Belegarbeit

im Fach Parallele Systeme
Thema:

Parallelisierung genetischer Algorithmen am Beispiel des TSP

im

Fachbereich IV
Angewandte Informatik - Master

Bearbeiter: Peter Horn B.Sc.
Bearbeiter: Josef Rossa B.Sc.

Betreuer: Sebastian Bauer M.Sc.

Datum: 9. August 2013

Hochschule für Technik
und Wirtschaft Berlin
University of Applied Sciences

Inhaltsverzeichnis

1 Rundreiseproblem

Beispiel:
Der Lastkraftwagen einer Spedition soll am Folgetag zehn deutsche Städte abfahren. Damit die Spedition Kosten spart, soll die Gesamtstrecke möglichst kurz sein. Der Lastkraftwagen soll also möglichst wenige Kilometer zurücklegen bis er am Abend wieder auf dem Parkplatz der Spedition parkt.

Das Beispiel stellt ein klassisches Optimierungsproblem vor. Die Optimierung zielt auf die Minimierung der Reiseroute ab. Genauer genommen handelt es sich folglich um ein Minimierungsproblem. In der Literatur wird das vorgestellte Problem als „Rundreiseproblem", „Problem des Handlungsreisenden" oder als „Traveling Salesman Problem" (TSP) referenziert.

Davon ausgehend, dass die Strecke zwischen zwei Städten in beide Richtungen die selben „Kosten" hat, ergeben sich:

$$\frac{(n-1)!}{2}$$

mögliche Routen zur Bestreitung der Rundreise (symmetrisches TSP). Bei unterschiedlichen „Kosten" ergeben sich:

$$(n - 1)!$$

verschiedene Routen (asymmetrisches TSP). Die umgekehrten Routen müssen hier also mit berücksichtigt werden. Im Bezug auf das Eingangsbeispiel ergeben sich bei symmetrischem TSP:

$$\frac{(10-1)!}{2} = 181.440$$

mögliche Routen. Bei asymmetrischem TSP verdoppelt sich diese Zahl. Um eine Vorstellung von dem Wachstum des Suchraums zu bekommen sei angemerkt, dass bei 15 Städten bereits über 43 Milliarden Routen existieren (symmetrisches TSP).
Das Finden der besten Reiseroute erfordert mit einem naiven Ansatz den Vergleich sämtlicher Reiserouten. Gefolgert aus den kombinatorischen Überlegungen ergibt sich für einen solchen Ansatz eine Komplexität von $\mathcal{O}(n!)$. Das Problem des Handlungsreisenden ist deshalb nicht in Polynomialzeit lösbar und liegt demnach in NP.

Nachdem NP harte Probleme praktisch nicht berechenbar sind, werden Ansätze gesucht die das Problem in Polynomialzeit lösen. Dabei kann man Ansätze die zu einer optimalen Lösung führen von solchen unterscheiden, die auf jeden Fall eine gute Lösung ermitteln.

Im Rahmen dieser Arbeit wird unter Anwendung des *Genetischen Algorithmus* (GA) eine Polynomialzeitlösung für das Rundreiseproblem errechnet. Diese Lösung stellt in der Regel keine optimale Lösung, abhängig von Zufall und Konfiguration jedoch eine recht gute Lösung dar.
Zur Verbesserung der Laufzeit liegt ein Schwerpunkt der Arbeit in der Parallelisierung des Algorithmus.

2 Genetischer Algorithmus

2.1 Theorie

Bei den genetischen Algorithmen handelt es sich um Algorithmen, die zur Lösung von Optimierungsproblemen eingesetzt werden.
Wie der Name schon sagt, lehnt sich der genetische Algorithmus an die biologische Evolution an. Dabei wird zwischen Genen, Chromosomen und Chromosomenpopulation (Population) unterschieden, die die Bestandteile des Algorithmus bilden. Demnach bilden die Chromosomen, die sich aus Genen zusammensetzen die Population.
Wenn wir diese biologische Zusammenhänge auf das zu lösende Optimierungsproblem übertragen, bilden die Gene die Bestandteile einer möglichen Lösung; die Chromosomen stellen jeweils eine mögliche Lösung dar und die Population umfasst die Menge von möglichen Lösungen.

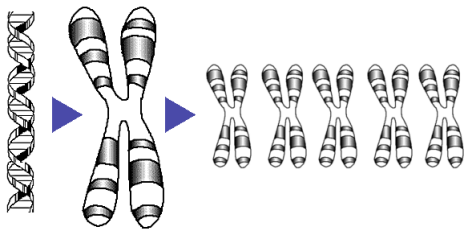

Abbildung 1: Bestandteile des Algorithmus (Gene, Chromosom, Population)

Durch geschicktes Verändern von Chromosomen innerhalb eines Generationszyklus wird die Population mit jeder neuen Generation ständig verbessert.
Ein Generationszyklus lässt sich in folgende Abschnitte unterteilen, die nacheinander durchlaufen werden:

