
Dokumentation

Optimierung des Einkaufens

vorgelegt an der Technischen Hochschule Köln, Campus Gummersbach
im Fach Architektur verteilter Systeme
im Studiengang Informatik Schwerpunkt Software Engineering

vorgelegt von: Ayhan Gezer, 11117870

Philipp Felix Schmeier, 11117804

Sheila Kolodziej, 11143470

Sümeyye Nur Öztürk, 11114188

Sven Thomas, 11117201

Volkan Yüca, 11117315

Prüfer: Prof. Dr. Lutz Köhler (Technische Hochschule Köln)

Gummersbach, 26. Januar 2021

Inhaltsverzeichnis

1	Aufgabenstellung.....	1
2	Grundlegende Dokumente	2
2.1	Lastenheft	2
2.2	Pflichtenheft	3
2.3	Glossar.....	6
2.4	Code Convention	8
3	Recherche und Grundentscheidung	9
3.1	Konkurrenzprodukte	9
3.2	Algorithmen	13
3.2.1	Travelling Salesman Problem (TSP)	13
3.2.2	Genetische Algorithmen (GA).....	14
3.2.3	Parallele Genetische Algorithmen	15
3.2.4	Kombination aus TSP und GA.....	17
3.3	Architektur	17
3.3.1	Mögliche Architekturen bei verteilten Systemen	17
3.3.2	Wahl der Architektur	18
4	Architekturdiagramm	20
5	Komponentenaufbau	21
5.1	Unity-Frontend	21
5.2	Scheduler	23
5.2.1	API-Aufbau.....	23
5.2.2	Technologie	24
5.2.3	Genutzte Kotlin Features.....	25
5.3	Worker	26
6	Implementierung.....	28
6.1	Unity-Frontend	28
6.1.1	Erstellung des Geschäfts	28
6.1.2	Benutzerinterface	30
6.1.3	Scheduler-Aufruf	33
6.1.4	Visualisierung.....	35
6.2	Scheduler	37
6.2.1	Der Ktor-Server	37
6.2.2	OpenAPI 3.0 Dokumentation.....	38
6.2.3	Einblick in den Code.....	39
6.3	Worker	43

6.3.1	Genetischer Algorithmus mit "Travelling Salesman Problem"	43
6.3.2	A*	45
7	Probleme.....	47
7.1	Allgemeine Probleme	47
7.2	Probleme Scheduler.....	48
7.3	Probleme Unity.....	48
8	Messungen.....	50
8.1	Verwendete Tools	50
8.2	Testszenarien.....	50
8.2.1	Parameter Konfiguration	50
8.2.2	Ergebnisqualität bei variierender Einkaufslistenlänge.....	52
8.2.3	Last- und Performancetest	53
8.3	Messergebnisse	54
8.3.1	Parameter Konfiguration	54
8.3.2	Ergebnisqualität bei variierender Einkaufslistenlänge.....	61
9	Fazit	64
9.1	Zeitmessung.....	65
9.2	Erfahrungsbericht.....	65
10	Quellenverzeichnis	68
10.1	Literatur.....	68
10.2	Internetquellen	68
11	Anhang	70
11.1	Berührungspunkte während des Projekts	70
11.2	Projektfeedback (lessons learned)	70
11.3	Zeitprotokolle.....	71
11.4	Paper	75

Abbildungsverzeichnis

Abbildung 1: Kaufland App	9
Abbildung 2: Besorger App.....	10
Abbildung 3: Pon App.....	11
Abbildung 4: Milk for Us App	12
Abbildung 5: Travelling Salesman Problem	14
Abbildung 6: Kreuzung zweier Eltern.....	17
Abbildung 7: Architekturdiagramm.....	20
Abbildung 8: Unity-Logo	21
Abbildung 9: Skizze von Dornseifer	22
Abbildung 10: OpenApi 3.0 Dokumentation des Schedulers.....	23
Abbildung 11: Beschreibung der Path Ressource.....	24
Abbildung 12: Extensionfunction für MutableList	25
Abbildung 13: Nutzung der Extensionfunction	26
Abbildung 14: Header der addWorker Funktion	26
Abbildung 15: Modelliertes Geschäft in Unity	28
Abbildung 16: Wegegraph anstelle von Luftlinien	30
Abbildung 17: Übersichts-Ansicht des Benutzerinterfaces.....	31
Abbildung 18: Einkaufslisten-Planer	32
Abbildung 19: Initialisierung des Einkaufslisten-Planers	33
Abbildung 20: Starten der Berechnung.....	34
Abbildung 21: Abfragen des aktuellen Rechenergebnisses	35
Abbildung 22: Visualisierung des Rechenergebnisses.....	36
Abbildung 23: Berechnung der Visualisierung	37
Abbildung 24: Initialisierung und Start des Servers.....	38
Abbildung 25: GET /worker Definition.....	38
Abbildung 26: Umsetzung der logRequest-Funktion.....	38
Abbildung 27: Abspeichern der Einkaufslistenelemente und des Navigationsgitters	40
Abbildung 28: Definition der Größe der Subpopulationen	40
Abbildung 29: Initialisierung des Workers beim Scheduler	41
Abbildung 30: Aktualisieren des Workers beim Scheduler.....	41
Abbildung 31: Löschen einer alten Einkaufsliste.....	42
Abbildung 32: Startpopulation generieren.....	43
Abbildung 33: Crossover	43
Abbildung 34: Mutation.....	44
Abbildung 35: Selektion der Überlebenden für nächste Generation.....	44
Abbildung 36: Berechnung bester Pfad	44
Abbildung 37: Pfadfindung ohne Atern	45
Abbildung 38: Pfadfindung mit A*	46
Abbildung 39: Beispielergebnis mit Dreiecksberechnungen	53
Abbildung 40: Abhängigkeit von WORKER_COUNT zu Distanz	55
Abbildung 41: Zeitlicher Verlauf der Distanzverbesserung.....	56
Abbildung 42: Zeit und Distanzen unterschiedlicher DEMO_INDIVIDUAL_SIZE Werte.....	57
Abbildung 43: Abhängigkeit von INDIVIDUAL_EXCHANGE_COUNT zu Distanz.....	58
Abbildung 44: Distanz in Abhängigkeit von der Mutationsrate	59
Abbildung 45: TournamentSize bei einer Populationsgröße von 1200.....	59
Abbildung 46: TournamentSize bei einer Populationsgröße von 2000.....	60
Abbildung 47: Qualität unterschiedlicher Einkaufslistenlängen	62
Abbildung 48: Qualität unterschiedlicher Einkaufslängen mit reduzierter Individuenzahl	62
Abbildung 49: Projektfeedback (lessons learned)	66
Abbildung 50: Berührungspunkte	67

Abbildung 51: Berührungspunkte während des Projekts.....	70
Abbildung 52: Berührungspunkte - Legende.....	70
Abbildung 53: Projektfeedback	70
Abbildung 54: Zeitprotokoll Studierende*r A	71
Abbildung 55: Zeitprotokoll Studierende*r B	71
Abbildung 56: Zeitprotokoll Studierende*r C	72
Abbildung 57: Zeitprotokoll Studierende*r D	72
Abbildung 58: Zeitprotokoll Studierende*r E	73
Abbildung 59: Zeitprotokoll Studierende*r F	73
Abbildung 60: Zeitprotokoll Alle zusammengefasst.....	74

Tabellenverzeichnis

Tabelle 1: Qualitätsanforderungen.....	2
Tabelle 2: Qualitätsanforderungen.....	5
Tabelle 3: Parameterkonfiguration des ersten Tests.....	61

1 Aufgabenstellung

Im Rahmen des Moduls „Architektur verteilter Systeme“ wird eine Anwendung zum optimierten Einkaufen erstellt, dessen Idee auf einem alltäglichen Problem beruht. Mit oder ohne Einkaufsliste ist es nahezu unmöglich den kürzesten Weg durch den Einkaufsladen zu finden und dabei an alle gewünschten Produkte zu gelangen. Meistens hat man zum Schluss immer noch ein paar Produkte offen und ist ggf. gezwungen noch einmal durch den gesamten Laden gehen, bevor man zur Kasse gehen kann. Dies ist ärgerlich und zeitaufwendig.

Die Anwendung soll dieses Problem beheben. Aus einer Auswahl von vordefinierten Einkaufsläden soll der Benutzer den gewünschten Laden auswählen können. Dort ist vermerkt, welches Produkt sich an welcher Stelle befindet. Der Benutzer kann daraufhin eine Einkaufsliste erstellen und anhand dieser sowie des Einkaufsladens wird der optimale Weg durch den Laden berechnet. Die Berechnung soll auf mehreren Rechnern stattfinden, um die Komplexität der verwendeten Algorithmen zeitsparend und qualitativ gewinnbringend zu verteilen. Anschließend wird dem Benutzer das Ergebnis grafisch angezeigt, sodass dieser durch den Laden navigiert wird.

Trotz der offenen zeitlichen Vorgaben wurde beschlossen dieses Projekt innerhalb eines Semesters abzuschließen. Dieser zeitliche Rahmen konnte nur eingehalten werden, indem man die Aufgabe zunächst nur auf einen Einkaufsladen beschränkt. Hierzu wurde der Lebensmittelladen „Dornseifer“ in Gummersbach ausgewählt.

2 Grundlegende Dokumente

2.1 Lastenheft

1. Zielbestimmung

Der Benutzer der Anwendung soll in der Lage sein mithilfe seiner eigenen Einkaufsliste den kürzesten Weg durch einen von ihm gewählten Einkaufsladen zu finden.

2. Produkteinsatz

Die Anwendung dient dem zeitlich effizienten Einkaufen des Benutzers. Zielgruppe sind alle einkaufenden Kinder, Jugendliche und Erwachsene.

3. Produktfunktionen

/LF10/ Einkaufsliste erstellen

/LF20/ Einkaufsliste

/LF30/ Navigation durch den Einkaufsladen

4. Produktdaten

/LD10/ Einkaufslistenelemente speichern /LD20/ Einkaufsladenstruktur speichern

5. Produktleistungen

/LL10/ Flüssige und echtzeitnahe Navigation durch den Einkaufsladen

6. Qualitätsanforderungen

Anforderung	sehr gut	gut	normal
Performance	x		
Erweiterbarkeit	x		

Tabelle 1: Qualitätsanforderungen

2.2 Pflichtenheft

1. Zielbestimmung

Der Benutzer dieser Anwendung soll in der Lage sein der Anwendung seine Einkaufsliste mitzuteilen. Daraufhin wird dem Benutzer den kürzesten Weg durch einen ausgewählten Einkaufsladen gezeigt.

(a) Musskriterien

- i. Benutzer kann Einkaufsliste erstellen
- ii. Benutzer kann Einkaufsliste vor dem Start der Navigation verändern
- iii. Einkaufsladenstruktur von „Dornseifer“ in Anwendung enthalten
- iv. Einkaufsladenstruktur von „Dornseifer“ wird in 2D bzw. 3D dargestellt
- v. Der kürzeste Weg durch den Einkaufsladen wird ermittelt
- vi. Grafische Darstellung der Navigation durch den Einkaufsladen
- vii.

(b) Wunschkriterien

- i. Während der Navigation kann der Benutzer die Einkaufsliste anpassen
- ii. Weitere Einkaufsläden stehen zur Auswahl
- iii. Anzeigen der zeitlichen Dauer des Einkaufs
- iv. Integration in eine App
- v. Berechnung Abbrechen

2. Produkteinsatz

(a) Anwendungsbereiche

- i. Einkauf
- ii. Alltag

(b) Zielgruppe

- i. Alle einkauffähigen Kinder, Jugendliche und Erwachsene

(c) Betriebsbedingung

- i. Alltägliche Umgebung

3. Produktfunktionen

/F10/ (/LF10/)

Einkaufsliste erstellen: Von Programmstart bis Navigationsstart

Ziel: Erstellen einer neuen Einkaufsliste

Vorbedingung: Navigation wurde noch nicht gestartet.

Nachbedingung Erfolg: Eine Einkaufsliste wurde erstellt.

Nachbedingung Fehlschlag: Eine Einkaufsliste konnte nicht erstellt werden.

Akteure: Benutzer

Auslösendes Ereignis: Benutzer wählt "Neue Einkaufsliste" aus.

Beschreibung:

- (a) Neue/Leere Einkaufsliste anzeigen
- (b) Einkaufslistenelemente hinzufügen lassen
- (c) Einkaufsliste wird weiterverarbeitet

/F20/ (/LF20/)

Einkaufsliste verändern: Von Programmstart bis Navigationsstart

Ziel: Benutzer verändert eine vorhandene Einkaufsliste

Vorbedingung: Eine Einkaufsliste ist vorhanden

Nachbedingung Erfolg: Einkaufsliste wurde verändert

Nachbedingung Fehlschlag: Einkaufsliste konnte nicht verändert werden

Akteure: Benutzer

Auslösendes Ereignis: Benutzer drückt Button „Einkauf planen“ im Frontend

Beschreibung:

- (a) Benutzer stellt sich eine Einkaufsliste zusammen
- (b) Die neu erstellte Einkaufsliste wird weiterverarbeitet

F30/ (/LF30/)

Navigation durch den Einkaufsladen: Von Navigationsstart bis Navigationssende

Ziel: Der kürzeste Weg durch den Einkaufsladen wird berechnet und dem Benutzer angezeigt

Vorbedingung: Einkaufsliste und Einkaufsladen wurden ausgewählt

Nachbedingung Erfolg: Der Benutzer wird durch den Einkaufsladen navigiert

Nachbedingung Fehlschlag: Die Navigation kann nicht gestartet werden

Auslösendes Ereignis: Spieler drückt Button „Starte Berechnung“

Beschreibung:

- (a) Anhand der Einkaufslistenelemente wird der kürzeste Weg von Ladeneingang bis zur Ladenkasse ermittelt.
- (b) Die Einkaufslistenelemente werden umsortiert und in eine Abholreihenfolge einge-reiht
- (c) Der Benutzer erhält eine grafische Navigation zu den Abholelementen mittels einer Linie.

4. Produktdaten

/D10/ (/LD10/)

Name des jeweiligen Produkts aus der Einkaufsliste

/D20/ (/LD20/)

Einkaufsladenstruktur:

- (a) Position der Regale, Wände, Kassen, Ein-/Ausgang
- (b) Position und Name der Produkte

6. Qualitätsanforderungen

Anforderung	sehr gut	gut	normal
Performance	x		
Erweiterbarkeit	x		

Tabelle 2: Qualitätsanforderungen

7. Benutzeroberfläche

- (a) Design ist dem Entwickler überlassen
- (b) Benutzerinteraktion ist dem Entwickler überlassen

8. Technische Produktumgebung

- (a) Mac oder Windows PCs

2.3 Glossar

Abholelement

Ein Element, das vom Benutzer in einer Einkaufsliste als Einkaufslistenelement eingetragen worden ist und einen Knoten im Wegegraph darstellt. Es wird vom Benutzer im Laufe der Navigation bzw. des Einkaufens eingekauft.

Allel

Konkrete Ausprägung eines Gens. Wird das Gen als Variable aufgefasst, so ist das Allel der Wert der Variablen.

Binär kodiertes Chromosom

Chromosom kann in diesem Zusammenhang mit Individuum gleichgesetzt werden. Anstelle des biologischen Erbguts werden beim binär kodierten Chromosom nur Nullen und Einsen gespeichert.

Crossover

Vorgang, bei dem zwei Abschnitte zweier Individuen/Chromosome ausgetauscht werden

Einkaufslistenelement

Ein Element, das vom Benutzer in einer Einkaufsliste eingetragen worden ist. Es besteht jeweils nur aus einem Produkt z.B. Salami.

Gen

Ein Gen stellt die einzelnen Abschnitte eines Individuums dar. Im Fall der Einkaufsoptimierung bedeutet dies, ein Gen stellt einen Teil der Route durch das Geschäft dar. Ein vom Benutzer ausgewähltes Produkt.

Individuum

Ein Individuum stellt eine Kombination von Genen dar. Es repräsentiert einen möglichen Weg durch das Geschäft. Dies wird durch die eindeutige Reihenfolge der Gene, also der Produkte dargestellt.

Masterpopulation

Die Masterpopulation wird einmalig für jede neue Einkaufsliste erzeugt und bildet die Grundlage für alle weiteren Populationen welche als Subpopulationen an die Worker zur Bearbeitung verteilt werden.

Master-Worker-Architektur

Die Architektur zum hierarchischen Verwalten des Zugriffs auf eine gemeinsame Ressource besitzt viele unterschiedliche Namen. Im Rahmen des Projektes wird der Begriff der Master-Worker-Architektur verwendet.

Population

Eine Population ist eine Sammlung von Individuen. Durch verschiedene Verfahren wie Selektion, Rekombinieren und Mutation (siehe Crossover), kann eine Population mehrere Generationen durchlaufen. Die Anzahl der Generationen sagt dabei aus, wie oft die Sammlung von Individuen durch die genannten Verfahren verändert wurden.

Scheduler

Der Scheduler stellt die Master-Komponenten in der Master-Worker-Architektur dar und ist für die Verwaltung der Kommunikation zwischen Workern und dem Unity-Frontend zuständig.

Unity-Frontend

Das Frontend, über welches der Nutzer die Anwendung mit allen ihren Komponenten steuert und verwendet.

Wegegraph

Der Wegegraph beschreibt die optimale Strecke vom Ladenanfang bis zum Ladenende und besitzt n Knoten für n Produkte, die auf der Einkaufsliste stehen.

Wegpunkt

Unterstützende Knoten bei der Berechnung der optimalen Strecke. Sie helfen dabei Abbiegungen oder schwierige Wege kollisionsfrei zu berechnen. Schwierige Wege entstehen dadurch, dass nicht jede Ladenstruktur konsequent wie ein Netz aufgebaut ist und dadurch Wege entstehen, die ohne zusätzliche Wegpunkte nicht realistisch dargestellt werden können.

Worker

Der Worker ist für die eigentliche Rechnung bzw. Generierung von Populationen zuständig und liefert an den Scheduler während der Laufzeit, verbesserte Populationsergebnisse anhand der Distanz.

2.4 Code Convention

Da die Komponenten in verschiedenen Sprachen geschrieben werden, wird kein einheitlicher Standard festgelegt. Für die jeweilige Programmiersprache werden die offiziellen Code Conventions als Standard angesehen und für die jeweilige Komponente eingehalten.

Kommentare sollen auf Englisch verfasst werden. Nach Möglichkeit sollte jede nicht triviale Methode mit einer Dokumentation versehen werden. Dazu gehören eine kurze Beschreibung der Aufgabe der Methode, sowie ggf. eine Erläuterung für die jeweiligen Parameter und den Rückgabewert der Funktion. Bei *trivialen* Methoden (z.B. Getter/Setter) kann darauf verzichtet werden.

3 Recherche und Grundentscheidung

3.1 Konkurrenzprodukte

Kaufland App

Die Kaufland App ermöglicht die Auswahl einer Wunschfiliale. Es können mehrere Einkaufslisten erstellt werden, die abgearbeitet werden können. In der Einkaufsliste können die Artikel nach Kategorien geordnet werden und die gewünschte Menge eines Artikels kann bestimmt werden. Außerdem bietet die App die Möglichkeit an, Artikel, die in der App in den Rezepten und Angeboten auftreten, direkt in die Einkaufsliste einzufügen.

Eine wichtige Funktionalität, die die Kaufland App nicht anbietet, ist die Wegweisung durch die Filiale. Aus Benutzersicht ist es nicht sinnvoll, dass beim Eintippen keine Vorschläge angezeigt werden und es somit keine Auswahlmöglichkeit gibt. Der Benutzer muss selbst den Artikelnamen eintippen. In der App sind nur die Artikel gespeichert, die in den Angeboten oder Rezepten auftauchen. Die restlichen Artikel, die man in die Einkaufsliste einfügen möchte, müssen selbst eingetragen werden.

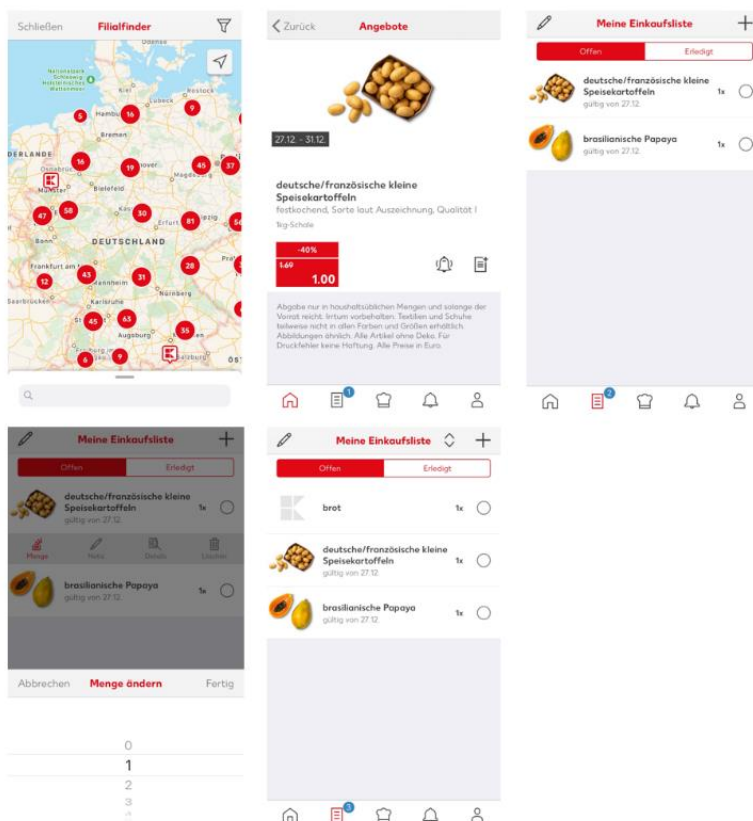


Abbildung 1: Kaufland App

Besorger App

Die Besorger-App ermöglicht es mehrere Einkaufslisten zu erstellen. Bei der Artikeleingabe wird eine Artikelliste angezeigt, aus der man entweder durch Scrollen und Anklicken oder durch Eingabe des Namens Artikel in die Einkaufsliste einfügen kann. Beim Hinzufügen der Artikel in die Einkaufsliste können die Menge, die Abteilung und der Händler (beispielsweise Aldi, Rewe, Rossmann etc.) eingegeben werden. Die Einkaufsliste kann anschließend nach Abteilung, nach Händler oder alphabetisch sortiert werden. Die verschiedenen Produktkategorien besitzen in der Besorger-App unterschiedliche Farben, was der Einkaufsliste Übersichtlichkeit verleiht.

Eine wichtige Funktionalität, die die Kaufland App nicht anbietet, ist die Auswahl der Filiale. Die Händlereingabe dient somit nur der Sortierung der Artikel und die Wegweisung durch die Filiale kann nicht erfolgen.

