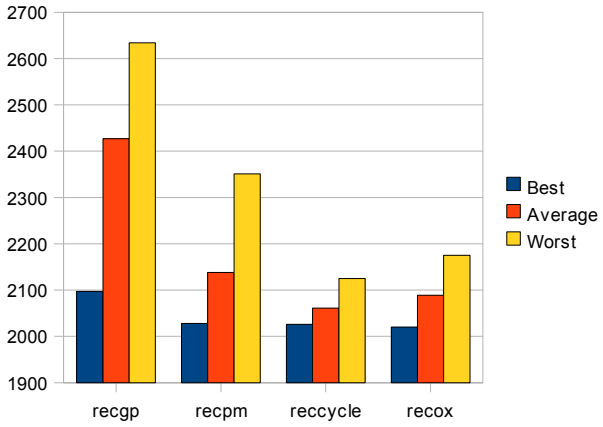

Abbildung 7.3: Test der Rekombinationsverfahren (Laufzeit beschränkt)

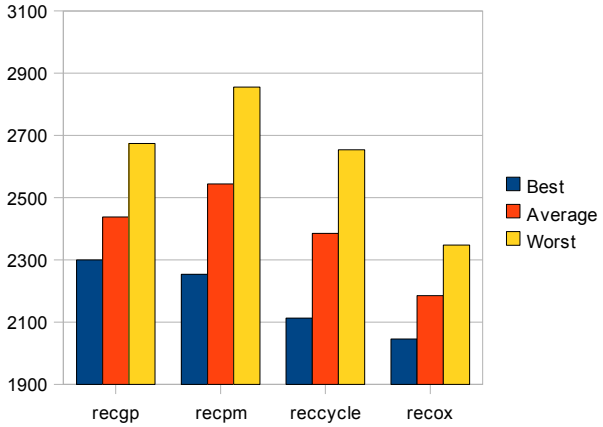

Abbildung 7.4: Test der Rekombinationsverfahren (Generationen beschränkt)

Rekombinationsverfahren sind einer der zeitintensivsten Bestandteile von genetischen Algorithmen, deshalb haben wir noch eine auf Zeitmessungen spezialisierte Testserie durchgeführt. Hierzu wurden um die vier Rekombinationsfunktionen jeweils eine Wrapperklasse geschrieben, welche die Anzahl der Ausführungen, sowie die Zeitverbrauch der Funktionen misst. Hier als Beispiel die Wrapperfunktion für `reccylce`:

```
1 function NewChrom = reccyclecount(OldChrom, RecRate)
2     global recombinationcounter recombinationtime;
3     recombinationcounter = recombinationcounter + 1;
4     tic;
5     NewChrom = reccycle(OldChrom, RecRate);
6     recombinationtime = recombinationtime + toc;
```

Verfahren	Aufrufe	Gesamtzeit	Aufrufzeit	GenAlgzeit	Anteil
recgp	297	30,40	0,1023576	33,58	90,52%
recpm	297	6,26	0,0210923	8,5	73,68%
reccycle	297	0,85	0,0028687	3,21	26,53%
recox	297	6,22	0,0209407	8,01	77,64%

Tabelle 7.3: Weitere Tests mit verschiedenen Rekombinationsverfahren

Die Spalten in Tabelle 7.3 beinhalten folgende Werte: Spalte 2 enthält die Anzahl der Aufrufe der Rekombinationsverfahren, Spalte 3 die gesamte Laufzeit, die auf das Rekombinationsverfahren abfällt. Die vierte Spalte zeigt die Laufzeit pro Aufruf, also Spalte 3 dividiert durch Spalte 2. Die fünfte Spalte beinhaltet die Gesamtlaufzeit des genetischen Algorithmus'. Die letzte Spalte enthält den aus Spalte 3 und 5 errechneten prozentualen Anteil des Rekombinationsverfahrens an der Gesamtlaufzeit.

Aus den Ergebnissen lässt sich deutlich erkennen, dass die Rekombinationsfunktion recgp mit 90% einen sehr entscheidenden Einfluss auf die Gesamtausführungszeit hat. Pro Ausführung wird eine zehntel Sekunde benötigt. `reccycle` hingegen benötigt etwa 2,8 Millisekunden. Dies ist ein enormer Geschwindigkeitsvorteil. Auch im Gegensatz zu `recpm` und `recox` ist die Funktion etwa sieben mal so schnell.

Auf Grund der klaren Überlegenheit von `reccycle` führen wir alle zukünftigen Testserien nur noch mit diesem Rekombinationsverfahren aus.