1) Fitnessberechnung
2) Selektion
3) Mutation
4) Crossover

Bei der *Fitnessberechnung* werden die Chromosomen anhand ihrer Gene bewertet bzw. die *Fitnesswerte* berechnet. Anschließend findet die *Selektion* statt. In diesem Schritt werden die Chromosomen mit

einem höheren bzw. besseren Fitnesswert bevorzugt ausgewählt. Die Auswahl von Chromosomen findet zufällig nach dem Prinzip *Ziehen mit Zurücklegen* statt, wobei die Wahrscheinlichkeit gezogen zu werden bei besseren Chromosomen gegenüber schlechten Chromosomen höher ist. Es erfolgen so viele *Ziehungen* wie es Chromosomen in der Population gibt. Das Ziel der Selektion ist es eine neue Population entstehen zu lassen, die gegenüber der vorherigen Population mehr gute Chromosomen aufweist.
Die während der *Selektion* mehrfach ausgewählten Chromosomen (geklonte Chromosomen) werden nun den genetischen Operationen: *Mutation* und *Crossover* unterzogen.
Die *Mutation* wird auf jedes einzelne geklonte Chromosom angewandt. Dabei werden innerhalb eines Chromosoms zufällig ausgewählte Gene vertauscht.
Nachdem alle geklonten Chromosomen *mutiert* wurden, erfolgt das sogenannte *Crossover*.
Das Ziel des *Crossover* ist es, durch das Vermischen von Genen zweier mutierter Chromosomen ein neues, besseres Chromosom zu kreieren. Demnach werden die Gene aus dem ersten Chromosom und die Gene aus dem zweiten Chromosom zu einem neuen Chromosom zusammengesetzt.
Nach dem *Crossover* ist eine neue Population mit neuen Chromosomen entstanden. Diese bildet die nächste Generation der ursprünglichen Population.

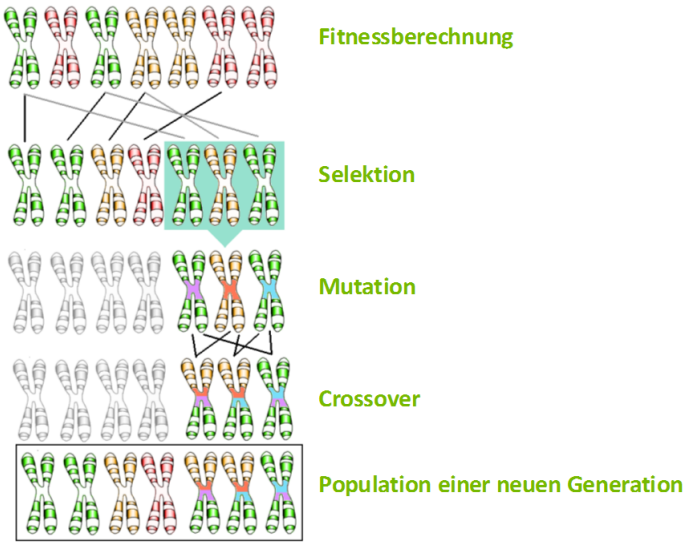

Abbildung 2: Generationszyklus

Die Abbruchbedingung des genetischen Algorithmus kann beliebig gewählt werden.
Typische Abbruchbedingungen sind:

- Erreichen einer bestimmten *Bestfitness*
- Ablauf einer bestimmten Zeit
- Anzahl der Generationszyklen

- Kombination der aufgeführten Abbruchbedingungen

2.2 Terminologie

Bezogen auf das Rundreiseproblem ergeben sich folgende Bestandteile des Algorithmus.
Die *Gene* entsprechen den einzelnen Städten die bereist werden müssen. Demnach setzt sich ein *Chromosom* aus allen Städten zusammen, welche die Route einer Rundreise bilden. Die *Population* stellt eine Menge von möglichen Routen dar.
Bei der *Fitnessberechnung* wird die Gesamtstrecke jeder einzelnen Route ausgerechnet. Die Chromosomen bzw. die Routen, die eine kürzere Strecke aufweisen, haben demnach eine höhere Wahrscheinlichkeit bei der *Selektion* ausgewählt zu werden.
Die *Mutation* eines Chromosoms erfolgt durch das Vertauschen zweier zufällig ausgewählter Gene. Das anschließende *Crossover*, welches auf die mutierten Chromosomen angewandt wird, vereint zwei Routen nach dem Reißverschlussprinzip zu einer neuen Route. Dabei wird für jede geklonte Route per Zufall eine weitere geklonte Route ausgewählt und die Gene bzw. die Städte aus den beiden Routen werden zur einer neuen Route zusammengesetzt. Das *Crossover* fügt alle ungeraden Gene aus dem ersten Chromosom und alle geraden Gene aus dem zweiten Chromosom zusammen.

2.3 Umsetzung

2.3.1 Initiale Population

Bevor der Algorithmus starten kann, muss einmalig eine zufällige Population erzeugt werden. Diese Population enthält eine Menge von zufällig erzeugten Chromosomen (= Rundreisen). Aufgabe dieses Moduls ist die Erzeugung dieser Chromosomen.

Jedes Chromosom (= jede Rundreise) kann als Städte Vektor interpretiert werden. Eine Stadt wird in der Applikation durch einen Integer Wert (Identifikationsnummer) repräsentiert. Der Chromosom Vektor enthält alle möglichen Stadt Identifikationsnummern. Der Ausgangs- und Zielort der Rundreise ist in diesem Vektor nicht enthalten, weil er für die Ermittlung neuer Rundreisen keine Rolle spielt.
Nachdem ein Chromosom als Vektor vorliegt, kann die Population als Matrix repräsentiert werden. Die Anzahl der Routen in der Population ist gleich der Zeilen der Matrix. Die Anzahl der Spalten entspricht der Anzahl der Städte.
Die Funktion, welche die initiale Population in Form dieser Matrix berechnet, erhält einen Vektor mit den Stadtidentifikationsnummer (ohne Ausgangs- und Zielort), die Größe des Vektorraums und einen Wert der anzeigt wie viele Chromosomen (Rundreisen) erzeugt werden sollen.
Aus den gegebenen Größen wird zunächst eine leere Matrix angelegt. Anschließend wird diese Matrix befüllt. Die Realisierung ist in der Abbildung 3 dargestellt. Für jede Zeile der Matrix (jede Route) werden die Stadtidentifikationsnummern beliebig in den Spalten der Matrix angeordnet. Die Stadtidentifikationsnummern werden dabei dem übergebenen Vektor entnommen und anschließen durch

```
for(i = 0; i < numberOfInitialChromosomes; i++) {

    for(k = 0; k < numOfCitiesWithoutStartCity; k++) {
        numOfCitiesLeft = numOfCitiesWithoutStartCity - k;
        randIndex = rand_r(&myseed) % numOfCitiesLeft;
        resultMatrix[i][k] = cityIDsWithoutStartCity[randIndex];
        cutArrayIndex(cityIDsWithoutStartCity, randIndex, numOfCitiesLeft);
    }

}
```

