Fahrzeugeinsatzplaung

Entwicklung einer Anwendung zur Optimierung

Oliver Pohling, David Mittelstädt, Henrik Voß

20. Juli 2014

Inhaltsverzeichnis

1 Einleitung

1.1 Einführung

Ziel dieses Projektes ist es, ein Programm zu entwickeln, mit dem eine Fahrzeugeinsatzplanung unter Berücksichtigung bestimmter Restriktionen durchgeführt werden kann.

1.2 Anforderungen

Dem Programm wird eine Konfiguration übergeben, in der folgende Parameter beschrieben sind:

- Fahrzeuge
  - Ein Fahrzeug kann einen Produkt transportieren
  - Durchschnittliche Geschwindigkeit
  - Kapazität (maximale Anzahl der Produkte)
  - Zeitfenster (Arbeitsbeginn und -ende)
  - Start- und End-Depot (Station)
- Produkte
  - Name des Produkts
- Station
  - Name der Station
  - X-Y-Koordinaten
- Auftrag
  - Name des Auftrages
  - Zu transportierendes Produkt
  - Anzahl der Produkte
  - Beliebige Auf- und Ablade-Station
  - Zeitfenster bei Auf- und Ablade-Station

Die Entfernung zwischen den Stationen kann über die Koordinaten berechnet werden.

1.3 Restriktionen

Folgende Restriktionen sind bei der Planung zu berücksichtigen:

- Alle Aufträge müssen erledigt werden
- Ein Fahrzeug darf nicht überladen werden
- Ein Fahrzeug startet und endet im Depot

- Ein Fahrzeug darf nur innerhalb seines Zeitfensters fahren
- Ein Produkt muss zuerst aufgeladen werden, bevor es abgeladen werden kann
- Das Auf- und Abladen muss innerhalb des Zeitfensters erfolgen

2 GenetischerAlgorithmus

Das nachfolgende Kapitel stellt den genetischen Algorithmus, als Verfahren zur Lösung komplexer Optimierungsprobleme, vor. Dazu werden im ersten Schritt die benötigten Grundlagen geschaffen und im Anschluss die speziellen Operatoren im Zusammenhang mit dem „PDPTW“ vorgestellt.

2.1 Grundlagen

Bei den genetischen Algorithmen handelt es sich um Verfahren, die zur Lösung komplexer Optimierungsaufgaben eingesetzt werden. Sie beruhen auf Methoden und Erkenntnisse der biologischen Genetik. Dabei dient insbesondere die Evolutionstheorie als Vorbild für die Entwicklung der genetischen Algorithmen, da die Evolution bislang gute Ergebnisse in der Natur geliefert hat. [5] [1]

Die grundlegende Arbeit, zur Schaffung dieser Verfahrenstechnik, wurde von I. Rechenberg mit dem 1960 erschienenen Werk „Evolutionsstrategie“ geschaffen. Auf diesem Wissen aufbauend, erfand John Holland 1975 die genetischen Algorithmen, die in seinem Werk „Adaption in Natural and Artificial Systems“ niedergeschrieben wurden.

Der genetische Algorithmus beruht dabei auf eine simple Verfahrenstechnik, welche nachfolgend dargestellt und erklärt wird:

- Initialisiere Population P
- Evaluiere alle Individuen in P
- Wiederhole, solange Abbruchbedingung nicht erreicht
  - Selektiere ein Individuenpaar x, y aus P als Eltern
  - Erzeuge zwei Nachkommen x‘, y‘ aus x und y durch Anwendung von Crossover-Operationen mit der Wahrscheinlichkeit p(cross)
  - Erzeuge modifizierte Nachkommen x‘‘, y‘‘ aus x‘ und y‘ durch Anwendung von Mutations-Operatoren mit der Wahrscheinlichkeit p(mut)
  - Füge x‘‘ und y‘‘ der Population P hinzu und entferne schlechtere Individuen
- Gib das beste Individuum aus P als Lösung aus.

Die einzelnen Schritte des genetischen Algorithmus werden dabei in den nachfolgenden Kapiteln anhand einer speziellen Problemstellung (PDPTW) konkretisiert und erläutert.

2.2 Repräsentation

Eine der größten Herausforderungen für die Entwicklung und den Erfolg des genetischen Algorithmus ist die Wahl einer geeigneten Repräsentationsebene. Dazu weißt der PDPTW zwei strukturell heterogene Teilprobleme auf. Zum einen muss eine Zusammenfassung von Aufträgen mit ihrer Zuordnung zu einem Fahrzeug als Gruppierungsproblem dargestellt werden. Auf der anderen Seite ist zusätzlich ein Reihenfolgeproblem zu lösen, indem eine zulässige Route bestimmt wird, in der die Ladeorte anzufahren sind. Aus diesen Einschränkungen folgt, dass

beispielsweise eine binäre Kodierung, die im klassischen genetischen Algorithmus angewandt wird, keine geeignete Repräsentationsform darstellt.

Bevor nun eine Möglichkeit der Darstellung der beiden genannten Aspekte vorgestellt wird, müssen die Begriffe des Individuums und des Gens in den Kontext des PDPTW gebracht werden. Ein Individuum ist im nachfolgenden ein kompletter Tourenplan, der aus der Gesamtheit aller Touren besteht, die zur Erfüllung aller Aufträge notwendig sind. Ein Gen hingegen besteht aus exakt einer Tour, der ein Fahrzeug und die verschiedene Aufträge zugeordnet sind. Dies soll die nachfolgende Abbildung 1 verdeutlichen:

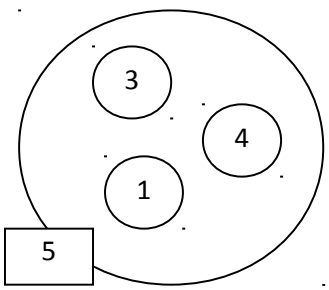
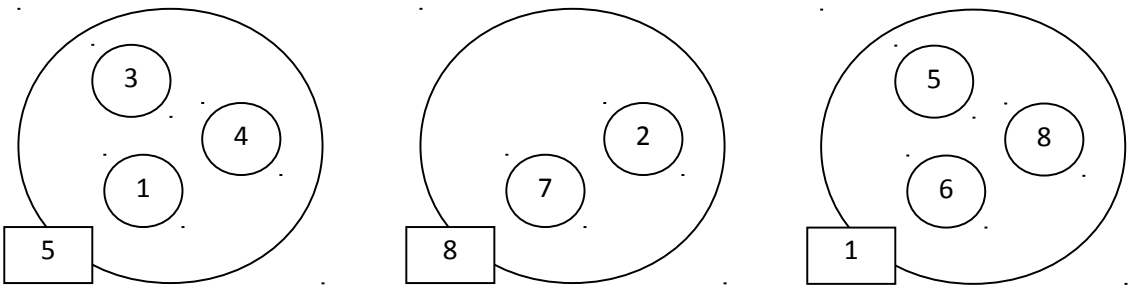


Abbildung 1: Repräsentation Gen (Tour)

Hier ist im unteren linken Bereich das verwendete Fahrzeug (Fahrzeug 5) zu finden, das für die Auslieferung der Aufträge 1, 3 und 4 verantwortlich ist. Desweiteren soll im späteren Verlauf zusätzlich die Reihenfolge identifizierbar sein, um genau zu wissen, wann welcher Auftrag ab- bzw. aufgeladen wurde. Ein Individuum stellt demnach die Gesamtheit aller Touren dar, die zur Erledigung aller Aufräge gebildet werden müssen. Dies verdeutlich die Abbildung 2:



(a) Tour 1 (b) Tour 2 (c) Tour 3

Abbildung 2: Repräsentation Individuum (Plan)

Hierbei handelt es sich um ein konkretes Individuum, das drei Fahrzeuge verwendet, um die gezeigten acht Aufträge zu erfüllen. Zusätzlich muss jedes Gen mit einer ergänzenden Datenstruktur assoziiert werden, so dass im späteren Verlauf der Reihenfolgeaspekt dargestellt werden kann. Diese Darstellung wird im Umfeld des genetischen Algorithmus als Phänotyp bezeichnet. Dies veranschaulicht die Abbildung 3 auf der nächsten Seite zu einer gegebenen Tour:

Auf diese Informationen haben die genetischen Operatoren unmittelbaren Zugriff, um eine möglichst große Varianz in die spätere Population einzubringen.

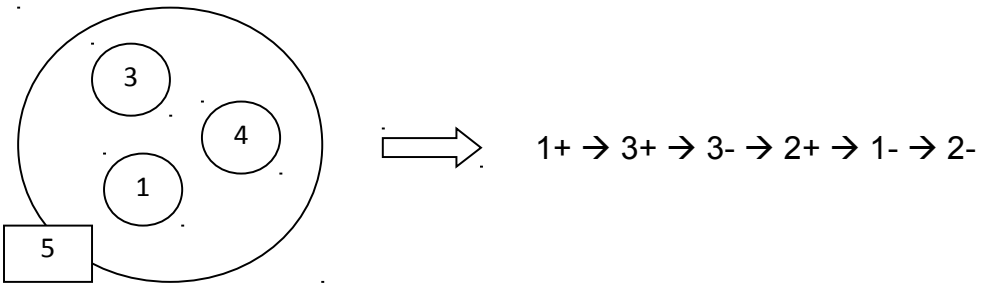


Abbildung 3: Aktionsreihenfolge aus einem Gen

2.3 Aufbau der Startpopulation

Bevor der genetische Algorithmus arbeiten kann, muss im ersten Schritt eine Startpopulation erzeugt werden, die für den Algorithmus als Ausgang dient. Dazu gibt es prinzipiell zwei verschiedene Vorgehensweisen:

1. Zufällige Auswahl von Individuen mit zufälligen Merkmalen
2. Auswahl von „guten“ Individuen, die durch bestimmte Konstruktionsheuristiken erzeugt wurden

Die zweite Variante bringt ein gewisses Risiko mit, da es beim genetischen Algorithmus passieren kann, dass die Suche, aufgrund bereits guter Lösungen, vorzeitigt konvergiert. Um diesem Problem entgegen zu wirken, wurde der Aufbau der Startpopulation mit beiden Möglichkeiten gekoppelt. Dies soll die Abbildung 4 verdeutlichen:

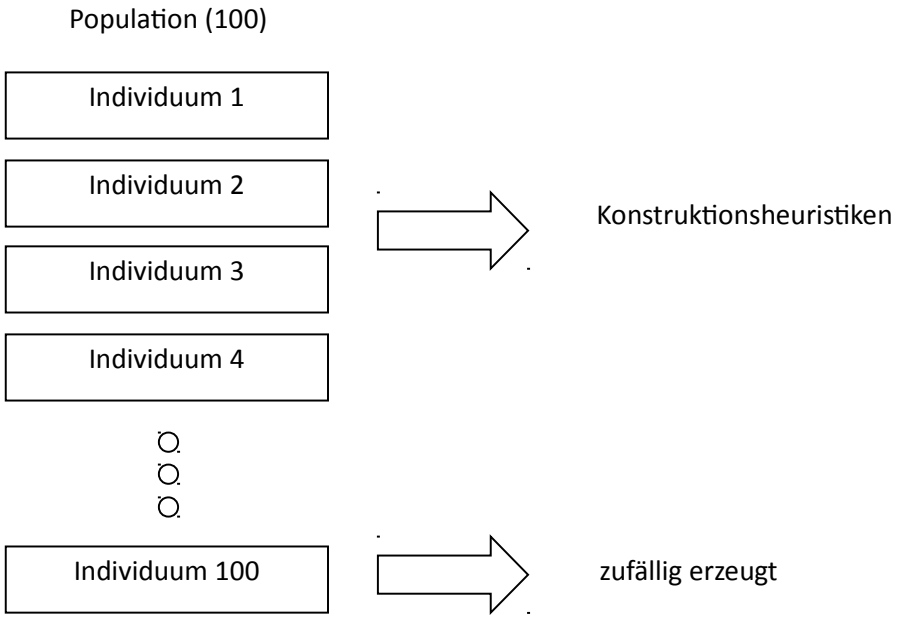


Abbildung 4: Aufbau der Startpopulation

Der Großteil der Population wird dabei zufällig erzeugt, wobei ein ausgewählter Teil sich den Konstruktionsheuristiken bedient. Dabei handelt es sich zum einen um das Nearest-Neighbor- und zum anderen um das Sweep-Verfahren, die in den folgenden Kapiteln kurz vorgestellt werden.

2.3.1 Nearest-Neighbor

Beim Nearest-Neighbor handelt es sich um ein bekanntes heuristisches Eröffnungsverfahren was beispielsweise zur Lösung des „Problems des Handlungsreisenden“ verwendet wird. Da das PDPTW eine Abwandlung von dieser Problemstellung ist, eignet sich diese Heuristik ideal für die Erzeugung von potentiellen Startindividuen. Dabei wird bei dieser Technik von einem beliebigen Startpunkt ausgehend der nächstdichteste Knoten besucht. Dies wird sukzessiv fortgesetzt, bis alle Knoten besucht wurden.