8 Mutation

> Beschreiben Sie mit Hilfe einiger Beispiel die Verfahren `mutswap`, `mutmove` und `mutinvert`. Warum ist es wohl sinnvoll, alle drei Mutationsoperatoren einzusetzen? Versuchen Sie, ein paar Testläufe mit unterschiedlichen Mutationsraten durchzuführen. [Erben, 2007]

Mutation ist die Veränderung von Individuen durch Veränderung einiger zufällig gewählter Gene. Die von GEATbx verwendeten Verfahren *Swap*, *Invert* und *Move* haben alle gemeinsam, dass Sie keine ungültigen Chromosome erzeugen, da sie nur die Positionen der Gene innerhalb des Chromosoms beeinflussen. Zur Veranschaulichung sind alle Mutationsverfahren in Abbildung fig:mutations dargestellt.

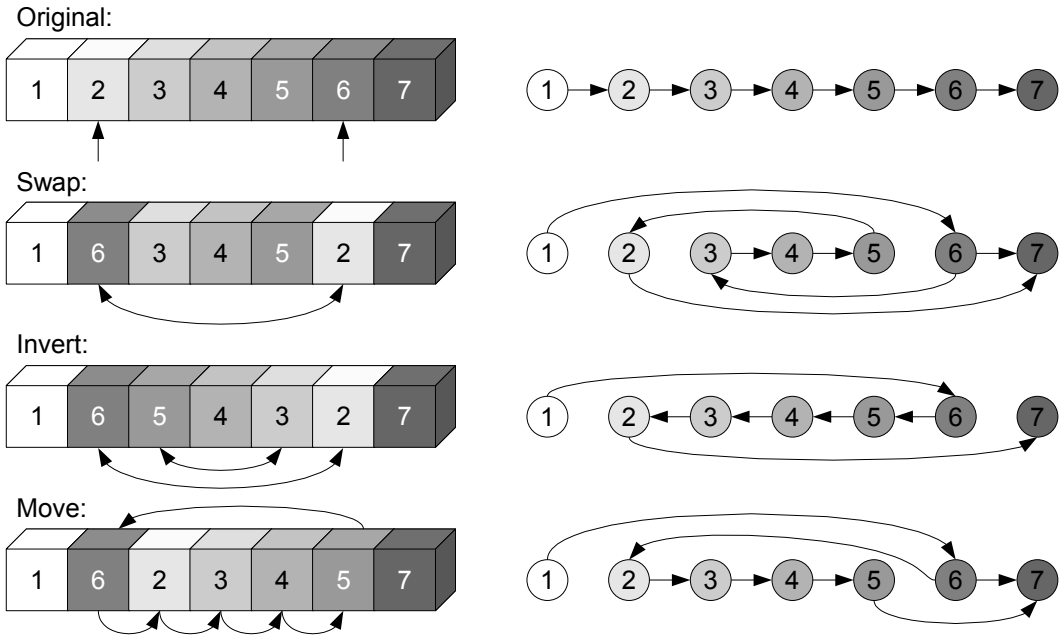

Abbildung 8.1: Vergleich der verschiedenen Mutationsverfahren

Es folgt eine kurze Beschreibung der einzelnen Verfahren.

Swap Bei dieser Mutationsart werden zwei Gene zufällig ausgewählt und einfach getauscht. Die Reihenfolge der anderen Gene wird nicht beeinflusst.

Invert Diese Mutationsart funktioniert prinzipiell wie Swap, nur werden nicht nur die ausgewählten Gene getauscht, sondern die Reihenfolge aller Gene, die dazwischen liegen, wird umgedreht.

Move Bei Move wird nur ein einzelnes Gen zufällig ausgewählt. Dann wird eine zufällige Position verschoben. Die Reihenfolge der anderen Gene bleibt unverändert, sie müssen lediglich „aufrücken“.

Die Frage, warum es sinnvoll ist, alle drei Mutationsverfahren einzusetzen, lässt sich damit beantworten, dass damit eine größere Variation der Chromosomen gesichert ist. Würde mehrfach hintereinander nur Swap eingesetzt, könnte es schnell passieren, dass vorherige Swaps wieder rückgängig gemacht werden und so schon verworfene Chromosomen wiederhergestellt werden. Außerdem kann je es mit verschiedenen Verfahren unterschiedlich lange dauern, bestimmte Probleme zu lösen. Gehen wir zum Beispiel davon aus, dass der optimale Pfad des in Abbildung 8.2 dargestellten Chromosoms einfach durchgehend von 1 bis 21 verläuft, so lässt sich dies am schnellst durch folgende Mutationen erreichen: *Move* Gen 7 an die 7. Position, *Invert* 9 bis 13 und *Swap* 16 und 20. Würde man nur das Swap-Verfahren verwenden, bräuchte man erheblich länger als drei Schritte.