Abbildung 3: Erzeugung der initialen Populationsmatrix

die Funktion *cutArrayIndex* aus diesem entfernt.
Am Ende der äußeren for-Schleife ist die initiale Populationsmatrix vollständig gefüllt.

2.3.2 Fitness Berechnung

Die Berechnung der Fitness eines Chromosoms bzw. einer Route ist relativ unproblematisch, man berechnet die Gesamtstrecke indem man die Entfernungen der Teilstrecken zwischen Start und Ziel aufaddiert.
Für die anschließend folgende Selektion muss die Überlebenswahrscheinlichkeit jedes einzelnen Chromosoms anhand seines Fitnesswertes berechnet werden. Da jedoch ein höherer Fitnesswert bzw. eine längere Gesamtstrecke zur einer kleineren Überlebenswahrscheinlichkeit führt, ist der Fitnesswert antiproportional zu der Überlebenswahrscheinlichkeit des Chromosoms.

```
double fitness_chrom = 0.0;
double surviving_population = 0.0;
double surviving = 0.0;
int i = 0;

// calculate fitness
for (i = 0; i < pop->chromArray_length; i++) {
    fitness_chrom = getChromosomeDistance(pool, pop, &pop->chromArray[i]);
    surviving = 1.0 / fitness_chrom;
    pop->chromArray[i].fitness = fitness_chrom;
    pop->chromArray[i].surviving = surviving;
    surviving_population += surviving;
}

// calculate surviving
for (i = 0; i < pop->chromArray_length; i++) {
    surviving = pop->chromArray[i].surviving / surviving_population;
    pop->chromArray[i].surviving = surviving;
}
```

Abbildung 4: Berechnung der Fitness und der Überlebenswahrscheinlichkeit

In der Abbildung 4 wird die Fitness und die Überlebenswahrscheinlichkeit aller in der Population

vorhandenen Routen berechnet.
In der ersten Schleife wird der Fitnesswert *fitness_chrom* und die vorläufige Überlebenswahrscheinlichkeit *surviving* jedes einzelnen Chromosoms berechnet. Um die angesprochene Antiproportionalität der Überlebenswahrscheinlichkeit zu der Fitness zu erreichen wird die *surviving* (Überlebenswahrscheinlichkeit) mit *1.0 / fitness_chrom* berechnet und zu der Gesamtwahrscheinlichkeit *surviving_population* dazu addiert.
Anschließend wird in der zweiten Schleife die tatsächliche Überlebenswahrscheinlichkeit jeder einzelnen Route bezogen auf die gesamte Population mit *surviving = pop-¿chromArray[i].surviving / surviving_population* berechnet.

2.3.3 Selektion

Bei der Selektion wird von einer festen Reihenfolge der Chromosomen innerhalb einer Population ausgegangen. Die Population stellt somit ein Chromosomen-Array dar.
Eine Population die aus vier Chromosomen besteht kann man sich wie in Abbildung 5 dargestellt vorstellen.

Abbildung 5: Selektion

Dabei besitzt jedes Chromosom seine Überlebenswahrscheinlichkeit bezogen auf die gesamte Population, die in der *Fitnessberechnung* ermittelt wurden. Nun wird bei der Auswahl jedes Mal eine Zufallszahl zwischen 0.0 und 1.0 (0%-100%) generiert anhand welcher bestimmt wird, welches Chromosom aus der Population gewählt wird. Wurde beispielsweise eine Zufallszahl 0.75 *gezogen*, so wird das *Chromosom 3* gewählt, da dieses an der 75%-Marke liegt.

Durch das Aufaddieren der Überlebenswahrscheinlichkeiten der einzelnen Chromosome in der Variable *survivingSum* nähert man sich der Zufallszahl *randomValue*. Ist die Summe der Überlebenswahrscheinlichk
größer oder gleich der Zufallszahl, wird das aktuelle Chromosom aus dem Populations-Array *&pop-¿chromArray[j]* ausgewählt.
Anschließend wird mit Hilfe der Methode *isInChromosomePointerArray* entschieden, ob das ausgewählte Chromosom in die Liste der geklonten bzw. mehrfach vorkommenden Chromosomen *cloneChromArray* aufgenommen wird. Diese wird, wie erläutert, im späteren Verlauf den genetische Operationen *Mutation* und *Crossover* unterzogen.

```
for (i = 0; i < chromArray_length; i++) {
    randomValue = (double) rand() / (double) RAND_MAX;
    survivingSum = 0.0;
    for (j = 0; j < chromArray_length; j++) {
        surviving = (*pop).chromArray[j].surviving;
        survivingSum += surviving;
        if (survivingSum >= randomValue) {
            if (!isInChromosomePointerArray(uniqueChromArray, &pop->chromArray[j])) {
                addToChromosomePointerArray(uniqueChromArray, &pop->chromArray[j]);
            } else {
                addToChromosomePointerArray(cloneChromArray, &pop->chromArray[j]);
            }
            break;
        }
    }
}
```