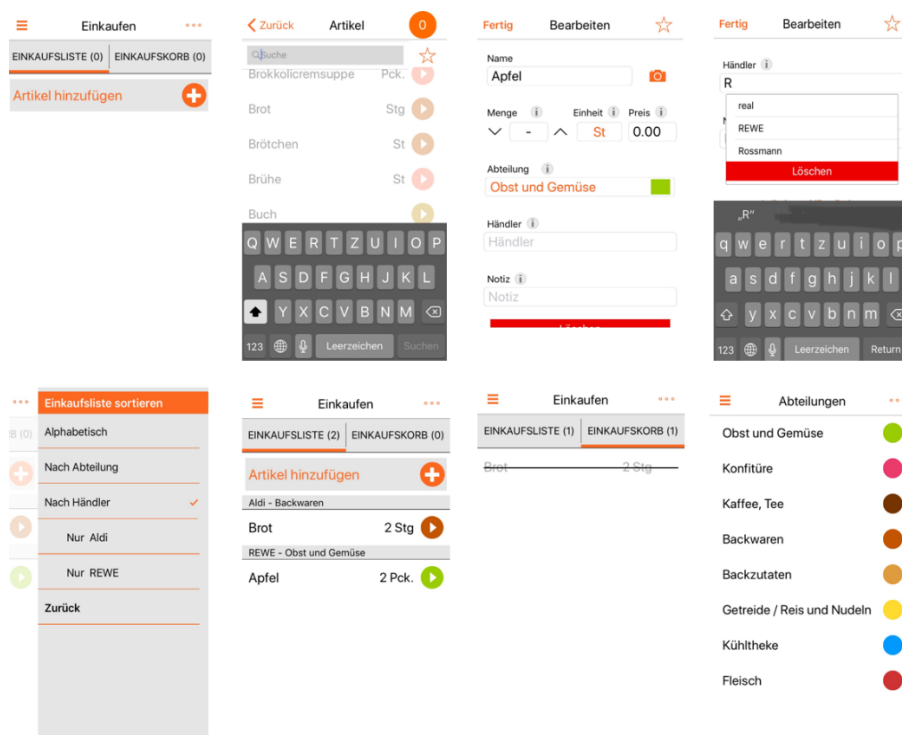


Abbildung 2: Besorger App

Pon – Die smarte Einkaufsliste

Die Pon-App ermöglicht es ebenfalls mehrere Einkaufslisten zu erstellen. Die Artikel, die in die Einkaufsliste hinzugefügt werden können, werden als eine Liste angezeigt. Wählt man einen Artikel aus, können Eigenschaften, wie zum Beispiel Menge und Sorte (bspw. Bio) festgelegt werden. Die vorher gekauften Produkte und Produkteigenschaften werden gespeichert, so dass diese nicht jedes Mal erneut eingegeben werden müssen. Einkaufslisten können nach Einkaufsläden sortiert werden. Bei jedem Artikel kann ausgewählt werden von welchem Einkaufsladen diese erworbt werden soll. Beim Hinzufügen der Artikel in die Einkaufsliste kann, nachdem der Einkaufsladen ausgewählt wurde, auf „Alternative hinzufügen“ gedrückt werden. Dadurch wird der Standort aktiviert und die Artikel können temporär abhängig vom aktuellen Standort in einen anderen Einkaufsladen, der sich gerade in der Nähe befindet, verschoben werden. Falls man in die Nähe des Einkaufsladen kommt, erfolgt eine GPS-basierte Erinnerung.

Die Pon-App bietet keinen Wegweiser durch die Filiale. Es ist jedoch möglich die Produkte so zu sortieren, wie sie im Einkaufsladen angeboten werden. Dies ist jedoch mit viel Aufwand verbunden, da man selbst die Kategorien sortieren muss und die Reihenfolge der Kategorien im Einkaufsladen kennen muss.

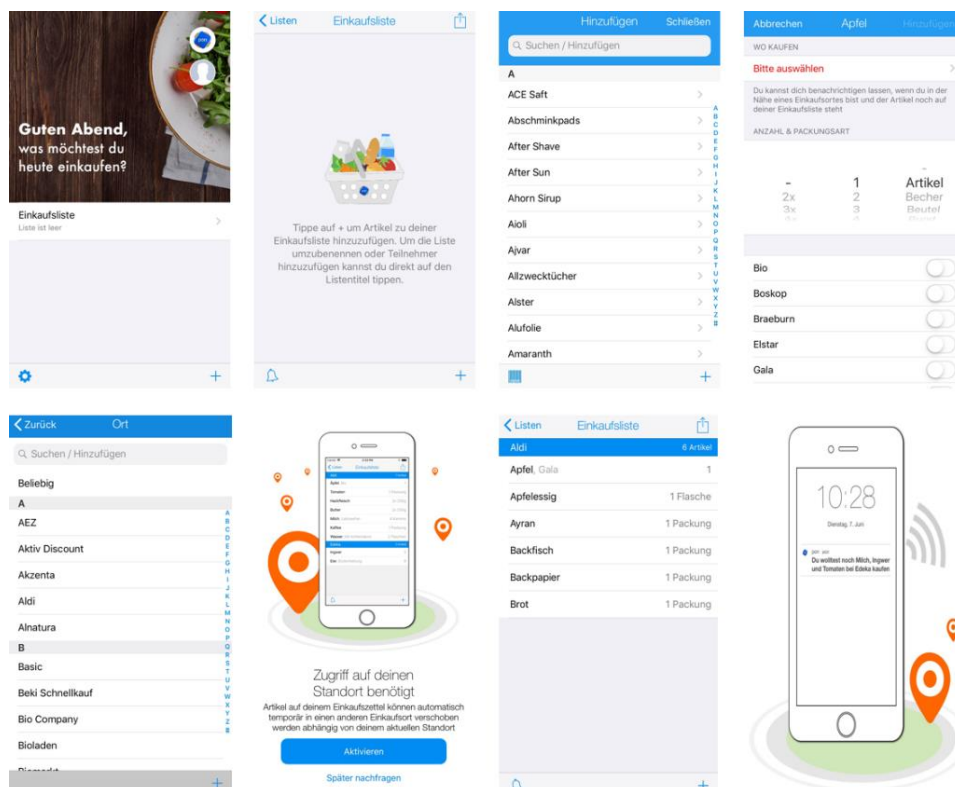


Abbildung 3: Pon App

„Milk for Us“: Der Einkaufszettel für Familie und WG

Mit der Milk for Us-App können mehrere Einkaufslisten erstellt werden. Da die Artikel in einer Datenbank gespeichert sind, können sie entweder aus der Liste ausgewählt oder gesucht werden. Die am häufigsten ausgewählten Produkte werden in der Einkaufsliste vorgeschlagen. Es besteht die Möglichkeit einen Einkaufsladen anzugeben und die Marktaufteilung zu bestimmen. Die einzelnen Kategorien können manuell, beispielsweise nach Reihenfolge der Abteilungen im Einkaufsladen, sortiert werden.

Die Milk for Us-App bietet ebenfalls keinen Wegweiser durch die Filiale. Es ist jedoch möglich, die Produkte so zu sortieren, wie sie im Einkaufsladen angeboten werden. Dies ist jedoch wie bei der Pon-App mit viel Aufwand verbunden, da man selbst die Kategorien sortieren muss und die Reihenfolge der Kategorien im Markt kennen muss.

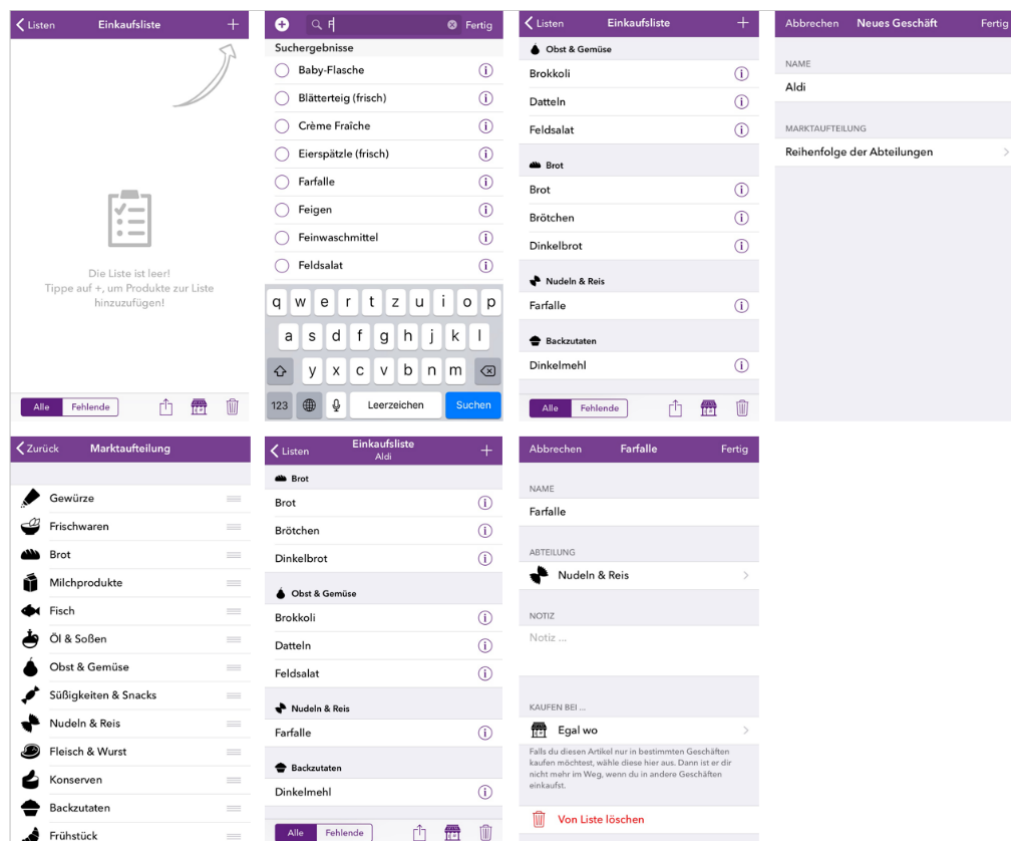


Abbildung 4: Milk for Us App

Fazit

Es sind zahlreiche Einkaufsliste-Apps vorhanden, die die Basisanforderungen an eine Einkaufsliste, also das Notieren oder Auswählen von Artikeln, erfüllen. Die Funktionalität nach Reihenfolge der Artikel im Einkaufsladen einzukaufen, bieten nur die Apps „Milk for Us“ und „Pon“. Hierbei muss jedoch die Reihenfolge der Artikel manuell eingetragen werden, was den ganzen Prozess aufwändiger und komplexer macht, da man nicht unbedingt die Reihenfolge der Artikel in einem Einkaufsladen kennt. Eine weitere gute Funktionalität bietet die Pon-App an. Der Benutzer bekommt eine Mitteilung, wenn er sich in der Nähe eines Marktes befindet. Nach der Konkurrenzanalyse wird deutlich, dass zwar viele Einkaufsliste-Apps vorhanden sind, jedoch keiner von denen die gewünschte Funktionalität, die in dem Projekt behandelt wird, erfüllt.

3.2 Algorithmen

3.2.1 Travelling Salesman Problem (TSP)

Das Problem des Handelsreisenden, auch bekannt als *Travelling Salesman Problem*, ist unter anderem in der theoretischen Informatik ein kombinatorisches Optimierungsproblem. Der „Handlungsreisende“ möchte verschiedene Orte besuchen. Die Aufgabe des Problems besteht darin, eine Reihenfolge mehrerer Orte so zu wählen, dass kein Ort außer dem ersten mehr als einmal besucht wird. Außerdem soll die Strecke des Handlungsreisenden so kurz wie möglich sein und einen Kreis bilden, d.h. der erste Ort ist gleich dem letzten Ort.¹

So wie bei vielen mathematischen Lösungen wird auch hier diese *reale* Situation durch ein einfacheres Modell abgebildet. Das Problem des Handlungsreisenden lässt sich mit einem Graphen modellieren. Die Knoten repräsentieren die Orte, zu denen der Handlungsreisende möchte, während die Kanten eine Verbindung zwischen diesen beschreiben. Zur Vereinfachung kann der Graph als **vollständig** angenommen werden, d.h. zwischen zwei Knoten existiert immer eine Kante. Selbst wenn zwischen zwei Knoten keine Kante existiert, wird in diesem Fall eine künstliche Kante eingefügt. Diese wird bei der Berechnung der kürzesten Strecke nicht mitberücksichtigt, da die Länge einer solchen Querverbindung niemals zu einer kürzesten Strecke beitragen würde.

Man unterscheidet zwischen asymmetrischem und symmetrischem Travelling Salesman Problem. Beim Asymmetrischen können die Kanten in Hin- und Rückrichtung unterschiedliche

¹ Vgl. Wikipedia: Problem des Handlungsreisenden https://de.wikipedia.org/wiki/Problem_des_Handlungsreisenden (Stand: 12.11.2019)

Längen besitzen. Deswegen wird diese Variante durch einen gerichteten Graphen repräsentiert, um die unterschiedlichen Weglängen abbilden zu können.

Beim symmetrischen Travelling Salesman Problem sind die Kantenlängen in beide Richtungen identisch. Deswegen wird meist ein ungerichteter Graph zur Modellierung verwendet.²

Bei n Städten beträgt die Anzahl der möglichen Wege $n!$, d.h. bei sehr hohen n wird es unmöglich in einer angenehmen Zeit alle Wege zu berechnen. Hierbei kann paralleles Rechnen die gesamte Rechenzeit reduzieren.³

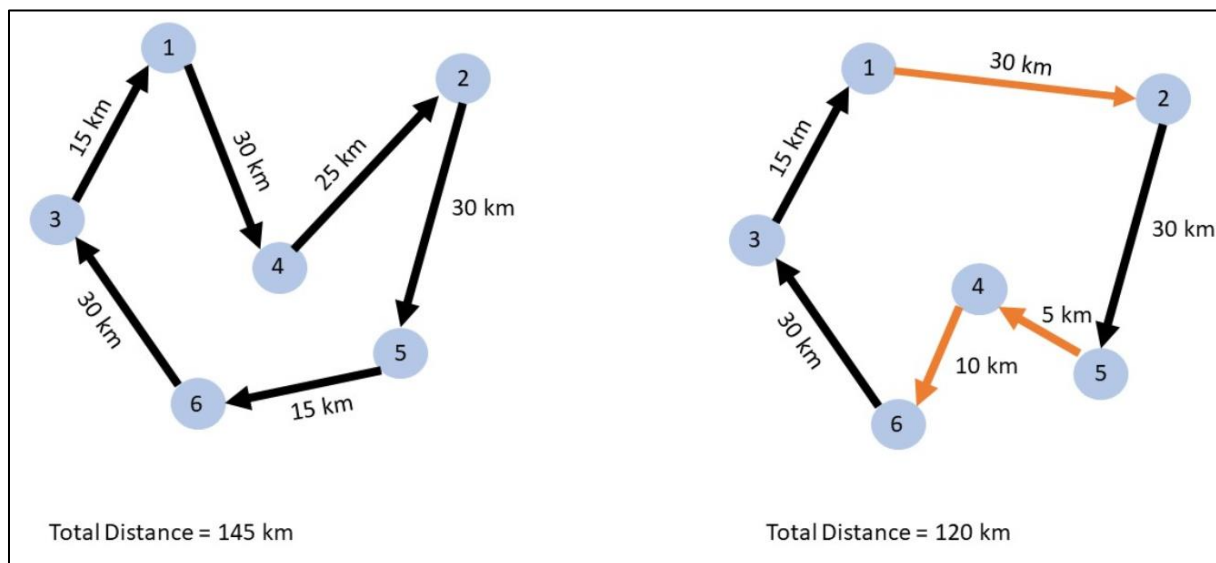


Abbildung 5: Travelling Salesman Problem

3.2.2 Genetische Algorithmen (GA)

Genetische Algorithmen gehören zu den sogenannten evolutionären Algorithmen. Ihre Idee ist es evolutionäre Prozesse nachzuahmen. Der Aufbau eines genetischen Algorithmus sieht wie folgt aus: Zunächst wird das zu optimierende Problem kodiert, d.h. auf ein binär kodierte Chromosom (siehe Glossar) abgebildet. Dann initialisiert man eine Ausgangspopulation, indem eine Population von Individuen (siehe Glossar) erzeugt und zufällig initialisiert wird. Man spricht hier von der Generation 0. Jedes Individuum wird daraufhin mit einer Fitnessfunktion bewertet, d.h. jedem einzelnen Chromosom wird eine reell wertige Zahl zugeordnet.

² Vgl. Wikipedia: Problem des Handlungsreisenden https://de.wikipedia.org/wiki/Problem_des_Handlungsreisenden (Stand: 12.11.2019)

³ Vgl. Borovska, Plamenka: Solving the Travelling Salesman Problem in Parallel by Genetic Algorithm on Multicomputer Cluster; International Conference on Computer Systems and Technologies, 2006.

Daraufhin werden jeweils zwei Elternteile mittels einer Selektionsvariable selektiert und es entstehen mittels einer gewählten Kreuzungsvariante Nachkommen dieser Elternteile. Da die Allele (siehe Glossar) der Nachkommen mutieren können, werden die Werte invertiert.

Dann wird die Population um die neuen Nachkommen ergänzt. Wird dadurch die Populationsgröße überschritten, werden mittels Ersetzungsstrategien alte Nachkommen durch neue ersetzt.

Diese Verfahrensschritte werden solange wiederholt, bis ein Abbruchkriterium erfüllt ist. Allein das Erreichen der Zielfunktion wird in den meisten Fällen nicht reichen, da genetische Algorithmen stochastische Suchalgorithmen sind, diese also das absolute Optimum nie erreichen würden. Das Kriterium sollte demnach einen Abbruch bei hinreichender Nähe zur Zielfunktion sein. Um ein zeitliches Abbruchkriterium zu erhalten kann man außerdem einen Abbruch nach einer maximalen Anzahl von Generationen festlegen. Dies bietet sich an, wenn der genetische Algorithmus sich dem Optimum nicht schnell genug nähert.⁴

3.2.3 Parallele Genetische Algorithmen

Um mittels eines genetischen Algorithmus ein gutes Ergebnis zu erzeugen, ist es bei komplexeren Problemen oftmals nötig, eine Vielzahl von Generationen zu durchlaufen. Es wurden bereits früh überlegt das bei genetischen Algorithmen genutzte Prinzip der Populationen zu nutzen, um diese über mehrere Systeme hinweg zu berechnen.⁵ Bei der Berechnung ist der Schritt der Evaluierung der Gene, also die Berechnung der Fitnessfunktion die zeitaufwendigste Operation.⁶

Die Herausforderung hinsichtlich der Parallelisierung beläuft sich somit auf der richtigen Verteilung der Gene, um eine gleichbleibende Last für alle beteiligten Komponenten zu gewährleisten. Zwar gibt es sogar Ansätze, in welchem nur einzelne Gene von einer Steuerungskomponente verteilt werden und somit ausschließlich die Berechnung der Fitness verteilt durchgeführt wird, allerdings entsteht dadurch eine große Last im genutzten Netz. Es haben sich daher verschiedene Methodiken entwickelt, welche eines gemein haben: Es werden gesamte Populationen aufgeteilt und verteilt. Folgend können neben der Berechnung der Fitness, auch weitere Schritte genetischer Algorithmen verteilt auszuführen werden, wie die Selektion, Rekombination und Mutation.

⁴ Vgl. Keen, Shawn: Ausarbeitung zu Genetischen Algorithmen <http://www.informatik.uni-ulm.de/ni/Lehre/SS04/ProsemSC/ausarbeitungen/Keen.pdf> (Stand:13.11.2019).

⁵ Vgl. W. F. Punch, 1998, "How Effective are Multiple Populations in Genetic Programming", Proceedings of the third Annual Genetic Programming Conference, July 22-25 1998. S. 313 ff.

⁶ Vgl. Chong, Fuey Sian, 1999: Java based Distributed Genetic Programming on the Internet. Technical Report CSRP-99-7, School of Computer Science, The University of Birmingham

Der große Unterschied gängiger Methodiken ist, wie der Austausch unter den unterschiedlichen Populationen gestaltet ist. Folgend werden das Inselmodell, das Pollenmodell und die Nearest-neighbor migration in Kürze erläutert.

3.2.3.1 Inselmodell

Das Inselmodell ist die einfachste Form der Parallelisierung genetischer Algorithmen. Es werden unterschiedliche Populationen generiert, welche unabhängig voneinander bearbeitet werden.⁷ Durch diese Methodik wird zwar die Berechnung oftmals nicht beschleunigt, allerdings verhindern die unterschiedlichen Startpunkte, dass das Endergebnis leicht in einem lokalen Maximum hängen bleibt. Ist es gewünscht, dass ein Austausch zwischen den Populationen existiert, kann das Modell erweitert werden. Die Erweiterung erfolgt durch den Austausch von Genen der verschiedenen Populationen nach einer festgelegten Anzahl an Generationen.^{8,9}

3.2.3.2 Pollenmodell

Das Pollenflugmodell orientiert sich am biologischen Vorbild des Pollenflugs durch den Wind. Nach Erzeugung der Startpopulationen werden diese räumlich zufällig verteilt und ein Wind simuliert. Populationen, die vom Wind betroffen sind, verlieren einen Anteil der Gene. Diese Gene werden, sofern vorhanden, an Populationen in Windrichtung verteilt. Verliert eine Population dadurch zu viele Gene, können weitere nach generiert werden.¹⁰

3.2.3.3 Nearest-neighbor migration

Beim Nachbarschaftsmodell oder Nearest-neighbor migration findet ein Genaustausch nur zwischen direkt benachbarten Populationen statt. Dies bietet den Vorteil einer starken Reduktion der Netzlasten gegenüber dem Pollenmodell. Es müssen keine zusätzlichen Berechnungen, wie die Simulation des Pollenflugs, durchgeführt werden. Anzumerken sei, dass die Nachbarschaftspopulationen nicht die Gene mit der maximalen Fitness austauschen, sondern die Gene zufällig gewählt werden. Dies ist erforderlich, damit Nachbarn nicht zwangsläufig in die gleichen lokalen Maxima laufen.^{11,12}

⁷ Vgl. Riedel, Marion: Parallele Genetische Algorithmen mit Anwendungen. Diss. 2002, TU Chemitz

⁸ Vgl. Nguyen, Minc Duc: Optimierung von Routingproblemen mit Genetischen Algorithmen auf Apache Spark. Diss. Hochschule für Angewandte Wissenschaften Hamburg, 2017, URL: http://edoc.sub.uni-hamburg.de/haw/volltexte/2018/4164/pdf/MinhDucNguyen_Optimierung_von_Routingproblemen_mit_Genetischen_Algorithmen_auf_Apache_Spark.pdf (Stand: 02.01.2020)

⁹ Vgl. Shrestha, Ajay ; Mahmood, Ausif: Improving Genetic Algorithm with Fine-Tuned Crossover and Scaled Architecture. In: Journal of Mathematics vol. 2016 (2016). – doi:10.1155/2016/4015845

¹⁰ Vgl. Riedel, Marion: Parallele Genetische Algorithmen mit Anwendungen. Diss. 2002, TU Chemitz

¹¹ Vgl. Riedel, Marion: Parallele Genetische Algorithmen mit Anwendungen. Diss. 2002, TU Chemitz

¹² Vgl. Tongchim, Shisanu und Chongstitvatana: Prabhas, Parallel genetic programming synchronous and asynchronous migration, Artif Life Robotics

3.2.4 Kombination aus TSP und GA

Bei einer Kombination aus dem Travelling Salesman Problem und genetischem Algorithmus bestehen die Individuen aus einem String aus Zahlen. Jede Zahl repräsentiert die Orte, die der Handelsreisende besuchen möchte. Außerdem repräsentieren die Zahlen die jeweilige Position.

Die Fitness Funktion repräsentiert die Länge der Reise. Dabei kann z.B. die euklidische Distanz zur Berechnung eines jeden Weges herangezogen werden. Die Auswahl erfolgt meistens über eine zufällige Wahl von k Individuen, aus denen die zwei fittesten zur Kreuzung ausgewählt werden. Die Kreuzung basiert über ein zufälliges Crossover-Prinzip (siehe Glossar), d.h. zwei Teile der Eltern werden mittels des Crossover-Punktes kopiert und es entstehen zwei Kinder. Die andere Hälfte des jeweiligen Kindes wird mit den fehlenden Städten des zweiten Elternteils aufgefüllt (siehe Abbildung 6).

Die mögliche Mutation wird durch ein zufälliges Ändern der Ortfolgenfolge hervorgerufen.

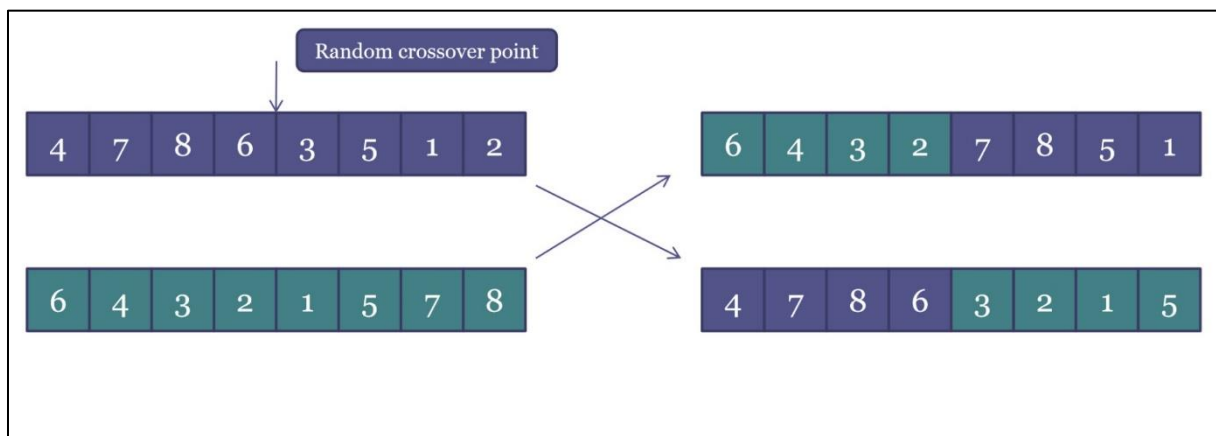


Abbildung 6: Kreuzung zweier Eltern

3.3 Architektur

3.3.1 Mögliche Architekturen bei verteilten Systemen¹³

In verteilten Systemen kommen mittlerweile viele verschiedene Architekturen zum Einsatz. Diese werden im folgenden Abschnitt kurz erläutert. Hierbei werden nicht alle möglichen Architekturen beschrieben, sondern nur die, die im Rahmen dieser Veranstaltung sinnvoll genutzt werden konnten.