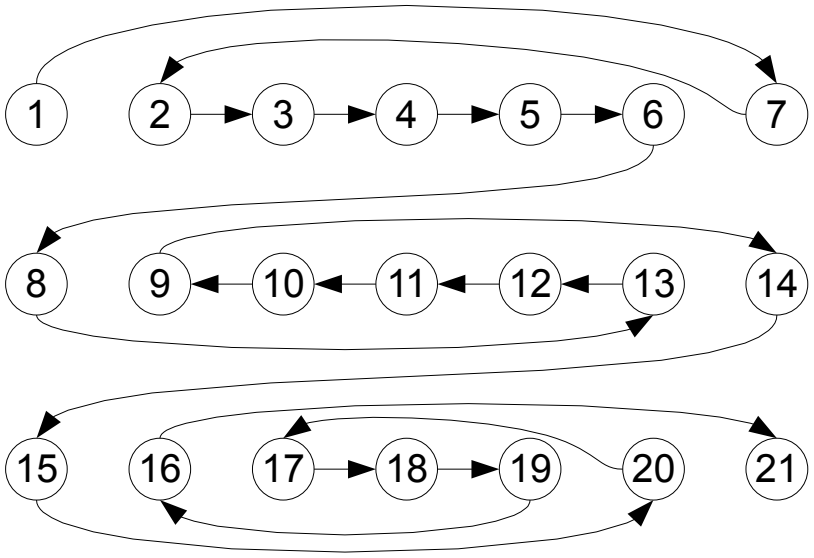

Abbildung 8.2: Durch mehrere Mutationsverfahren lösbares Problem

Nun führen wir noch Tests mit verschiedenen Einstellungen der Mutationsraten für Swap, Invert und Move durch. Dabei ist zu beachten, dass wir zwei getrennte Testserien ausführen, einmal mit ganzen Zahlen größer gleich 1 und einmal mit Dezimalzahlen zwischen 0 und 1. Dies liegt schlichtweg daran, dass GEATbx daran verschluckt, wenn man diese Parameterklassen mischt. Die für den Test verwendeten Parameter können Tabelle 8.1 entnommen werden.

Parameter	Wert
Filename	'bays29'
TestNumber	10
PopSize = 95	
Subpop	3
Generations	100
Minutes = 0.5	
Migrationtopology	2
Migrationrate	0.15
Migrationintervall	35
Selection	'seltour'
Recombination	'reccycle'

Tabelle 8.1: Verwendete feste Parameter beim Test der Mutationsverfahren

Bei genauer Betrachtung der Ergebnisse in Tabelle 8.2 kann man erkennen, dass der Move-Wert am wenigsten Auswirkungen hat, aber er scheint bei 1 den Besten Wert zu haben. Der Invert-Wert ist bei 1 und 2 recht gut. Danach stark abfallend. Es wird weiterhin 1 verwendet. Auch wenn der beste Wert bei 3 3 1 liegt scheint 1 dennoch der beste Wert für Swap zu sein. Der beste Wert 3 3 1 wird zum Schluss noch einmal getestet.

Swap	Move	Invert	Best	Average	Worst	Average Time
1	1	1	2155	2378	2788	2,29
1	1	2	2230	2347	2504	2,39
1	2	1	2183	2392	2505	2,38
1	3	2	2237	2385	2624	2,56
1	5	1	2160	2400	2632	2,7
1	5	2	2202	2371	2648	2,71
1	10	1	2223	2390	2546	2,84
3	3	1	2061	2311	2597	2,5
5	1	1	2249	2360	2510	2,5
5	2	1	2106	2400	2720	2,56

Tabelle 8.2: Tests mit verschiedenen Mutationsraten

Bei den Werten zwischen 0 und 1 (Tabelle 8.3) ist eindeutige eine Verbesserung zu Erkennen um so höher die Werte steigen. Es wird weiterhin 1 1 1 verwendet.Die Kombination 0.5 0.5 1 wird im Abschlusstest noch einmal berücksichtigt.