Abbildung 6: Selektion Code

2.3.4 Mutation

Bei der Mutation werden zwei zufällige Gene eines jeden Chromosomes in einer Population vertauscht. Die Funktion, welche die Mutation realisiert enthält dazu einen Zeiger auf eine Population Struktur. Die Struktur enthält sämtliche Chromosomen als Array.

```
for (i = 0; i < rows; i++) {
    randIndex1 = rand_r(&myseed) % columns;
    do {
        randIndex2 = rand_r(&myseed) % columns;
    } while (randIndex1 == randIndex2);
    Chromosome *chrom = &pop->chromArray[i];
    rem_valIndex1 = chrom->cityArray_without_startCity[randIndex1];
    chrom->cityArray_without_startCity[randIndex1] = chrom->cityArray_without_startCity[randIndex2];
    chrom->cityArray_without_startCity[randIndex2] = rem_valIndex1;
}
```

Abbildung 7: Mutation einer Population

Wie in Abbildung 7 zu sehen, werden für jedes Chromosom einer Population zwei unterschiedliche Zufallszahlen erzeugt. Eine Chromosom Struktur enthält wieder die Stadtidentifikationsnummern als Vektor (vgl. 2.3.1). Die Zufallszahlen sprechen nun einen Index dieses Vektors an und vertauschen die dazugehörigen Inhalte. Am Ende der for-Schleife wurde in jedem Chromosom der übergebenen Population eine Stadtidentifikationsnummer mit einer anderen vertauscht. Auf diese Weise ergeben sich neue, wenn auch nicht sehr verschiedene Routen. Einer intensiveren Veränderung der Route dient das Crossover.

2.3.5 Crossover

Das Zusammenführen zweier Chromosomen zu einem neuen Chromosom nach dem Reißverschlussprinzip (*Crossover*) ist nicht unproblematisch, da die Genreihenfolge bzw. die Reihenfolge der Städte einer Route variiert.

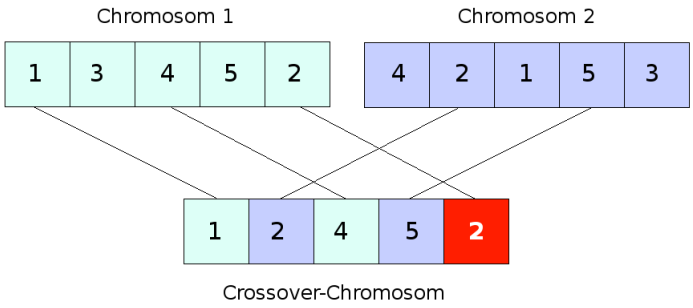

Abbildung 8: Problematik des Reißverschlussprinzips beim Crossover

In der Abbildung 8 sieht man bei dem letzten Gen des neuen *Crossover-Chromosoms*, das dieses an der zweite Stelle bereits vorkommt. Demnach lässt sich das Reißverschlussprinzip nicht zu 100% umsetzen.
Daher erfolgt das Crossover in zwei Schritten. Im ersten Schritt wird versucht das Reißverschlussprinzip anzuwenden. Ist dies an den entsprechenden Stellen des neuen Chromosoms nicht möglich, wird das Gen nicht gesetzt.
Im zweiten Schritt erfolgt dann die *Reparatur* des *Crossover-Chromosoms*. Es werden fehlende Gene ermittelt und nach dem Zufallsprinzip in die *Genlücken* eingesetzt.

3 Parallelisierung des Genetischen Algorithmus

3.1 Möglichkeiten

Abbildung 9 kategorisiert die verschiedensten Möglichkeiten der Parallelisierung genetischer Algorithmen.

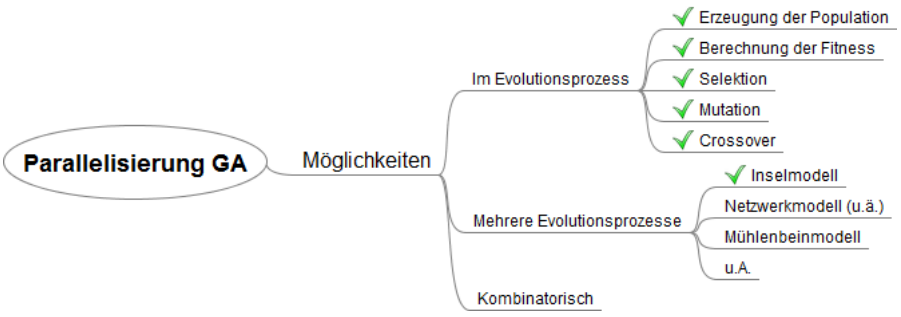

Abbildung 9: Möglichkeiten der Parallelisierung GA

In der ersten Kategorie handelt es sich um die Parallelisierung im Evolutionsprozess. Wie bereits im Abschnitt 2 erläutert durchläuft der GA mehrere Evolutionszyklen. Vor dem ersten Evolutionszyklus wird meist zufällig eine initiale Population erzeugt. Wie noch gezeigt wird, lässt sich die Erzeugung der initialen Population sehr gut parallelisieren (vgl. Abschnitt 3.2.1). Der Mehrwert bezüglich der

Gesamtlaufzeit ist allerdings gering, weil die Erzeugung der initialen Population eben nur einmal stattfindet.
Anders verhält es sich bei den Modulen im Evolutionsprozess die in jedem Evolutionszyklus wieder ausgeführt werden. In einem späteren Abschnitt wird gezeigt, dass sich die Berechnung der Fitness, die Selektion, die Mutation und der Crossover einzeln parallelisieren lassen. Im Abschnitt 5 wird vorgestellt welche Veränderung sich im Bezug auf die Gesamtlaufzeit ergeben hat.