¹³ Vgl. Schill, Alexander; Springer, Thomas: Verteilte Systeme: Grundlagen und Basistechnologien, 2.Aufl. 2012, Berlin, Heidelberg, Springer Berlin Heidelberg

Die wohl bekannteste Architektur ist die **Master-Slave-/Master-Worker-Architektur**. Wie der Name vermuten lässt gibt es einen Teilnehmer mit der Rolle des Masters und viele Teilnehmer mit der Rolle des Workers. Nur der Master kann unaufgefordert auf eine gemeinsame Ressource zugreifen. Die Worker können dies erst, wenn sie vom Master dazu aufgefordert werden (Polling). Vorteil ist hier, dass die Zugriffsverhältnisse klar aufgeteilt sind. Allerdings ist diese Architektur ein wenig einschränkend, da die Worker nicht untereinander kommunizieren können und das Polling der Worker durch den Master ineffizient ist.

Eine etwas gleichberechtigtere Variante vom Master-Worker -Prinzip ist die **Peer-to-Peer** Verbindung, bei der alle Teilnehmer gleichberechtigt sind und sowohl Dienste in Anspruch nehmen als auch Dienste zur Verfügung stellen können. In der Praxis werden aber auch hier die Rechner oft in Aufgabenbereiche eingeteilt, um Ordnung in die Aufgabenverteilung zu bringen. Eine dritte Möglichkeit ist die **N-Tier-Architektur**. Dies ist eine Schichtenarchitektur, in der Aspekte des Systems einer Schicht zugeordnet werden. Bei einer drei-Schichten Architektur hat man beispielsweise eine Präsentations-, Anwendungs- und Persistenzschicht. Diese Art der Architektur eignet sich vor allem für komplexe Softwaresysteme. Vorteil ist hier eine zentrale Datenhaltung, sowie die Skalierbarkeit, da man beliebig viele Schichten konstruieren kann. Aber auch die Fehlertoleranz ist hier deutlich besser als in anderen Architekturen.

Eine weitere Möglichkeit ist eine **Serviceorientierte (SOA)-Architektur**. Hier wird sich oft an Geschäftsprozessen orientiert, deren Abstraktion die Grundlage für Serviceimplementierung liefert. Maßgeblich sind hier nicht die technischen Einzelaufgaben, sondern die Zusammenfassung dieser Aufgaben zu einer größeren, komplexeren Aufgabe. Dadurch wird die Komplexität der einzelnen Anwendungen hinter den standardisierten Schnittstellen verborgen. Vorteile dieser Architektur sind eine hohe Flexibilität und eine mögliche Senkung des Programmieraufwandes, da bestimmte Services ggf. wiederverwendet werden können.

Um das Zusammenspiel der einzelnen Komponenten durch Ereignisse zu steuern, wurde die **Event-driven-Architektur** entwickelt. Dabei kann ein Ereignis sowohl von außen als auch vom System selbst ausgelöst werden. Voraussetzung zum Einsatz dieser Architektur ist allerdings, dass alle Teilnehmer, die bei der Abwicklung des Ereignisses beteiligt sind, miteinander kommunizieren können. Sie kann als Ergänzung zur service-orientierten Architektur verwendet werden, da auch dort Dienste durch Ereignisse ausgelöst werden können.

3.3.2 Wahl der Architektur

Hinsichtlich der zuvor vorgestellten Architekturen und der zuvor beschriebenen Aufgabenstellung kann man die N-Tier-Architektur nicht empfehlen. In diesem Projekt ist eine hohe Skalierbarkeit nicht notwendig. Ebenso bietet eine serviceorientierte Architektur für dieses Projekt

keinen Mehrwert, da sich die Flexibilität unserer Anwendung lediglich auf die unterschiedlichen Einkaufsläden beschränkt. Als Ergänzung zur serviceorientierten Architektur entfällt damit auch die Event-driven-Architektur. Sie kann abgrenzend zur SOA genutzt werden, aber hinsichtlich der verwendeten Algorithmen bietet sich eine Master-Worker oder eine Peer-to-Peer Architektur an.

Betrachtet man den genetischen Algorithmus genauer, stellt man fest, dass es reicht, wenn die Kommunikation nur zwischen dem Master und den Workern stattfindet. Die Worker berechnen die neue Generation und müssen das Ergebnis nur dem Master mitteilen. Eine Kommunikation zwischen den einzelnen Workern ist demnach nicht notwendig, somit wird in diesem Projekt die Master-Worker Architektur angewendet.

4 Architekturdiagramm

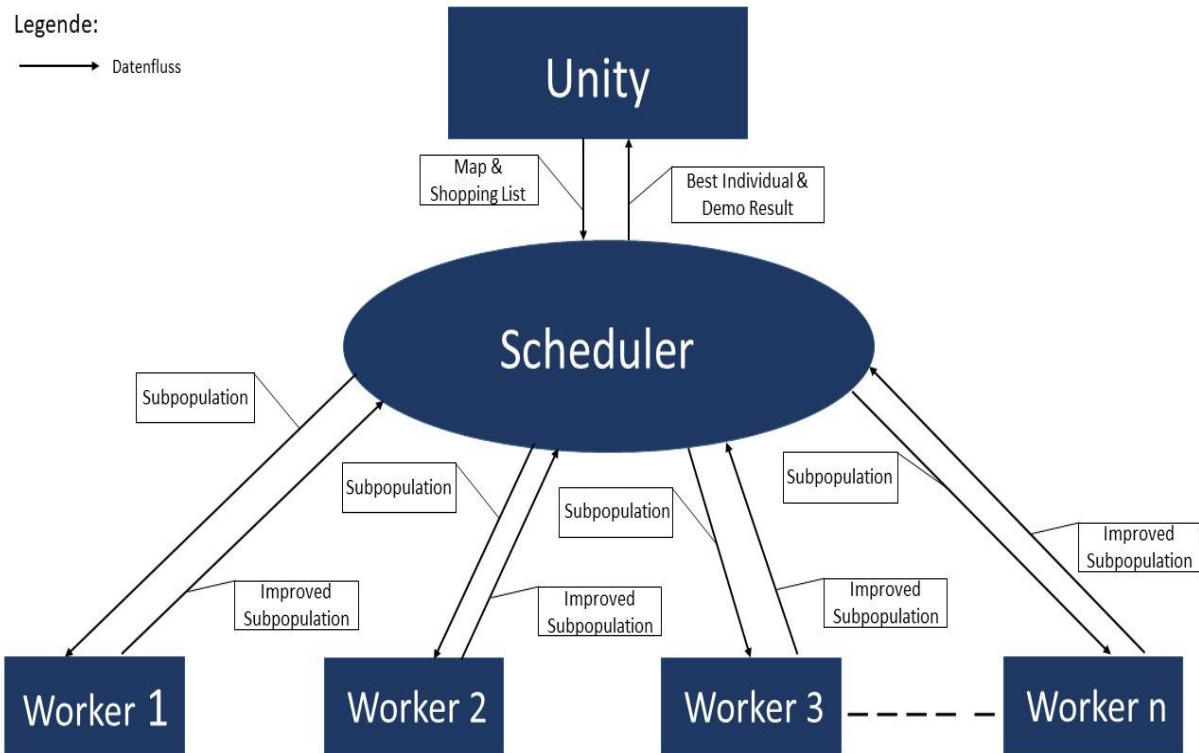


Abbildung 7: Architekturdiagramm

Die Abbildung 7 beschreibt die Architektur der Anwendung. In Unity wird die Karte von „Dorns-eifer“ erstellt und die Möglichkeit angeboten, eine Einkaufsliste zu erstellen. Nachdem man die Einkaufsliste erstellt hat, wird diese Einkaufsliste zusammen mit der Karte an den Scheduler geschickt. Der Scheduler dient als Schnittstelle zwischen der Unity-Instanz und den Workern. Dieser übergibt Subpopulation an die einzelnen Worker, die diese abfragen. Die Scheduler generieren mittels genetischer Algorithmen und Travelling Salesman Problem mehrere Gene-rationen und schicken diese weiter an die Worker. Die Worker ermitteln und übergeben die verbesserte Subpopulation anschließend an den Scheduler weiter, der die kürzeste Distanz, also das beste Individuum von allen Workern bestimmt. Zusammen mit einem Zwischener-gebnis wird dies an Unity übergeben, wo das Ergebnis visualisiert wird.

5 Komponentenaufbau

5.1 Unity-Frontend



Abbildung 8: Unity-Logo

Für die graphische Oberfläche wird Unity verwendet. Mit Unity können neben eigenen Spielen auch 3D-Modelle, Texturen, Animationen und viele andere komplexe Projekte realisieren werden. In dieser Oberfläche sind die verschiedenen Läden mit den Artikelstandorten in einer 3D-Welt abgebildet. Ebenso wird hier die Einkaufsliste erstellt und das Ergebnis präsentiert. Somit stellt die Unity Anwendung sowohl den Start- als auch den Endpunkt der verteilten Anwendung dar.

Die Funktionalitäten, die das Unity Frontend anbietet, sind:

- (1) Visuelle Abbildung der Läden mit Artikelstandorten.
- (2) Generierung einer abstrahierten Darstellung eines Ladens für den Algorithmus.
- (3) Bereitstellung eines UI zur Erstellung einer Einkaufsliste.
- (4) Start des Algorithmus durch Übergabe der Daten an den Scheduler.
- (5) Visualisierung des Ergebnisses.

Um das Frontend in Unity umzusetzen, wurde zuerst die folgende Skizze (vgl. Abbildung 9) von dem Einkaufsladen „Dornseifer“ erstellt um den Einkaufsladen, wie er in Unity aussehen soll, zu veranschaulichen. Dabei wurden die einzelnen Artikel zuerst in Kategorien zusammengefasst. Bei Kategorien, die mehrmals auftreten, kann in der Zukunft eine Spezifikation (z.B. Angabe der Marke) erfolgen.

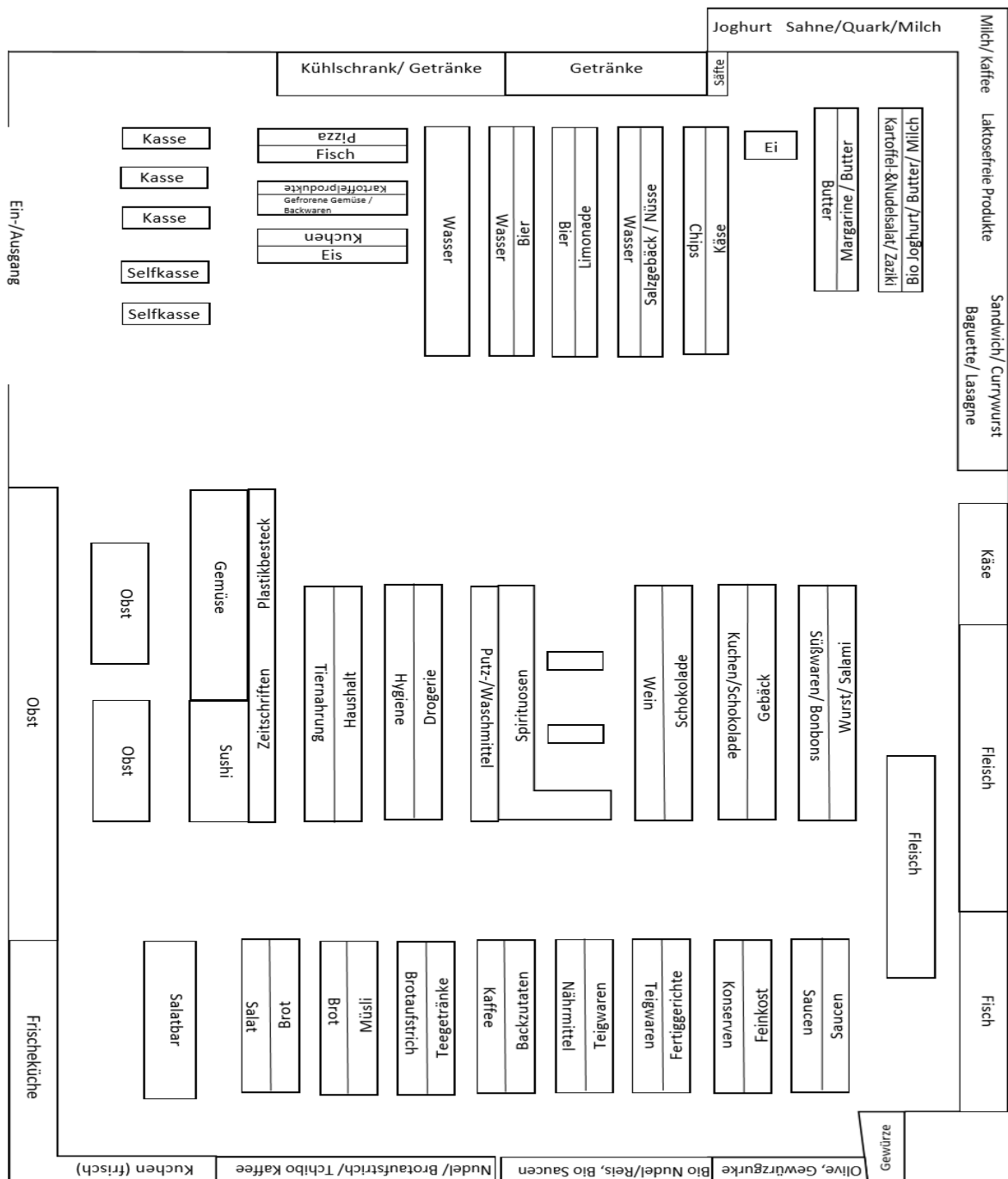


Abbildung 9: Skizze von Dornseifer

Nachdem der Nutzer eine Einkaufsliste erstellt und zur Berechnung freigegeben hat, wird diese zusammen mit den Ladendaten (Artikelstandorte etc.) an den Scheduler übergeben. Anschließend wird in regelmäßigen Abständen das aktuelle Zwischenergebnis, welches im Scheduler vorliegt, abgerufen und live im Userinterface angezeigt. Nachdem die gesamte Berechnung abgeschlossen ist, erhält man den besten Weg von dem Scheduler zurück und die Einkaufsliste wird sortiert ausgegeben. Um den optimalen Weg abzugehen, sollte der Benutzer die Einkaufsliste in der angezeigten Reihenfolge abarbeiten.

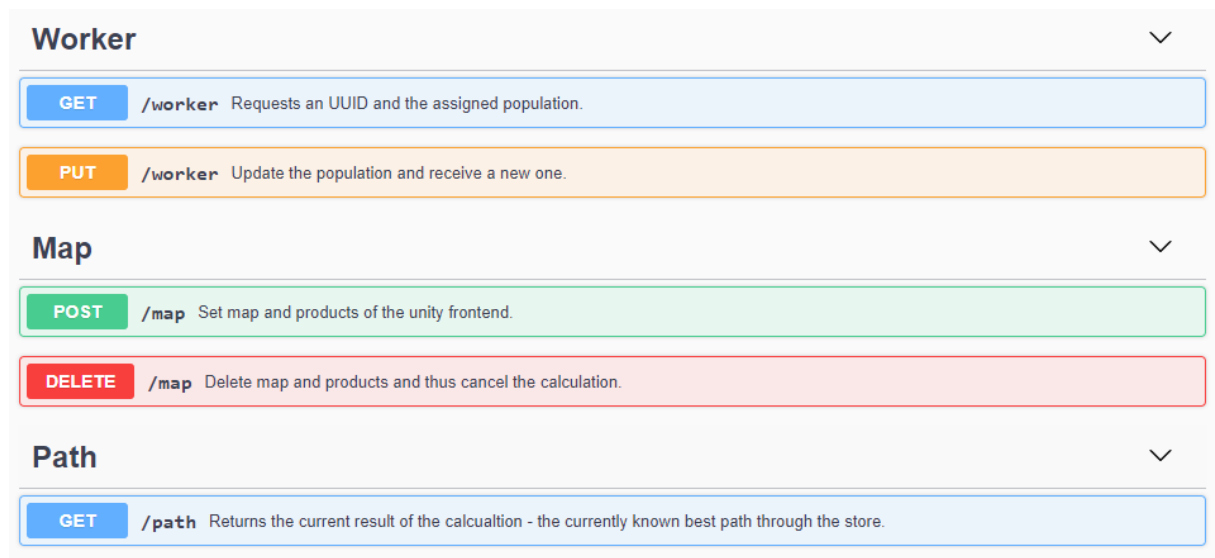
5.2 Scheduler

Der Scheduler bildet die Schnittstelle zwischen der Unity-Instanz und den einzelnen Workern ab. Über eine REST-API (siehe Scheduler Ressourcentabelle) können sich unter anderem die Worker anmelden und die Daten aktualisieren.

Hat der Scheduler eine Produktliste erhalten, erzeugt dieser eine Masterpopulation. Worker erhalten auf Anfrage folgend Subpopulationen zur Bearbeitung. Ist ein Worker mit der Bearbeitung einer Subpopulation fertig, sendet er diese an den Scheduler zurück und erhält eine neue Subpopulation zur Bearbeitung. Bei jeder Aktualisierung einer Subpopulation, werden Gene angelehnt an das Nachbarschaftsmodell (siehe Kapitel Nearest-neighbor migration) ausgetauscht. Dadurch wird gewährleistet, dass nicht alle Worker dieselbe Population besitzen. Dies könnte vor allem dann auftreten, wenn die Einkaufsliste relativ klein ist.

5.2.1 API-Aufbau

Die OpenApi 3.0 Dokumentation des Schedulers (siehe Abbildung 10) beschreibt, auf welche Daten die einzelnen Komponenten Zugriff haben und deren Beziehung.



Worker ▼	
GET	<code>/worker</code> Requests an UUID and the assigned population.
PUT	<code>/worker</code> Update the population and receive a new one.
Map ▼	
POST	<code>/map</code> Set map and products of the unity frontend.
DELETE	<code>/map</code> Delete map and products and thus cancel the calculation.
Path ▼	
GET	<code>/path</code> Returns the current result of the calculation - the currently known best path through the store.

Abbildung 10: OpenApi 3.0 Dokumentation des Schedulers

Die Dokumentation bildet für alle möglichen REST-Aufrufe detaillierte Informationen zu erwarteten Parametern, erfolgreicher Antwort oder möglicher Fehlercodes übersichtlich ab. Ein Auszug für die Path Ressource wird in Abbildung 11 dargestellt.

GET
/path
Returns the current result of the calculation - the currently known best path through the store.

Parameters

Try it out

No parameters

Responses

Code	Description	Links
200	<p>Currently the best path. If Path is set in addition to the DemoPath, the result is the preliminary end result.</p> <div> Media type <div>application/json</div> Controls Accept header. </div> <div> Example Value Schema </div> <pre> { "Items": [{ "position": { "x": "51.04", "y": "7.23" }, "name": "Sushi" }], "DemoItems": [{ "position": { "x": "51.04", "y": "7.23" }, "name": "Sushi" }], "Distance": 310 } </pre>	No links
204	The worker have not yet delivered a result.	No links
503	There are currently no registered workers working on a solution.	No links

Abbildung 11: Beschreibung der Path Ressource

5.2.2 Technologie

Der Scheduler wurde in Kotlin implementiert. Kotlin wird von JetBrains entwickelt und ist eine statisch typisierte Programmiersprache, welche unter anderem in Java Byte Code übersetzt werden kann und folglich auf der Java Virtual Maschine lauffähig ist. Mit Veröffentlichung im Jahre 2011 ist Kotlin im Verhältnis zu Java oder C eine junge Sprache. Trotz dieser kurzen

Lebenszeit ist Kotlin bereits weit verbreitet und wird etwa in der Android Entwicklung als bevorzugte Sprache von Google empfohlen.¹⁴

Auch im Backend kann Kotlin eingesetzt werden und so bietet z.B. das Ktor Framework eine einfache Möglichkeit asynchrone Server- und Clientsysteme zu entwickeln. Auch in diesem Projekt wurde Ktor genutzt, um den Scheduler als Webserver zu implementieren, da im Team bereits Einsatzerfahrung existierte.

5.2.3 Genutzte Kotlin Features

Kotlin stellt verschiedene spezielle Features bereit, welche während der Implementierung des Schedulers genutzt wurden. Einige werden zu Beginn anhand ihrer Codebeispiele zum besseren Verständnis erläutert.

Beispielsweise sind Funktionen höherer Ordnung, also Funktionen, welche selbst Funktionen als Übergabeparameter entgegennehmen können in Kotlin nativ implementiert. Zwar ist dies auch seit der Version 8 in Java möglich, allerdings bietet Kotlin durch die native Implementierung einige Quality of Live Features mehr, welche während der Entwicklung direkt berücksichtigt werden konnten.

Extensionfunctions sind Funktionen, welche eine bereits vorhandene Klasse erweitern können, ohne dass eine explizite Vererbungshierarchie o. ä. angelegt werden muss. So war es im Projekt möglich die Klasse `MutableList`, um eine zusätzliche `get` Methode zu erweitern (siehe Abbildung 12), welche anstelle des Index, wie bei der Standardimplementierung, eine Bedingung entgegennehmen kann.

```
3  /**
4   * @return the first found element matching the [condition] or null if none was found
5   */
6  fun <T> MutableList<T>.get(condition: (T) -> Boolean): T? {
7      this.forEach { it: T
8          if (condition(it)) return it
9      }
10
11     return null
12 }
```

Abbildung 12: Extensionfunction für `MutableList`

¹⁴ Vgl. Haase, Chet 2019: Google I/O 2019: Empowering developers to build the best experiences on Android + Play @ONLINE. URL: <https://android-developers.googleblog.com/2019/05/google-io-2019-empowering-developers-to-build-experiences-on-Android-Play.html> (Stand: 18.06.2019)

Durch die zusätzliche Methode war es im weiteren Verlauf möglich, übersichtlicheren Code zu erstellen. So konnte diese genutzt werden, um ein Workerobjekt aus der Liste aller Worker nur mit Angabe der UUID zu erhalten, wie in Abbildung 13 zu sehen ist.

```
val worker = scheduler.workers.get { it.uuid == UUID.fromString(workerId) }
```

Abbildung 13: Nutzung der Extensionfunktion

Die letzten und wichtigsten Features von Kotlin, welche hier erwähnt werden sollen, sind Coroutines und suspend Funktionen. In der heutigen Zeit, und besonders bei Serveranwendungen ist es wichtig, dass Server asynchron arbeiten können, um auch große Lasten performant managen zu können. Kotlin bietet mittels Coroutines Entwicklern die Möglichkeit, synchronen Code zu schreiben, welcher asynchron lauffähig ist. Kotlin selbst sorgt im Hintergrund dafür, dass Abhängigkeiten eingehalten werden. Ein Beispiel für die Nutzung von Coroutinen kann im REST-Service gefunden werden. Alle Funktionen, welche Clientanfragen verarbeiten sind suspend (siehe Abbildung 14) und können so asynchron und mehrfach vom Server ausgeführt werden.

```
private suspend fun addWorker(call: ApplicationCall) {
```

Abbildung 14: Header der addWorker Funktion

5.3 Worker

Der Worker steht ständig in Verbindung mit dem Scheduler, um neue Individuen abfragen zu können. Der Scheduler liefert den Worker eine "Start" Sub Population und daraufhin rechnet der Worker, mittels genetischen Algorithmus und Travelling Salesman Problem eine neue Sub Population und liefert diese an den Scheduler. Die durch den Worker ermittelte Distanz von den Individuen der Population, kann der Worker den besseren Pfad ermitteln. Zurzeit werden 100 Generationen verwendet, es wurde bei der Anzahl von Generationen darauf geachtet, dass die Zahl nicht so groß ist und auch nicht so klein. Denn kleine Anzahl von Generationen führt dazu, dass ein "idealer" Ergebnis verpasst wird. Eine große Anzahl von Generationen führt dazu, dass man auf Dauer neue Generationen generiert, obwohl das beste Ergebnis schon erreicht wurde (Verschwendete Leistung). Es wurde also anhand mehrerer Ausführungen von verschiedener Anzahl der Generationen festgestellt, dass die Anzahl 100 für den Worker ideal ist. Anschließend dazu, achtet der Worker anhand der Mutationsrate, wie oft eine Mutation ausgeführt werden soll. Eine Mutation wird für den Fall benötigt, dass sich die Pfade

bzw. Individuen nicht mehr verbessern. In diesem Fall wird “mutiert”, in dem man die Individuen zufällig neu generiert. Diese werden dann in dem weiteren Verlauf der Berechnung verwendet, um eventuell bessere Ergebnisse erzielen zu können.