Swap	Move	Invert	Best	Average	Worst	Average Time
0,5	0,5	0,5	2107	2357	2505	2,04
0,5	0,5	1	2033	2272	2517	2,11
0,5	0,75	1	2113	2343	2630	2,21
0,75	0,5	0,5	2163	2313	2527	2,1
0,75	0,5	0,75	2223	2324	2480	2,16
0,75	0,5	1	2162	2304	2397	2,25
0,75	0,75	0,75	2150	2266	2439	2,23
1	0,75	0,75	2123	2314	2604	2,28
1	1	0,75	2081	2320	2730	2,31
1	1	1	2101	2297	2519	2,32

Tabelle 8.3: Tests mit verschiedenen Mutationsraten

9 Optimale Einstellungen

Geben Sie die Kombination von Operatoren und Parameterwerten an, die Ihnen am besten erscheint. Führen Sie mit Ihrem so konfigurierten Algorithmus auch ein paar Testläufe mit den TSP-Beispielen `bayg29.tsp` und `berlin52.tsp` durch und geben Sie die Resultate an. [Erben, 2007]

9.1 Letzte Untersuchung der Populationsgröße

Da sich nun viele Werte verändert haben, überprüfen wir ein letztes Mal den Einfluss der Populatonsgröße. Dazu lassen wir in mehreren Testserien die Populationsgröße mit Schrittweite 5 von 20 bis 150 variieren. Bei jeder Testserie legen wir eine andere Beschränkung fest. Einmal wird die Anzahl der Generationen auf 100 (Tabelle 9.1) beschränkt,die restlichen Testserien beschränken die Laufzeit auf 30 Sekunden (Tabelle 9.2) und – nachdem das Verfahren jetzt erheblich schneller läuft – auch auf 10 Sekunden (Tabelle 9.3) sowie 2 Sekunden (Tabelle 9.4). Aus Platzgründen listen wir nur die zehn jeweils besten Durchschnittswerte. Die vollständigen Daten liegen im CSV-Format auf der beiliegenden CD.

PopSize	Best	Average	Worst	Average Time	Average Generations
85	2140	2254	2396	2,61	100
110	2111	2240	2433	3,30	100
135	2083	2200	2386	3,89	100
145	2107	2236	2375	4,14	100
160	2067	2225	2481	4,49	100
170	2076	2245	2446	4,71	100
175	2065	2199	2339	4,88	100
185	2103	2224	2296	5,15	100
190	2038	2242	2505	5,25	100
200	2034	2236	2501	5,49	100

Tabelle 9.1: Tests mit verschiedenen Populationsgrößen mit Generationenbeschränkung

Abbildung 9.1 zeigt die vier Testserien im Vergleich. Bei der auf 100 Generationen begrenzten Tests sieht man, dass das Ergebnis um so besser wird um so größer die Popu-

PopSize	Best	Average	Worst	Average Time	Average Generations
115	2020	2082	2259	30,07	977,00
140	2020	2062	2171	30,09	828,80
145	2020	2076	2153	30,09	817,00
160	2020	2077	2202	30,09	735,40
170	2026	2055	2101	30,09	702,30
175	2020	2076	2228	30,08	685,80
180	2026	2054	2118	30,09	664,20
185	2026	2082	2239	30,09	655,30
195	2020	2057	2152	30,10	622,40
200	2020	2075	2239	30,10	604,20

Tabelle 9.2: Tests mit verschiedenen Populationsgrößen mit 30 Sekunden Laufzeitbeschränkung

PopSize	Best	Average	Worst	Average Time	Average Generations
20	2028	2110	2303	10,07	1061,90
55	2020	2095	2217	10,07	582,50
100	2020	2078	2159	10,08	367,30
110	2026	2104	2315	10,08	336,40
125	2020	2100	2274	10,08	304,40
150	2033	2111	2290	10,08	257,60
180	2020	2087	2186	10,09	216,90
190	2026	2096	2172	10,09	207,30
195	2028	2094	2202	10,09	200,80
200	2026	2100	2177	10,10	193,80

Tabelle 9.3: Tests mit verschiedenen Populationsgrößen mit 10 Sekunden Laufzeitbeschränkung

lation ist. Dies fällt am Anfang deutlicher aus als am Schluss. Ab 130 recht konstant. Bei 30 Sekunden Laufzeit ist das Ergebnis ähnlich und bei 10 Sekunden bleibt das Ergebnis recht gleich. Für die auf 2 Sekunden beschränkte Testserie wird das Ergebnis mit wachsender Populationsgröße immer schlechter. Insgesamt entscheiden wir uns für 175 als optimale Populationsgröße.