Neben der Parallelisierung im Evolutionsprozess gibt es die Möglichkeit mehrere Evolutionsprozesse gleichzeitig zu starten. Jeder Evolutionsprozess bringt dann eigene Ergebnisse hervor. Diese Möglichkeit der Parallelisierung ist insbesondere in der Kombination mit dem *Grid Computing* interessant. Jeder Knoten im *Grid* könnte beispielsweise einen eigenen Evolutionsprozess behandeln.
Agieren die einzelnen Evolutionsprozesse unabhängig voneinander ist die Rede von einem *Inselmodell*. Für den Fall, dass die Evolutionsprozesse Informationen (bspw. gute Chromosomen) austauschen gibt es eine Reihe unterschiedlicher Modelle. Das *Netzwerkmodell* und das *Mühlenbeinmodell* seien exemplarisch genannt.

Letztendlich gibt es noch die Möglichkeit genetische Algorithmen mit anderen Algorithmen zu kombinieren. Beispielsweise kann ein *Greedy* Algorithmus parallel zum Evolutionsprozess gute Chromosomen generieren und diese an den Evolutionsprozess übergeben. Damit der *Greedy* mehrere Ergebnisse liefert kann er etwa immer nur auf eine Teilstrecke angewandt werden, während der andere Teil der Strecke statisch ist. Dies sei nur als ein Beispiel zur Parallelisierung durch Kombination aufgeführt. Im Prinzip lässt sich der genetische Algorithmus mit sämtlichen Optimierungsansätzen für eine Problem kombinieren. Es ist dabei stets vorteilhaft wenn der genetische Algorithmus parallel nach neuen Lösungen sucht.
Die Haken in Abbildung 9 zeigen, an welchen Stellen der genetische Algorithmus im Rahmen dieser Arbeit parallelisiert wurde. Es wird deutlich, dass noch weitere Möglichkeiten zur Optimierung der Ergebnisfindung offen sind.

3.2 Umsetzung

3.2.1 Initiale Population

Die Erzeugung der initialen Population (vgl. Abschnitt 2.3.1) kann auf einfache Weise parallelisiert werden. Jeder *Thread* erzeugt dabei einen Teil der Gesamtpopulation. Wie viele Chromosomen ein *Thread* erzeugt wird berechnet indem die Anzahl der Chromosomen in der initialen Population durch die Anzahl der *Threads* geteilt wird. Sofern die Anzahl der Chromosomen nicht ohne Rest durch die Anzahl der *Threads* geteilt werden kann übernimmt der letzte *Thread* die Erzeugung der übrigen Chromosomen.
Die äußere for-Schleife dargestellt in Abbildung 3, steht also in Abhängigkeit der verfügbaren *Threads*. Die dahingehende Modifikation ist in Abbildung 10 abgebildet. Die Variable *arrays_per_this_thread* enthält dabei die Anzahl der Chromosomen die dieser *Thread* erzeugen soll.

```
int counter = thread_num * arrays_per_thread;

for(i = counter; i < (counter + arrays_per_this_thread); i++) {
```

Abbildung 10: Modifikation der Erzeugung der initialen Populationsmatrix

3.2.2 Fitness Berechnung

Bei der Parallelisierung der Fitnessberechnung wurden die beiden Schleifen mit *omp for* parallelisiert (siehe Abbildung 11).

```
// calculate fitness
#pragma omp parallel shared(surviving_population)
#pragma omp for private(i,fitness_chrom, surviving)
    for (i = 0; i < pop->chromArray_length; i++) {
        fitness_chrom = getChromosomeDistance(pool, pop, &pop->chromArray[i]);
        surviving = 1.0 / fitness_chrom;
        pop->chromArray[i].fitness = fitness_chrom;
        pop->chromArray[i].surviving = surviving;
#pragma omp critical
        surviving_population += surviving;
    }

// calculate surviving
#pragma omp parallel
#pragma omp for private(i,surviving)
    for (i = 0; i < pop->chromArray_length; i++) {
        surviving = pop->chromArray[i].surviving / surviving_population;
        pop->chromArray[i].surviving = surviving;
    }
```

Abbildung 11: Fitnessberechnung

Da die Berechnung der Fitness jedes einzelnen Chromosoms unabhängig voneinander erfolgt, können die Threads die Population einfach unter sich aufteilen.
Die Berechnung der Überlebenswahrscheinlichkeit ist jedoch von der gesamten Population abhängig. So muss bei der Ermittlung der *surviving_population* die Aufsummierung *surviving_population += surviving* synchronisiert werden. Anschließend kann die zweite Schleife problemlos, entsprechend der Fitnessberechnung, parallelisiert werden.