2.3.2 Sweep

Der Sweep-Algorithmus bedient sich einer anderen Technik zum Finden des nächstmöglichen Knotens. Hier wird von einem beliebigen Knoten ausgehend, der Knoten gewählt, der den geringsten Winkel von einer definierten Startkante aufweist. Dies kann sowohl im oder entgegengesetzt dem Uhrzeigersinn durchgeführt werden.

2.4 Individuenauswahl (Selektion)

Die Selektion bestimmt welche Individuen sich paaren dürfen, und erzeugt aus Ihnen die Menge der Nachkommen. Dabei kann die Auswahl entweder komplett zufällig stattfinden oder ins direkte Verhältnis zur Fitness eines Individuums gesetzt werden. Wenn man sich für eine reine fitnessproportionale Selektion entscheidet, so ist ein verhältnismäßig niedriger Selektionsdruck vorhanden, da der genetische Algorithmus verhältnismäßig lange konvergiert. Abhilfe schafft dort eine Ranglistenbasierte Selektion. Bei diesem Verfahren wird die Selektionswahrscheinlichkeit nicht mehr ins direkte Verhältnis zur Fitness gesetzt. Stattdessen werden die unterschiedlichen Individuen nach ihrer Fitness sortiert, so dass die Tour mit dem besten Ergebnis an oberster Stelle der Population steht. Im nächsten Schritt erfolgt die Auswahl der möglichen Kinder für die oberen Elemente der Rangliste mit einer höheren Wahrscheinlichkeit als tieferliegende Individuen.

Zusätzlich wird auch hier auf ein Verfahren zurückgegriffen, welches die möglichen Eltern rein zufällig wählt. Dies hat den Vorteil, dass auch Individuen mit schlechter Fitness als potentieller Elternteil gewählt wird. Dadurch werden ggf. mehr genetische Informationen in die nächste Population übertragen und eine zu frühe Konvergenz, aufgrund von sehr guten Fitnesswerten, verhindert.

Die Kombination der beiden Techniken veranschaulicht die Abbildung 5 auf der nächsten Seite. Dort wird als mögliches Elternpaar zum einen das Individuum 2 aufgrund der Fitness ausgewählt. Zum anderen wird Individuum 100 rein zufällig gewählt. Dadurch besteht der Vorteil, dass das Individuum, trotz schlechter Fitness, ggf. eine gute Kombination hervorbringt.

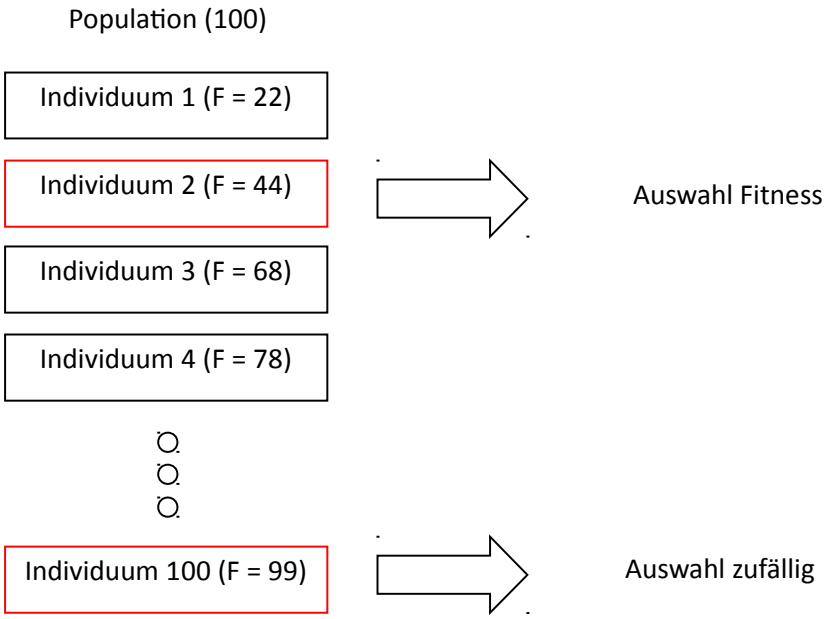


Abbildung 5: Individuenauswahl

2.5 Rekombination (Crossover)

Bei der Rekombination, auch als Crossover bezeichnet, handelt es sich um den Vorgang, aus zwei Elternpaaren neue Individuen zu erzeugen. Dieses Verfahren stellt dabei einen der wichtigsten Suchoperatoren für genetische Algorithmen dar. Dabei werden die Nachkommen aus den Informationen der gekreuzten Elternpaare systematisch zusammengesetzt. Gesteuert wird der Crossover durch eine Wahrscheinlichkeit, die bestimmt ob entweder nur einfache Kopien der Eltern erzeugt werden, oder ein Operator zur Erzeugung der Nachkommen verwendet wird. Dazu werden nachfolgend zwei verschiedene Varianten vorgestellt.

2.5.1 Crossover mit Control-String

Bei dieser Variante der Rekombination wird, auf Grundlage der gewählten Eltern, ein Bit-String gebildet, der letztendlich zu der Entscheidung führt, ob sich der Nachkomme aus einem Teil des ersten oder zweiten Elternteils zusammensetzt. Im Kontext der Fahrzeugeinsatzplanung soll dies die Tabelle 1 verdeutlichen:

Tabelle 1: Crossover mit Control-String

Fahrzeug 1 (Plan 1):	$1+ \rightarrow 4+ \rightarrow 1- \rightarrow 7+ \rightarrow 7- \rightarrow 4-$
Control-String:	1 0 0 0 1 1 0 1 0 1
Fahrzeug 1 (Plan 2):	$1+ \rightarrow 5+ \rightarrow 5- \rightarrow 1-$
Nachkomme:	$1+ \rightarrow 5+ \rightarrow 5- \rightarrow 1- \rightarrow 4+ \rightarrow 7+ \rightarrow 7- \rightarrow 4-$

Oben und unten ist jeweils eine Tour dargestellt, die jeweils einem Elternindividuum entnommen wurde. In der Mitte befindet sich der „Control-String“, wobei die Länge der Summe

der Aufträge (Be- und Entladung) der beiden Eltern entspricht. Nun nimmt ein Element des Strings zufällig eine Ausprägung von „1“ oder „0“ an, wobei die „1“ besagt, dass das Element aus Fahrzeug 1 von Plan 1 für den Nachkommen gewählt wird. Auf gleiche Weise geschieht dies mit der „0“ für Fahrzeug 1 aus Plan 2. Bereits gewählte Aufträge werden dabei nicht mehr berücksichtigt. Letztendlich entsteht der unten in der Tabelle 1 auf der vorherigen Seite aufgeführte Nachkomme.

2.5.2 Crossover nach Falkenauer

Die zweite Variante für einen möglichen Crossover orientiert sich an dem gruppenzentrierten Operator, der von Falkenauer vorgeschlagen wurde. Im Wesentlichen besteht er aus vier Phasen die nachfolgend in der Abbildung 6 dargestellt und erläutert werden.

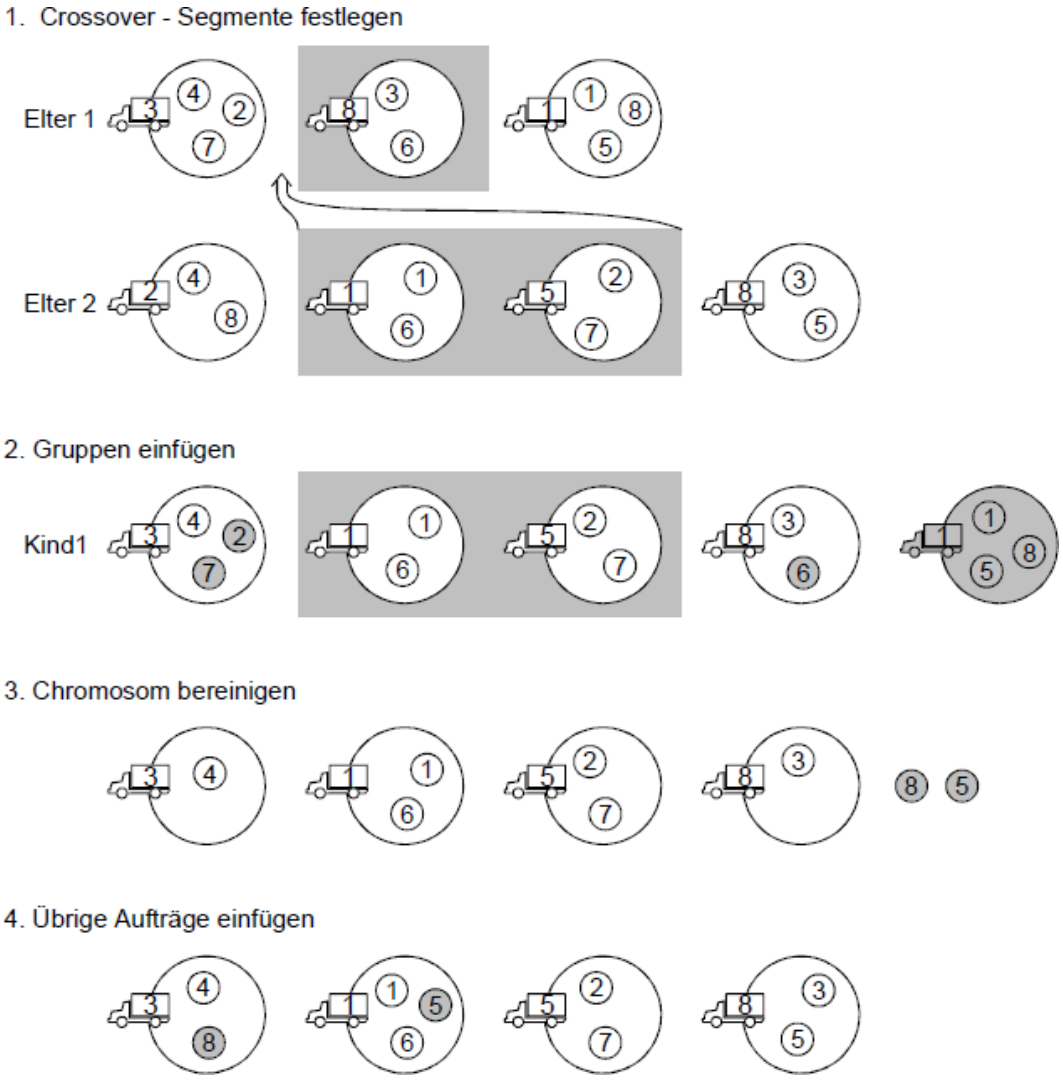


Abbildung 6: CrossOver nach Falkenauer [1]

1. In der ersten Phase werden in jedem der beiden Eltern zufällig je zwei Crossover-Punkte

gebildet, die innerhalb von jedem Elternteil den Anfang und das Ende des Crossover-Segments darstellen.

2. Hier wird das Crossover-Segment des zweiten Elternteils an den ersten Crossover-Punkt des ersten Elternteils angefügt. Die dazugehörigen Touren werden vom Elternteil unmittelbar übernommen, also vorerst ohne Neukonstruktion. Als Folge kann eine Anzahl von Aufträgen mehreren Fahrzeugeinsatzplänen zugeordnet sein. Desweiteren kann es auch vorkommen, dass mehrere Touren vom gleichen Fahrzeug bedient werden.
3. In dieser Phase werden alle auftretenden Probleme aus Punkt 2 aufgelöst. Dabei wird so vorgegangen, dass vorerst alle neuerhaltenden Touren unangetastet bleiben und die Touren des empfangenden Elternteils modifiziert werden, um alle Konflikte zu lösen. Dazu werden alle Touren eliminiert, die auf ein Fahrzeug referenzieren, das neu hinzugefügt wurde (im Beispiel Fahrzeug 1). Falls anschließend noch einzelne Aufträge doppelt vorhanden sind, so werden auch diese aus den Touren des empfangenden Elternteils eliminiert (im Beispiel die Aufträge 2, 6 und 7). Nach Abschluss dieser Operation sind einzelne Aufträge (im Beispiel 5 und 8) noch vorerst keiner Tour zugeordnet.
4. Die verbleibenden Aufträge werden rein zufällig an die vorhandenen Touren vergeben. Dazu kann beispielsweise auch ein komplett neues Fahrzeug allokiert werden. Nach Abschluss von Schritt 4 liegt letztendlich als Ergebnis ein vollständiges Kind vor.

Gegebenenfalls können die Schritte 2-4 mit vertauschten Rollen der Eltern wiederholt werden, falls man ein zweites Kind erzeugen möchte.

2.6 Mutation

Unter der Mutation, im Zusammenhang zum Genetischen Algorithmus, versteht man die Art und Weise, um aus einem Elternteil ein neues Individuum durch zufällige Veränderung zu erzeugen. Diese Änderung wird dabei auch hier durch eine Wahrscheinlichkeit gesteuert, die angibt, ob eine Mutation an einem Individuum stattfindet oder nicht. Sie dienen hauptsächlich dazu, eine gewisse Inhomogenität und Divergenz mit in die Population herein zu bringen, was durch die Rekombination unter Umständen nicht möglich ist. Zusätzlich wird eine frühzeitige Konvergenz des Algorithmus verhindert und der Selektionsdruck abgeschwächt.