Anscheinend verbessert sich das Ergebnis nur innerhalb der ersten paar hundert Generationen stark. Danach setzt sich das Ergebnis in einem lokalen Maximum fest. Diese

PopSize	Best	Average	Worst	Average Time	Average Generations
20	2156	2393	2644	2,07	216,90
25	2153	2339	2629	2,07	197,10
30	2104	2286	2552	2,07	171,00
35	2209	2369	2499	2,07	157,30
45	2104	2321	2500	2,07	133,10
55	2270	2414	2547	2,07	111,90
60	2142	2379	2785	2,07	104,00
65	2122	2353	2605	2,07	100,10
90	2182	2373	2543	2,07	75,90
105	2212	2410	2650	2,07	65,00

Tabelle 9.4: Tests mit verschiedenen Populationsgrößen mit 2 Sekunden Laufzeitbeschränkung

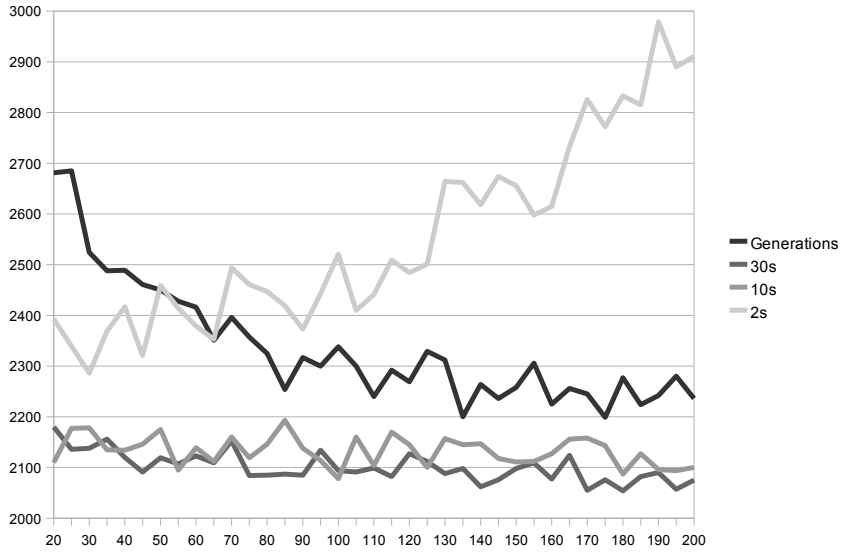

Abbildung 9.1: Einfluss der Populationensgröße auf die Mittelwerte(verschiedene Beschränkungen)

Generationen sollten auf jeden Fall ausgenutzt werden. Dies ist der Grund für die zunehmend schlechte Performance der Tests mit 2 Sekunden Laufzeitbeschränkung. Später kommt es allerdings mehr auf eine große Populationsgröße an um das Problem des lokalen Maximums zu lösen.

9.2 Finale Testserien mit bays29

Parameter	Wert
Number	10
Subpop	3
PopSize	175
Migrationtopology	2
Migrationrate	0.15
Migrationintervall	35
Selection	'seltour'
Recombination	'reccycle'
Swap	1
Move	1
Invert	1

Tabelle 9.5: Parameter der bisherigen optimalen Konfiguration

Wir führen jetzt noch einige letzte Testserien mit den im Verlauf der Ausarbeitung aufgetauchten unschlüssigen Parametern durch. Unsere bisherige Optimalkonfiguration kann Tabelle 9.5 entnommen werden. Wir testen jetzt zum einen mit dieser Konfiguration

(in Tabelle 9.6 als „Normal“ bezeichnet), aber auch jeweils eine Testserie, in der wir selsus als Selektionsverfahren, recox als Rekombinationsverfahren sowie die Mutationsraten 3 3 1 und 0.5 0.5 1 verwenden. Bei jeder abweichenden Testserie wird nur ein Parameter variiert, alle anderen sind wie in Tabelle 9.5. Jede diese Testserien lassen wir in vier Varianten laufen: Generationen auf 400 beschränkt, Laufzeit beschränkt auf 10, 30 und 120 Sekunden.