3.2.3 Selektion

Auch die Selektion lässt sich problemlos parallelisieren. Da jede Auswahl eines Chromosoms unabhängig von der Selektion-Ergebnismenge ist, führt jeder Thread die entsprechende Anzahl an *Ziehungen* der Chromosome aus der Populationsmenge durch. Bei drei Threads und einer Populationsgröße von 90 Chromosomen, führt beispielsweise jeder Thread 30 *Ziehungen* durch.
Die Ergebnismenge der Selektion bilden zwei Arrays: *uniqueChromArray* und *cloneChromArray*, wobei die erste Menge eindeutig ist und die zweite die geklonten Chromosomen beinhaltet. Da alle Threads die beiden Arrays mit Hilfe der Methode *addToChromosomePointerArray* befüllen bzw. schreibend darauf zugreifen, muss an dieser Stelle synchronisiert werden.

```
#pragma omp parallel shared(uniqueChromArray, cloneChromArray)
    {
        unsigned int myseed = omp_get_thread_num() + time(NULL );
#pragma omp for private(i,j,randomValue,survivingSum,surviving)
        for (i = 0; i < chromArray_length; i++) {
            randomValue = (double) rand_r(&myseed) / (double) RAND_MAX;
            survivingSum = 0.0;
            for (j = 0; j < chromArray_length; j++) {
                surviving = (*pop).chromArray[j].surviving;
                survivingSum += surviving;
                if (survivingSum >= randomValue) {
#pragma omp critical
                    if (!isInChromosomePointerArray(uniqueChromArray, &pop->chromArray[j])) {
                        addToChromosomePointerArray(uniqueChromArray, &pop->chromArray[j]);
                    } else {
                        addToChromosomePointerArray(cloneChromArray, &pop->chromArray[j]);
                    }
                    break;
                }
            }
        }
    }
```

Abbildung 12: Selektion

3.2.4 Mutation

Bei der Parallelisierung der Mutation (vgl. Abschnitt 2.3.4) übernimmt jeder *Thread* eine bestimmte Anzahl an Chromosomen der übergebenen Population. Wie viele Chromosomen ein *Thread* mutiert wird analog zur Vorgehensweise in Abschnitt 3.2.1 errechnet.

```
for(i = counter; i < (counter + rows_per_this_thread); i++) {
```

Abbildung 13: Modifikation der Mutation einer Population

Abbildung 13 zeigt, dass die for-Schleife mit der über alle Chromosomen der Population iteriert wird abhängig von der Anzahl der *Threads* ist. Auch diese Verfahren ist angelehnt an die Modifikation in Abschnitt 3.2.1.

3.2.5 Crossover

Ein einzelner Crossover-Vorgang ist, von den anderen Crossover-Vorgängen unabhängig und kann in einem beliebigen Thread ausgeführt werden. Lediglich beim Schreiben in die Crossover-Ergebnismenge muss synchronisiert werden.

4 Applikation

4.1 Technologien

Die Verwendung eines C basierten *Frameworks* zur Parallelisierung (OpenMP) war Voraussetzung bei diesem Projekt. Da zur Erstellung der grafischen Oberfläche die Wahl auf JavaFX 2.2 als GUI

Framework viel, war man auf Java und C festgelegt.

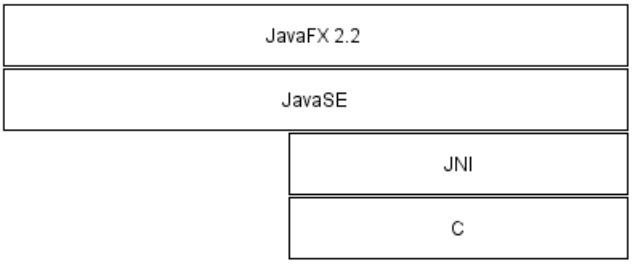

Abbildung 14: Eingesetzte Technologien

Abbildung 14 visualisiert den Aufbau der eingesetzten Technologien. JavaFX 2.2 dient, wie bereits erwähnt, als Basis API für die grafische Oberfläche. Die Business Logik ist in Standard Java realisiert. Hauptbestandteile der Business Logik ist ein Service zum Laden der persistierten Applikationsdaten (Städte und Entfernungen), ein Service zum Zugriff auf Konfigurationen und ein Service der den Algorithmus steuert. Der GA Service bedient sich intern dem Java Native Interface (JNI) um auf die in C implementierte Algorithmus Funktionalität zuzugreifen. In C existiert jeweils eine parallele und eine serielle Realisierung des Algorithmus.
Näheres zur Systemarchitektur findet sich im folgenden Unterabschnitt.

4.2 Architektur

Die Systemarchitektur besteht im Wesentlichen aus drei funktionalen Einheiten (vgl. Abbildung 15).

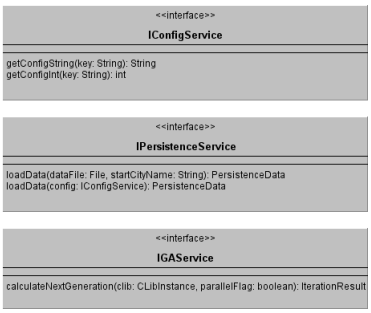

Abbildung 15: Funktionale Schnittstellen

Die Realisierung der *IConfigService* Schnittstelle bietet Zugriff auf wichtige Systemkonfigurationen. Die Konfigurationen werden aus einer *properties Datei* geladen. Beispielweise kann darin die Populationsgröße, oder die Anzahl der zu berechnenden Generationen vorgegeben werden.
Die *IPersistenceService* Schnittstellen Realisierung lädt persistierte Daten der Applikation. Die Daten werden in einer *PersistenceData* Instanz (siehe Abbildung 16) zusammengeführt.
Die Daten enthalten alle Orte und deren Entfernung (Datenbasis der Routenberechnung).
Den eigentlichen Algorithmus realisiert die implementierende Klasse der *IGAService* Schnittstelle.