Innerhalb dieser Arbeit wurden insgesamt sieben verschiedene Mutationsoperatoren entwickelt, welche mit einer Gewichtung dem genetischen Algorithmus hinzugefügt werden können. Diese werden im nachfolgenden jeweils mit einer Abbildung vorgestellt.

2.6.1 Aufteilung der längsten Route

Dieser Operator überprüft die Länge von jeder Tour innerhalb einer Fahrzeugeinsatzplanung. Die Tour, welche die längste Strecke anhand der gefahrenen Kilometer aufweist (im Beispiel Fahrzeug 5), wird entfernt und auf die verbleibenden Touren aufgeteilt (siehe Abbildung 7 auf der nächsten Seite und Abbildung 8 auf der nächsten Seite). Dabei werden die Aufträge rein zufällig an die anderen Touren vergeben.

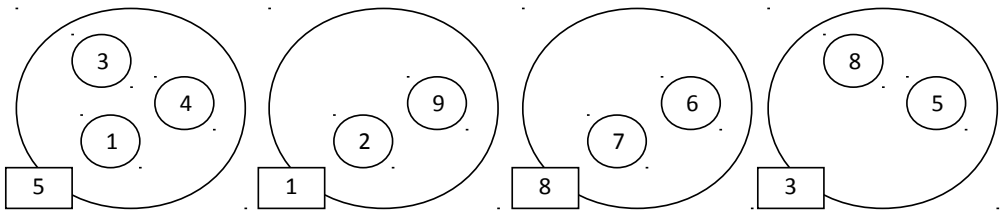


Abbildung 7: Ausgangsindividuum

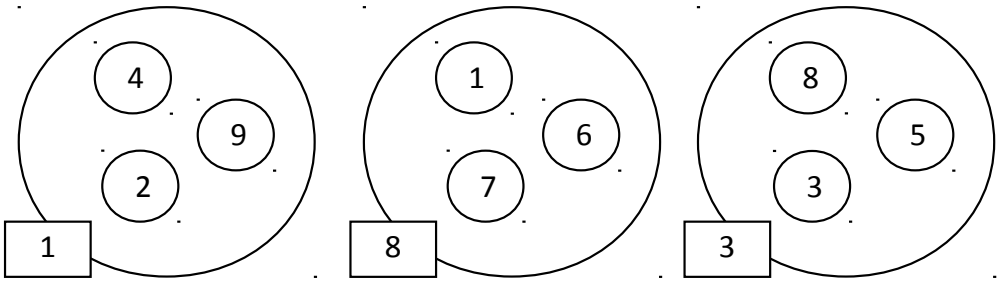


Abbildung 8: Neues Individuum

2.6.2 Aufteilung der kleinsten/zufälligen Tour

Diese beiden Mutationsoperatoren arbeiten auf die gleiche Weise wie unter Abschnitt 2.6.1 auf der vorherigen Seite beschrieben. Dabei wird hier der mögliche Kandidat entweder zufällig oder anhand der kleinsten Tour gewählt. Da die Arbeitsweise nahezu identisch ist, wurde hier auf eine zusätzliche Abbildung verzichtet.

2.6.3 Verschiebung einer/mehrerer Aktion(en) innerhalb einer Tour

Dieser Operator geht noch einen Schritt weiter und verschiebt innerhalb einer Tour eine Aktion. Bei einer Aktion kann es sich um ein Auf- oder Abladen eines bestimmten Auftrags handeln, was über die assoziierte Datenstruktur zu einer Tour gewährleistet wird. (siehe Abbildung 9 auf der nächsten Seite). Ein weiterer Operator verschiebt zusätzlich nicht nur eine Aktion, sondern ganze Subrouten, die in ihren Grenzen zufällig gewählt werden.

Die Repräsentation (links in Abbildung 9 auf der nächsten Seite) bleibt davon unberührt, da es sich um eine reine Manipulation der Datenstruktur handelt. Konkret wird in diesem Beispiel das Abladen des Auftrags 3 mit dem Aufladen des Auftrags 2 vertauscht. Dies kann zur Folge habe, dass falsche Lösungen entstehen, da nach dem Operator keine Überprüfung und ggf. eine Korrektur stattfindet. Der Umgang mit der genannten Problematik wird jedoch im Abschnitt 2.8 auf Seite 16 erläutert.

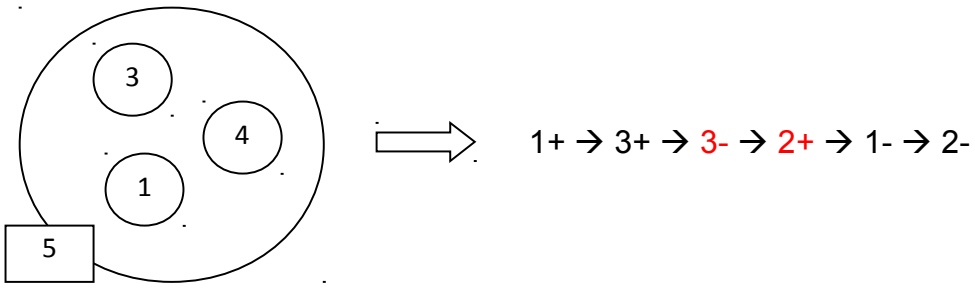


Abbildung 9: Verschiebung einer/mehrerer Aktion(en) innerhalb einer Tour

2.6.4 Verschiebung von Subtouren über eine Tour hinaus

Die bisherigen Operatoren haben lediglich ein Gen des Individuums mutiert. Um eine größere Varianz, unter dem Einfluss der Mutation, zu erschaffen wurde zusätzlich eine Möglichkeit entwickelt, wo ein Tauschen von Subtouren zwischen den unterschiedlichen Genen eines Individuums stattfindet. Dies veranschaulicht die Abbildung 10:

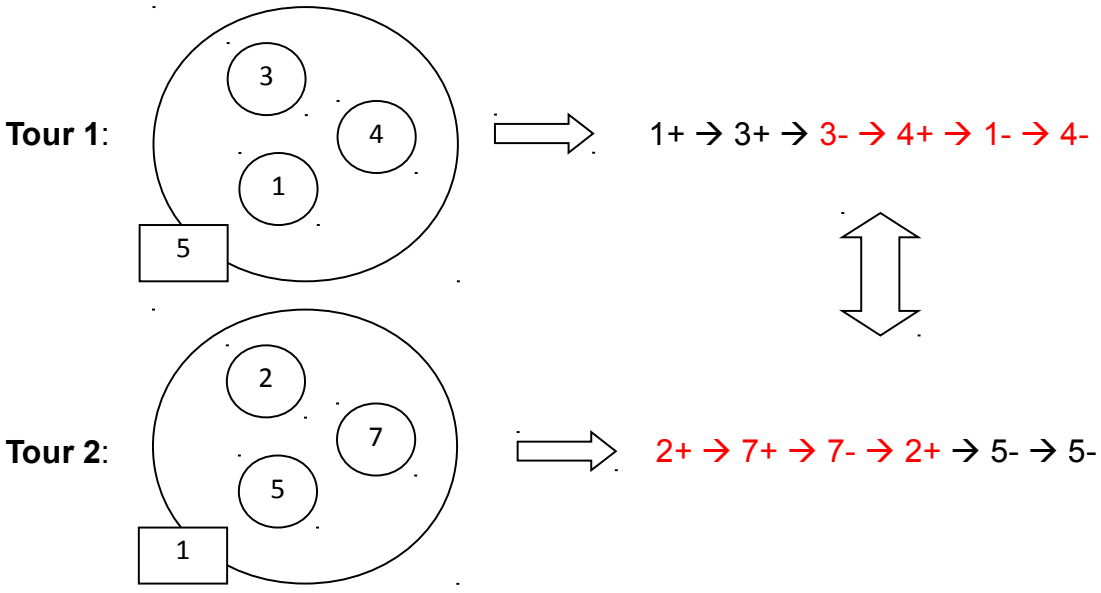


Abbildung 10: Verschiebung von Subtouren über eine Tour hinaus

Dadurch entstehen letztendlich zwei komplett neue Touren innerhalb einer Fahrzeugeinsatzplanung, wobei diese Mutation Einfluss auf die Repräsentationsebene hat, da beispielsweise die Order 2 und 7 von Tour 2 nun von der Tour 1 ausgeführt wird.

2.6.5 Kombination von 2 Touren

Den letzten genetischen Operator für eine Mutation stellt die Kombination von 2 Touren zu einer neuen Tour dar. Das Fahrzeug, das die Aufträge abliefert, wird zufällig aus den beiden

vorhandenen gewählt. Dies wird durch die Abbildung 11 veranschaulicht:

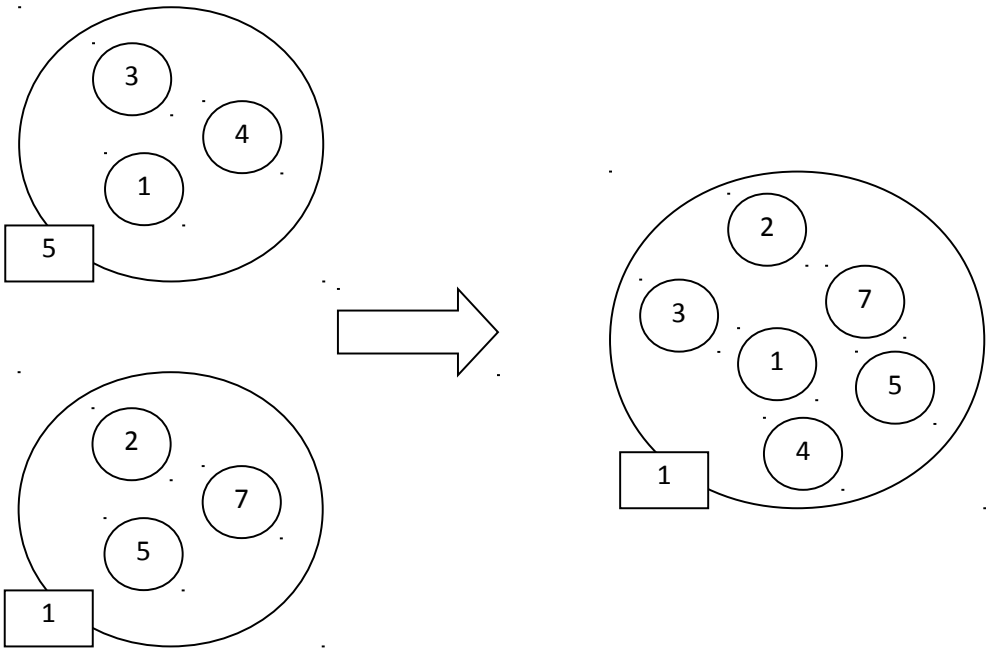


Abbildung 11: Kombination von zwei Touren

2.7 Fitnesstest

Um die Individuen einer Population bewerten zu können, werden sie einem Fitnesstest unterzogen. Der hierbei ermittelte Fitnesswert gibt dann Aufschluss über die Qualität eines Individuums. In der Fahrzeugeinsatzplanung ist die Qualität eines Plans von verschiedenen Komponenten abhängig, beispielsweise von der Länge, von der Anzahl der verwendeten Fahrzeuge oder von der Art und Anzahl der Restriktionsverletzungen. Der in diesem Projekt verwendete Fitnesstest ist modular aufgebaut und sieht für die unterschiedlichen Komponenten jeweils einen separaten Baustein vor. Folgende Bausteine sind in der Applikation vorhanden:

- Länge eines Plans (*LengthFitnessTest*)
- die Anzahl der verwendeten Fahrzeuge für einen Plan (*VehicleFitnessTest*)
- Verletzung der Kapazitätsbedingungen (*CapacityFitnessTest*)
- Verletzung der Zeitfenster der Fahrzeuge (*VehicleMakeSpanFitnessTest*)
- Verletzung der Zeitfenster der Aufträge (*LoadingFitnessTest*)

Je nach Wunsch der Optimierung können diese Bausteine dann zusammengesetzt werden. Hierbei sind alle Kombinationen der genannten Bausteine möglich, auch die Verwendung eines einzelnen Bausteins ist vorgesehen.

Jeder Baustein berechnet hierbei seinen eignen Fitnesswert und je nach Zusammensetzung des Fitnesstests werden diese dann addiert. Ziel ist es die Summe der einzelnen Fitnesswerte zu minimieren. Die Problematik beim Berechnen der Gesamtfitness besteht darin, dass unterschiedliche Einheiten verwendet werden, zum Beispiel Mengenangaben für die Anzahl der Fahrzeuge oder Zeitangaben für Verletzungen der Zeitfenster. Deshalb wird als Basis für die jeweiligen Fitnesswerte die Länge eines Plans verwendet. Je nach Baustein wird die Länge dann mit einem bestimmten Faktor multipliziert. Dieser Faktor ergibt sich aus der Wichtigkeit eines Bausteins und aus einem spezifischen Wert, beispielsweise die Anzahl der verwendeten Fahrzeuge. Die Wichtigkeit eines Bausteins kann von außen angegeben werden. Dadurch ist es einem Anwender möglich, festzulegen welche Komponenten für ihn wichtig sind, zum Beispiel möchte Anwender A die kürzeste Strecke zurücklegen und nimmt dafür Verletzungen der Zeitfenster in Kauf, dagegen möchte Anwender B unbedingt alle Restriktionen einhalten und akzeptiert einen längeren Weg.