Der Worker hat keine eigene Schnittstelle. Die einzige Kommunikation, die vom Worker ausgeht, ist mit dem Scheduler. Es benutzt die Schnittstellen von Kapitel 5.2.1 API-Aufbau (siehe Ressource: “Worker”).

6 Implementierung

6.1 Unity-Frontend

6.1.1 Erstellung des Geschäfts

Die Grundlage für die Berechnung eines optimalen Weges, um alle Artikel von der Einkaufsliste möglichst zügig einzukaufen, ist in dem Unity-Frontend gesetzt. In Unity sind alle Geschäfte, für die der Nutzer eine Berechnung durchführen kann, in einer 3D-Welt modelliert. Die Planung zur Abbildung des Geschäfts wurde dazu bereits in Kapitel 5.1 durch das Anfertigen einer Skizze vollzogen. Es wurden nun alle Regale mit der im Verhältnis zur 3D-Welt ungefähren echten Größe und deren Produkt bzw. Produktkategorie so angeordnet, dass diese die Struktur des echten Geschäfts widerspiegeln. Dabei ist ein Meter in Unity ebenfalls auf eine Längeneinheit übersetzt. Die folgende Abbildung 15 zeigt das für das Projekt modellierte Geschäft „Dornseifer“ in der 3D-Welt.



Abbildung 15: Modelliertes Geschäft in Unity

Wenn über das UI, mehr dazu im Kapitel 6.1.2, die Berechnung gestartet wird, transformiert das Unity-Frontend die Einkaufsliste mit allen relevanten Objekten (z.B. Regale, Kassen, etc.) in ein für die Berechnung geeignetes JSON-Format. Das JSON beinhaltet dann alle Objekte mit ihren Produkten und Positionen in der 3D-Welt.

Allerdings gehen bei der Transformation der 3D-Welt in das JSON-Format einige Eigenschaften dieser relevanten Objekte verloren. Bei den meisten ist dies nicht weiter von Wichtigkeit, da die Berechnung ohnehin nicht von den Eigenschaften, wie beispielsweise Regalhöhe, beeinflusst wird. Jedoch gehen ebenfalls die Informationen verloren, wie die Struktur des Geschäftes und die damit verbunden Laufwege und Blockaden aussehen. Diese Informationen sind keineswegs irrelevant für die Berechnung des optimalen Weges, da ohne diese eine Berechnung mit Luftlinien durchgeführt würde. Das hat zur Folge, dass bei der Berechnung des optimalen Weges nicht nur davon ausgegangen wird, dass der Benutzer über die Regale klettert, sondern im Falle eines mehrstöckigen Gebäudes auch durch die Decke bzw. den Boden gehen kann. Um dem Vorzubeugen, wird neben der Einkaufsliste in dem JSON auch ein zuvor in Unity modellierter Graph mitgegeben, welcher die ungefähren realen Laufwege abbildet. Auf Grundlage dieses Graphen kann dann bei der Berechnung die reale Distanz anstelle der Luftliniendistanz berechnet und verwendet werden.

Der Graph setzt sich in Unity aus einer Vielzahl an unsichtbaren Wegpunkten zusammen, welche jeweils alle ihre Nachbarpunkte kennen. Zusätzlich zu den unsichtbaren Wegpunkten besitzt jedes Regal bzw. Theke ebenfalls einen eigenen dieser Wegpunkte, welche den Interaktionspunkt mit dem Regal definiert und sich in den Graphen nahtlos eingliedert. Bei der Transformation wird jedem Wegpunkt die eindeutige Instanz-ID, welche von Unity selbst an alle Objekte vergeben wird, verwendet, um die Nachbarpunkte im JSON zu referenzieren. Alle Nachbarpunkt-Referenzen werden in Unity während der Platzierung bzw. Änderung der Wegpunkte und Regale automatisch auf Grundlage der Distanz und strukturellen Blockaden ermittelt. Somit müssen bei der Modellierung des Geschäfts nur die Knoten, jedoch nicht die Kanten des Graphen manuell erstellt werden. Die folgende Abbildung 16 zeigt die Übersicht des Graphen in Unity (links im Bild, lila Linien stellen Kanten und graue Punkte Knoten des Graphen da) und das daraus resultierende JSON (rechts im Bild).

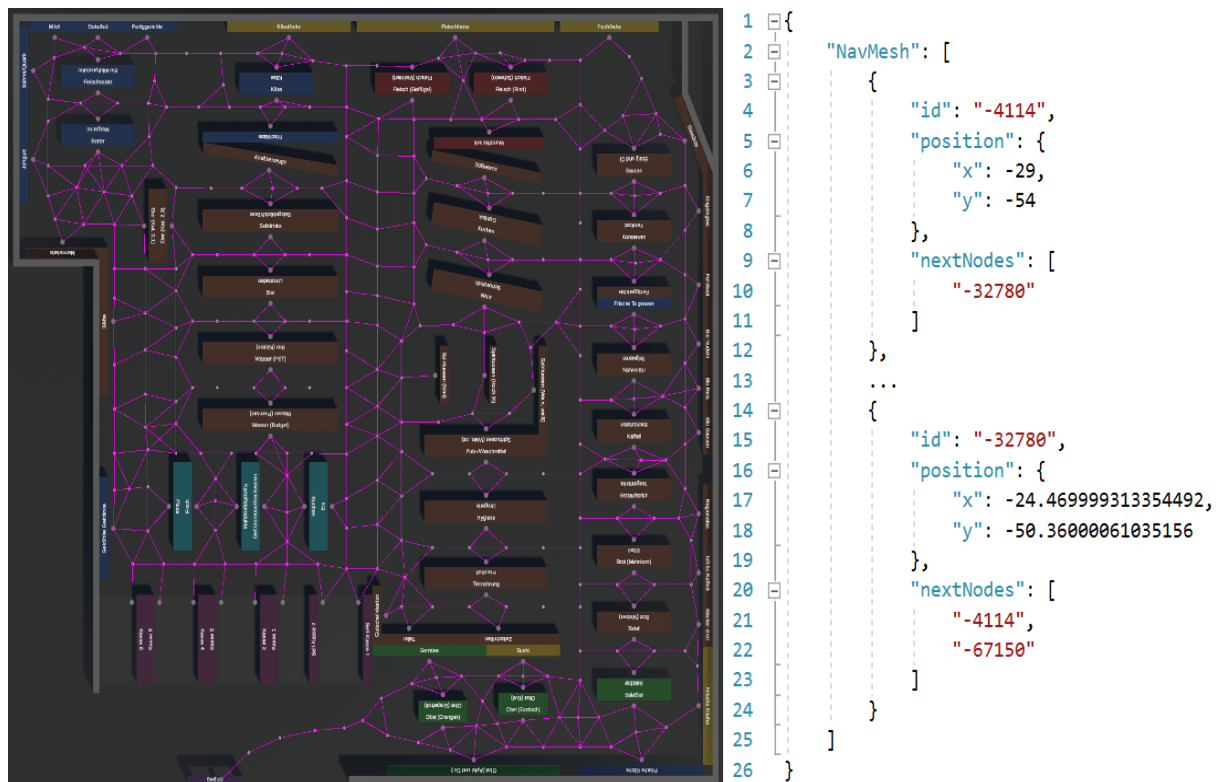


Abbildung 16: Wegegraph anstelle von Luftlinien

6.1.2 Benutzerinterface

Das Benutzerinterface (UI) stellt die Schnittstelle zwischen Nutzer und der gesamten Anwendung dar. So landet der Nutzer beim Starten des Unity-Frontends auf der Übersichts-Ansicht des modellierten Geschäftes (vgl. Abbildung 17). In der oberen linken Ecke der Übersicht findet sich ein Bereich, in welchem während und nach einer Berechnung verschiedene Statistiken zur Berechnung angezeigt werden. Rechts oben hat der Nutzer die Möglichkeit, über einen Button den Einkauf zu planen. Direkt darunter befindet sich zur einfachen Test-Handhabung ein Eingabefeld für die IP-Adresse des Schedulers. Unten rechts befindet sich abschließend noch eine Legende, welche die verschiedenen Farben der Regale und anderen modellierten Objekte der Geschäfts-Übersicht erläutert.



Abbildung 18: Einkaufslisten-Planer

Beim Öffnen des in Abbildung 18 gezeigten Planungs-UI wird die linke Liste mit den Produkten des Geschäfts immer dynamisch gefüllt. Das heißt, wenn die Modellierung des Ladens geändert wird (z.B. Produkte neu hinzukommen oder Regale wegfallen), müssen diese Änderungen nicht noch in der UI-Liste vorgenommen werden.

In der Awake-Methode des UIs wird die dynamische Erstellung des Laden-Inventars durchgeführt (vgl. Abbildung 19). Der gesamte Quellcode kann auf [GitHub](#) eingesehen werden. Diese Methode wird automatisch von Unity aufgerufen, wenn ein Objekt instanziiert wird. Zu Beginn der Methode werden dazu alle Regale des Geschäftes abgerufen. Dies wird über die `GetComponentInChildren`-Methode von Unity bewerkstelligt. Diese liefert eine Liste an allen Objekten zurück, die in diesem Fall von Typ `Shelf` sind und hierarchisch dem Objekt `shelfParent` untergeordnet sind. Die hierarchische Ordnung wird im Editor von Unity in der sogenannten Hierarchy festgelegt. In dieser sind alle Objekte einer Szene vorhanden und gruppiert. So sind alle Regale des Geschäftes in einem Container zusammen gruppiert, welcher als `shelfParent` der gezeigten Awake-Mode bekannt ist. Nach dem Holen aller Regale wird in Zeile 31 die Liste alphabetisch sortiert, um dem Nutzer das Suchen von bestimmten Produkten zu vereinfachen. In dem `foreach`-Loop in Zeile 34 werden dann für alle Regale die Schaltflächen erstellt (Z. 35), konfiguriert (Z. 37) und in die UI-Liste gelegt (Z. 36).

In Zeile 40 wird die Einkaufsliste einer neuen, leeren Liste zugewiesen. In dieser werden die Referenzen der Produkte gespeichert, welcher der Nutzer in seine Einkaufsliste transferiert hat, um daraus das JSON für die Berechnung zu erstellen.

Die Zeilen 44 bis 47 und 49 bis 52 fügen zu der Einkaufsliste noch die nicht anklickbaren Elemente „Eingang“ und „Kasse x“ zu, um das Setup des Planungs-UI zu vollenden.

```
29 private void Awake() {
30     var shelves = new List<Shelf>(shelfParent.GetComponentInChildren<Shelf>());
31     shelves.Sort((x, y) => x.productName.CompareTo(y.productName)); //Sort list alphabetical
32
33     //For every shelf add a button in the list planner
34     foreach(var shelf in shelves) {
35         var item = Instantiate(itemPrefab);
36         item.transform.SetParent(shopViewPortContent);
37         item.Setup(this, shelf, item.transform.GetSiblingIndex());
38     }
39
40     shoppingList = new List<ShopAsset>();
41
42
43     //Add the entrypoint and checkout to the shopping list as not clickable buttons
44     entrypointBtn = Instantiate(itemPrefab);
45     entrypointBtn.transform.SetParent(listViewPortContent);
46     entrypointBtn.Setup(this, GetEntrypoint(), -1);
47     entrypointBtn.buttonComponent.interactable = false;
48
49     checkoutBtn = Instantiate(itemPrefab);
50     checkoutBtn.transform.SetParent(listViewPortContent);
51     checkoutBtn.Setup(this, ChooseRandomCheckout(), -1);
52     checkoutBtn.buttonComponent.interactable = false; ;
53 }
```

Abbildung 19: Initialisierung des Einkaufslisten-Planers

6.1.3 Scheduler-Aufruf

Nachdem der Nutzer die Einkaufsliste zusammengestellt und die Berechnung gestartet hat, wird über die UI-Komponente die Transformation der Liste in ein JSON-Objekt, das Hinzufügen des Wegegraphen und das Senden der Daten an den Scheduler in der Methode in Abbildung 20 ausgelöst. Dazu werden zu Beginn alle offenen UI-Fenster geschlossen (Z. 23) und ein Textfeld mit dem Hinweis, dass die Berechnung läuft, angezeigt (Z. 24, 25). Um eine Kapselung der im REST-Aufruf erforderlichen Daten und der der 3D-Welt herzustellen, werden die jeweiligen Objekte der Einkaufsliste in Zeile 28 in eine extra Klasse, die die Struktur des JSON widerspiegelt kopiert. Das Senden der Daten an den Scheduler geschieht dann in Zeile 35. Die Klasse SchedulerRestClient ist nach einem Singleton-Pattern aufgebaut. Dies erlaubt es, über das statische Feld Instance auf die Instanz dieser einen Klasse zuzugreifen. Die Klasse selbst ist nicht statisch, da in der Klasse für die Rest-Aufrufe jeweils eine Coroutine von Unity verwendet werden muss, welche nicht in statischen Klassen verfügbar ist. Mitgegeben werden neben der kopierten Einkaufsliste und der Scheduler Adresse noch zwei Methodenverweise (Delegates), welche sowohl während der Berechnung als auch nach dem Erhalt des endgültigen Rechenergebnisses aufgerufen werden.

```

22 public void StartCalculation() {
23     CloseAllUis();
24     loadingPanel.GetComponentInChildren<TextMeshProUGUI>().text = "Berechnung läuft...";
25     loadingPanel.SetActive(true);
26
27     //Create node list and set the statics for waypoint count
28     var nodes = NodeModel.CreateList(plannerPanel.GetShoppingList());
29     statisticsUI.UpdateWaypointCnt(nodes.Count);
30
31
32     Debug.Log("Posting shopping list...");
33     var hostUrl = schedulerIpField.text;
34
35     SchedulerRestClient.Instance.StartCalculationForShoppinglist(nodes, hostUrl, ProcessIntermediateResult, ProcessCalculationResult);
36     statisticsUI.ResetAndStartTimer();
37 }

```

Abbildung 20: Starten der Berechnung

In der SchedulerRestClient-Klasse wird nach dem Senden der Daten an den Scheduler automatisch solange nach dem Endergebnis gefragt, bis dieses vorliegt, oder die Berechnung vom Nutzer abgebrochen wurde. Dies geschieht in der Methode in der Abbildung 21. Die Methode selbst wird als Coroutine ausgeführt, um ein Einfrieren der Anwendung beim Warten auf eine REST-Response zu verhindern. Die Scheduler-IP, sowie die zwei Methodenverweise, welche während und nach der Berechnung ausgeführt werden sollen, werden an diese von den vorherigen Aufrufen weitergegeben.

Das Fragen nach dem Ergebnis wird in einer Endlos-Schleife in Zeile 111 durchgeführt. In den darauffolgenden Zeilen 112 und 113 wird nach dem Ergebnis gefragt. Wenn der Response „200 OK“ zurückkommt, wird in Zeilen 118 und 119 der Response-Body in eine Klasse geparsed und als result gesetzt. Mit diesem ersten Ergebnis wird dann die Methode aufgerufen, welche als intAction (intermediate Action) übergeben wurde. Als Zwischenaktion wird beispielsweise das aktuelle Zwischenergebnis im UI angezeigt. Wenn in der Response das Endergebnis vorliegt, die Items-Liste nicht null ist, wird die Endlosschleife beendet und die Methode mit der Endergebnis-Aktion wird ausgeführt. Solange kein Endergebnis vorliegt und der Nutzer die Berechnung nicht abgebrochen hat, wird einen kurzen Moment gewartet (Z. 129), bis erneut nach dem Ergebnis gefragt wird.

```

107 private IEnumerator QueryCalculationResult(string hostUrl, Action<PathResponse> intAction, Action<PathResponse, bool> actionOnResult) {
108     Debug.Log("Checking for result...");
109     PathResponse result = null;
110
111     while(result == null && result.Items == null && !cancelCalculation) {
112         var request = CreateGetCalculatedWaypointsRequest(hostUrl);
113         yield return request.SendWebRequest();
114
115         if(!request.isNetworkError && request.responseCode == 200) {
116             Debug.Log("Got result.");
117
118             var response = Encoding.UTF8.GetString(request.downloadHandler.data);
119             result = JsonUtility.FromJson<PathResponse>(response);
120
121             if(result != null) intAction(result);
122
123         } else if(request.isNetworkError) {
124             Debug.LogWarning($"Can't get result. Network-Error: {request.isNetworkError}, Response-Code: {request.responseCode}");
125             break;
126         }
127     }
128
129     yield return new WaitForSeconds(delayBetweenRequests);
130 }
131
132 calculationActive = false;
133 actionOnResult(result, cancelCalculation);
134
135 cancelCalculation = false;
136 }

```

Abbildung 21: Abfragen des aktuellen Rechenergebnisses

6.1.4 Visualisierung

Sobald ein Rechenergebnis vorliegt (dabei werden sowohl Zwischenergebnisse als auch das Endergebnis berücksichtigt), wird der berechnete Weg in Unity dem Nutzer angezeigt. So kann unter anderem während der laufenden Berechnung der sich ändernde Weg beobachtet werden, als auch nach der Berechnung das Ergebnis in Form von Linien, die sowohl die Luftlinien als auch den Laufweg des Ergebnisses in dem Geschäft darstellen. Neben diesen Linien werden in dem Statistik-Bereich noch Informationen, wie z.B. die berechnete Gesamtdistanz, die reale Distanz, die Zeit vom Abschicken der Daten bis zum Erhalt des Ergebnisses und noch weitere angezeigt. Die Abbildung 22 stellt einen beispielhaften Überblick über die genannten Funktionalitäten.



Abbildung 22: Visualisierung des Rechenergebnisses

Das Anzeigen der Linien wird von einer LineRenderer-Komponente übernommen. Diese ist einer der Komponenten, die von Unity bereitgestellt werden. Vereinfacht erläutert, wird der LineRenderer-Komponente ein Array an Positionen gegeben, über die dieser dann die Linie aufspannt. Die Zusammenstellung des Positions-Arrays geschieht in der Methode in Abbildung 23. Diese wird bei Erhalt des Endergebnisses mit den sortierten Wegpunkten, welche einen optimalen Weg repräsentieren, aufgerufen. Die von der Berechnung stammenden Wegpunkte sind allerdings nur basierend auf dem in Kapitel 6.1.1 erstellen Graphen, wodurch diese nicht den echten kürzesten Laufweg widerspiegeln. Aus diesem Grund werden auch zwei verschiedene Linien in Abbildung 22 angezeigt.

Die türkise Linie ist aus exakt den Wegpunkten, die aus der Berechnung stammen erstellt worden (nicht in der Methode in Abbildung 23 gezeigt) und die grüne Linie spiegelt den Weg wider, welcher dem kürzesten realen Weg in Unity selbst entspricht. Dieser wird in der Schleife in Zeile 49 in Abbildung 23 berechnet. Dazu wird von jedem aus dem Ergebnis stammenden Wegpunkt der NavMeshPath, eine Unity-Repräsentation eines Weges von Punkt A nach Punkt B über den Unity-Internen Wegegraphen, berechnet (Z. 50). Der NavMeshPath selbst beinhaltet dann eine Liste von Wegpunkten, welche die Ecken und Abbiegungen des Unity-Weges definieren, über welche dann in einer weiteren Schleife iteriert wird (Z. 52). In dieser Schleife

werden die Wegpunkte des NavMeshPath mit einem Höhenoffset versehen, damit die angezeigte Linie nicht im Boden, und damit auch sichtbar, verläuft (Z. 53). Nachdem alle Punkte für die Linie berechnet sind, werden diese dem LineRenderer in den Zeilen 58 und 59 übergeben. Der Aufruf der Simplify-Methode in Zeile 61 vereinfacht die Liste mit den Punkten, wenn Punkte dicht beieinander liegen, wodurch die Linie glatter verläuft.

```
44 public void DisplayNavPath(List<Vector3> waypoints) {
45     List<Vector3> linePoints = new List<Vector3>();
46     NavMeshPath path = new NavMeshPath();
47
48     //Calculate a path from each self to the next one
49     for(int i = 0; i < waypoints.Count - 1; i++) {
50         if(NavMesh.CalculatePath(waypoints[i], waypoints[i + 1], NavMesh.AllAreas, path)) {
51             foreach(var point in path.corners) {
52                 linePoints.Add(new Vector3(point.x, point.y + hightOffsetPath, point.z));
53             }
54         }
55     }
56
57     rendererPath.positionCount = linePoints.Count;
58     rendererPath.SetPositions(linePoints.ToArray());
59
60     rendererPath.Simplify(1);
61 }
62
```

Abbildung 23: Berechnung der Visualisierung

6.2 Scheduler

6.2.1 Der Ktor-Server

Wie bereits im Kapitel 5.2.2 beschrieben, handelt es sich bei Ktor um ein Kotlin-Server-Framework. Ktor kann leicht über Einbinden der entsprechenden Abhängigkeiten in Gradle eingebunden werden. Hierbei musste bereits eine Entscheidung getroffen werden, da Ktor eine Abstraktion für verschiedene Server-Engines wie Netty, Apache, CIO, u. W. bietet. Es wurde sich für die Netty-Engine entschieden, da diese alle benötigten Grundfunktionalitäten eines http-Servers abdeckt.

Nach der Einbindung kann der Server u.a. mittels der `embeddedServer`-Funktion erstellt und konfiguriert werden, wie in Abbildung 24 zu sehen ist. Zur Wahrung der Übersichtlichkeit wurde die Beschreibung der Routen in ein eigenes Objekt verlagert.

```

1  import io.ktor.server.engine.embeddedServer
2  import io.ktor.server.netty.Netty
3  import scheduler.RestService
4
5  fun main() {
6      embeddedServer(Netty, port = 8080) { this: Application
7          RestService.initRouting( application: this)
8      }.start(wait = true)
9  }

```

Abbildung 24: Initialisierung und Start des Servers

Im RestService können die einzelnen Routen, welche in Kapitel 5.2.1 API-Aufbau in der REST-Tabelle beschrieben sind, abgebildet werden. Ktor stellt für alle http-Verben entsprechende Funktionen zur Verfügung, welche auf http/s-Anfragen des konfigurierten Ports horchen und diese bearbeiten. Die Abbildung 25 zeigt die GET-Anfrage, welche auf einen möglicherweise übergebenen Query-Parameter UUID prüft und entsprechende suspend-Methoden zur Weiterverarbeitung aufruft.

```

45  get( path: "/worker") { this: PipelineContext<Unit, ApplicationCall>
46      logRequest(call)
47      call.parameters["uuid"]?.let { respondPopulation(call, it) } ?: addWorker(call)
48  }

```

Abbildung 25: GET /worker Definition

Über den weitergereichten call-Parameter vom Typ ApplicationCall besteht die Möglichkeit auf alle benötigten Informationen wie Requestbody oder sonstige zuzugreifen. Abbildung 26 zeigt, die logRequest Funktion, welche mithilfe des ApplicationCalls alle relevanten Daten zum Anfragenden in die Konsole schreibt.

```

319  private fun logRequest(call: ApplicationCall) {
320      println("${call.request.httpMethod.value} ${call.request.path()} from ${call.request.origin.remoteHost}")
321  }

```

Abbildung 26: Umsetzung der logRequest-Funktion

6.2.2 OpenAPI 3.0 Dokumentation

Die OpenAPI Spezifikation dient zur Beschreibung von REST-Schnittstellen. Zur Gewährleistung einer guten Übersicht und einer einfachen Nutzung des Schedulers wurde eine API-Dokumentation nach OpenAPI 3.0 Standards angefertigt. Für die Erstellung der Spezifikation gibt es verschiedene Möglichkeiten. Wählt man bei der Entwicklung einen API-First-Ansatz,

existieren einige Tools, die aus handgeschriebenen Spezifikationsdateien unterschiedlichste Servertemplates generieren. Da zum Zeitpunkt der Entscheidung für OpenAPI bereits der Server zu großen Teilen programmiert war, wurde gegen diese Möglichkeit entschieden. Viele große Serverframeworks, wie bspw. Spring Boot bieten integrierte Lösungen, um z. B. mittels Annotationen aus dem erstellten Code eine API-Dokumentation zu generieren. Dies ist besonders im produktiven Umfeld hilfreich, in welchem neue Endpunkte hinzukommen können.