Test	Best	Average	Worst	Average Time	Average Generations
Normal	2020	2093	2221	19,3	400,0
recox	2026	2062	2152	60,6	400,0
selsus	2031	2121	2266	21,3	400,0
3 3 1	2028	2115	2257	22,3	400,0
0.5 0.5 1	2020	2074	2143	16,8	400,0
Normal	2020	2085	2272	120,1	2241,5
recox	2026	2071	2219	120,1	760,6
selsus	2026	2090	2145	120,1	2105,6
3 3 1	2028	2156	2348	120,1	2143,0
0.5 0.5 1	2020	2063	2159	120,1	2691,4
Normal	2020	2078	2192	30,1	709,2
recox	2020	2088	2213	30,2	206,3
selsus	2026	2137	2322	30,1	628,0
3 3 1	2020	2158	2416	30,1	607,6
0.5 0.5 1	2020	2093	2211	30,1	803,8
Normal	2020	2058	2147	10,1	231,8
recox	2103	2292	2507	10,2	69,0
selsus	2028	2159	2310	10,1	204,3
3 3 1	2020	2220	2630	10,1	199,0
0.5 0.5 1	2026	2131	2328	10,1	260,4

Tabelle 9.6: Finale Tests

Alle Testserien liefern gute Ergebnisse, auch schon nach 10 Sekunden. Es fällt auf, dass die Mutationsraten 0.5 0.5 1 konstant bessere Werte liefern, weshalb wir unsere optimale Konfiguration entsprechend anpassen.

Die Route mit dem besten Ergebnis ist in Abbildung 9.2 zu sehen.

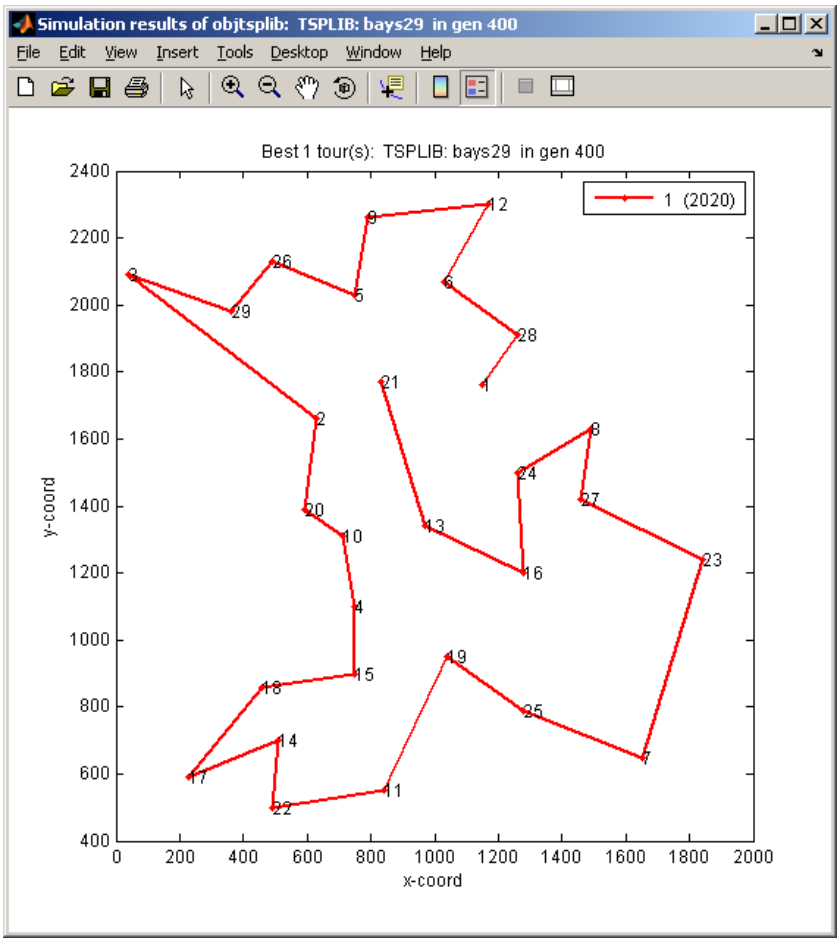

Abbildung 9.2: Bestes Ergebnis für `bays29`

9.3 Test der optimalen Konfiguration mit neuen Daten

Nachdem unsere endgültige optimale Konfiguration jetzt feststeht (Parameter in Tabelle 9.7, lassen wir unseren Algorithmus noch auf ein paar neue Datensätze los, um zu sehen, wie er sich macht.

Die Datei `bayg29.tsp` enthält die bekannten 29 bayrischen Städte, nur mit geographischen Entfernungen statt Straßen. Bei diesem Problem findet unser Algorithmus den besten Weg in guter Zeit. Dieser ist in Abbildung 9.3 zu sehen.