Eine Besonderheit bei diesem Fitnesstest stellen die Bausteine mit den Restriktionsverletzungen dar. Bei diesen Bausteinen wird folgendes Verhalten umgesetzt: mehrere kleine Verletzungen sind nicht so schlimm wie eine große, beispielsweise ist die Qualität eines Plans besser, wenn zwei Fahrzeuge um jeweils eine halbe Stunde ihr Zeitfenster verfehlen als wenn ein Fahrzeug um eine Stunde sein Zeitfenster verfehlt. Dafür wird eine exponentielle Bestrafung eingesetzt.

2.8 Bildung der Nachfolgegeneration

Über die Mechanismen der Selektion hinaus, die in Abschnitt 2.4 auf Seite 9 definiert wurden, wird sowohl die komplette Ausgangspopulation, als auch die neue Population, für die Nachfolgepopulation berücksichtigt. Dazu findet auch an der Neuen wiederum eine Sortierung statt. Zum einen werden alle invaliden Lösungen, d.h. Lösungen, wo beispielsweise ein Ent-

vor dem Beladen erfolgt, ans Ende der Population verschoben. Danach erfolgt eine Sortierung anhand der Fitness. Demnach werden beiden Populationen zu einer Nachfolgepopulation der definierten Menge (im Beispiel 100) zusammengefasst. Zur Veranschaulichung dient die Abbildung 12:

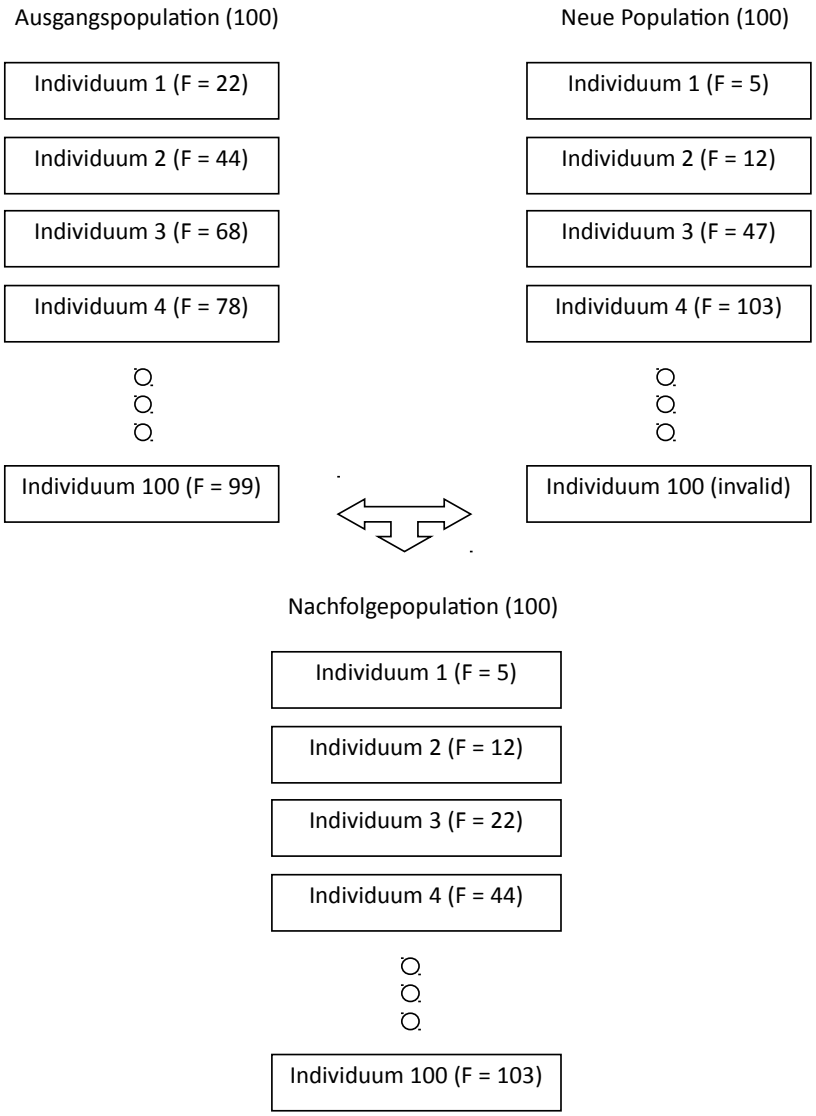


Abbildung 12: Bildung der Nachfolgegeneration

Das Resultat ist die neue Nachfolgepopulation, die wieder in den genetischen Algorithmus einfließt, bis eine Abbruchbedingung erreicht wird. Diese Verfahrenstechnik hat den Vorteil, dass bereits potentiell gute Lösungen, die in der Ausgangspopulation vorhanden waren, erhalten bleiben und nicht durch die neugebildete Population überschrieben werden.

2.9 Abbruchbedingung

Als letzten entscheidenden Punkt muss beim genetischen Algorithmus eine Abbruchbedingung definiert werden, wann die Suche abgeschlossen ist. Dazu wurden zwei Varianten entwickelt:

1. Der genetische Algorithmus wird solange durchlaufen, bis eine definierte Anzahl von Iterationen (z.B. 5000) erreicht wurde.
2. Die Fitness der besten Lösung entspricht einem definierten Abstand von der Durchschnittsfitness von allen Lösungen zu dieser Lösung

Die erste Bedingung wird für den Fall benötigt, dass der genetische Algorithmus nicht unendlich oft durchlaufen wird. Eine Iteration entspricht dabei dem Weg von einer Ausgangspopulation bis hin zu Neubildung der Nachfolgegeneration.

Die zweite Bedingung ist die eigentlich entscheidende, da der Algorithmus solange laufen soll, bis keine bessere Lösung mehr gefunden wird. Ist dies der Fall, so kann die genetische Suche abgeschlossen werden und das erste Individuum in der Population entspricht der optimalen Lösung für eine gegebene Konfiguration.

3 Softwarearchitektur

3.1 Systemdesign

Das Systemdesign setzt sich im wesentlichen aus den folgenden vier Modulen zusammen:

1. Konfigurationsgenerierung
2. Erzeugen der Startpopulation durch Konstruktionsverfahren
3. Genetisches Optimierungsverfahren
4. Ergebnisvisualisierung

Desweiteren sind zwei Datenmodelle definiert:

1. Konfiguration-Datenmodell
2. Ergebnis-Datenmodell

Das Aktivitätsdiagramm aus Abbildung 13 zeigt den groben Ablauf der Anwendung.

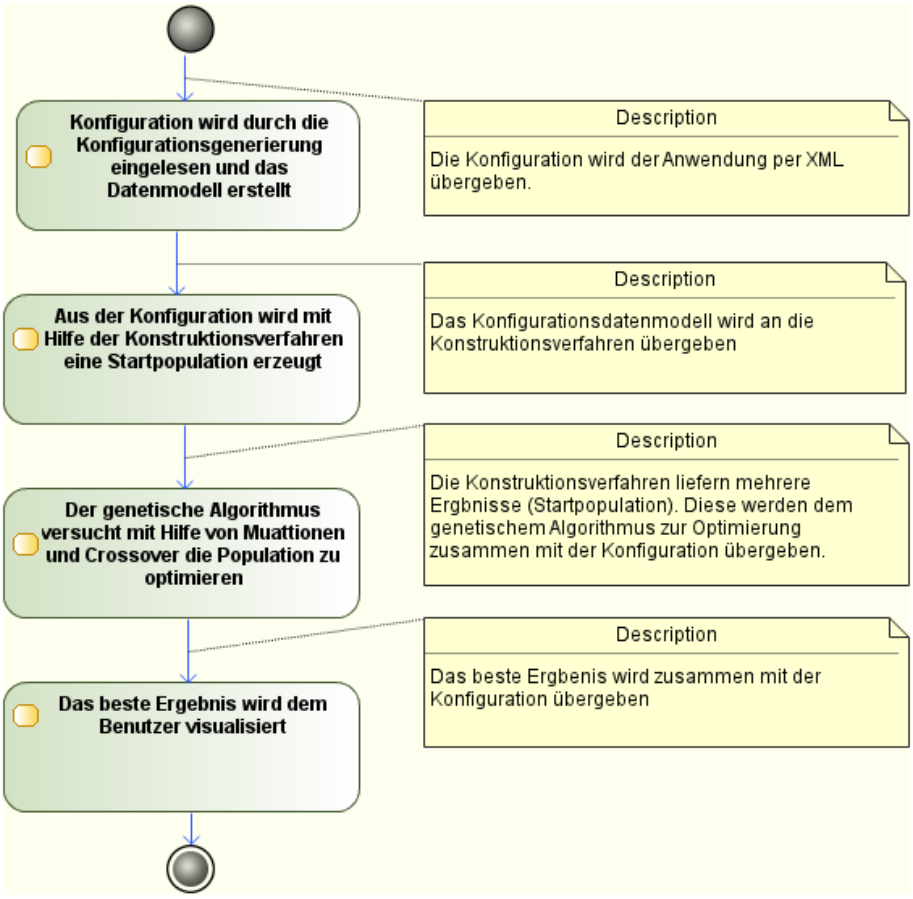


Abbildung 13: Aktivitätsdiagramm: Grobe Ablauf der Anwendung

3.2 Konfigurations-Datenmodell

Das Konfiguration-Datenmodell beschreibt die Konfiguration, die vom Anwender zur Verfügung gestellt wird (siehe Abschnitt 1.2 auf Seite 4). Sie umfasst Stationen, Fahrzeuge, Produkte, Aufträge und Zeitfenster (siehe Abbildung 14).

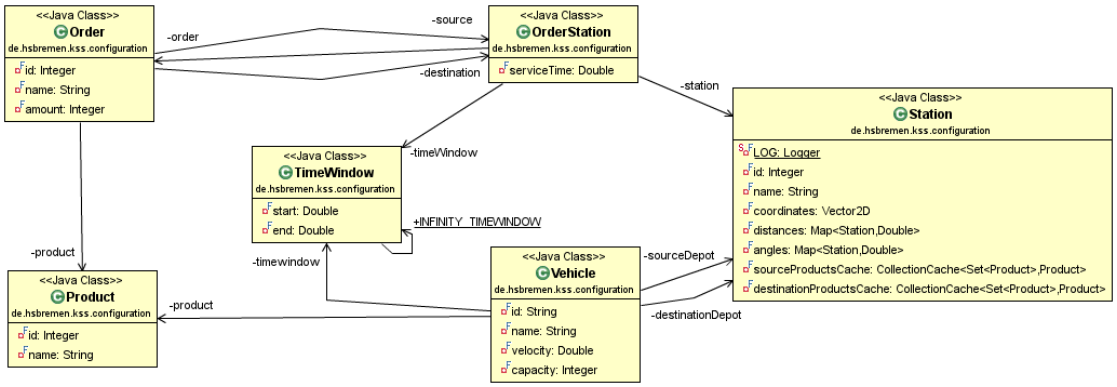


Abbildung 14: Repräsentation eines Plans mit Touren und Aktionen

3.3 Ergebnis-Datenmodell

Das Ergebnis wird mit Hilfe des Ergebnis-Datenmodells repräsentiert (siehe Abbildung 15 auf der nächsten Seite). Ein Ergebnis ist ein Plan, der mehrere Touren enthält, die wiederum mehrere Aktionen enthalten. Dabei wird eine Tour von einem Fahrzeug durchgeführt.

Eine Aktion hat drei grundlegende Eigenschaften. Sie wird an einer Station durchgeführt, sie hat eine Ausführungszeit und muss in einem bestimmten Zeitfenster durchgeführt werden. Die Fahrten zwischen Stationen werden nicht mit einer Aktion definiert. Sie ergeben sich implizit aus zwei aufeinander folgenden Aktionen. Die folgenden Aktionen sind definiert:

FromDepotAction Das Fahrzeug beginnt seine Tour im Depot (Station).

OrderLoadAction Das Fahrzeug läd an einer Station einen Auftrag auf.

WaitingAction Das Fahrzeug wartet an einer Station.

OrderUnloadAction Das Fahrzeug läd an einer Station einen Auftrag ab.

ToDepotAction Das Fahrzeug beendet seine Tour am Depot (Station).