Bei Kotlin und Ktor handelt es sich um eine recht junge Programmiersprache und ein noch jüngeres Framework, wodurch Ktor bisher keine native Implementierung bietet. Die aktive Community hat allerdings bereits verschiedene Implementierungen wie ktor-swagger¹⁵ oder Ktor-OpenAPI-Generator¹⁶ bereitgestellt. Nach kleineren Testimplementierungen musste allerdings festgestellt werden, dass diesen noch verschiedene Features fehlen. So ist es beim Ktor-OpenAPI-Generator momentan nicht möglich, unterschiedliche HTTP-Status-Codes mit entsprechender Beschreibung aus dem Code zu generieren. Aufgrund dieser Sachlage wurde auf die automatische Generierung der Dokumentation verzichtet und diese wurde manuell nach den OpenAPI 3.0 Standards erstellt. Weiterhin spricht für die manuelle Dokumentation, dass es sich bei diesem Projekt nicht um ein produktives System handelt, in welchem die API-Dokumentation schnelllebig sein können.

6.2.3 Einblick in den Code

Wird der Scheduler gestartet, wartet dieser auf Anfragen vom Unity-Frontend oder von den Workern. In dem normalen Ablauf schickt das Unity-Frontend als erstes eine Liste von Einkaufslistenelementen und zusätzlich noch Navigationspunkte als JSON. Dort sieht man Einkaufslistenelemente, die sich in *Items* befinden. Items besitzen als erstes Element immer den *Eingang* und als letztes Element immer eine *Kasse*.

Zusätzlich zu den Einkaufslistenelementen wird auch ein Wegegraph in Form des NavMesh mitgeschickt. Dies wurde nachträglich hinzugefügt und beschreibt, wie in Kapitel 6.1 erläutert das zusätzliche Navigationsgitter, das aufgrund des Problems mit der Berechnung über Luftlinien (siehe Kapitel 7.3 Probleme Unity) entstanden ist.

¹⁵ Implementierung durch nielsfalk vgl. <https://github.com/nielsfalk/ktor-swagger>

¹⁶ Implementierung durch papsign vgl. <https://github.com/papsign/Ktor-OpenAPI-Generator>

```

204  /**
205   * save sent json to map
206   */
207  private suspend fun saveMap(call: ApplicationCall) {
208      try {
209          val string = call.receiveTextWithCorrectEncoding()
210          val unityData = gson.fromJson(string, UnityMapStructure::class.java)
211
212          scheduler.products = unityData.products
213          scheduler.map = unityData.navMesh
214          scheduler.calculationRunning = true
215      } catch (e: Exception) {
216          println("Could not read map")
217          e.printStackTrace()
218      }
219
220
221      ensureMapAndProducts { products, navMesh ->
222          scheduler.createPopulation(products)
223          call.respondText(gson.toJson(UnityMapStructure(products, navMesh)), ContentType.Application.Json, HttpStatus.OK)
224      } ?: respondJsonError(call)
225  }
226

```

Abbildung 27: Abspeichern der Einkaufslistenelemente und des Navigationsgitters

Wie man der Abbildung 27 entnehmen kann liest der Scheduler das JSON ein und speichert beide Komponenten in `products` bzw. `map` ab. Sobald beides im Scheduler vorhanden ist, wird der `HttpStatusCode OK` zurückgesendet und mithilfe der Einkaufslistenelemente eine Masterpopulation erstellt (Abbildung 27 Zeile 221 ff.).

Um später die Populationen angemessen verteilen zu können, wird die Größe der Population im Vorfeld durch die Konstante `POPULATION_SIZE` definiert. Diese Konstante besitzt eine Abhängigkeit zu der maximalen Anzahl der erwarteten Worker und zu der Anzahl der Einkaufslistenelemente. Nähere Informationen zur Bestimmung dieser Konstante befinden sich in Kapitel 7.2 Probleme Scheduler.

Nachdem die Masterpopulation erfolgreich erstellt worden ist, wird diese in Subpopulationen aufgeteilt, damit jeder Worker eine Subpopulation erhalten kann. Um dies zu erreichen, wird eine Subpopulationsgröße wie in Abbildung 26 definiert.

```

val subPopSize = populationSize / (WORKER_COUNT+1)

```

Abbildung 28: Definition der Größe der Subpopulationen

Mithilfe dieser Größe werden `(WORKER_COUNT+1)` Subpopulationen der Größe `subPopSize` erstellt und mit den Elementen aus der Masterpopulation gefüllt. Da die Reihenfolge hierbei keine Rolle spielt, wird vereinfacht immer das erste Element genommen.

Währenddessen können sich Worker beim Scheduler anmelden. Bei der Anmeldung wird dem jeweiligen Worker eine UUID und eine zufällige Subpopulation zugeordnet. Außerdem wird die

IP-Adresse und die derzeitige Zeit in einem Worker Objekt abgespeichert (siehe Abbildung 29).

```
val worker = Worker(UUID.randomUUID(), workerAddress, subPopulation, LocalDateTime.now())
```

Abbildung 29: Initialisierung des Workers beim Scheduler

Wurde der Worker erfolgreich erstellt, wird der http Status OK zurückgegeben.

Der Worker erhält eine Subpopulation und den Wegegraph, damit dieser arbeiten kann. Sind alle Berechnungen auf Seiten des Workers abgeschlossen, wird die Population dem Scheduler mitgeteilt.

Sowohl die einzelnen Individuen der mitgeschickten Population als auch das beste Individuum werden aktualisiert und abgespeichert (siehe Abbildung 30).

```
if (scheduler.subPopulations.any { subPop -> subPop.worker == it }) {  
    it.subPopulation.updateIndividuals(parsedPopulation)  
    scheduler.updateBestIndividual(it.subPopulation)  
  
    it.changePopulation(newPopulation)  
    scheduler.evolvePopulation()  
} else {  
    it.changePopulation(newPopulation)  
}
```

Abbildung 30: Aktualisieren des Workers beim Scheduler

Bei der Abspeicherung des besten Individuums musste ein Kriterium gefunden werden, so dass das Unity-Frontend nicht gleich das erstbeste Ergebnis erhält und dieses anzeigt. Es wurde sich für zwei weitere Konstanten (**MIN_DELTA** und **DEMO_INDIVIDUAL_SIZE**) entschieden. Mithilfe dessen wird bestimmt, ob das "beste Individuum" das aller beste Individuum ist oder nur ein temporäres bestes Individuum. **DEMO_INDIVIDUAL_SIZE** gewährleistet, dass eine gewisse feste Anzahl an Ergebnissen von den Workern abgewartet wird, ehe das zweite Kriterium überprüft wird. Dabei werden die Ergebnisse in einer Liste gespeichert. Bei der Überprüfung Mithilfe von **MIN_DELTA** wird die Liste aller bisherigen Ergebnisse herangezogen, die nach Distanz sortiert ist. Die Differenz des schlechtesten und des besten muss kleiner als der Wert der Konstante sein. Erst dann wird es als das aller beste Ergebnis anerkannt und separat abgespeichert. Nähere Informationen zur Bestimmung dieser Konstanten siehe Kapitel 7.2 Probleme Scheduler.

Aus der Masterpopulation wird daraufhin eine Subpopulation geholt, die dann mit der derzeitigen Population ausgetauscht wird, um ein lokales Maximum zu vermeiden. Die neue Population erfährt wie im Worker einen beliebigen Austausch eines Individuums (siehe Kapitel 3.2.2 Genetische Algorithmen (GA) bzw. Kapitel 6.3.1 Genetischer Algorithmus mit “Travelling Salesman Problem”).

Daraufhin wird die Zeit des Workers auf den aktuellen Zeitstempel aktualisiert, damit dieser nicht vom Scheduler rausgeworfen wird. Danach wird überprüft, ob es alte Worker gibt, die eine Zeit lang keine Antwort mehr zurückgegeben haben und gelöscht werden können. Die gewünschte Zeit, in der die Worker maximal antworten können wird in einer Konstante **WORKER_RESPONSE_TIME** festgesetzt und liegt derzeit bei 2 Minuten. Zum Schluss wird die neue Population zum Worker geschickt, damit dieser weiterarbeiten kann.

Während Unity ständig beim Scheduler nach neuen Ergebnissen fragt, wird jedes Ergebnis von den Workern vom Scheduler an das Unity-Frontend weitergegeben. Durch die unterschiedliche Abspeicherung des temporären und des besten Ergebnisses weiß Unity, ob es sich bei der Nachricht um die letzte Nachricht handelt oder nicht.

Die Möglichkeit besteht, dass der Nutzer über das Unity-Frontend mitten in der Berechnung einer Navigation diese Abbricht und eine neue Einkaufsliste schickt. Hierbei muss beachtet werden, dass sämtliche Bestandteile der alten Berechnung bereinigt werden, d.h. das beste Ergebnis, sowie das beste Individuum müssen neben der Subpopulation des Workers und des temporär besten Individuums gelöscht werden (siehe Abbildung 31). Danach kann der Scheduler eine neue Einkaufsliste in Form von Einkaufslistenelementen und Wegegraph wie es am Anfang dieses Kapitels beschrieben worden ist, abspeichern und den Workern eine neue Aufgabe geben.

```
/**
 * delete map and set calculationRunning flag to false
 */
private suspend fun deleteMap(call: ApplicationCall) {
    scheduler.bestDistance = 0
    scheduler.bestIndividual = null
    scheduler.subPopulations.clear()
    scheduler.demoIndividual.clear()

    scheduler.calculationRunning = false

    println("Map has been deleted")
    call.respondText( text: "Current map has been deleted", ContentType.Text.Plain, HttpStatusCode.OK)
}
```

Abbildung 31: Löschen einer alten Einkaufsliste

6.3 Worker

6.3.1 Genetischer Algorithmus mit “Travelling Salesman Problem”

Die Implementierung der genetischen Algorithmen fand im Worker statt.

Da der Worker eine Subpopulation vom Scheduler bekommt, wendet der Worker auf diese Subpopulation den genetischen Algorithmus an. Es werden dabei 100 Generationen generiert und am Ende wird die beste Population behalten.

Im ersten Schritt wird deshalb eine Start Population entsprechend den vom Scheduler gegebene Population erstellt. Diese ist anfangs leer aber enthält dieselbe Größe wie die übergebene Population.

```
public static Population evolvePopulation(Population pop) {  
    Population newPopulation = new Population(pop.populationSize(), false);
```

Abbildung 32: Startpopulation generieren

Die Population muss jetzt mit Individuen befüllt werden und dazu werden die Individuen der übergebenen Population mittels tournamentSelection für die Rekombination (Crossover) ausgewählt. Die Funktion tournamentSelection wählt ein Individuum aus der Population aus, und diese wird dann im späteren Verlauf, für die Crossover Funktion verwendet.

Durch das Crossover entsteht ein neues Individuum (child) der dann in die neue Population hinzugefügt wird (siehe Kapitel 3.2.4). Dieser Vorgang wird solange ausgeführt, bis die Schleife die Populationsgröße erreicht hat.

```
for (int i = elitismOffset; i < newPopulation.populationSize(); i++) {  
    // Select parents  
    IndividualPath parent1 = tournamentSelection(pop);  
    IndividualPath parent2 = tournamentSelection(pop);  
    // Crossover parents  
    IndividualPath child = crossover(parent1, parent2);  
    // Add child to new population  
    newPopulation.savePath(i, child);  
}
```

Abbildung 33: Crossover

Wichtig: Die Crossover Funktion, die in Abbildung 33 gezeigt wird, rekombiniert alle Gene (Produkte) außer den ersten und letzten. Denn das erste Gen repräsentiert den Eingang des Ladens und die letzte Gene den Ausgang (Kasse).

Im nächsten Schritt erfolgt die Mutation der Individuen und die neu generierte Population wird zurückgegeben. Die Mutation ist eine zufällige Mutation und auch hier wird darauf geachtet, dass die Mutation den Anfang und Ende der Individuen nicht verändert.

Eine Mutation findet nicht immer statt, sondern nur ab und zu für den Fall, dass sich die Population nicht mehr "verbessert" (siehe Kapitel 5.3 für detaillierte Erläuterung).

```
for (int i = elitismOffset; i < newPopulation.populationSize(); i++) {  
    mutate(newPopulation.getPath(i));  
}  
  
return newPopulation;
```

Abbildung 34: Mutation

Nachdem die Fitnessfunktion auf der neu generierten Population angewendet wurde, werden nun die Überlebenden für die nächste Generation selektiert.

```
private static IndividualPath tournamentSelection(Population pop) {  
    // Create a tournament population  
    Population tournament = new Population(tournamentSize, false);  
    // For each place in the tournament get a random candidate path and  
    // add it  
    for (int i = 0; i < tournamentSize; i++) {  
        int randomId = (int) (Math.random() * pop.populationSize());  
        tournament.savePath(i, pop.getPath(randomId));  
    }  
}
```

Abbildung 35: Selektion der Überlebenden für nächste Generation

Als letztes wird der beste Pfad (Individuum) ausgewählt und zurückgegeben.

```
IndividualPath fittest = tournament.getFittest();  
return fittest;
```

Abbildung 36: Berechnung bester Pfad

Der beste Pfad wird durch die Fitness bzw. Distanz Wert des Pfades (Individuum) ermittelt.

6.3.2 A*

Für dieses Projekt wurde der A* Algorithmus verwendet, da es für die „reale Distanz“ zwischen Gegenständen im Einkaufsladen benötigt wurde. Die einzelnen Distanzen zwischen den „Gegenständen“ bzw. „Produkten“, wurde also mittels A* ermittelt. A* löst dieses Problem, weil es für Pfadfindung Probleme geeignet ist.

Pfadfinder geben die Möglichkeit vorausschauend zu planen, anstatt bis zum letzten Moment zu warten, um zu entdecken, dass es ein Problem bei der Pfadfindung gibt. Es gibt einen Kompromiss zwischen der Planung mit Pfadfindern und der Reaktion mit Bewegungsalgorithmen. Die Planung ist im Allgemeinen langsamer, führt aber zu besseren Ergebnissen. An dieser Stelle macht es auch Sinn die Parallele Programmierung einzusetzen, damit auch dadurch die Ergebnisse schneller berechnet werden können. Erreicht wird dadurch ein Gleichgewicht zwischen einem schnellen Ergebnis und einem qualitativen Ergebnis. Wenn wir, wie in Abbildung 37 dargestellt, als Beispiel zwei Punkte nehmen (Start - Rot und Ende – Blau), kann man bei „Greedy“ Algorithmen entdecken das die Ergebnisse nicht so richtig erscheinen:

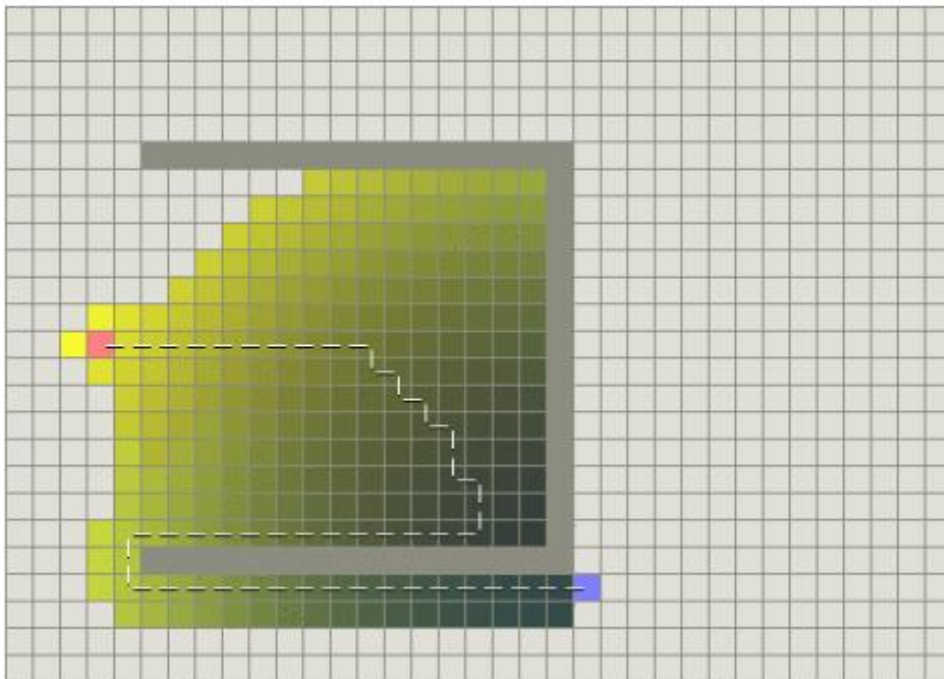


Abbildung 37: Pfadfindung ohne A Stern¹⁷

¹⁷ Vgl. Red Blob Games: Introduction to A*. Dijkstra'Algorithm and Best-Find-Search, 08.04.2020, URL: <https://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html> (Stand: 10.04.2020)

Wird A* verwendet, ist das ganze „realistischer“:

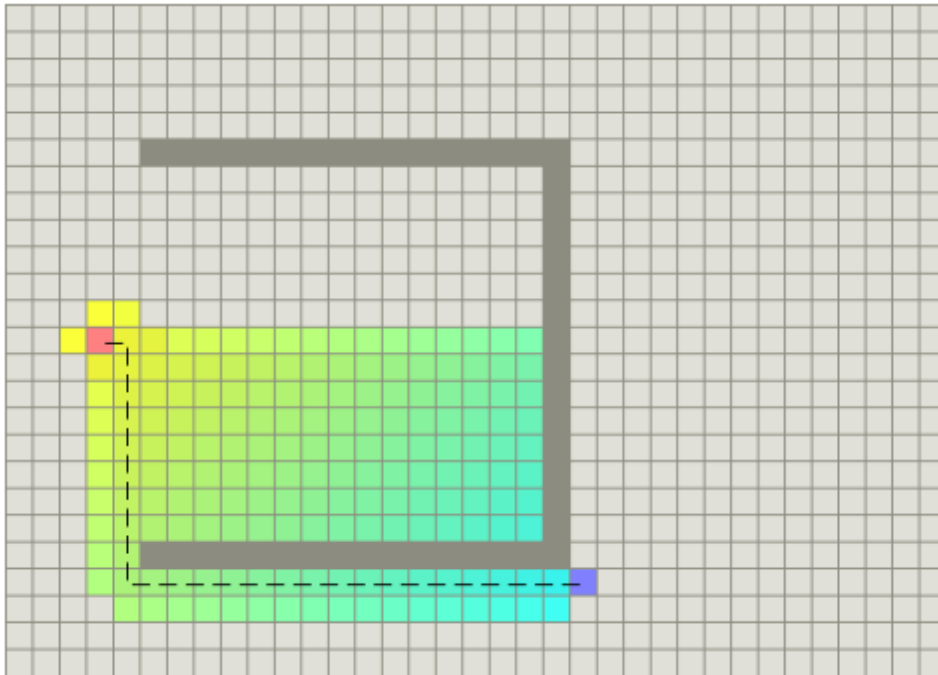


Abbildung 38: Pfadfindung mit A*¹⁸

¹⁸ Vgl. Red Blob Games: Introduction to A*. The A* Algorithm, 08.04.2020, URL: <https://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html> (Stand: 10.04.2020)

7 Probleme

Im Folgenden werden die Probleme erläutert, die während der Bearbeitung des Projektes allgemein und in den einzelnen Bereichen aufgetreten sind.

7.1 Allgemeine Probleme

Umgang mit Mac

Da alle Gruppenmitglieder Windows-Kenntnisse besitzen, war der Umgang mit Mac für alle gewöhnungsbedürftig. Insbesondere waren die Tastenkombinationen, die nicht identisch sind mit den Tastenkombinationen von Windows sind, am Anfang ein Hindernis, da diese herausgefunden werden mussten.

Java Version auf Mac Rechner

Ein weiteres Problem war die Java Version auf den Mac Rechnern im Raum. Die Mac Rechner besitzen die Java Version 8. Da die Rechner der Gruppenmitglieder eine höhere Java Version besitzen, musste jedes Mal, wenn auf den eigenen Rechnern eine Änderung am Code vorgenommen wurde, die Java Version angepasst werden.

Einheitliches JSON Format

Die Einigung auf ein einheitliches JSON Format war schwierig. Obwohl sich darauf geeinigt wurde, dass die Komponente Unity das Format vorgibt und eine Demo hochgeladen wurde, die für alle verfügbar war, war die Fehlersuche aufgrund fehlerhafter JSON Strukturen dennoch ein signifikanter Zeitfaktor.

JSON Darstellung

Ein kleiner Fehler tauchte bei der Darstellung der Inhalte im JSON auf, wo deutsche Spezialzeichen verwendet wurden. Das Problem wurde durch das Einfügen von „UTF8“ in den Code gelöst. Ebenfalls wurde der ContentType vergessen anzugeben.

Fehlercodes

Die Kommunikation der Fehlercodes war ebenfalls schwierig. Oft wurden von einer Komponente Fehlercodes bereitgestellt beziehungsweise weitergeschickt, diese jedoch von der anderen Komponente nicht abgefangen und bearbeitet. Dies war ein ausdrücklicher Wunsch im

Team, der allerdings unter den einzelnen Teammitgliedern unterschiedliche Priorität aufweist und dementsprechend nur dürftig bearbeitet worden ist.

7.2 Probleme Scheduler

Beim Testen der Funktionalität im Scheduler, gab es bei der Wahl der Workergröße und der Populationsgröße ein Abhängigkeitsproblem, denn diese konnte nicht beliebig gewählt werden. Es sollten immer mehr Subpopulationen als Worker vorhanden sein, damit alle registrierten Worker jederzeit Zugriff auf eine Subpopulation haben. Es wäre zwar möglich gewesen mit mehr Workern zu arbeiten, allerdings war es das Ziel Wartezeiten seitens der Worker zu vermeiden, daher mussten beide Variablen geschickt gewählt werden. Es entstanden zwei Gleichungen:

- I. $\text{Workersize} + 1 > \text{Subpopulation}$
- II. $\text{Subpopulation} = \text{Populationsgröße} / (\text{Workersize} + 1)$

Mittels Ersetzungsverfahren ergibt sich somit:

- $\text{Workersize} + 1 > \text{Populationsgröße} / (\text{Workersize} + 1)$
- $(\text{Workersize} + 1)^2 > \text{Populationsgröße}$

Beispiel: Für 2 Worker folgt daraus eine Populationsgröße von max. 8.

7.3 Probleme Unity

Darstellung der Berechnungen in Unity

Bei der Berechnung der kürzesten Distanz via Luftlinien ergaben sich einige Probleme hinsichtlich der realen Gegebenheiten, die nur Unity kennt. Zum Beispiel kann der Algorithmus bei der Darstellung nicht wissen, dass er nicht durch den Kassenbereich laufen soll, sondern durch den gekennzeichneten Eingang. Das hat auch zur Folge das Produkte in der Nähe der Kasse oft zu früh angelaufen werden, obwohl man die in einem realen Einkauf erst später einkaufen würde.

Ebenfalls ein Problem sind allgemein die Luftlinien, da dadurch Regale nicht berücksichtigt werden. Befindet sich Produkt A auf der einen Seite des Regals und Produkt B auf der anderen Seite des Regals, ist dies via Luftlinie oft der kürzeste Abstand und wird hintereinander eingekauft. Ein Nutzer muss allerdings bei der Darstellung um das Regal herumlaufen und somit ist dies kein guter Ansatz.

Lösung: Ein zusätzliches "Gitter" von Punkten wird übergeben (siehe Abbildung 16).

8 Messungen

8.1 Verwendete Tools

Für die Parameter Konfiguration und die variable Einkaufslistenlänge mussten die Voraussetzungen auf seitens der Worker implementiert werden, damit die Ergebnisse aus den Testläufen in die Konsole geschrieben werden konnten. Für die Vorbereitung der einzelnen Testszenarien wurde Excel verwendet. Zur Visualisierung wurde sich für Chart.js verwendet, da es zum einen flexibel ist aber auch zum anderen leicht anzuwenden ist.

Für den Last- und Performancetest sollte Apache Bench verwendet werden, da die Verwendung von Apache Bench einfach über die Kommandozeile realisierbar ist.