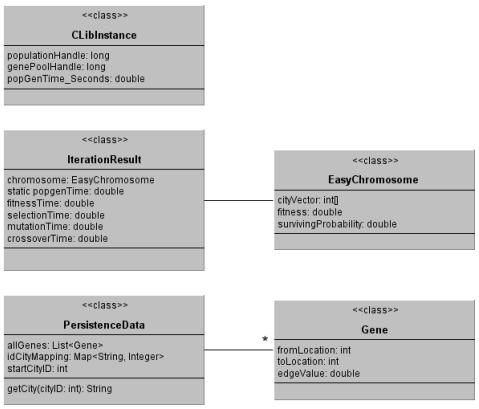

Abbildung 16: Wesentliche Klassen

Mit jedem Aufruf der Methode *calculateNextGeneration* wird die nächste Generation berechnet und als *IterationResult* (siehe Abbildung 16) zurückgegeben. In dem Klassendiagramm ist zu erkennen, dass darin auch die Zeitmessung festgehalten ist. Wie bereits im vorhergehenden Abschnitt dargelegt ist der Algorithmus in C implementiert. Die Klasse *CLibInstance* hält die Referenzen für den Zugriff auf die C Funktionalität vor.
Die grafische Oberfläche ist nach dem *MVVM Pattern* realisiert. Die Anzeige (*View*) ist dabei an ein entsprechendes Datenmodell (*Model*) gebunden und aktualisiert sich automatisch sobald Änderungen an dem Model vorgenommen werden.
In der Anlage befinden sich Bildschirmfotos der drei hauptsächlichen Benutzer Dialoge. In Abbildung 19 ist ein Graph zu erkennen, der sowohl die Fitness des besten gefundenen Chromosoms, als auch die des besten Chromosoms aus der zuletzt berechneten Population visualisiert. Abbildung 20 zeigt einen Dialog in dem die Zeitmessungen bezogen auf die einzelnen Schritte des Algorithmus dargestellt werden, während in Abbildung 21 die aktuell beste gefundene Route angezeigt wird.

5 Ergebnisse

5.1 Rundreiseproblem

Im Rahmen dieser Arbeit soll eine Polynomialzeitlösung für das Rundreiseproblem errechnet werden. Anfangs wurde propagiert, dass sich die Anwendung des *Genetischen Algorithmus* dazu eignet. Abbildung 17 zeigt nun den Kurvenverlauf anhand welchem die These überprüft werden soll.
Auf der X Achse des Graphen sind die Generationen aufgetragen. Die Y Achse bildet die Fitnesswerte ab. Die grüne Kurve visualisiert den zur Generation besten gefundenen Fitnesswert insgesamt. Die gelbe Kurve zeigt hingegen den besten Fitnesswert der jeweiligen Generation. Wenn es Abweichungen der Kurven gibt bedeutet das, dass die beste gefundene Lösung, etwa durch Selektion und Mutation, wieder verworfen wurde.
In der Tendenz des Generationsverlaufs ist zu erkennen, dass die Fitnesswerte zunächst relativ hoch sind, anschließend schnell fallen und bis zum Ende der Generationszyklen dann kontinuierlich aber

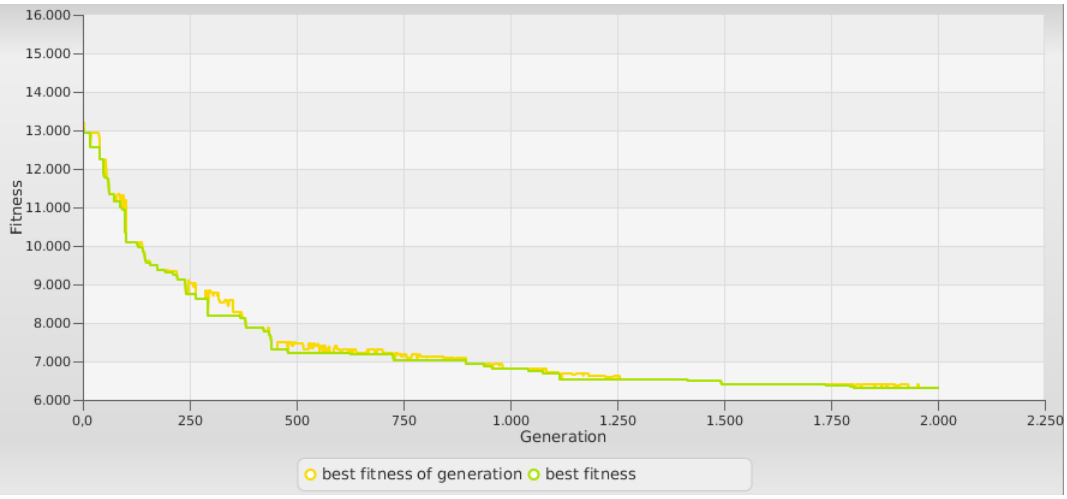

Abbildung 17: Fitnessverlauf über 2000 Generationszyklen

immer langsamer abnehmen. Dieser Verlauf entspricht exakt dem erwarteten Verlauf. Anfangs werden zufällig Lösungen generiert. Diese Lösungen sind in der Regel nicht gut. Der *Genetische Algorithmus* findet dann anfangs schnell bessere Lösungen. In späteren Generationen ist es nicht mehr so leicht bessere Lösungen zu finden, deshalb nimmt das Kurvengefälle hier ab.
Verglichen mit Ansätzen die alle Möglichen Rundreisen betrachten kommt der *Genetische Algorithmus* sehr schnell zu einem akzeptablen Ergebnis. Im Kurvenverlauf von Grafik 17 lagen dem Algorithmus 15.000 zufällige Chromosomen zugrunde. Innerhalb von 2.000 Generationszyklen hat es der Algorithmus geschafft diese 15.000 Chromosomen so zu mutieren, dass sich der Wert der besten Rundreise von anfangs knapp über 13.000 Kilometer auf am Ende 6.320 Kilometer verbessert hat.