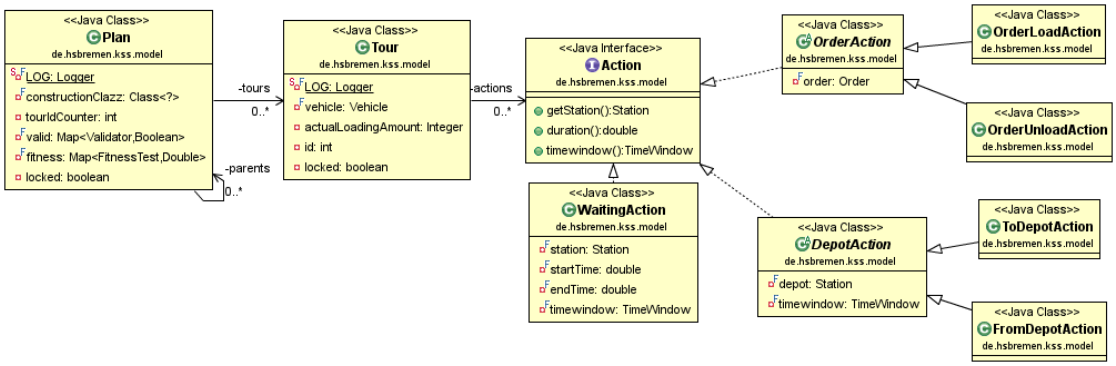


Abbildung 15: Repräsentation eines Plans mit Touren und Aktionen

3.4 Konfigurationsgenerierung

Die Konfigurationsgenerierung-Modul erzeugt eine Instanz des Konfigurations-Datenmodells. Diese Instanz ist Basis für alle weiteren Berechnungen in der Anwendung. Das Datenmodell kann aus unterschiedlichen Quellen generiert werden.

3.5 Erzeugen der Startpopulation

Wie in Abschnitt 2.3 auf Seite 8 beschrieben, ist es notwendig eine Startpopulation zu erzeugen. Für das Erzeugen der Population ist die Klasse `PopulationGeneratorImpl` zuständig. Ihr werden beliebige Konstruktionsverfahren (`Construction`), eine Konfiguration und die Größe der Population übergeben. Der Generator erzeugt dann automatisch die Startpopulation.

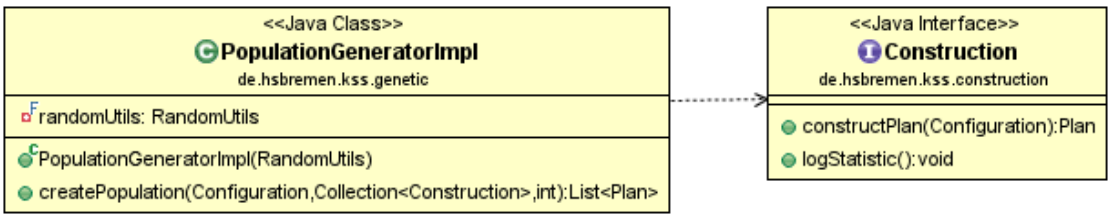


Abbildung 16: Erzeugung der Startpopulation

3.6 Genetischer Algorithmus

Die Softwarearchitektur des genetischen Algorithmus ist sehr modular aufgebaut. Für die in Abschnitt 2 auf Seite 6 beschriebenen Merkmale (Individuenauswahl, Rekombination, Mutation, Fitnesstest, Bildung der Nachfolgegenerqation und Abbruchbedingung) sind Interfaces definiert. Die Klasse `GeneticAlgorithmImpl` ist nur zuständig für die Steuerung der einzelnen Module (Merkmale). Die Abbildung 17 auf der nächsten Seite zeigt anhand eines

Klassendiagramms das modulare Konzept.

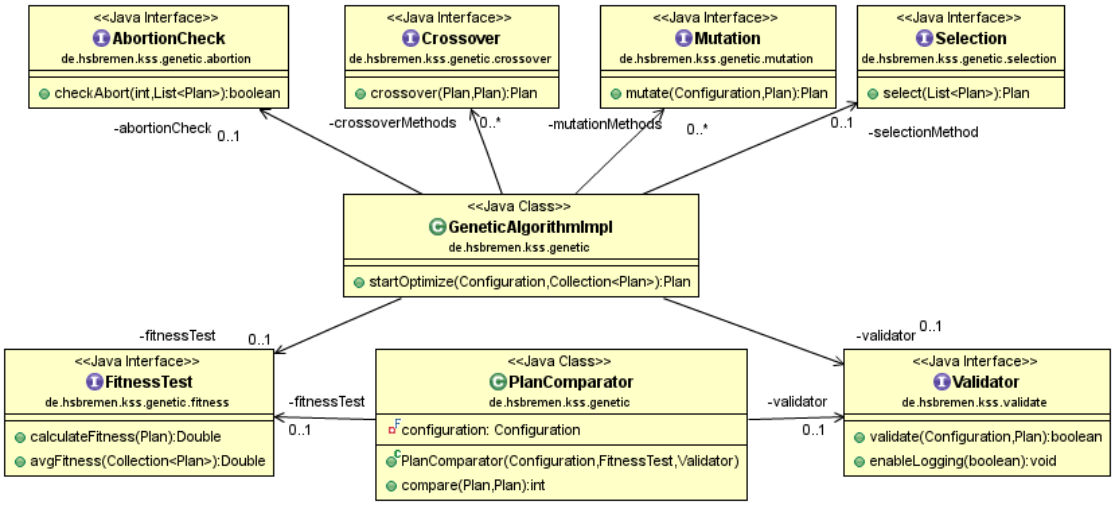


Abbildung 17: Klassendiagramm vom genetischen Algorithmus

4 Umsetzung

4.1 Entwicklung

Die Anwendung wurde mit Java 7 entwickelt. Als Entwicklungsumgebung wurde Eclipse in Verbindung mit Maven eingesetzt.

4.2 Verwendete Bibliotheken

Im Rahmen der Entwicklung wurden diverse OpenSource-Bibliotheken eingeführt, um die Entwicklung zu beschleunigen und die Qualität der Software zu steigern. Im folgenden werden die verwendeten Bibliotheken beschrieben:

Apache Commons Math (Version 3.2, Apache License 2.0) [13] Stellt mathematische Funktionen zur Verfügung. Wird unteranderem für die Berechnung der Entfernungen verwendet.

Apache Commons Lang (Version 3.3.1, Apache License 2.0) [12]

Apache Commons Collections (Version 4.0, Apache License 2.0) [11] Erweitert das Java-Collections-Framework.

Joda-Time (Version 2.3, Apache License 2.0) [6]

JUnit (Version 4.11, Eclipse Public License) [7] JUnit ist die Standard-Bibliothek für Unit-Tests unter Java.

Hamcrest (Version 1.3, BSD 3-Clause) [4] Mit Hamcrest ist es möglich bei Tests „sprechendere“ Ausdrücke zu formulieren.

EasyMock (Version 3.2, Apache License 2.0) [2] EasyMock ermöglicht ein einfaches Erstellen von Mock-Objekten für den Unit-Test.

Simple Logging Facade for Java (SL4J) (Version 1.7.6, MIT license) [10] SLF4J ist eine Logging-Schnittstelle und wird zur Ausgabe auf der Konsole verwendet.

Logback (Version 1.1.1, Eclipse Public License v1.0 und LGPL 2.1) [9] Wird als Implementierung für SLF4J eingesetzt.

Google Guave (Version 17.0, Apache License 2.0) [3] Guave ist eine Sammlung von Softwarebibliotheken. Es wird ausschließlich der EventBus für die Kommunikation zwischen den Softwarekomponenten verwendet.

JFreeChart (Version 1.0.17, LPGL) [8] JFreeChart wird zur Darstellung der Graphen verwendet.

4.3 Populationssortierung

Das Sortieren der Population geschieht mit Hilfe des `PlanComparators` (Abbildung 18). Dieser sortiert zuerst mit Hilfe des `Validators` die gültigen und ungültigen Individuen und im Anschluss die beiden Gruppen nach ihrem Fitnesswert. Somit stehen gültige Individuen immer über ungültigen Individuen.

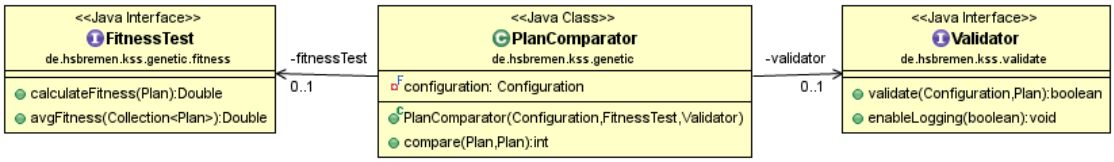


Abbildung 18: Sortierung der Population

4.4 Validation

Es wurden zwei Validatoren (siehe Abbildung 19) implementiert. Der `FullValidator` prüft alle Restriktionen (vgl. Abschnitt 1.3 auf Seite 4). Er wird am Ende der Optimierung verwendet um das beste Ergebnis zu überprüfen. Für das Sortieren der Population wird hingegen der `RightOrderValidatorImpl` verwendet. Er führt eine schwächere Prüfung durch und ignoriert unteranderem die Verletzungen von Zeitfenstern.

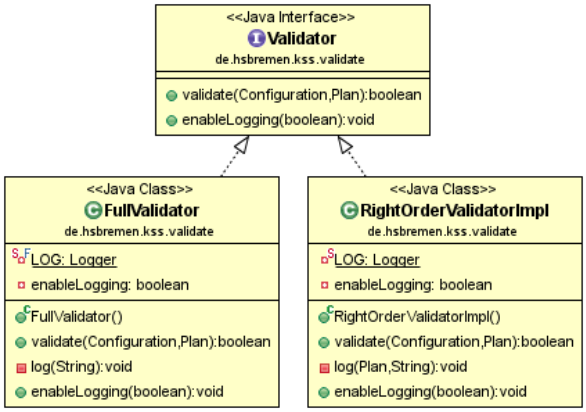


Abbildung 19: Implementierte Validatoren

4.5 Fitnesstest

Abbildung 20 zeigt die implementierten Fitness Tests. Sie lassen sich dynamisch zusammenschalten und konfigurieren. Bei der Implementierung wurde unter anderem darauf geachtet, dass eine Zeitfenster-Verletzung von einer Stunde schlechter bewertet wird, als zwölf Verletzungen von jeweils fünf Minuten.

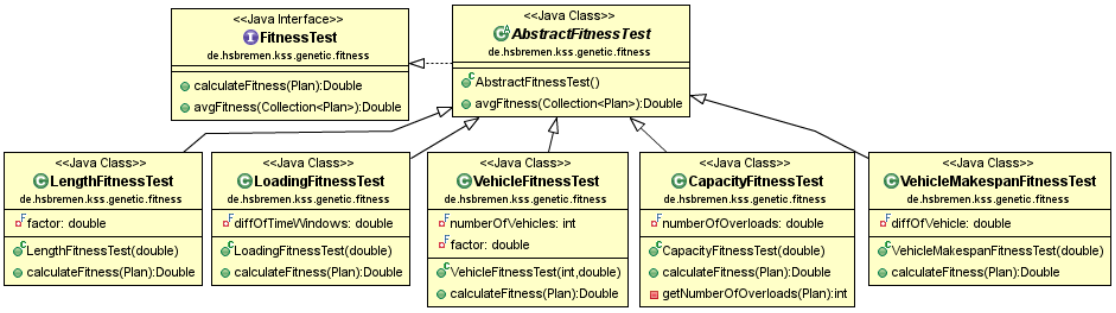


Abbildung 20: Implementierte Fitness Tests

4.6 Selektion

Es wurden drei unterschiedliche Selektionsverfahren entwickelt (siehe Abbildung 21). Die `RandomSelektion` wählt zufällig ein Individuum aus der Population aus. Die `OnyTheBestSelection` wählt aus den Besten (konfigurierbar) zufällig ein Individuum aus. Die `LinearDistributionSelection` wählt die Individuen mit einem höheren Rang mit einer höheren Wahrscheinlichkeit aus. Die Abbildung 22 auf der nächsten Seite verdeutlicht die unterschiedlichen Selektionsarten anhand von drei Graphen.