8.2 Testszenarien

Dieses Kapitel dient als Einleitung für die Messergebnisse. Es werden in dem Abschnitt der Parameter Konfiguration die einzelnen Parameter kurz vorgestellt und im Anschluss daran grob erläutert wie die Testszenarien erstellt und durchgeführt wurden.

Im nächsten Abschnitt wird daraufhin auf die Einkaufslistenlänge eingegangen, da dies ebenfalls einen signifikanten Einfluss auf die zeitliche Dauer und die Qualität des Endresultats hat.

Am Ende wird der Plan für den Last- und Performancetest vorgestellt, welches aufgrund von Corona nicht getestet werden konnte.

8.2.1 Parameter Konfiguration

Innerhalb der Anwendung wurden sowohl vom Scheduler als auch vom Worker einige Konstanten festgelegt, die so konfiguriert werden müssen, dass das beste Ergebnis entsteht. Im folgenden Abschnitt werden die verwendeten Konstanten kurz erläutert.

WORKER_COUNT: Damit der Scheduler genug Subpopulationen für die Worker bereithalten kann, wurde die Anzahl der maximalen Worker, die sich beim Scheduler anmelden können, durch diese Variable festgelegt.

POPULATION_SIZE: Beschreibt im Scheduler die Größe der Population. Diese Konstante ist abhängig von **WORKER_COUNT** da diese durch die Anzahl der Worker geteilt wird um genug Subpopulationen für die Worker zu erstellen. Somit haben beide Konstanten Einfluss auf die Anzahl der Individuen pro Subpopulation (siehe Probleme Scheduler).

DEMO_INDIVIDUAL_SIZE: Die Worker schicken dem Scheduler regelmäßig deren Ergebnisse, die vom Scheduler zunächst gespeichert werden und im Frontend als vorläufige Ergebnisse präsentiert werden. Die Größe dieser Konstante gibt an, ab wie vielen vorläufigen Ergebnissen das beste Ergebnis auftreten und als Endergebnis an Unity geschickt werden kann. Es werden nur so viele vorläufige Ergebnisse abgespeichert, wie der Wert der Konstanten. Bei weiteren Ergebnissen werden die ältesten gelöscht. Dies hängt zusätzlich von MIN_DELTA ab.

MIN_DELTA: Wurde definiert, um festzustellen, ob die Werte sich nicht mehr ändern und ob das beste Ergebnis an Unity geschickt werden kann. Wie vorher erwähnt werden die vorläufigen Ergebnisse in einer Liste gespeichert. Diese Liste ist nach den besten Ergebnissen sortiert. Es wird geprüft ob die Differenz des schlechtesten und des besten Ergebnisses unter MIN_DELTA liegen. Ist das der Fall, so wird das beste Ergebnis an das Frontend geschickt.

INDIVIDUAL_EXCHANGE_COUNT: Dieser Wert bestimmt beim Scheduler wie oft ein Genaustausch stattfinden soll. Bei diesem Genaustausch wird die letzte Subpopulation genommen und stellt den zufälligen Pool an Genen dar. Pro Subpopulation wird das letzte Individuum gelöscht und zufällig aus dem Pool an Genen ein Gen herausgenommen und dort hinzugefügt. Dieses Verfahren geschieht bei einem Wert von zehn exakt zehnmal.

mutationRate: Auf der Seite der Worker beschreibt diese Konstante die Wahrscheinlichkeit einer Mutation. Der Wert liegt dabei zwischen 0 und 1. Es wird überprüft, ob ein zufälliger Wert zwischen 0 und 1 unterhalb des Wertes der Konstanten liegt. Ist dies der Fall wird eine Mutation herbeigeführt.

tournamentSize: Dieser Wert gibt an, wie oft und dementsprechend wie viele Individuen aus einer Population ausgewählt werden, um daraus die beste (fitteste) Population zu entnehmen. Dies geschieht immer dann, wenn die Eltern für das Crossover ausgewählt werden. Man stellt fest, dass dieser Wert in Abhängigkeit zu der Populationsgröße steht, denn wird dieser Wert über die Populationsgröße gewählt, ist die Wahrscheinlichkeit von doppelten Individuen sehr hoch. Dies wäre eine Verschwendung an Rechenzeit.

Da hier mit genetischen Algorithmen gearbeitet wurde, musste jeder Test mehrmals wiederholt werden, damit eine Tendenz festgestellt werden kann. Aus den Versuchen wurde jeweils der Durchschnitt berechnet. Aufgrund der Dauer jedes einzelnen Tests war es nicht möglich eine ausreichend große Menge von Wiederholungen durchzuführen, da dies sonst den Rahmen dieses Projektes zeitlich überspannt hätte. Deshalb wurde genauer auf zufällig sehr gute und sehr schlechte Ergebnisse geachtet und diese ggf. aus den Ergebnissen herausgenommen

oder durch weitere Versuche ersetzt, sofern diese den Durchschnitt verfälschten. Die gesamte Parameterkonfiguration während des Testens spielte eine geringere Rolle, weswegen sie nur am Rande bzw. gar nicht erwähnt wird. Ziel dieser Testreihe ist es den Einfluss einzelner Parameter auf das Gesamtergebnis zu bestimmen. Deswegen werden lediglich die wichtigen Veränderungen der einzelnen Parameter erwähnt.

Die Parameterwerte sind teilweise willkürlich und teilweise durch vorherige Testergebnisse entstanden, weswegen der Abstand zwischen diesen Werten variiert. Wurde z.B. eine sehr große Differenz zwischen Parameterwert 1 und Parameterwert 2 festgestellt, wurde individuell entschieden, ob ein Parameterwert zwischen den beiden getesteten sinnvoll wäre oder nicht. In diesem Test wurde immer die gleiche Einkaufsliste gewählt, um die Distanzen besser zu unterscheiden. Die Einkaufsliste findet sich unter der großen vordefinierten Einkaufsliste im Unity Frontend.

8.2.2 Ergebnisqualität bei variierender Einkaufslistenlänge

Die im Abschnitt 8.2.1 durchgeführten Tests besaßen eine maximale Worker Anzahl von 12 und eine maximale Populationsgröße von 1500. Die daraus resultierende Individuengröße beträgt somit 18000. Da immer dieselbe Einkaufsliste mit 20 Produkten ausgewählt wurde, existierten immer 20! mögliche Reihenfolgen von Produkten. Nur eine Reihenfolge davon repräsentiert die zu findende kürzeste Distanz. Es ist demnach recht unwahrscheinlich, dass bei den gewählten Parametern ein optimales Ergebnis entsteht.

Deswegen wurde eine weitere Testreihe durchgeführt, die diesmal die Qualität des Ergebnisses in Abhängigkeit von der Einkaufslistenlänge bestimmt. Dazu wurden unterschiedliche Einkaufslisten mit zufällig gewählten Produkten erstellt. Das Ergebnis wurde mit Zahlen zwischen 1 und 3 bewertet, wobei eins ein perfektes Ergebnis darstellt und 3 ein sehr schlechtes Ergebnis darstellt. 2 definiert Fälle, bei denen man nicht auf Anhieb sagen konnte, ob es einen kürzeren Weg gibt oder nicht. Außerdem wurden Werte im Bereich 2 definiert, wenn es zu Dreiecksberechnungen kam, wo es nicht ultimativ ausschlaggebend ist, ob man nun Produkt i oder Produkt $i+1$ zuerst kauft.

Als *Dreiecksberechnung* werden alle Berechnungen bezeichnet, die wie in Abbildung 39 wie ein Dreieck aussehen. Diese Fälle wurden in der Regel selten vom Algorithmus aufgelöst, da die Distanzveränderung nur relativ klein ausfällt. Je nach Größe des Dreiecks wurden bessere oder schlechtere Bewertungen abgegeben.



Abbildung 39: Beispielergebnis mit Dreiecksberechnungen

Ebenfalls in diese Testreihe flossen die Ergebnisse aus den vorherigen Tests mit ein. Die Kenntnisse von dort wurden angewendet, um das Gesamtergebnis zu verbessern. Angefangen wurde mit derselben Parameterkonfiguration wie bei den meisten Tests davor. Daraufhin wurden Schritt für Schritt Verbesserungen vorgenommen.

8.2.3 Last- und Performancetest

Der Last -und Performancetest sollte mit Apache Bench durchgeführt werden. Allerdings war es nicht möglich aufgrund von Corona die Tests in der Technischen Hochschule Köln durchzuführen.

Unity, Scheduler und die Worker kommunizieren über die jeweiligen Schnittstellen, somit bilden die Schnittstellen eine wichtige Rolle für das Projekt. Aus diesem Grund ist es sehr wichtig an den Schnittstellen Stresstest, Dauerlasttest, Performancetests und Skalierbarkeitstest durchzuführen.

Bei dem Stresstest soll der Moment gesucht werden, wo die Antwortleistung exponentiell sinkt, wenn die Arbeitslast erhöht wird. Denn genau an diesem Punkt ist die maximale Kapazität des Systems erreicht.

Beim Dauerlasttest soll die Stabilität, den Ressourcenbedarf und das Antwortzeitverhalten überprüft werden. (Die Dauer beträgt hierbei mindestens 12h.)

Beim Performancetest soll das Zeitverhalten überprüft werden. Die Fragestellung: „Wie effizient sind die jeweiligen endpoints?“ soll beantwortet werden.

Beim Skalierbarkeitstest soll herausgefunden werden, wie intensiv das System genutzt werden kann und ob es sich "skalieren" lässt, wenn weitere Hardware verwendet wird. Mit anderen Worten, ob mehr Hardware mehr Arbeitslast bewältigen kann und trotzdem gute Antwortzeiten gewährleistet sind.

8.3 Messergebnisse

8.3.1 Parameter Konfiguration

Im ersten Test wurde festgestellt, ob man die Konstante **WORKER_COUNT** der tatsächlich genutzten Anzahl an Workern anpassen sollte. Setzt man die Konstante auf 12, so erstellt der Scheduler 12+1 Subpopulationen. Werden allerdings nur sechs Worker benutzt, so bleiben bei der Berechnung immer 7 Subpopulationen unbenutzt. Eine längere Berechnung, die sich hier vermuten lässt, wurde anhand der Testergebnisse nicht nachgewiesen. Bei einem Verhältnis von 12 zu 6 betrug die durchschnittliche Rechenzeit 7,58 Minuten während bei einem Verhältnis von 6 zu 6 die Rechenzeit auf circa 14 Minuten anstieg.

WORKER_COUNT bestimmt allerdings auch die Größe der Individuen. Bei einer definierten Populationsgröße von 1200 beträgt die Individuengröße $1200/12 = 100$. Jeder Worker erhält vom Scheduler 100 (verschiedene) Individuen. Würde man die Konstante auf die sechs Worker anpassen, so ergäbe sich eine Individuengröße von 200.

Aus den vorher erwähnten Zeitmessungen ergibt sich demnach, dass je kleiner die Individuengröße ist, desto schneller die Worker das Ergebnis berechnen. Dies lässt sich dadurch erklären, dass im Worker durch jedes einzelne Individuum gegangen wird und dort Mutationen und Crossover stattfinden.

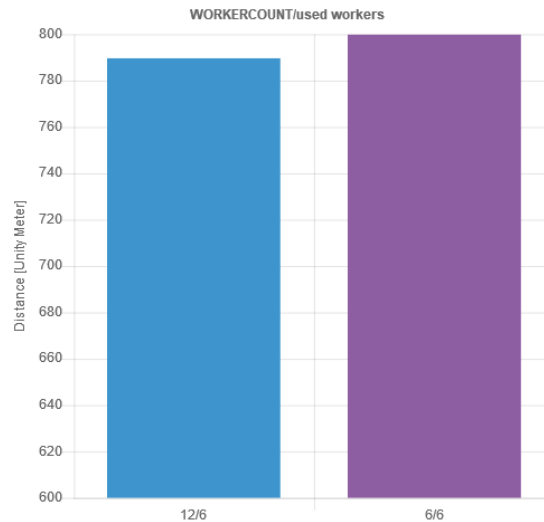


Abbildung 40: Abhängigkeit von `WORKER_COUNT` zu Distanz

Die Individuengröße hat einen viel größeren Einfluss auf die Rechenzeit als die Anzahl der Subpopulationen, die beim Scheduler nicht benutzt werden. Trotzdem sollte die Individuengröße nicht reduziert werden, da eine kleinere Individuengröße auch die Möglichkeit ein besseres Ergebnis zu erzielen reduziert. Die Wahrscheinlichkeit ist bei einer kleineren Individuengröße wesentlich geringer, dass das beste bzw. ein sehr gutes Ergebnis direkt am Anfang dabei ist. Wie man an Abbildung 41 erkennt ist dieser Aspekt ebenfalls sehr relevant, da der genetische Algorithmus in den ersten Sekunden viel effizienter Distanzverbesserungen erzielt als mit wachsender Rechenzeit. Dabei ist es nicht widersprüchlich, dass im Allgemeinen bei der verwendeten Einkaufsliste von zwanzig Produkten nur gute Ergebnisse bei längerer Rechenzeit festgestellt worden sind, da dies ebenfalls auf die Individuenanzahl zurückgeführt werden kann. Zwanzig Produkte ergeben $20!$ mögliche Individuen. Bei einer größeren Populationsgröße und bei gleichbleibender Workeranzahl auch bei einer damit einhergehenden größeren Individuenanzahl, ist die Wahrscheinlichkeit höher, dass ein sehr gutes Ergebnis direkt am Anfang mit dabei ist.

Umso interessanter war es, dass das Ergebnis bei der kleineren Individuengröße besser war (Abbildung 40). Da hier allerdings relativ wenig Wiederholungen durchgeführt worden sind, sollte man dieses Ergebnis nicht genau nehmen, denn diese Ergebnisse sind für eine Tendenz doch recht ähnlich. Es wird demnach empfohlen diese Konstante den tatsächlich genutzten Workern anzupassen.

Im nächsten Test wurde das **MIN_DELTA** überprüft. Diese Konstante wurde definiert, um ein zeitlich besseres Ergebnis zu erzielen, jedoch gerade bei Produkten, die nah beieinander sind oder die eine ähnliche Entfernung zueinander haben, kann es vorkommen, dass gerade im späteren Verlauf der Berechnung es zu minimalen Verbesserungen kommt. Abbildung 41 ist

hierfür ein sehr gutes Beispiel. Bei einem MIN_DELTA von 10 wäre die kleine markierte Verbesserung irrelevant gewesen und das Programm hätte früher ein bestes Ergebnis berechnet als ein MIN_DELTA von 0. Kritischer Punkt ist hier, wenn kurz nach dieser kleinen Verbesserung eine weitere Verbesserung eintritt, die bei einem MIN_DELTA von 10 nicht berücksichtigt werden konnte. Die Testergebnisse zeigten allerdings, dass es keinen großen zeitlichen Unterschied zwischen diesen Varianten gibt, denn Test 1 lieferte eine Durchschnittszeit von 14 Minuten, während der zweite Test eine Durchschnittszeit von ca. 15 Minuten lieferte. Demnach sollte man auf die Qualität des Ergebnisses setzen und bei einem MIN_DELTA von 0 bleiben.

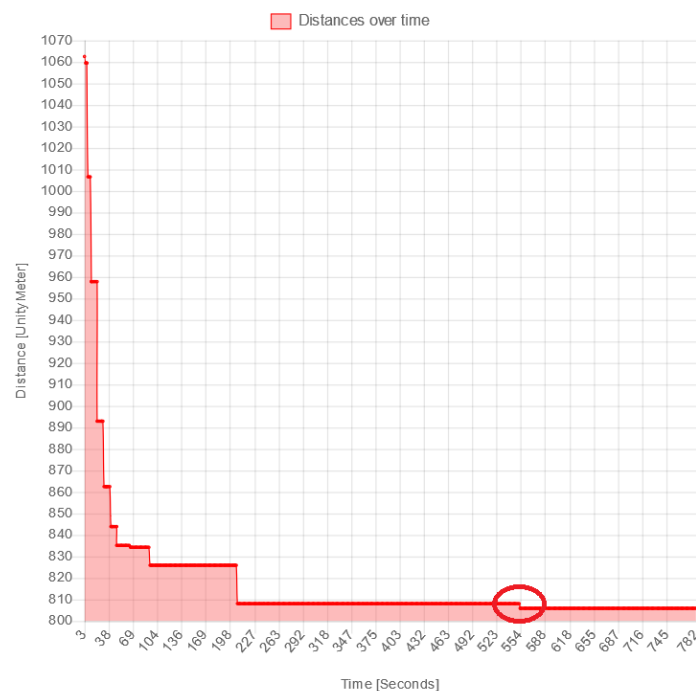


Abbildung 41: Zeitlicher Verlauf der Distanzverbesserung

Ebenfalls an Abbildung 41 erkennbar ist, dass der Algorithmus mit zeitlichem Wachstum immer länger für Verbesserungen benötigt. Möchte man dem Algorithmus die Möglichkeit geben noch kleine Verbesserungen zu finden, so bietet sich eine relativ hohe **DEMO_INDIVIDUAL_SIZE** an. In Abbildung 42 sind die Testergebnisse mit unterschiedlichen Werten für diese Konstante aufgetragen. Für diesen Test sind sechs Worker benutzt worden. Hier sieht man erneut, dass diese Testergebnisse lediglich Tendenzen widerspiegeln. Während man im zeitlichen Verlauf eine klare Tendenz dahingehend sieht, dass eine größere DEMO_INDIVIDUAL_SIZE mit einer erhöhten Rechenzeit korreliert, sieht man im Distanzverlauf keine genaue Tendenz. Hier lässt sich lediglich vermutlich, dass hier eine steigende DEMO_INDIVIDUAL_SIZE mit besseren Ergebnissen korreliert, da dadurch der Algorithmus eine Chance bekommt, noch Verbesserungen zu finden. Da diese kleine, späte Verbesserung im Vergleich zu den Testläufen allerdings relativ selten ist, sollte man hier mehr Wert auf die Rechenzeit legen und je nach

Anzahl der Worker nicht zu hohe Werte benutzen. Erhöht sich die Anzahl der Worker, so verringert sich die Rechenzeit, da mehr Instanzen parallel an einem besten Ergebnis arbeiten und man kann die DEMO_INDIVIDUAL_SIZE nach Bedarf erhöhen.

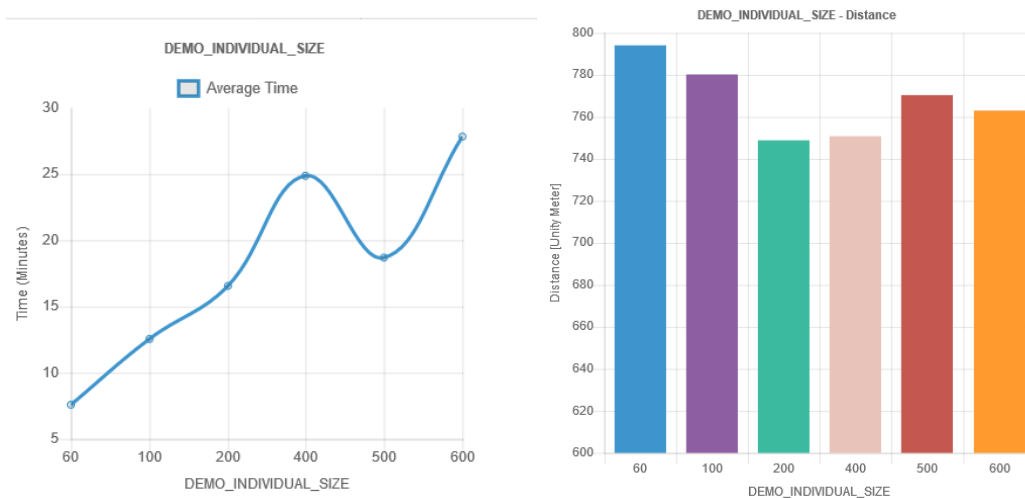


Abbildung 42: Zeit und Distanzen unterschiedlicher DEMO_INDIVIDUAL_SIZE Werte

Für die **POPULATION_SIZE** wurden 12 Worker und folgende Werte benutzt: Das Minimum von 169 (siehe Probleme Scheduler), damit jeder Worker mindestens eine Population erhält, 600 und 1500.

Grobe Wertveränderungen ließen bereits eine klare Tendenz erkennen. Bei der Minimumgröße der Population benötigte man die kürzeste Rechenzeit, erhielt allerdings im Vergleich zu den anderen Tests für diese Konstante das schlechteste Ergebnis. Die Zeit wurde mit wachsender Populationsgröße ebenfalls größer, während die Distanz sich weiter verbesserte. Auch hier gilt demnach, dass man ein Kompromiss zwischen guter Distanz und Zeitaufwand finden muss, allerdings erhöht wie erwähnt, eine größere Population die Wahrscheinlichkeit für ein besseres Ergebnis direkt am Anfang und trägt somit essenziell zu einem sehr guten Endergebnis bei. Dieser Wert sollte so hoch eingestellt werden, dass die vorhanden Worker die Ergebnisse in einer maximal vertretbaren Rechenzeit liefern.

Beim **INDIVIDUAL_EXCHANGE_COUNT** handelt es sich um eine Wiederholungszahl d.h. definierbare Werte zwischen 0 und der Populationsgröße. Getestet wurden die Fälle, die in Abbildung 23 auf der horizontalen Achse aufgetragen sind. Eine Erhöhung dieser Konstante wird empfohlen, da diese die Chancen erhöht ein lokales Optimum zu umgehen bzw. um ein besseres Ergebnis zu erzielen. Wird diese Konstante allerdings zu hoch, ist die Durchmischung der Individuen zu hoch und führt zu keiner Distanzverbesserung. Zeitlich dagegen lässt sich bei der Veränderung der Konstante nichts feststellen, da alle Testläufe ca. 12 Minuten gedauert haben. Um dem Algorithmus eine zusätzliche Chance für Verbesserung zu geben,

wird ein Wert von ca. 15 empfohlen. Nach der Abbildung zu urteilen würde sich zwar ein Wert von ca. 50 anbieten, jedoch ist dies sehr abhängig von der Populationsgröße und könnte bei kleineren Populationsgrößen zu wesentlich schlechteren Ergebnissen führen. Da die Veränderung der Konstante bei einer großen Populationsgröße einen kleinen Einfluss aufweist, sollte man für den Allgemeinfall (d.h. für verschiedene Populationsgrößen) einen kleineren Wert benutzen, um auch dort die Qualität nicht zu reduzieren.

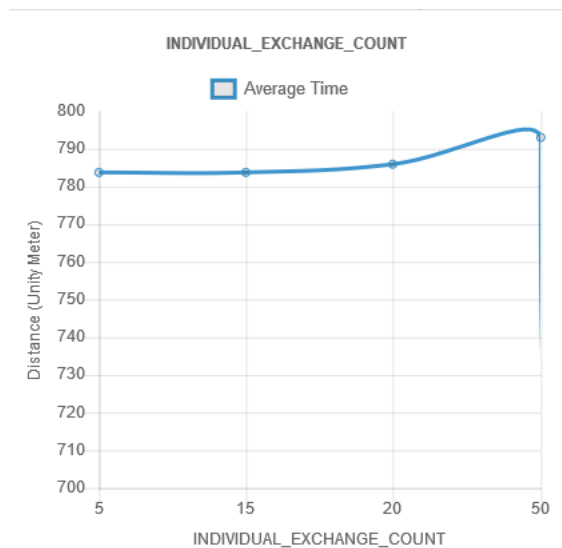


Abbildung 43: Abhängigkeit von `INDIVIDUAL_EXCHANGE_COUNT` zu Distanz

Bei der **Mutationsrate** mussten wesentlich kleinere Veränderungen der Konstante vorgenommen werden, da es sich hier um eine Prozentzahl handelt, d.h. definierbare Werte zwischen 0 und 1. Bereits kleine Veränderungen haben teilweise zu großen Veränderungen geführt. Dies erkannte man an den stärker variierenden Ergebnissen einzelner Wiederholungen mit gleicher Parameterkonfiguration. In Abbildung 44 sieht man die getesteten Mutationsraten. Man erkennt deutlich, dass diese Konstante nicht höher als 0.4 gesetzt werden sollte, da dies zu einer zu großen Durchmischung der Individuen führt und somit eher schlechtere Distanzen erstellt werden. Aufgrund der stark variierenden Ergebnisse wird bei dieser Konstante ein Wert zwischen 0.05 und 0.2 empfohlen.