Die Datei `berlin52.tsp` enthält 52 Standorte innerhalb Berlins. Hier finden wir nicht den optimalen Weg von 7542 – diesen konnten wir durch GEATbx beiliegende Daten ermitteln. Wir kommen mit einem besten Wert von 7797 recht nahe daran, unsere Durchschnittswerte sind aber allgemein etwas schlechter. Die Laufzeit ist hier deutlich länger,

was aber mit der erhöhten Komplexität zusammenhängen dürfte. Zudem müssen hier die Entfernungen erst berechnet werden, da sie in der Datei nicht explizit angegeben werden. Wir haben deshalb für Berlin eine zusätzliche Testserie mit großzügigerer Laufzeitbeschränkung durchgeführt. Das Ergebnis wird in Abbildung 9.4 dargestellt.

Parameter	Wert
Number	10
Subpop	3
PopSize	175
Migrationtopology	2
Migrationrate	0.15
Migrationintervall	35
Selection	'seltour'
Recombination	'reccycle'
Swap	0.5
Move	0.5
Invert	1

Tabelle 9.7: Parameter der endgültigen optimalen Konfiguration

Filename	Best	Average	Worst	Average Time	Average Generations
bays29	2020	2079	2297	16,9	400,0
bayg29	1610	1681	1766	16,7	400,0
berlin52	8431	9018	9654	25,1	400,0
berlin52	8252	8814	9697	64,8	1000,0
bays29	2020	2076	2194	30,1	676,5
bayg29	1634	1677	1772	30,1	676,4
berlin52	8317	9149	10294	30,1	464,8
berlin52	7797	8542	9367	120,1	1836,0