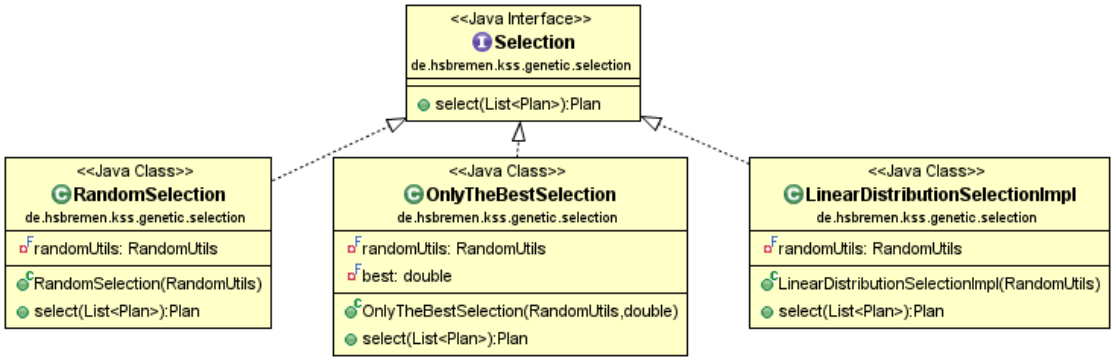
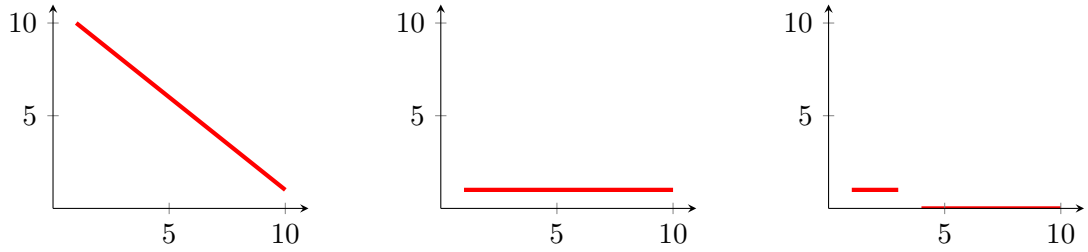


Abbildung 21: Implementierte Selektionen



(a) LinearSelectionImpl (b) RandomSelektion (c) OnlyTheBestSelection

Abbildung 22: Selektion bei einer Populationsgröße von 10 Individuen (X: Individuenrang, Y: Wahrscheinlichkeit)

4.7 Mutation

Im Klassendiagramm (Abbildung 23 auf der nächsten Seite) sind alle implementierten Mutationen dargestellt. Instanzen dieser Mutationen werden dem genetischen Algorithmus übergeben. Dieser wählt dabei zufällig eine Mutation aus und wendet sie auf ein Individuum an. Es können auch „null“-Mutationen übergeben werden. Dies bedeutet, dass das Individuum nicht mutiert wird. Hierüber lässt sich also die Mutationsrate steuern.

Die Mutationen sind so implementiert, dass sie möglichst nur gültige Lösungen liefern (Kein Abladen vor dem Aufladen; nur Produkte laden, die auch mit dem Fahrzeug transportiert werden können).

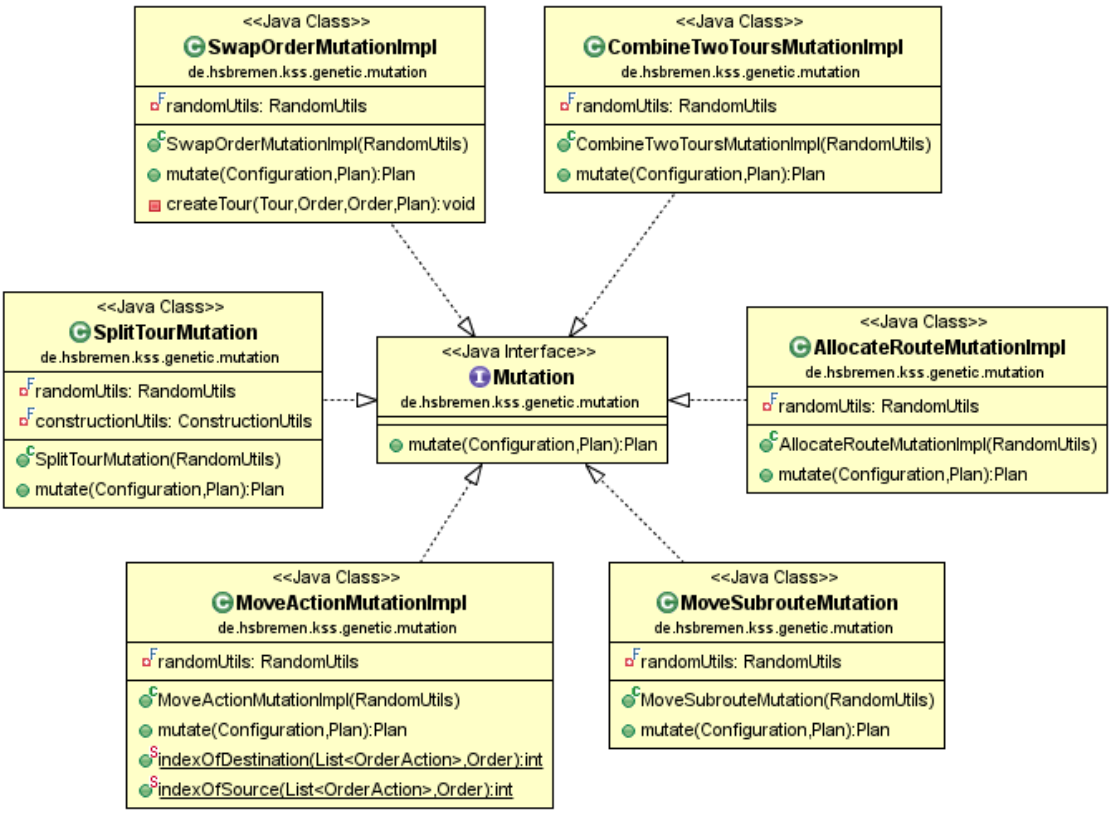


Abbildung 23: Implementierte Mutationsalgorithmen

4.8 Grafische Oberfläche

Die drei folgenden Abbildungen zeigen die grafische Oberfläche. Sie wird beider jeder Iteration aktualisiert und zeigt somit immer den aktuellen Stand.

Abbildung 24 auf der nächsten Seite zeigt eine Deutschlandkarte auf welcher das beste Individuum abgebildet wird. Die unterschiedlichen Farben symbolisieren unterschiedliche Touren.

Abbildung 25 auf der nächsten Seite zeigt die relativen Aufbau des Fitnesswertes. Aus diesem Graphen lässt sich entnehmen, ob es noch Verletzungen von Restriktionen gibt und was der genetische Algorithmus zur Zeit optimiert.

Abbildung 26 auf Seite 29 zeigt die Entwicklung des Fitness Wertes und der Gesamtstreckenlänge. Der Fitness Wert sinkt stetig. Die Streckenlänge kann sich auch wieder verschlechtern, wenn hierdurch andere Verletzung vermieden werden.

Abbildung 24: Das beste Ergebnis wird auf einer Deutschlandkarte dargestellt

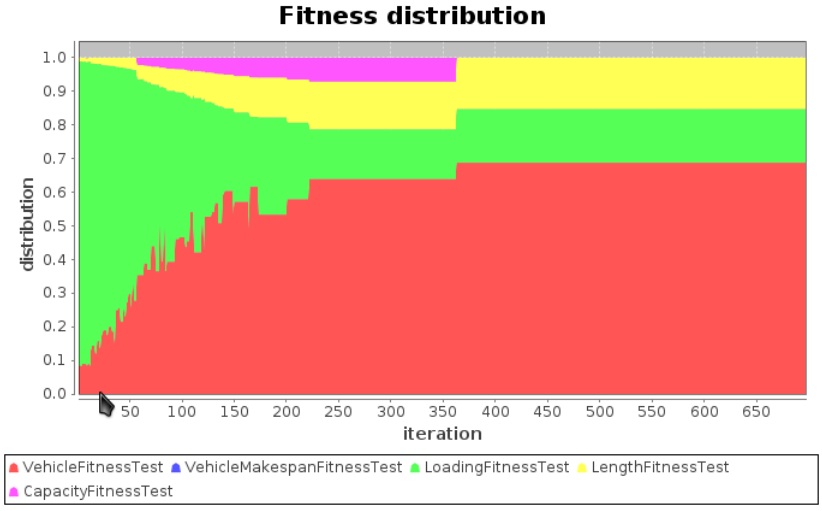


Abbildung 25: Verteilung des Fitnesswertes

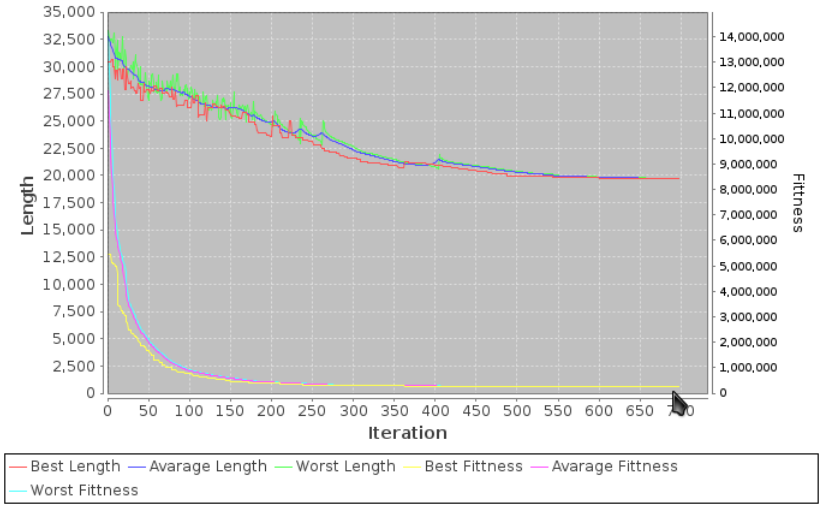


Abbildung 26: Fitness Wert und Länge des besten und schlechtesten Individuum + Durchschnitt über die gesamt Population

5 Experimente

In den Experimenten wird der genetische Algorithmus auf seine Funktionsweise überprüft. Außerdem soll der Einfluss der Änderungen der Komponenten auf das Ergebnis des Algorithmus untersucht werden. Zuerst wird hierfür eine Zielsetzung definiert. Darauf folgen dann eine Beschreibung der verwendeten Konfiguration für die Experimente, die Durchführung sowie eine Auswertung zum Abschluss dieses Kapitels.

5.1 Zielsetzung

Mit Hilfe der Experimente soll festgestellt werden, ob der genetische Algorithmus das erwartete Verhalten, sukzessive Verbesserung der Individuen, aufweist. Wie bereits in den vorherigen Kapiteln beschrieben, besteht der genetische Algorithmus aus verschiedenen Komponenten. Die Funktionsweise einzelner Komponenten wurde bereits mit Unittests nachgewiesen. In den nachfolgenden Experimenten soll nun das Zusammenspiel dieser Komponenten beobachtet werden sowie deren Auswirkung auf das Ergebnis des genetischen Algorithmus. Dabei spielen folgende Bestandteile des genetischen Algorithmus eine zentrale Rolle:

- Mutation
- Selektion
- Crossover
- Fitnesstest

An diesen Bestandteilen werden während der Experimente Veränderungen vorgenommen und die Auswirkungen auf das Ergebnis untersucht.

Um die Qualität eines Ergebnisses beschreiben zu können, müssen vor den Experimenten Indikatoren festgelegt werden. Für die hier durchgeführten Experimente werden folgende Indikatoren verwendet:

- Länge eines Plans
- Anzahl der Fahrzeuge
- Anzahl der Iterationen
- Anzahl und Art der Restriktionsverletzungen

Als Indikator hätte auch der Fitnesswert dienen können, allerdings kann sich darunter wenig vorgestellt werden. Die Laufzeit des Algorithmus dient hierbei nicht als Indikator, da diese von System zu System variieren kann. Grundsätzlich lässt sich aber sagen, dass der genetische Algorithmus in einer angemessenen Zeit ein Ergebnis liefert. Bei den Experimenten gibt es eine Vielzahl von Parametern, die beim Ausführen der Applikation angegeben und somit verändert werden können, beispielsweise die Populationsgröße, Anzahl der Produkte, Generierung der Startpopulation, usw.. Um nicht eine endlose Zahl an Experimenten durchzuführen, ist es wichtig sich vorher zu überlegen, welche dieser Parameter verändert werden können und welche gleich bleiben sollen.

5.2 Konfiguration

Für die durchzuführenden Experimente soll die Konfiguration simpel gehalten werden, um die zuvor genannten Ziele zu erreichen und die Experimente in einem überschaubaren Zeitraum durchführen zu können. In Tabelle 2 ist die verwendete Konfiguration aufgelistet. Die Namen der Mutationen und der Fitnessbausteine sind dem Quellcode entnommen.

Tabelle 2: Konfiguration

Paramter	**Wert**
Seeding	0
Anzahl der Produkte	1
Anzahl der Fahrzeuge	10
Anzahl der Aufträge	50
Zeitfenster der Fahrzeuge	ja
Zeitfenster der Aufträge	ja
Startpopulation	zufällig
Populationsgröße	200
MoveActionMutation	20
MoveSubrouteMutation	10
SwapOrderMutation	10
AllocateRouteMutation	6
CombineTwoToursMutation	1
SplitTourMutation	1
NullMutation	1, entspricht einer Mutationsrate von 98%
Crossover	ja/nein
Selektion	zufällig/linear
LengthFitness	1
VehicleFitness	1; 2
CapacityFitnessTest	5
VehicleMakeSpanFitnessTest	1,2
LoadingFitnessTest	1,2
Anordnung der Stationen	Deutschland-Karte mit gleichmäßiger Verteilung
Abbruchkriterium	Die Differenz zwischen der besten und der durchschnittlichen Population beträgt weniger als 0,0001 oder 2000 Iterationen sind erreicht.

5.3 Durchführung

Die Durchführung der Experimente setzt sich aus verschiedenen Einzelexperimente zusammen. Zunächst wird ein allgemeines Experiment durchgeführt. Als Grundlage für den Nachweis der Funktionalität des genetischen Algorithmus dient hierbei eine Karte (siehe Abschnitt 4.8 auf Seite 27) mit den Station und den Touren eines Plans. Im Anschluss werden dann spezielle Experimente durchgeführt, welche die Auswirkungen von Mutationen, Selektion, Crossover und Fitnessbausteinen auf das Ergebnis des genetischen Algorithmus zeigen sollen.

