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: Sven Thomas, 11117201

Philipp Felix Schmeier, 11117804

Sheila Kolodziej, 11143470

Ayhan Gezer, 11117870

Volkan Yüca, 11117315

Sümeyye Nur Öztürk, 11114188

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

Gummersbach, …

Technology
Arts Sciences
TH Köln

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Aufgabenstellung

Im Rahmen des Moduls „Architektur verteilter Systeme“ wird eine Anwendung zum optimierten Einkaufen erstellt.

Die Idee basiert auf einem alltäglichen Problem, das besonders in „fremden“ bzw. großen Einkaufsläden auftritt. 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 muss daraufhin eine Einkaufsliste erstellen und anhand dieser beiden Komponenten 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 die Berechnung grafisch angezeigt, sodass dieser durch den Laden navigiert wird.

Aufgrund des eher geringen zeitlichen Rahmens dieser Veranstaltung wurde beschlossen, dass diese Aufgabe zunächst anhand des eher kleinen Lebensmittelladens „Dornseifer“ in Gummersbach bearbeitet wird.

2 Grundlegende Dokumente

2.1 Lastenheft

1. **Zielbestimmung**

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

2. **Produkteinsatz**

Das Produkt dient des zeitlich effizienten Einkaufs des Benutzers. Zielgruppe sind alle einkaufenden Kinder, Jugendliche und Erwachsene.

3. **Produktfunktionen**

/LF10/ Einkaufsliste erstellen /LF20/ Einkaufsliste verändern /LF30/ Lebensmittelladen auswählen /LF40/ Navigation durch den Lebensmittelladen

4. **Produktdaten**

/LD10/ Einkaufslistenelemente speichern /LD20/ Lebensmittelladenstruktur speichern

5. **Produktleistungen**

/LL10/ Flüssige und echtzeitnahe Navigation durch den Lebensmittelladen

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 mithilfe seiner eigenen Einkaufsliste den kürzesten Weg durch einen von ihm gewählten Lebensmittelladen zu finden.

(a) **Musskriterien**

i. Benutzer kann Einkaufsliste erstellen
ii. Benutzer kann Einkaufsliste speichern
iii. Benutzer kann Einkaufsliste vor dem Start der Navigation verändern
iv. Lebensmittelladenstruktur von „Dornseifer“ in Anwendung enthalten
v. Lebensmittelladenstruktur von „Dornseifer“ wird in 2D bzw. 3D dargestellt
vi. Benutzer kann „Dornseifer“ als Lebensmittelladen auswählen
vii. Der zeitlich effizienteste Weg durch den Lebensmittelladen wird ermittelt
viii. Grafische Darstellung der Navigation durch den Lebensmittelladen
ix. Möglichkeit zur (zukünftigen) Erweiterung der Auswahl von Lebensmittelläden

(b) **Wunschkriterien**

i. Während der Navigation kann der Benutzer die Einkaufsliste anpassen
ii. Weitere Lebensmittelläden 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 drückt Button “Neue Einkaufsliste”

Beschreibung:

(a) Neue/Leere Einkaufsliste anzeigen

(b) Einkaufslistenelemente hineinschreiben lassen

(c) Einkaufsliste wird gespeichert

/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 „Einkaufsliste bearbeiten“

Beschreibung:

(a) Benutzer wählt eine Einkaufsliste aus

(b) Ausgewählte Einkaufsliste wird angezeigt

(c) Benutzer verändert Einkaufsliste

(d) Veränderungen werden gespeichert

F30/ (/LF30/)

Lebensmittelladen auswählen: Von Programmstart bis Navigationsstart

Ziel: Benutzer wählt einen Lebensmittelladen aus

Vorbedingung: Mindestens ein Lebensmittelladen wurde in die Anwendung integriert

Nachbedingung Erfolg: Ein Lebensmittelladen wurde ausgewählt

Nachbedingung Fehlschlag: Es konnte kein Lebensmittelladen ausgewählt werden

Akteure: Benutzer
Auslösendes Ereignis: Benutzer drückt Button „Lebensmittelladen auswählen“
Beschreibung:

(a) Benutzer wählt einen vordefinierten Lebensmittelladen aus
(b) Auswahl wird gespeichert

F40/ (/LF40/)
Navigation durch den Lebensmittelladen: Von Navigationsstart bis Navigationssende
Ziel: Der zeitlich effizienteste Weg durch den Lebensmittelladen wird berechnet und dem Benutzer angezeigt
Vorbedingung: Einkaufsliste und Lebensmittelladen wurden ausgewählt
Nachbedingung Erfolg: Der Benutzer wird durch den Lebensmittelladen navigiert
Nachbedingung Fehlschlag: Die Navigation kann nicht gestartet werden
Auslösendes Ereignis: Spieler drückt Button „Navigation starten“
Beschreibung:

(a) Anhand der Einkaufslistenelemente wird der zeitlich effizienteste Weg von Ladeneingang bis zur Ladenkasse ermittelt.
(b) Die Einkaufslistenelemente werden umsortiert und in eine Abholreihenfolge eingereiht
(c) Der Benutzer erhält eine grafische Navigation zu dem ersten Abholelement mittels einer Linie und einem Bild des Einkaufslistenelements
(d) Benutzer folgt Navigation
(e) Benutzer vermerkt Erfolg durch Anklicken des „Häkchen“-Symbols oder Misserfolg durch Anklicken des „Kreuz“-Symbols
(f) Erfolg/Misserfolg wird in der Einkaufsliste gespeichert durch „Häkchen“ oder „Kreuz“ Symbol hinter dem jeweiligen Einkaufslistenelement
(g) Der Benutzer erhält eine grafische Navigation zu dem nächsten Abholelement mittels einer Linie und einem Bild des Einkaufslistenelements
(h) Wurde das letzte Abholelement erfolgreich/nicht erfolgreich in den Einkaufswagen gelegt, wird die Navigation beendet.

4. **Produktdaten**

/D10/ (/LD10/)
Name des jeweiligen Lebensmittels aus der Einkaufsliste

/D20/ (/LD20/)

Lebensmittelladenstruktur:

(a) Position der Regale, Wände, Kassen, Ein-/Ausgang

(b) Position und Name der Lebensmittel

5. **Produktleistungen**

/L10/ (/LL10/)

Der aktuelle Standort des Benutzers muss regelmäßig/echtzeitnah aktualisiert werden

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

Master-Worker-Architektur

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

Worker

TODO: Ayhan und Volkan.

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.

Einkaufslistenelement

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

Abholelement

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

Einkaufsknoten

Jedes einzelne Einkaufslistenelement beschreibt einen Einkaufsknoten bei der Berechnung und Navigation der optimierten Strecke durch den Lebensmittelladen. Im Optimalfall besucht der Benutzer jeden einzelnen Knoten und kauft bei n Einkaufsknoten n verschiedene Produkte ein.

Berechnungsknoten

Unterstützende Knoten bei der Berechnung der optimalen Strecke. Sie helfen dabei Abbiegungen oder schwierige Wege kollisionsfrei zu berechnen.

Kommentiert [ST1]: Wegpunkt?

Navigationsgraph

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

Population

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. ~~Population, die im Gegensatz zu den anderen Populationen im Rahmen des Schedulers erzeugt wird.~~

2.4 Code Convention

Da die Komponenten in verschiedenen Sprachen geschrieben worden sind, kann man sich nicht auf einen einheitlichen Standard festsetzen. 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.

Kommentiert [ST2]: Wegegraph?

Kommentiert [PS3]: Formulierung ggf. ein wenig abschwächen, da dies momentan nicht der Wirklichkeit entspricht und auch nicht für JEDE Methode nützlich ist.

3 Recherche und Grundentscheidung

3.1 Konkurrenzprodukte

Kaufland App

Die Kaufland App ermöglicht die Auswahl von Wunschfiliale. Es können mehrere Einkaufslisten erstellt werden, die abgearbeitet werden können. Es ist möglich die gewünschte Menge eines Artikels beim Einfügen in die Einkaufsliste einzugeben oder später zu ändern. Die Einkaufsliste kann nach Kategorien geordnet werden. Artikel aus Angebote und Rezepte können direkt in die Einkaufsliste hinzugefügt werden.
Die Kaufland App bietet keinen Wegweiser durch die Filiale. Artikel, die nicht in Angebote oder Rezepte stehen, müssen eingegeben werden. Es gibt keine Auswahlmöglichkeit.

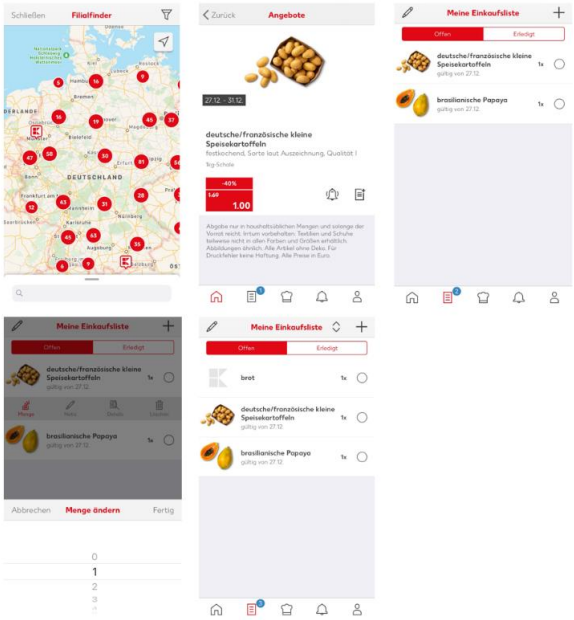


Abbildung 1: Kaufland App

Besorger App

Bei der Besorger App können mehrere Einkaufslisten erstellt werden. Bei der Artikeleingabe wird eine Artikelliste angezeigt. Man kann die Artikelliste entweder durch scrollen durchgehen oder den Namen eingeben, damit passende Artikelvorschläge angezeigt werden. Beim Hinzufügen der Artikel können die Menge, die Abteilung und der Händler (bspw. Aldi, Rewe, Rossmann etc.) eingegeben werden. Die in die Einkaufsliste eingefügte Artikel können alphabetisch, nach Abteilung oder nach Händler sortiert werden. Außerdem besitzen die verschiedenen Kategorien unterschiedliche Farben.
Die Händlereingabe dient nur zur Übersicht in der Einkaufsliste. Es können keine Filialen ausgewählt werden. Die Besorger App bietet somit keinen Wegweiser durch die Filiale.