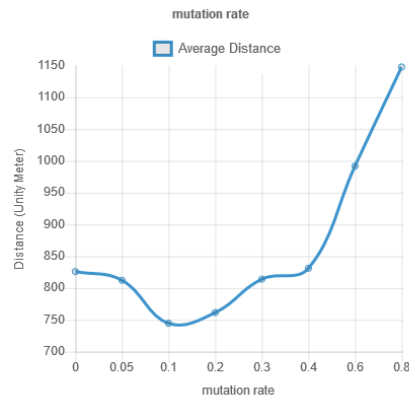


Abbildung 44: Distanz in Abhängigkeit von der Mutationsrate

Die Mutationsrate bezieht sich auf die einzelnen Produkte eines Individuums, d.h. dass die Länge der Einkaufsliste ebenfalls einen Einfluss auf das Endergebnis bei gleichbleibender Mutationsrate aufweisen könnte. Dies wurde im Rahmen dieses Projektes nicht genauer untersucht.

Die **tournamentSize** benötigte die größte Anzahl an Tests, da der definierbare Bereich zwischen 0 und der Populationsgröße liegt, aber auch bereits relativ kleine Veränderungen der Konstante zu großen Distanzveränderungen führte wie man in Abbildung 45 erkennen kann. Hier wurde die Populationsgröße von 1200 gewählt. Man erkennt eine Tendenz, dass die tournamentSize bei ca. 40% der Populationsgröße liegen sollte, um ein möglichst gutes Endresultat zu erzielen. Wie man allerdings bei der tournamentSize von 1000 sieht, gibt es auch hier trotz mehrmaliger Wiederholungen und Datensatzbereinigung starke Schwankungen.

Aus diesem Grund wurde eine weitere Testreihe mit einer Populationsgröße von 2000 durchgeführt, um eine ähnliche Tendenz festzustellen. Diese Ergebnisse sind in Abbildung 46 dargestellt.

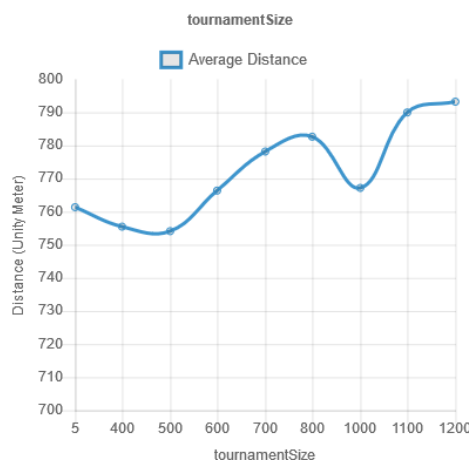


Abbildung 45: TournamentSize bei einer Populationsgröße von 1200

Die Distanzveränderungen sind bei dem genetischen Algorithmus eigentlich viel zu klein, um wirklich eine Tendenz feststellen zu können. Trotzdem erkennt man, dass bei einer tournamentSize von 1000, das der Hälfte der Populationsgröße entspricht, eine Distanzverbesserung erfolgt ist. Dies könnte ein Zufall oder die Bestätigung der Resultate aus der vorherigen Testreihe sein. Weitere Test wurden dahingehend nicht durchgeführt. Aufgrund der vorliegenden Resultate wird eine tournamentSize zwischen 40% und 50% der Populationsgröße empfohlen.

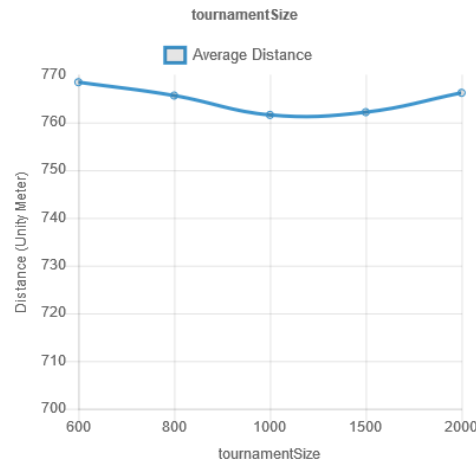


Abbildung 46: TournamentSize bei einer Populationsgröße von 2000

Die einzelnen Parameter sind teilweise stark abhängig voneinander, sodass die oben aufgeführten Testergebnisse nicht zu einem optimalen Ergebnis führen. Es ist sogar so, dass die oben genannten empfohlenen Werte zusammen zu einem schlechteren Durchschnittswert führen, als die teilweise in den Tests erreichen Durchschnittswerte. Dies kann daran liegen, dass bestimmte Abhängigkeiten nicht betrachtet worden sind, die einen starken Einfluss auf das Endresultat besitzen. Daher wurde eine weitere Testreihe durchgeführt, die eine variierende Einkaufsliste vorsieht.

8.3.2 Ergebnisqualität bei variierender Einkaufslistenlänge

Als Testreihe wurde folgende Parameterkonfiguration benutzt, die in etwa so aussieht wie die Parameterkonfiguration aus den vorherigen Tests aus Abschnitt 8.3.1:

WORKER_COUNT	8
Tatsächlich genutzte Anzahl an Workern	8
POPULATION_SIZE	1200
DEMO_INDIVIDUAL_SIZE	200
MIN_DELTA	0
INDIVIDUAL_EXCHANGE_COUNT	10
mutationRate	0.2
tournamentSize	5

Tabelle 3: Parameterkonfiguration des ersten Tests

Aus den Testergebnissen und den Überlegungen lässt sich auch hier vermuten, dass mit hoher Einkaufslistenlänge ein schlechteres Ergebnis zu erwarten ist. Dies wurde, wie man aus Tabelle 3 entnehmen kann, tendenziell bestätigt. Auch hier kann man erkennen, dass es sich hier um Wahrscheinlichkeiten handelt. Daher ist das schlechte Ergebnis bei sechs Produkten nicht verwunderlich. Die Verwendung von genetischen Algorithmen in diesem Kontext zeigt zum ersten Mal einen wesentlichen Nachteil. Da es sich um Wahrscheinlichkeiten handelt, kann man nie genau wissen, ob der Algorithmus einen guten oder einen schlechten Durchlauf hat. Der Benutzer ist demnach mehr oder weniger der Willkür des Algorithmus ausgeliefert. Nichtsdestotrotz kann man die allgemeine Wahrscheinlichkeit für ein gutes Ergebnis erhöhen. Für den ersten Test bekam jeder Worker eine Population mit 1200 Individuen, dies entspricht einer Gesamtindividuenanzahl von 9600 (siehe Tabelle 3).

Im nächsten Test wurden die Individuen auf 7200 reduziert und erneut alle Einkaufslistenlängen getestet. Das Ergebnis in Abbildung 47 ist zwar nicht sehr eindeutig, allerdings lässt sich die Tendenz dahingehend bestätigen, dass bereits bei weniger Produkten ein schlechteres Ergebnis erzielt wird. Der Durchschnittswert aller Bewertungen im ersten Test liegt bei 1.875, während der Durchschnittswert im zweiten Test bei 1,767 liegt. Man erkennt demnach eine kleine Veränderung und kann sagen, dass eine Erhöhung der Individuen zu einer Verbesserung des Endergebnisses gerade hinsichtlich längerer Einkaufslisten führen kann.

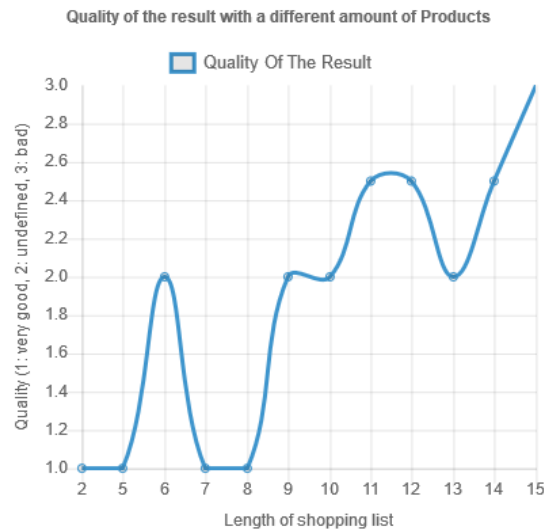


Abbildung 47: Qualität unterschiedlicher Einkaufslistenlängen

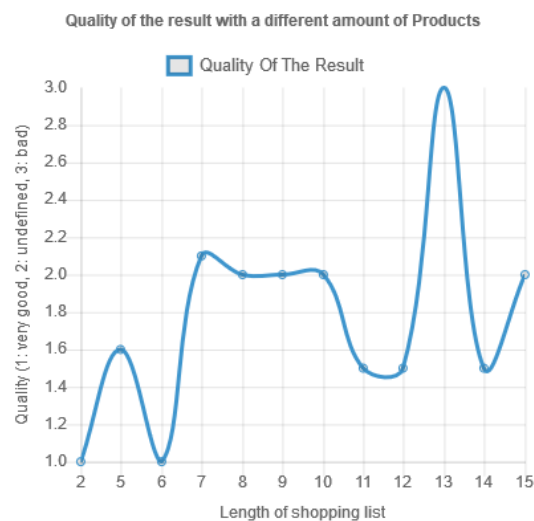


Abbildung 48: Qualität unterschiedlicher Einkaufslängen mit reduzierter Individuenzahl

Schwankungen können eventuell durch viel größere Individuen reduziert werden, da es dann wahrscheinlicher wird ein gutes Ergebnis zu erzielen. Dies wurde allerdings nicht weitergehend untersucht.

Neben der Bewertung der Distanz wurde auch die jeweilige Rechenzeit beobachtet. Im ersten Test schickte jeder Worker ca. 25 Ergebnisse zum Scheduler, um die DEMO_INDIVIDUAL_SIZE zu erreichen. Im zweiten Test wurde dies auf 50 erhöht. Die Rechenzeit verdoppelte sich dabei. Eine Steigerung der DEMO_INDIVIDUAL_SIZE und damit eine Steigerung der Wiederholungen pro Worker ist nur dann zu empfehlen, wenn nachträglich die Worker noch Distanzverbesserungen finden. Dies war allerdings im Rahmen dieser Testreihe nie der

Fall, da es ein relativ seltenes Ereignis ist und ebenfalls abhängig von der Einkaufslistenlänge ist. Ebenfalls auffällig war die Kombination aus der Individuengröße von 9600 und der Wiederholungen von 50 Mal. Hier erhielt man eine weitere durchschnittliche Verbesserung um ca. 0,1. Dies bestätigt die im vorherigen Abschnitt erwähnte Charakteristik des genetischen Algorithmus, dass die meisten Verbesserungen am Anfang gefunden werden und eine erhöhte Individuenzahl zu einer erhöhten Wahrscheinlichkeit für Verbesserungen führen. Somit führen 25 bis 50 Wiederholungen pro Worker durchschnittlich zu einem guten Ergebnis und eine Erhöhung ist nicht notwendig.

Die letzte Testreihe bestand darin die Algorithmusparameter (INDIVIDUAL_EXCHANGE_COUNT, mutationRate und tournamentSize) mittels der in den vorherigen Tests aus dem Abschnitt Parameter Konfiguration ermittelten Bereiche anzupassen und zu kontrollieren, ob diese das Endresultat verbessern. Verwendet wurde ein INDIVIDUAL_EXCHANGE_COUNT von 15, eine mutationRate von 0.1 und eine tournamentSize von 400 welches ca. 40% der Populationsgröße entspricht. Nach der durchschnittlichen Bewertung zu urteilen gab es keine Verbesserung, allerdings gab es in dieser Testreihe auch nie eine sehr schlechte Berechnung. Im Allgemeinen scheint diese Konfiguration zu einem stabileren Gesamtverlauf zu führen, welches sehr positiv zu bewerten ist.

Aus allen bisherigen Tests wurden essenzielle Informationen gewonnen. Zum einen kann man sagen, dass man für den genetischen Algorithmus die größtmögliche Anzahl an Individuen benötigt, die man in der jeweiligen Situation erzeugen kann. Je höher die Individuenzahl, desto höher ist die Wahrscheinlichkeit für ein sehr gutes Endergebnis. Gerade bei langen Einkaufslisten ist dies ein wesentlicher Faktor für die Benutzbarkeit dieses Programms hinsichtlich eines praxisorientierten Einsatzes. Um eine größtmögliche Individuenzahl zu gewinnen muss man die Faktoren WORKER_COUNT und POPULATION_SIZE maximieren.

Andererseits möchte man bei einer Iterationsmenge von maximal 50 pro Worker bleiben, um die Rechenzeit möglichst gering zu halten. Diese Iterationsmenge ist definiert durch die Division von DEMO_INDIVIDUAL_SIZE mit WORKER_COUNT. Um dies zu minimieren sollte die Anzahl der Worker möglichst groß sein und die DEMO_INDIVIDUAL_SIZE dahingehend angepasst werden, um auf die gewünschte Iterationsmenge zu kommen.

Zusammenfassend sollte man bei der Anwendung dieses Programms die größtmögliche Workeranzahl und die damit maximal rechentechnisch vertretbarste Populationsgröße wählen und die DEMO_INDIVIDUAL_SIZE individuell dahingehend anpassen, um ein optimales Ergebnis zu erzielen. Alle nicht erwähnten Konstanten sind wie aus Abschnitt Parameter Konfiguration zu wählen.

9 Fazit

Ziel der vorliegenden Arbeit war es, durch die Verwendung von Paralleler Programmierung die Optimierung des Einkaufens zu verbessern und am Ende eine Messung durchzuführen, die dann beweist, dass die Parallele Programmierung Probleme „schneller“ löst als die klassische Programmierung. Es wurden verschiedene Algorithmen wie z. B. A Stern, Travelling Salesman und Genetische Algorithmen verwendet. Bei der Anwendung von Paralleler Programmierung tauchte hierbei der Begriff „Parallele genetische Algorithmen“ auf. Durch die Anwendung solcher Algorithmen basierend auf einer Architektur, konnte das Ziel der Arbeit teilweise erfüllt werden.

Jedoch konnte die Vergleichsmessung von mehreren Rechnern (Workern) nicht durchgeführt werden. Erwartet war hierbei, dass am Ende ein Vergleich der Laufzeit in Zusammenhang mit der Anzahl von mehreren Rechnern entsteht (Beispielweise, dass zwei Rechner eine höhere Laufzeit haben, als wenn das Problem mit vier Rechner gelöst wird). Aufgrund der Covid-19-Pandemie konnte dieser Vergleich leider nicht erzeugt werden, denn für die Messung wurden mehrere Rechner benötigt und die PC-Räume von TH-Köln standen wegen der Pandemie nicht zur Verfügung.

Ursprünglich war es geplant, ein „Paper“ über dieses Projekt zu veröffentlichen, welches aber aufgrund des „unvollständigen“ Projektes zumindest aktuell noch nicht veröffentlicht wird. Das Paper wurde in den Anhang beigelegt (Kapitel 11.3) und soll eine kurze Übersicht vom ganzen Projekt erschaffen.

Im Grunde hat sich die freie Gestaltung des Projektes, die Anfangsmotivation und die interessante Aufgabe positiv auf das Team bewirkt. Die Größe des Teams war in Ordnung und hat vor allem dabei geholfen, dass neue Ideen ins Projekt einfließen und das Problem somit besser und effektiver gelöst werden konnte. Eine noch größere Gruppe wäre an dieser Stelle nicht zu empfehlen, aufgrund Terminfindungen, Abstimmungen oder Kommunikation könnten sich Schwierigkeiten ergeben. Die verschiedenen Perspektiven/Erfahrungen der Teammitglieder wurden auch betrachtet. Dies hat den Vorteil mit sich gebracht, dass bestimmte Probleme schnell gelöst werden konnten.

Aufgrund der Covid-19-Pandemie hat sich das Projekt sehr verzögert. Grund dafür war das fehlende Feedback zur Bewältigung der Covid-19-Pandemie. Hierbei wurden jegliche Kommunikationsmittel verwendet, um das Problem mit den PC-Räumen lösen zu können. Zusätzlich kam dazu noch ein Problem, nämlich ein Umzug der PC-Räume in einen anderen Raum. Das Team hatte also doppelte Schwierigkeiten, um die fehlenden Messungen durchführen zu können.

Zusätzlich gab es Probleme mit dem Transponder Verleih. Jedes Mal, als das Team ein Meeting hatte, konnte der PC-Raum nicht besucht werden. Grund dafür war es, dass die vorherigen Besucher die Transponder auch mitgenommen haben und nicht am Pförtner abgegeben haben. Ein Teamkollege hatte jedoch glücklicherweise einen "privaten" Transponder im Besitz, die dann auch verwendet wurde.

Es wird in der Zukunft erhofft, dass die fehlenden Messungen für das Projekt vom aktuellen Team (oder von einem anderen Team) nach der Covid-19-Pandemie, durchgeführt werden. Somit könnte dieses Projekt endgültig vervollständigt werden und das Paper kann damit auch veröffentlicht werden.

Abschließend kann aber nicht gesagt werden, dass dieses Projekt wegen der Covid-19-Pandemie komplett gescheitert ist. Das Team hat in diesem Projekt viele neue Methoden kennengelernt und konnte die Theorie der Parallelen Programmierung, in die Praxis umsetzen.

9.1 Zeitmessung

Die Zeiterfassung erfolgte mithilfe einer Aufwanderfassungstabelle, welcher in Excel erstellt wurde. Dabei hat jedes Gruppenmitglied seinen Aufwand in einer separaten Datei gepflegt. Die Zeiterfassung erfolgte in den Kategorien Recherche, Implementierung, Entscheidung, Dokumentation, Team-Meeting und Andere.

Die Zeiterfassung der einzelnen Gruppenmitglieder sind in Kapitel 11.3 Zeitprotokolle dargestellt.

9.2 Erfahrungsbericht

Insgesamt hat das Projekt und das Thema allen Gruppenmitgliedern gefallen. In Abbildung 49 sind Bewertungen hinsichtlich verschiedener Aspekte des Projektes von jedem Gruppenmitglied zusammengefasst. Dabei konnte jedes Mitglied die Bewertungen „minus-minus“, „minus“, „plus“ und „plus-plus“ je Aspekt vergeben. In dem Diagramm sind diese dann als Zahlen kommutiert dargestellt (bspw. „minus-minus“ als -2). Hinsichtlich der Gruppendynamik ist zu erkennen, dass dieser Teilaspekt des Projektes am wenigsten zufriedenstellend funktioniert hat. Ebenso ist bei der Kommunikation noch Verbesserungsbedarf für zukünftige Projekte vorhanden.

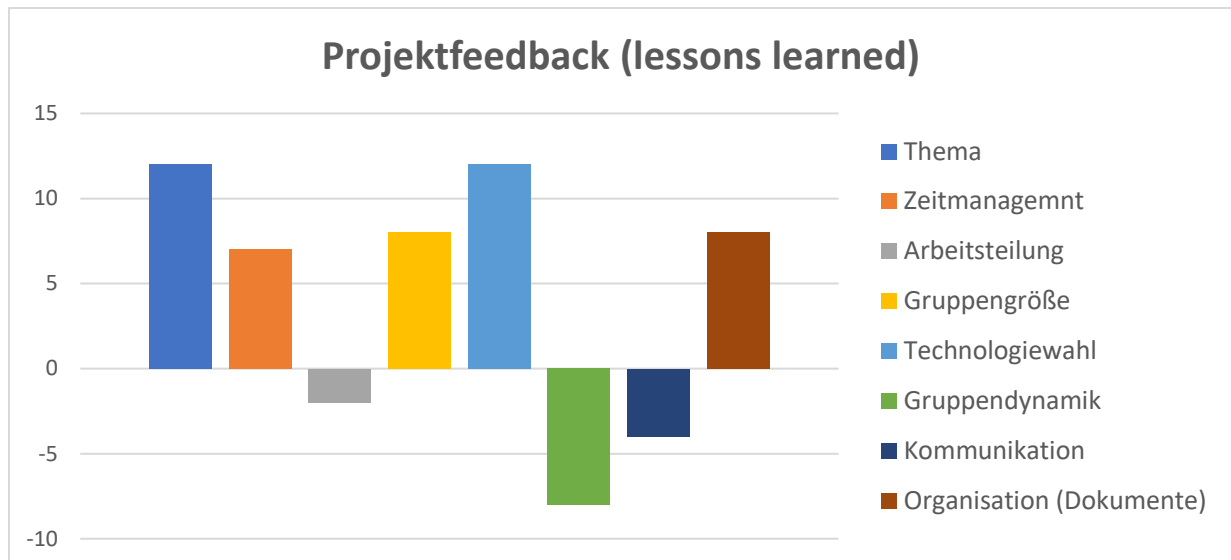


Abbildung 49: Projektfeedback (lessons learned)

Die folgenden Kreisdiagramme in Abbildung 50 zeigen die Fortschritte der Gruppenmitglieder während des Projektes in den einzelnen Bereichen. Da während des Projektes mit iMacs gearbeitet wurde, haben alle große Fortschritte bei der Bedienung von Apple Rechnern gemacht. Aufgrund dessen, dass alle Gruppenmitglieder Java sehr gut beherrschen, konnten keine Fortschritte notiert werden. Auffallend ist, dass die meisten sich im Rahmen des Projektes mit Kotlin auseinandergesetzt haben und einer von den sechs Gruppenmitgliedern gute Vorkenntnisse mitgebracht hat. Mit Genetische Algorithmen hatten sich alle vor dem Projekt schonmal auseinandergesetzt (sei es in Mathe oder in Programmierfächern). Alle haben sehr große Fortschritte gemacht und sind nahezu „Experte“ geworden. Vor dem Projekt hatten sich alle mit Verteilten Architekturen auseinandergesetzt. Es ist deutlich zu erkennen, dass alle während des Projektes große Fortschritte gemacht haben.

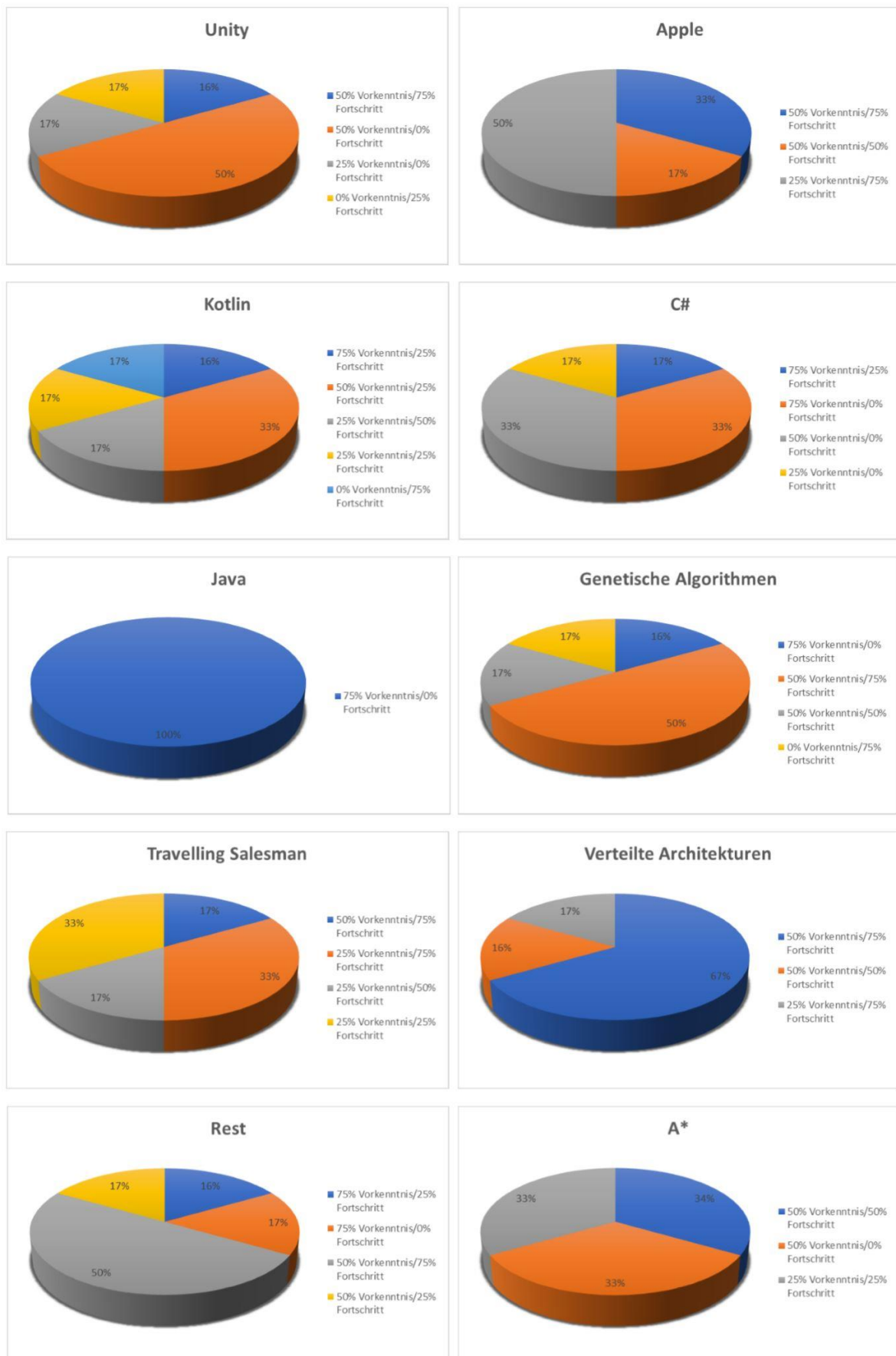


Abbildung 50: Berührungspunkte

10 Quellenverzeichnis

10.1 Literatur

Plamenka Borowska: Solving the Travelling Salesman Problem in Parallel by Genetic Algorithm on Multicomputer Cluster; International Conference on Computer Systems and Technologies, 2006.