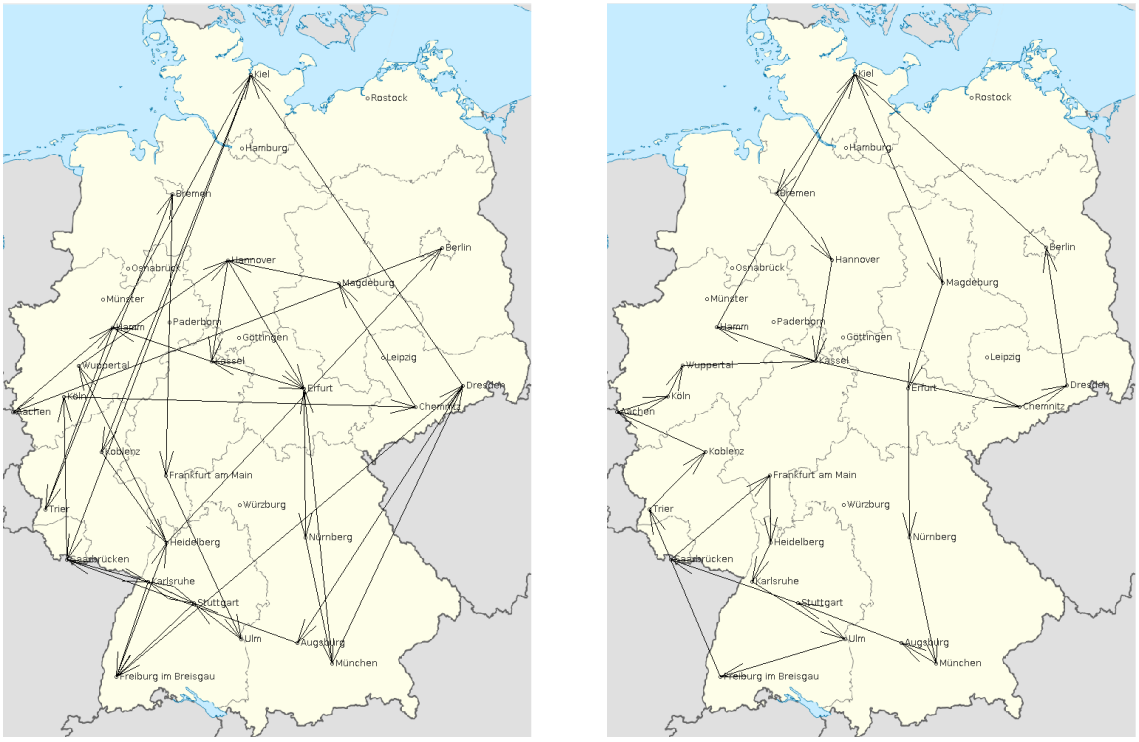
5.3.1 Allgemein

Für dieses allgemeine Experiment wurde die in Abschnitt 5.2 auf der vorherigen Seite aufgelistete Konfiguration minimal verändert. Anstatt der erwähnten zehn Fahrzeuge wird diesmal nur ein Fahrzeug verwendet und es werden nur 20 Aufträge bearbeitet. Dies dient der besseren Übersicht auf den verwendeten Karten. In Abbildung 27(a) auf der nächsten Seite ist zu sehen, wie die Touren des Plans nach der ersten Iteration verlaufen. Es gibt sehr viele Überschneidungen der einzelnen Touren und es sind sehr lange Touren vorhanden. Insgesamt wirkt die Verteilung sehr unstrukturiert. In Abbildung 27(b) auf der nächsten Seite ist dann zu sehen, wie die Touren nach 246 Iterationen angeordnet sind. Hierbei ist ein deutlicher Unterschied zu Abbildung 27(a) auf der nächsten Seite sichtbar. Es sind wesentlich weniger Überschneidungen zu erkennen. Außerdem werden kaum noch lange Touren absolviert. Insgesamt wirkt die Anordnung deutlich strukturierter. Das sind typische Merkmale für die Optimierung in der Fahrzeugeinsatzplanung mit einem genetischen Algorithmus und können daher als Beweis für die Funktionalität des Algorithmus verwendet werden. Allerdings ist auch nach 246 Iterationen noch nicht die beste Lösung erreicht worden, da immer noch ein paar Überschneidungen vorhanden sind. Der genetische Algorithmus kann bzw. sollte also noch verbessert werden.

5.3.2 einzelne Mutationen

Mit diesem Experiment soll die Auswirkung einer einzelnen Mutation auf das Ergebnis des genetischen Algorithmus untersucht werden. Dazu wird immer nur eine Mutationsart für den genetischen Algorithmus verwendet. Für dieses und alle nachfolgenden Experimente gilt wieder die in Kapitel Abschnitt 5.2 auf der vorherigen Seite beschriebene Konfiguration. Um die Mutationen bewerten zu können, wurde vor den Experimenten ein Referenzwert aufgenommen. Bei diesem Wert handelt es sich um die beste Lösung aus der zufällig generierten Startpopulation. Dieser Wert wird nicht nur für dieses Experiment, sondern auch für alle nachfolgenden als Referenz verwendet. Der beste Plan aus der Startpopulation hat die folgenden Eigenschaften:

- Länge: 28622
- Anzahl Fahrzeuge: 10
- Verletzungen: 30 Zeitfenster (TW) können nicht eingehalten werden und zweimal ist ein Fahrzeug überladen (O).



(a) Iteration 1

(b) Iteration 246

Abbildung 27: Vergleich zwischen Iteration 1 und 246

In Klammern stehen die Abkürzungen, welche in den nachfolgenden Tabellen für die Restriktionsverletzungen verwendet werden.

Bei diesem Experiment werden alle Fitnessbausteine, kein Crossover und die zufällige Selektion verwendet. Außerdem wird die NullMutation nicht separat untersucht, da diese nur für die Angabe der Mutationsrate zuständig ist. In Tabelle 3 sind die Ergbenisse dieses Experimentes dargestellt.

Tabelle 3: einzelne Mutationen

Mutation	**Länge**	**Iteration**	**Autos**	**Verletzung**
MoveActionMutation	20697	336	10	12 TW, 1 O
MoveSubrouteMutation	19327	664	10	9 TW, 1 O
SwapOrderMutation	17336	2000	10	5 TW, 1 O
AllocateRouteMutation	25517	2000	10	38 TW, 1 O
CombineTwoToursMutation	28622	2000	10	30 TW, 2 O
SplitTourMutation	28622	9	10	30 TW, 2 O

Die Auswirkungen der einzelnen Mutationen auf das Resultat des genetischen Algorithmus unterscheiden sich deutlich, zum Beispiel erreicht man mit der *SwapOrderMutation* schon ein

relativ gutes Endergebnis. Im Vergleich zum Startwert konnte sowohl die Länge als auch die Anzahl der Verletzungen deutlich reduziert werden. Dagegen können die Mutationen *CombineTwoToursMutation* und *SplitTourMutation* keine Verbesserungen des Ausgangswertes herbeiführen. Allerdings wird es mit keiner Mutation geschafft die Verletzungen der Restriktionen komplett zu beseitigen.

5.3.3 zusammengesetzte Mutationen

Als nächstes wird das Zusammenspiel der Mutationen untersucht. Im Gegensatz zum ersten Experiment werden die Mutationen nun nacheinander hinzugefügt und zusammen für den genetischen Algorithmus verwendet. Die Mutationen werden von oben nach unten hintereinander hinzugefügt.

Wie bereits im ersten Experiment werden alle Fitnessbausteine, kein Crossover und die zufällige Selektion verwendet. Dieses Mal wird auch die NullMutation verwendet. Die Ergebnisse sind in Tabelle 4 zu sehen.

Tabelle 4: zusammengesetzte Mutationen

Mutation	**Länge**	**Iteration**	**Autos**	**Verletzung**
MoveActionMutation	20697	336	10	12 TW, 1 O
MoveSubrouteMutation	19873	385	10	11 TW, 1 O
SwapOrderMutation	16154	681	10	
AllocateRouteMutation	15884	681	10	1 O
CombineTwoToursMutation	16071	1153	10	
SplitTourMutation	15546	871	10	1 O
NullMutation	15957	933	10	

Grundsätzlich lässt sich sagen: je mehr Mutationen verwendet werden, desto besser ist das Ergebnis des genetischen Algorithmus. Die Länge eines Plans wird kleiner und die Verletzungen werden deutlich reduziert, teilweise sind gar keine mehr vorhanden. Beim Hinzufügen der *CombineTwoToursMutation* wird zwar die Länge wieder größer, dafür verschwinden aber die Verletzungen. Das gleiche gilt für die *NullMutation*.

5.3.4 gleichverteilte Mutationen

Auch in diesem Experiment werden die Mutationen nacheinander hinzugefügt, diesmal haben sie allerdings alle die gleiche Wahrscheinlichkeit. Mit diesem Experiment soll nachgewiesen werden, dass die Verteilung der Mutationen Einfluss auf das Ergebnis des genetischen Algorithmus hat. Als Vergleichswerte dienen die Ergebnisse aus Tabelle 4. Es werden wieder alle Fitnessbausteine, zufällige Selektion und kein Crossover verwendet. In Tabelle 5 auf der nächsten Seite sind die Ergebnisse aufgelistet. Nachdem Hinzufügen der NullMutation beträgt die Mutationsrate 86%.

Durch das Verändern der Wahrscheinlichkeiten der einzelnen Mutationen werden auch an-

Tabelle 5: gleichverteilte Mutationen

Mutation	**Länge**	**Iteration**	**Autos**	**Verletzung**
MoveActionMutation	20697	336	10	12 TW, 1 O
MoveSubrouteMutation	18762	572	10	8 TW, 1 O
SwapOrderMutation	16347	673	10	1 TW
AllocateRouteMutation	14390	1596	10	
CombineTwoToursMutation	15360	2000	10	1 O
SplitTourMutation	15283	745	10	
NullMutation	16418	626	10	1 TW

dere Ergebnisse erzielt. Diese liegen größtenteils in den Bereichen der Ergebnisse mit der Mutationsverteilung aus der Konfiguration. Allerdings wird hier beim Hinzufügen der *AllocateRouteMutation* ein absoluter Bestwert erzielt, der kürzeste Plan ohne einen einzigen Verstoß gegen die Restriktionen. Dagegen wird beim Hinzufügen der *NullMutation* das Ergebnis schlechter als bei seinem Referenzwert. Das deutet darauf hin, dass die Mutationsrate möglichst hoch liegen sollte.

5.3.5 Selektion

In diesem Experiment soll der Unterschied zwischen zufälliger und linearer Selektion untersucht werden. Wie bereits im vorherigen Experiment werden die Mutationen wieder nacheinander hinzugefügt. Daher können die Werte aus Tabelle 4 auf der vorherigen Seite als Referenzwerte betrachtet werden. Dieses Mal wird aber statt der zufälligen Selektion, die Lineare verwendet. Es werden wieder alle Fitnessbausteine und kein Crossover verwendet. In Tabelle 6 sind die Ergebnisse der linearen Selektion dargestellt.

Tabelle 6: Selektion

Mutation	**Länge**	**Iteration**	**Autos**	**Verletzung**
MoveActionMutation	20460	341	10	10 TW, 1 O
MoveSubrouteMutation	19213	413	10	9 TW
SwapOrderMutation	15647	550	10	1 O
AllocateRouteMutation	16622	803	10	1 O
CombineTwoToursMutation	16696	630	10	1 TW
SplitTourMutation	16452	638	10	
NullMutation	15969	661	10	

Beim Vergleich der zufälligen und linearen Selektion sind die Anzahl der Iterationen auffällig. Bei der linearen Selektion werden grundsätzlich deutlich weniger Iterationen als bei der zufälligen Selektion benötigt. Dafür sind die Ergebnisse aber qualitativ nicht ganz so gut.

Die Verletzungen sind zwar ziemlich ähnlich aber die Länge unterscheidet sich dann doch etwas. Bei der linearen Selektion sind Pläne mit der gleichen Mutationen länger als bei der zufälligen Selektion.

5.3.6 Crossover

In diesem Experiment wird die Auswirkung des Crossovers auf das Ergebnis des genetischen Algorithmus untersucht. Wie bereits in den Experimenten davor, werden die Mutationen wieder nacheinander hinzugefügt. Es wird wieder die zufällige Selektion verwendet. Daher können die Werte aus Tabelle 4 auf Seite 34 wieder als Referenz genommen werden. Es werden wieder alle Fitnessbausteine verwendet, diesmal wird der genetische Algorithmus aber mit dem Crossover ausgeführt. In Tabelle 7 sind die Ergebnisse für dieses Experiment aufgelistet.