Tabelle 9.8: Finale Tests

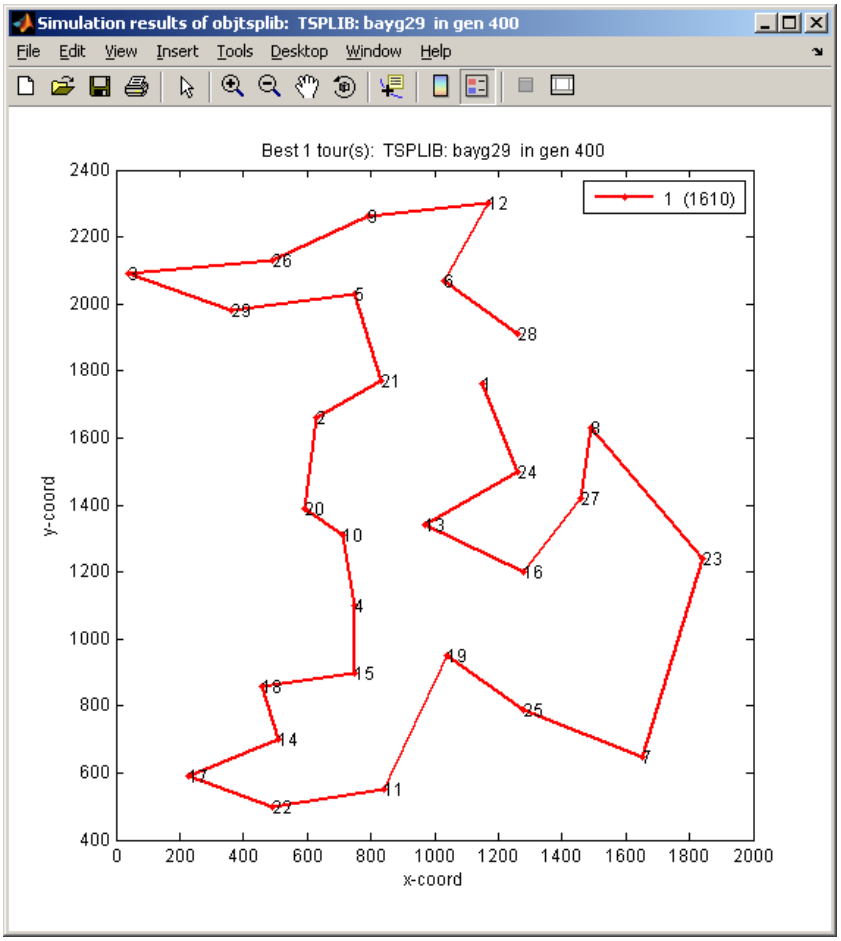

Abbildung 9.3: Bestes Ergebnis für `bayg29`

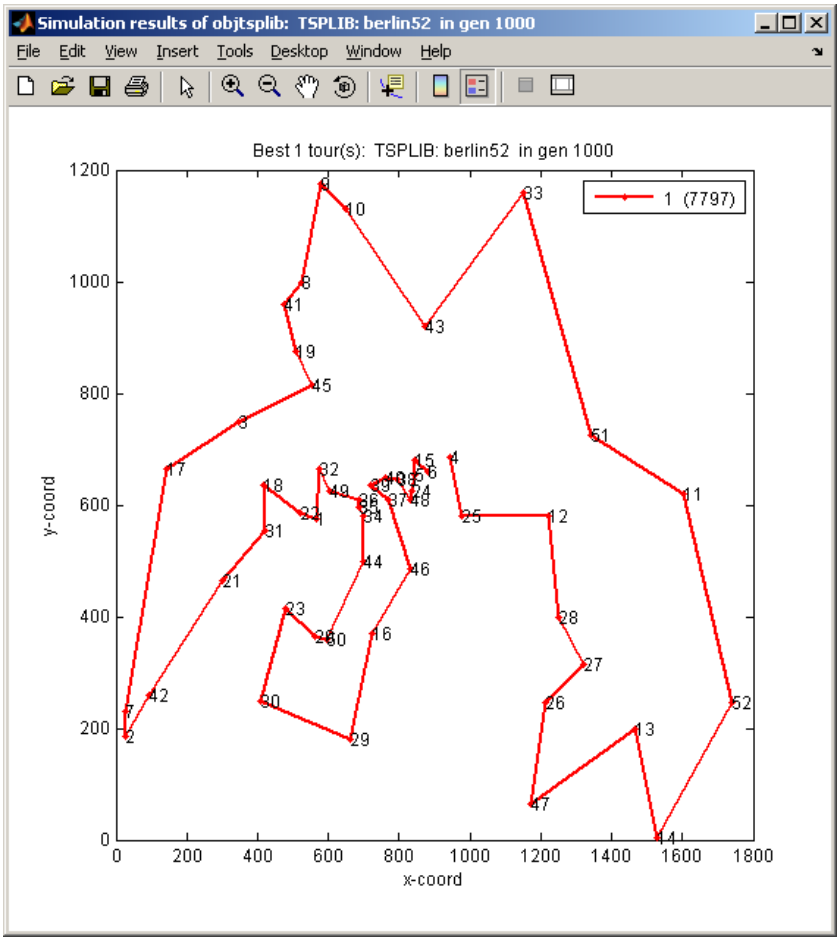

Abbildung 9.4: Bestes Ergebnis für `berlin52`

10 Fazit

Alles in allem konnten wir das ursprüngliche Skript `demotsplib.m` in jeglicher Hinsicht verbessern. Wir erhalten bessere Werte bei nur einem Bruchteil der vorherigen Laufzeit. Insbesondere im Bereich der Rekombinationsverfahren gab es viel Optimierungsspielraum . Im direkten Vergleich der jeweils 2007 und 2008 zu implementierenden Verfahren reccycle und recox ist unsere Einschätzung, dass reccycle immer dann die bessere Wahl ist, wenn es auf eine gute Laufzeit ankommt.

Und wenn die genetischen Algorithmen nicht in einem lokalen Maximum hängen geblieben sind, entwickeln sie sich noch heute weiter.

Literaturverzeichnis

[Chern 2005] Chern, Prof. Maw-Sheng: *Genetic Algorithms: Traveling Salesman Problem.* Version: Oktober 2005. `http://chern.ie.nthu.edu.tw/gen/GA-TSP.pdf`, Abruf: 9. Juli 2008

[Erben 2007] Erben, Prof. Dr. W.: *Genetische Algorithmen - Übungsaufgabe 3 (Version SS 2007).* Mai 2007

[Erben 2008a] Erben, Prof. Dr. W.: *Genetische Algorithmen - Übungsaufgabe 3 (Version SS 2008).* Mai 2008

[Erben 2008b] Erben, Prof. Dr. W.: *Vorlesungsunterlagen Genetische Algorithmen.* März 2008

[Pohlheim 2006] Pohlheim, Hartmut: *GEATbx: Genetic and Evolutionary Algorithm Toolbox for use with MATLAB Documentation.* Version: Dezember 2006. `http://www.geatbx.com/docu/index.html`, Abruf: 8. Juli 2008