10.2 Internetquellen

Chong, Fuey Sian, 1999: Java based Distributed Genetic Programming on the Internet. Technical Report CSRP-99-7, School of Computer Science, The University of Birmingham

Ohne Verfasser: Problem des Handlungsreisenden, 21.11.2019,
URL: https://de.wikipedia.org/wiki/Problem_des_Handlungsreisenden (Stand: 21.11.2019)

Ohne Verfasser: Unity Logo,
URL: <https://gamestudiotower.files.wordpress.com/2014/11/unity-logo.png> (Stand: 10.12.2019)

Haase, Chet 2019: Google I/O 2019: Empowering developers to build the best experiences on Android + Play @ONLINE. URL: <https://android-developers.googleblog.com/2019/05/google-io-2019-empowering-developers-to-build-experiences-on-Android-Play.html> (Stand 18 Juni 2019)

Nguyen, Minc Duc. Optimierung von Routingproblemen mit Genetischen Algorithmen auf Apache Spark. Diss. Hochschule für Angewandte Wissenschaften Hamburg, 2017. URL: http://edoc.sub.uni-hamburg.de/haw/volltexte/2018/4164/pdf/MinhDucNguyen_Optimierung_von_Routingproblemen_mit_Genetischen_Algorithmen_auf_Apache_Spark.pdf (Stand: 02.01.2020)

Riedel, Marion: Parallele Genetische Algorithmen mit Anwendungen. Diss. 2002, TU Chemnitz

Shrestha, Ajay ; Mahmood, Ausif: Improving Genetic Algorithm with Fine-Tuned Crossover and Scaled Architecture. In: Journal of Mathematics vol. 2016 (2016). – doi:10.1155/2016/4015845

Shawn, Keen: Genetischen Algorithmen. Abbruchkriterien (o.J.),
URL: <http://www.informatik.uni-ulm.de/ni/Lehre/SS04/ProsemSC/ausarbeitungen/Keen.pdf> (Stand:13.11.2019)

Red Blob Games: Introduction to A*. Dijkstra'Algorithm and Best-Find-Search, 08.04.2020,
URL: <https://theory.stanford.edu/~amitp/GameProgramming/AStarComparison>.

Red Blob Games: Introduction to A*. The A* Algorithm, 08.04.2020, URL: <https://theory.stanford.edu/~amitp/GameProgramming/AStarComparison>.

W. F. Punch, 1998, "How Effective are Multiple Populations in Genetic Programming", Proceedings of the third Annual Genetic Programming Conference, July 22-25 1998. pp. 313-318.

Tongchim, Shisanu und Chongstitvatana: Prabhas, Parallel genetic programming synchronous and asynchronous migration, Artif Life Robotics

10.3 Quellcode

Der Vollständige Quellcode kann auf <https://github.com/pschm/AVS> eingesehen werden.

11 Anhang

11.1 Berührungspunkte während des Projekts

Berührungspunkte während des Projekts

	Ayhan	Philipp	Sheila	Sümeyye	Sven	Volkan
Apple	+++	++	++	+++	++	++
Unity	o	o	+	o	+++	o
C#	o	o	o	o	+	o
Kotlin	+	+	++	+	o	+
Java	o	o	o	o	o	o
Genetische Algorithmen	+++	+++	+++	+++	++	o
Traveling Salesman	++	+	+++	+++	+	+++
Verteilte Architekturen	+++	+++	+++	+++	++	+++
REST	+++	+++	+++	+	+	o
A*	++	o	o	o	o	++

Abbildung 51: Berührungspunkte während des Projekts

Legende

Vor diesem Projekt...

... war mir dies völlig unbekannt	
... habe ich mal davon gehört, bin aber nie/selten in Berührung damit gekommen	
... habe ich mich schon einmal damit auseinandergesetzt, aber bin kein Experte	
... war ich bereits Experte in diesem Bereich	

Nach diesem Projekt...

... gibt es keinen Fortschritt (z.B. da es nicht mein Aufgabengebiet war)	o
... habe ich es im groben kennengelernt, obwohl es nicht mein Aufgabengebiet war (An	+
... habe ich gute Fortschritte gemacht, bin aber noch kein Experte	++
... habe ich sehr große Fortschritte gemacht und bin (nahezu) Experte	+++

Abbildung 52: Berührungspunkte - Legende

11.2 Projektfeedback (lessons learned)

Was fand ich gut, was nicht so gut, was würde ich anders machen wenn noch mal von vorne anfangen würde

Projektfeedback (lessons learned)

	Ayhan	Philipp	Sheila	Sümeyye	Sven	Volkan
Thema	++	++	++	++	++	++
Zeitmanagemnt	+	+	+	+	++	+
Arbeitsteilung	+	-	-	+	-	-
Gruppengröße	+	+	++	++	+	+
Technologiewahl	++	++	++	++	++	++
Gruppendynamik	--	-	-	-	--	-
Kommunikation	+	--	-	+	--	-
Organisation (Dokumente)	+	+	+	++	+	++

Bewertung:

--

-

++

Abbildung 53: Projektfeedback

11.3 Zeitprotokolle

Studierende*r A

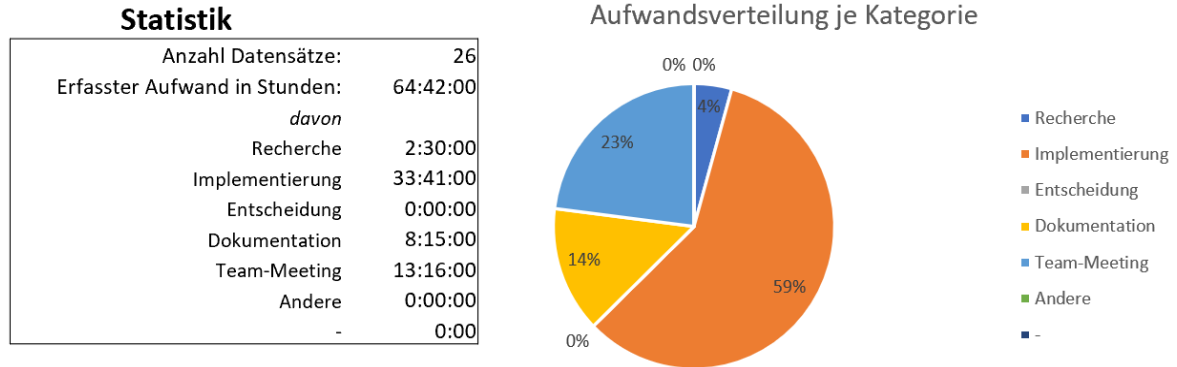


Abbildung 54: Zeitprotokoll Studierende*r A

Studierende*r B

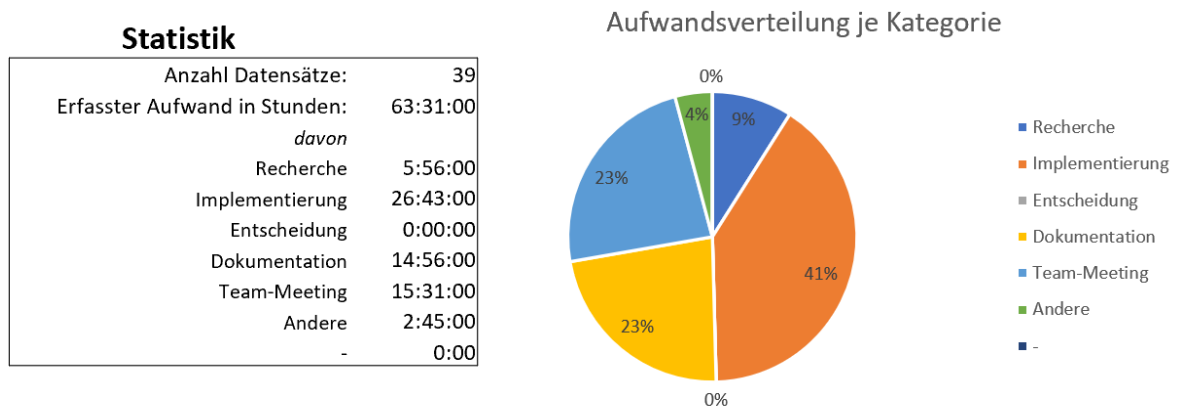


Abbildung 55: Zeitprotokoll Studierende*r B

Studierende*r C

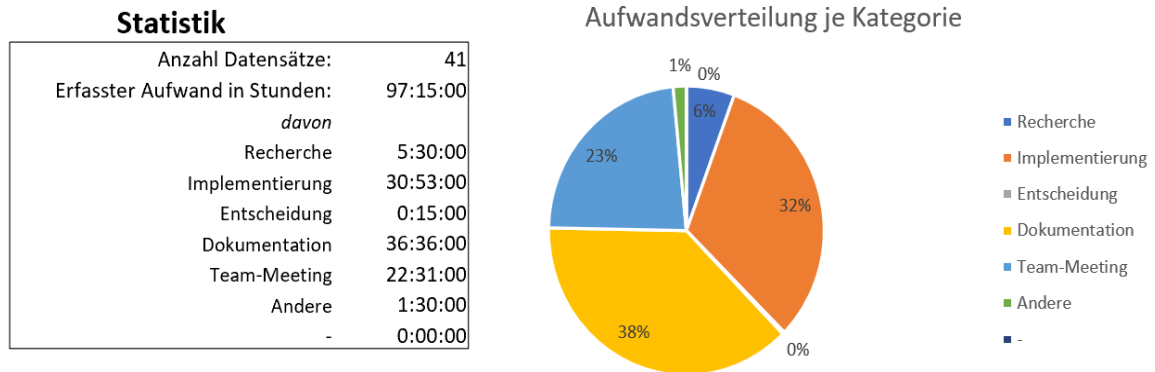


Abbildung 56: Zeitprotokoll Studierende*r C

Studierende*r D

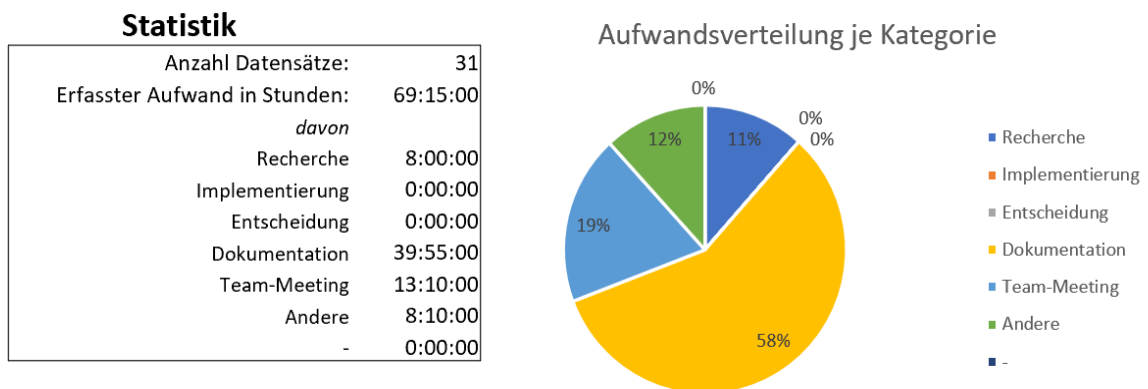


Abbildung 57: Zeitprotokoll Studierende*r D

Studierende*r E

Statistik

Anzahl Datensätze:	80
Erfasster Aufwand in Stunden:	75:40
davon	
Recherche	4:10
Implementierung	33:55
Entscheidung	3:05
Dokumentation	10:15
Team-Meeting	21:55
Andere	2:20
-	0:00

Aufwandsverteilung je Kategorie

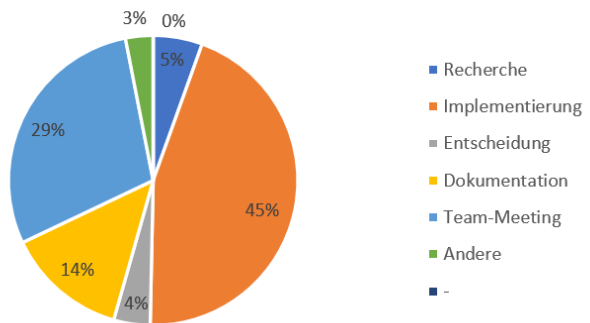


Abbildung 58: Zeitprotokoll Studierende*r E

Studierende*r F

Statistik

Anzahl Datensätze:	27
Erfasster Aufwand in Stunden:	62:22:00
davon	
Recherche	2:30:00
Implementierung	33:41:00
Entscheidung	0:00:00
Dokumentation	14:55:00
Team-Meeting	13:16:00
Andere	0:00:00
-	0:00:00

Aufwandsverteilung je Kategorie

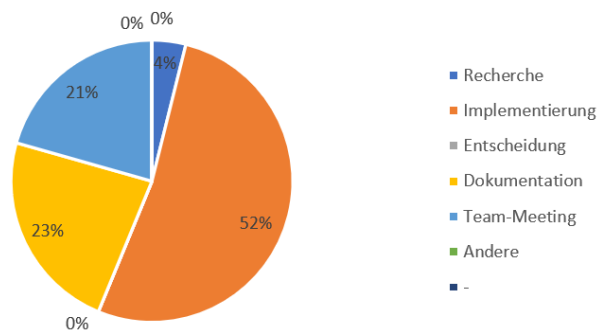


Abbildung 59: Zeitprotokoll Studierende*r F

Alle zusammengefasst

Statistik	
Anzahl Datensätze:	244
Erfasster Aufwand in Stunden:	432:45
davon	
Recherche	32:36
Implementierung	161:53
Entscheidung	3:20
Dokumentation	126:52
Team-Meeting	99:39
Andere	14:45
-	0:00

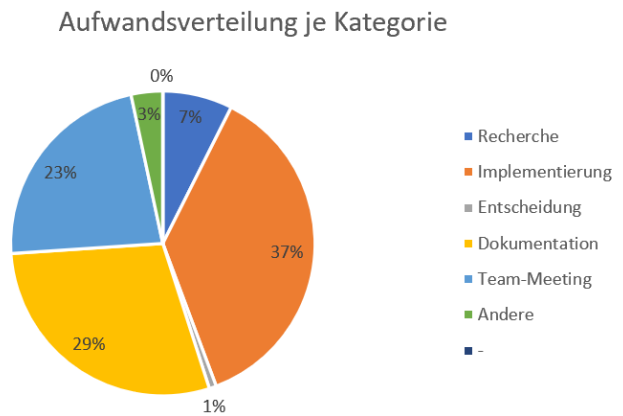


Abbildung 60: Zeitprotokoll Alle zusammengefasst

11.4 Paper

Optimized Shopping, solving TSP using a parallel Genetic Algorithm

Gezer, Ayhan
Cologne, Germany
ayhan.gezer@th-koeln.de

Kolodziej, Sheila
Solingen, Germany
sheila.kolodziej@th-koeln.de

Öztürk, Sümeyye Nur
Cologne, Germany
suemeyye_nur.oetuerk@th-koeln.de

Schmeier, Philipp Felix
Gummersbach, Germany
philipp.schmeier@th-koeln.de

Thomas, Sven
Gummersbach, Germany
sven.thomas@th-koeln.de

Yüca, Volkan
Cologne, Germany
volkan.yueca@th-koeln.de

ABSTRACT

This paper discusses one approach of solving the traveling salesman problem. During the process of development a shopping list was used to simplify the problem. Through the usage of selected products and their sorting in the store a short (shopping) route should be found. The developed solution is based on parallel genetic algorithms.

Therefore a three-part architecture was designed to use the full advantages of the parallel genetic algorithms. It consists of a display component, which was implemented as a Unity frontend, and a master-worker architecture. The master component ensures communication with the frontend and manages the master population, which are processed by the workers.

Author Keywords

traveling salesman; tsp; genetic algorithm; parallel genetic algorithm; distributed systems; th köln; architecture of distributed systems

INTRODUCTION

This project is part of the module Architecture of Distributed Systems in the Master program Computer Science (specializing in software engineering) at the [TH Köln](#). The aim of the module is to work on a problem that can be solved faster or better by using a distributed architecture. Students are free to choose the topic.

Implementation and testing is done due to the provided PC pool of 12 Mac Pros. These computers were used to carry out the Measurement.

PROBLEM DESCRIPTION

The Traveling Salesman Problem is a well known optimization problem. Our main objective is finding the optimal route through a chosen store while using a shopping list to provide a stress-free shopping experience for customers.

In this project the developed solution will be tested by using a local market (Rewe Dornseifer in Gummersbach forum) as an example. Range and scale are abstracted to ensure a clear presentation.

ALGORITHM

When finding the algorithm, the main focus was on the possible parallelization, which made the choice of a genetic algorithm. The advantage of genetic algorithms is that they can be parallelized particularly well in a variety of ways. When calculating genetic algorithms, the most time-consuming operation is determining the fitness of an individual [1]. Since the distribution of the fitness calculation over a wide number of components would generate a large network traffic, the distributed calculation of entire populations was decided.

To ensure an exchange between many populations, different models, such as the island model [2], the pollen model [3] and the nearest neighbor migration model [3, 4] were compared and analyzed.

The nearest neighbor migration was chosen for the algorithm because it offers the advantage of generating a low network load (especially compared to the pollen model) and at the same time ensuring a continuous exchange of individuals among the populations.

Further information on configuration can be found in the Configuration of the Algorithm chapter.

ARCHITECTURE

The architecture was divided into three components. Firstly, the pure frontend, which was implemented in Unity. Secondly, the implementation of the algorithm in its own master/worker architecture. The master, hereinafter referred to as the scheduler, ensures the exchange between the workers and with the frontend. The workers receive populations and evolve them.

In the design there was a strong focus on the loose coupling of the components so that the failure of individual components does not limit the functionality of the entire system.

REST was used for the exchange between the components. This guarantees a free choice of technology for the individual components.

COMPONENTS

Unity-Frontend

The frontend was developed in Unity and, in addition to the visual display, is also responsible for generating the shopping list and a navigation mesh that shows all producty and usable walkways. After creating a shopping list, it is sent to the scheduler. By regularly asking the scheduler, new calculation results are queried and displayed.

Scheduler

The scheduler represents the master component and therefore offers a REST API, which can be viewed in the [OpenApi 3.0 documentation](#). The scheduler is written in Kotlin and Ktor is used as the server framework.

After receiving a new shopping list, a master population is created. All workers who are already registered or who will register in the future will be assigned a subpopulation for processing. If a worker delivers an updated population, an individual exchange is initiated with the neighboring population.

Worker

The worker maps the computing component and is implemented as a simple Java application. At the beginning, the worker logs on to the scheduler and waits for a shopping list and thus population to be available. After receiving the population and the navigation mesh, the population is processed.

CONFIGURATION OF THE ALGORITHM

With genetic algorithms, there are many possible adjustments that can be tuned to influence the result. Different configurations were therefore tested for scheduler and workers.

The most important are named below and briefly explained. More detailed explanations and specific values can be found in the [complete documentation](#) (available only in german).

Scheduler-Config

- **Worker-Count:** The number of simultaneous workers supported by the scheduler. Based on this, the number of subpopulations is calculated
- **Master-Population-Size:** The size of the master population.
- **Demo-Individual-Size:** Before a final result is passed on to the frontend, a number of individuals is held, which consists of the best individuals from the processed populations. Only when this minimum number is reached is a result passed on.
- **Min-Delta:** The min delta ensures that no final result is passed on to the frontend as long as there are still large discrepancies between the demo individuals collected. So there is still a lot of room for improvement.
- **Individual-Exchange-Count:** This determines how many individuals are exchanged between the subpopulations.

Worker-Config

- **Mutation-Rate:** The mutation rate describes the frequency of the mutation of a single gene.
- **Tournament-Size:** The tournament size describes how many individuals are randomly drawn from the population for the selection.

MEASUREMENT

The measurements were performed in the scheduler using the Char.js library. Before the actual measurements and thus the comparison of the performance was carried out using different numbers of workers, various tests were used out to find the optimal parameters.

This generated attention to important details that had to be adjusted for the different numbers of workers. Using the example of the scheduler setting worker-count, this means, it is not enough just to adjust this setting to the number of workers actually used. A test with six real workers and a worker count of 12 resulted in a computing time of 7.58 minutes. However, when the worker counts were adjusted to six, the computing time was extended to 14 minutes. After further testing, this is related to the population size. Due to the halved number of workers, the population size per worker doubled, which led to considerable increased computing times.

The knowledge gained from the parameter tests was subsequently be used for the tests with differentiating real worker numbers.

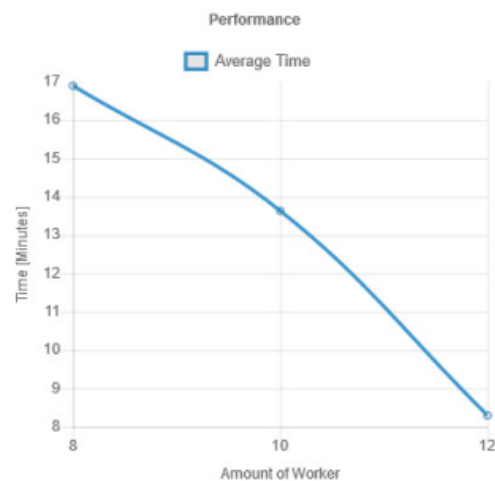


Figure 1. Comparison of computing time with different numbers of workers

The tests with different numbers of workers quickly showed how much the parallelization helps to shorten the calculation. As in figure 1 to see, the calculated time is greatly reduced. The originally planned tests with fewer workers were avoided due to the longer and longer runtimes.

CONCLUSION

Even if genetic algorithms do not guarantee optimal results, they offer a good opportunity to approach problems that cannot be solved in a finite amount of time by pure computing power to such an extent that they become qualitative, processable data. Combining these advantages with the strengths of a distributed calculation, complex problems can be solved in

acceptable times. Using the example of the shopping list, it is sufficient to write it before the actual shopping, when you arrive at the store, the route is ready and usable.

If you are interested in the application and procedure, the entire documentation (only available in German) and all sources are available in our [GitHub repository](#).

ACKNOWLEDGMENTS

We thank the lecturers of the module, Prof Dr. Lutz Köhler and M. Sc. Pascal Schönthier for the support during the semester and the opportunity to write and publish this paper.

REFERENCES

- [1] Fuey Sian Chong. 1999. Java based Distributed Genetic Programming on the Internet. Technical Report CSRP-99-7, School of Computer Science, The University of Birmingham. (1999). Retrieved November 17, 2019.
- [2] Minc Duc Nguyen. 2017. *Optimierung von Routingproblemen mit Genetischen Algorithmen auf Apache Spark*. Ph.D. Dissertation. Hochschule für Angewandte Wissenschaften Hamburg, Hamburg, Germany. Retrieved January 02, 2020 from http://edoc.sub.uni-hamburg.de/haw/volltexte/2018/4164/pdf/MinhDucNguyen_Optimierung_von_Routingproblemen_mit_Genetischen_Algorithmen_auf_Apache_Spark.pdf.
- [3] Marion Riedel. 2002. *Parallele Genetische Algorithmen mit Anwendungen*. Ph.D. Dissertation. TU Chemnitz, Chemnitz, Germany. Retrieved January 05, 2020.
- [4] Shisanu Tongchim and Prabhas Chongstitvatana. 2001. Parallel genetic programming synchronous and asynchronous migration. *Artif Life Robotics* (2001).