Tabelle 7: Crossover

Mutation	**Länge**	**Iteration**	**Autos**	**Verletzung**
MoveActionMutation	19930	405	10	12 TW, 1 O
MoveSubrouteMutation	20139	562	10	9 TW, 1 O
SwapOrderMutation	15730	823	10	1 O
AllocateRouteMutation	15518	1544	10	1 TW, 1 O
CombineTwoToursMutation	17069	1027	10	1 O
SplitTourMutation	15598	1111	10	
NullMutation	15501	1409	10	1 O

Auch bei der Verwendung des Crossovers werden die Werte grundsätzlich besser, wenn eine weitere Mutation hinzukommt. Die Länge des Plans sowie die Verletzungen liegen in ähnlichen Bereichen wie bei der Verwendung des genetischen Algorithmus ohne Crossover. Interessant sind hierbei die Anzahl der Iterationen. Diese liegen deutlich über denen ohne Crossover, beispielsweise bei der *SplitTourMutation* 871 (ohne Crossover) zu 1111 (mit Crossover).

5.3.7 Fitness

Mit diesem Experiment soll die Funktion des Fitnesstests nachgewiesen sowie deren Auswirkung auf den genetischen Algorithmus. Dafür werden die Bausteine des Fitnesstests nacheinander hinzugefügt und das Ergebnis untersucht. Die Vorgehensweise ist damit nahezu identisch zur Untersuchung der Mutationen. Allerdings wird beim Experiment zum Fitnesstest darauf verzichtet die Bausteine separat zu testen. Separate Tests sind nicht sinnvoll, da der genetische Algorithmus dann schnell beendet wird, z.B. wenn nur der Baustein für die Kapazitätsbedingungen getestet wird, endet der genetische Algorithmus sobald eine Lösung ohne Verletzung gefunden wird. Es werden alle Mutationen, die zufällige Selektion und kein Crossover für dieses Experiment verwendet. In Tabelle 8 auf der nächsten Seite sind die Ergebnisse dargestellt.

Die Ergebnisse zeigen, dass der Fitnesstest enormen Einfluss auf den Ablauf und das Ergebnis des genetischen Algorithmus hat. Wenn nur nach der Länge optimiert wird, wird zwar ein

Tabelle 8: Fitness

Fitness	**Länge**	**Iteration**	**Autos**	**Verletzung**
LengthFitness	6528	826	1	87 TW, 46 O
VehicleFitness	8109	656	1	87 TW, 46 O
CapacityFitnessTest	15427	343	1	91 TW
VehicleMakeSpanFitnessTest	15675	543	1	91 TW
LoadingFitnessTest	15957	933	10	

sehr kurzer Plan gefunden, allerdings werden die Zeitfenster und Überladungen komplett außer Acht gelassen und die Verletzungen sind außerordentlich hoch. Da diese Restriktionen vom Fitnesstest nicht beachtet werden, wird auch nur ein Fahrzeug für den Plan benötigt. Anders sieht es aus, wenn der Baustein zur Bewertung der Kapazität hinzugefügt wird. Es sind dann keine Überladungen mehr vorhanden, dafür werden aber ein paar mehr Zeitfenster verletzt und der Plan wird deutlich länger. Mit dem Hinzufügen des Bausteins zur Bewertung der Zeitfenster für die Aufträge werden nun alle Restriktionen beachtet und das Ergebnis verändert sich dahingehend. Es treten weder Verletzungen der Kapazität noch der Zeitfenster auf. Dafür wird der Plan aber nochmals etwas länger und es werden wieder zehn Fahrzeuge benötigt.

5.4 Auswertung

Mit den Experimenten konnte die Funktionalität des genetischen Algorithmus nachgewiesen werden. Sowohl die Visualisierung des Plans als auch die charakteristischen Werte, z.B. die Länge oder Anzahl der Verletzungen, zeigen, dass der entwickelte Algorithmus zur Optimierung in der Fahrzeugeinsatzplanung eingesetzt werden kann. Allerdings muss erwähnt werden, dass es noch Potential zur Verbesserung gibt. Vor allem die Visualisierung zeigt, dass es noch einige Verbesserungsmöglichkeiten gibt.

Die Experimente haben außerdem gezeigt, dass es sehr viele Möglichkeiten gibt den genetischen Algorithmus zu beeinflussen. Das Ergebnis kann zum Beispiel durch Mutationen, Selektion, Crossover oder den Fitnesstest verändert werden. Mit diesen Parametern kann der Algorithmus spezifischen Problemstellungen angepasst werden, z.B. kann mit der linearen Selektion schneller ein Ergebnis gefunden werden, welches aber nicht ganz so gut sein muss, wie bei der zufälligen Selektion. Mutationen sollten grundsätzlich nicht einzelnen verwendet werden, sondern immer in Kombination. Dabei wurden die besten Ergebnisse erzielt. Außerdem sollte die Mutationsrate ziemlich hoch liegen (>90%). Auch der Fitnesstest spielt eine wichtige Rolle bei dem entwickelten Algorithmus, da er sehr starken Einfluss auf das Ergebnis hat.

Während der Entwicklung der Applikation und der Durchführung der Experimente wurden schon relativ gute Werte für die Parameter der Mutation und des Fitnesstests gefunden. Aber auch hier gibt es noch Verbesserungspotential. Durch Kombinieren verschiedener Werte für die Parameter kann der genetische Algorithmus sicherlich noch verbessert werden. Hierbei sollte aber immer daran gedacht werden, dass die Optimierung nicht nur von den Parametern

abhängt, sondern auch von der Konfiguration. Wenn eine bestimmte Kombination von Parametern bei einem Problem zu einer richtig guten Lösung führt, muss dies nicht zwangsläufig bei einem anderen Problem auch so sein.

Während der Experimente veränderte sich die Zahl der Fahrzeuge nicht, ausgenommen vom Experiment zum Fitnesstest. Das liegt daran, dass die Verletzung der Zeitfenster sehr stark bestraft wird. Daher werden immer alle Fahrzeuge benötigt und die Anzahl kann nicht minimiert werden. Es wurden aber auch Experimente durchgeführt, in denen die Zeitfenster sehr groß waren. Hier klappte dann auch die Minimierung der verwendeten Fahrzeuge.

6 Ausblick

Zum Zeitpunkt der Abgabe dieses Projektes, lieferte der genetischen Algorithmus schon relativ gute Ergebnisse. Trotzdem könnte der genetische Algorithmus noch erweitert werden, um bessere Ergebnisse zu liefern. Es wäre denkbar weitere Mutationen hinzuzufügen, beispielsweise eine Mutation zum Drehen von Teilen einer Tour. Um die Ergebnisse des genetischen Algorithmus zu verbessern, könnten auch noch unterschiedliche Kombinationen der Parameter getestet werden, beispielsweise die Verteilung der bereits vorhandenen Mutationen. Ebenfalls eine Möglichkeit bessere Ergebnisse zu erzielen, ist die Implementierung eines weiteren Crossovers bzw. Verbesserung des vorhandenen.

Die Applikation bietet dem Anwender mit verschiedenen Diagrammen Informationen über den Ablauf des genetischen Algorithmus, z.B. Fitness und Länge eines Plans in Abhängigkeit der Iterationen oder Verteilung der Fitnesswerte. Allerdings könnten auch noch weitere Diagramme bzw. Informationen hinzugefügt werden, z.B. Anzahl der Fahrzeuge, Wartezeiten oder Gesamtfahrzeit.

In diesem Zusammenhang wäre es für den Anwender vorteilhaft, wenn er zur Laufzeit des genetischen Algorithmus Änderungen an den Parametern vornehmen könnte. Eine Möglichkeit wäre das dynamische Ändern der Werte für die Wichtigkeit der Fitnessbausteine.

Momentan wird die Applikation noch aus der Entwicklungsumgebung Eclipse gestartet und die Änderung der Parameter muss direkt im Sourcecode erfolgen. Die Änderung der Parameter erfolgt zwar an einer zentralen Stelle, jedoch muss man Kenntnis über den Aufbau des Programms besitzen. Hier wäre es wünschenswert, wenn die Parameter von außen in die Applikation gegeben werden können. Bestenfalls sollte dies mit einer grafischen Eingabemaske realisiert werden.

Außerdem sind Verbesserungen an der Softwarearchitektur denkbar, beispielsweise könnte noch die Dependency-Injection eingeführt werden, um Abhängigkeiten zwischen Objekten zentral verwalten zu können.

Es wurden bis jetzt nur für einige Module Unittests geschrieben. Es wäre wichtig, dass fehlende Unittests noch ergänzt werden, um nicht offensichtliche Fehler zu finden und im Anschluss zu beheben.

Eine weitere Überlegung wäre, die Werte (Länge eines Plans, Anzahl Fahrzeuge, Fitnesswert) jeder Iteration in eine Datei auszugeben, um im Anschluss detaillierte Auswertungen durchführen zu können.

Momentan beinhaltet die Applikation noch nicht die Umsetzung der Synchronisation zwischen zwei Aufträgen mit verschiedenen Produkten. Diese Funktionalität müsste in der weiteren Entwicklung noch hinzugefügt werden.

7 Fazit

Das Ziel dieses Projektes, die Implementierung eines genetischen Algorithmus zur Optimierung in der Fahrzeugeinsatzplanung, wurde erreicht. Hierbei wurden neue Erfahrungen im Bereich der Fahrzeugeinsatzplanung gesammelt. Die dort vorhandenen Problemstellungen sowie möglichen Lösungswege wurden kennengelernt. Zur Umsetzung des genetischen Algorithmus konnte auf bereits vorhandene Kenntnisse der Softwareentwicklung zurückgegriffen werden. Dadurch bot dieses Projekt eine gute Kombination zwischen neuem und bereits vorhandenem Wissen.

Der genetische Algorithmus ist ein sehr interessanter Ansatz für die Lösung von Optimierungsproblemen. Die Grundidee des genetischen Algorithmus lässt sich relativ einfach umsetzen und liefert gute Ergebnisse. Ein weiterer interessanter Aspekt des genetischen Algorithmus ist seine Vielfältigkeit. Er lässt sich für eine Vielzahl von Optimierungsproblemen einsetzen und könnte daher immer wieder Verwendung in den verschiedensten Gebieten finden.

Ein weiterer wichtiger Punkt bei der Durchführung dieses Projektes war die Komponente Zufall. Es ist erstaunlich, wie viel beim genetischen Algorithmus dem Zufall überlassen wird und trotzdem gute Ergebnisse erzielt werden.

Während dieses Projektes wurde auch gelernt, wie die Ergebnisse bewertet werden können. Gerade in einem komplexen Gebiet wie der Fahrzeugeinsatzplanung ist es unheimlich schwierig die gefundenen Lösungen einschätzen zu können. Hierbei sind Visualisierungen unheimlich wichtig. Die Verteilung der Touren gibt Auskunft über die Qualität eines Plans, z.B. hat ein guter Plan kaum Überschneidungen und Schleifen. Auch der genetische Algorithmus kann über Visualisierungen bewertet werden, da er einen charakteristische Graph vorweist und ein enger Zusammenhang zwischen dem besten Wert und dem Durchschnittswert einer Population besteht (Selektionsdruck).

Literatur

[1] Bortfeldt, Andreas ; Fischer, Torsten ; Homberger, Jörg ; Pankratz, Giselher ; Strangmeier, Reinhard: Planen, Lernen, Optimieren. (2003)

[2] EasyMock contributors: *EasyMock*. `http://easymock.org/`. Version: 2014. – [Online; 9. Juli 2014]

[3] Google: *Guave*. `https://code.google.com/p/guava-libraries/`. Version: 2014. – [Online; 9. Juli 2014]

[4] hamcrest.org: *Hamcrest*. `http://hamcrest.org/`. Version: 2014. – [Online; 9. Juli 2014]

[5] Jih, Wan-rong ; Hsu, Y: A family competition genetic algorithm for the pickup and delivery problems with time window. In: *Bulletin of the College of Engineering* 90 (2004), S. 121–130

[6] Joda: *Joda-Time - Java date and time API*. `http://www.joda.org/joda-time/`. Version: 2014. – [Online; 9. Juli 2014]

[7] JUnit: *JUnit*. `http://junit.org/`. Version: 2014. – [Online; 9. Juli 2014]

[8] Object Refinery: *JFreeChart*. `http://www.jfree.org/jfreechart/`. Version: 2014. – [Online; 9. Juli 2014]

[9] QOS.ch: *Logback Project*. `http://logback.qos.ch/`. Version: 2014. – [Online; 9. Juli 2014]

[10] QOS.ch: *Simple Logging Facade for Java (SLF4J)*. `http://http://www.slf4j.org/`. Version: 2014. – [Online; 9. Juli 2014]

[11] The Apache Software Foundation: *Commons Collections*. `http://commons.apache.org/proper/commons-collections/`. Version: 2014. – [Online; 9. Juli 2014]

[12] The Apache Software Foundation: *Commons Lang*. `http://commons.apache.org/proper/commons-lang/`. Version: 2014. – [Online; 9. Juli 2014]

[13] The Apache Software Foundation: *Commons Math: The Apache Commons Mathematics Library*. `http://commons.apache.org/proper/commons-math/`. Version: 2014. – [Online; 9. Juli 2014]