5.2 Parallelisierung

Die Ergebnisse der Parallelisierung sind in der Ergebnistabelle in Abbildung 18 zusammengefasst. Eine Zeile der Tabelle entspricht einer Konfiguration des *Genetischen Algorithmus*. In jeder Konfiguration wurde der Algorithmus zwei Mal auf äquivalenter Hardware durchlaufen. Jeder der Rechner auf welchen die Tests durchgeführt wurden hatte vier Prozessorkerne. Die Zeiten in der Tabelle sind Durchschnittswerte der beiden Durchläufe. Die Fitnesswerte entsprechen dem jeweils besten Wert der Durchläufe.
Am Beispiel von Zeile eins wird demonstriert wie sich die Tabelle lesen lässt:

Bei einer Populationsgröße von 1.000 Chromosomen und 500 Generationszyklen benötigte der Algorithmus in der seriellen Variante zwei Sekunden. In der parallelen Variante dauerte die Berechnung 15 Sekunden. Das bedeutet, dass die parallele Berechnung um den Faktor 6,64 langsamer durchgelaufen ist. Die beste gefundene Fitness in dieser Konfiguration lag bei 9.366 Kilometern.

Ein wesentliches Ergebnis der Tabelle ist, dass die serielle Variante bei kleinen Populationen um ein

Populationsgröße	Iterationen	seriell in Sek.	parallel in Sek.	Verbesserung (Faktor)	beste Fitness
1000	500	2	15	6,64	9366
1000	2000	9	57	6,13	8235
5000	500	39	31	1,27	8819
5000	2000	157	124	1,27	7690
15000	500	324	147	2,21	7644
15000	2000	1291	579	2,23	6320
50000	100	699	261	2,67	10424

Abbildung 18: Ergebnistabelle der Parallelisierung

vielfaches schneller ist als der parallele Ansatz. Eine Mögliche Ursache für diese Phänomen könnte die Sicherstellung der *Cache-Kohärenz* sein.
Bei größeren Populationen wirkt sich die Parallelisierung dann aber doch positiv aus. Bereits bei einer Populationsgröße von 5.000 Chromosomen ergibt sich eine zeitliche Verbesserung um den Faktor 1,27. Mit zunehmender Populationsgröße wird die Zeitersparnis noch deutlicher. Die besten Fitnesswerte erzielte die Konfiguration mit 15.000 Chromosomen in der Population beim Durchlauf von 2.000 Generationen.

Literatur

[HAR07] Harbich, Steffen *„Einführung genetischer Algorithmen mit Anwendungsbeispiel“* (2007)
URL: wwwisg.cs.uni-magdeburg.de/sim/vilab/2007/papers/12_genetisch_sharbich.pdf
(Zugriff 17.04.2013 19:33 Uhr)

Abbildungsverzeichnis

Abbildung 19: GUI Graph

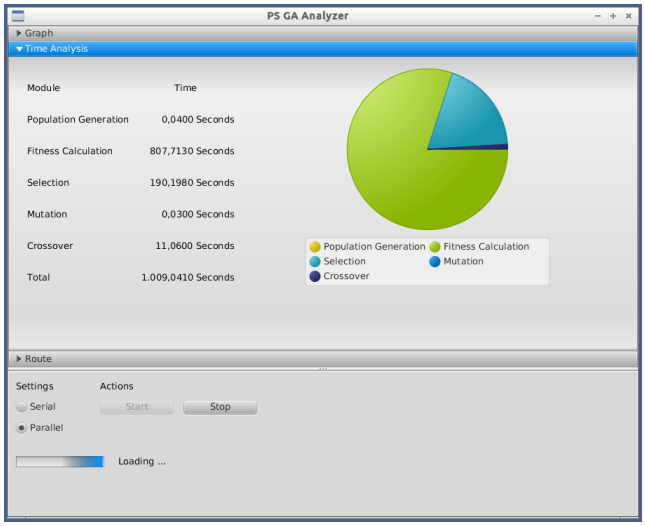

Abbildung 20: GUI Zeitauswertung

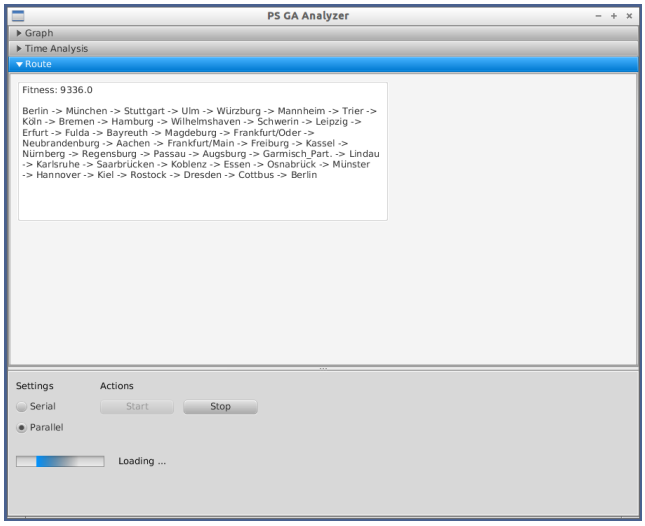

Abbildung 21: GUI Route