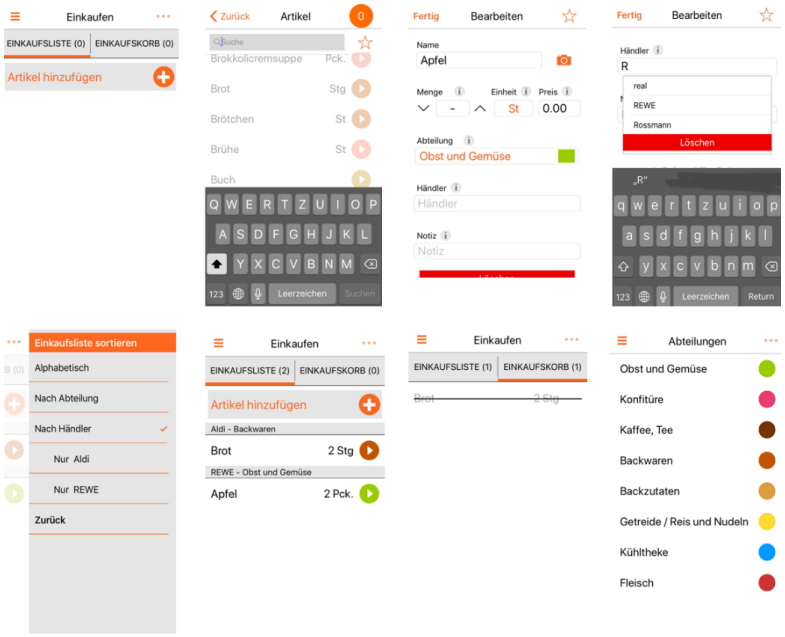


Abbildung 2: Besorger App

Pon – Die smarte Einkaufsliste

Die Pon-App ermöglicht es ebenfalls mehrere Einkaufslisten zu erstellen. Die Artikel, die man in die Einkaufsliste hinzufügen möchte, 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 gemerkt, sodass man diese nicht jedes Mal erneut eingeben muss. Einkaufslisten können nach Einkaufsladen sortiert werden. Man kann bei jedem Artikelauswählen, von welchem Einkaufsladen man diese erwerben möchte. Beim Hinzufügen der Artikel in die Einkaufsliste kann, nachdem man den Einkaufsladen ausgewählt hat, 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.

Kommentiert [PS4]: Klingt irgendwie komisch.

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 selber die Kategorien sortieren muss und die Reihenfolge der Kategorien im Einkaufsladen kennen muss.

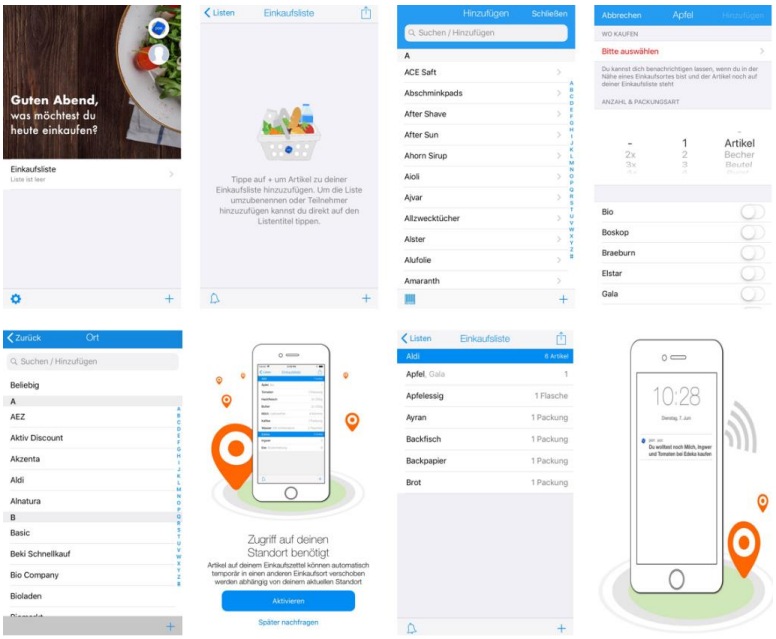


Abbildung 3: Pon App

„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 neuen Einkaufsladen anzulegen 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 auch in hier möglich die Produkte so zu sortieren, wie sie im Einkaufsladen angeboten werden. Dies ist jedoch mit viel Aufwand verbunden, da man selber die Kategorien sortieren muss und die Reihenfolge der Kategorien im Markt kennen muss.

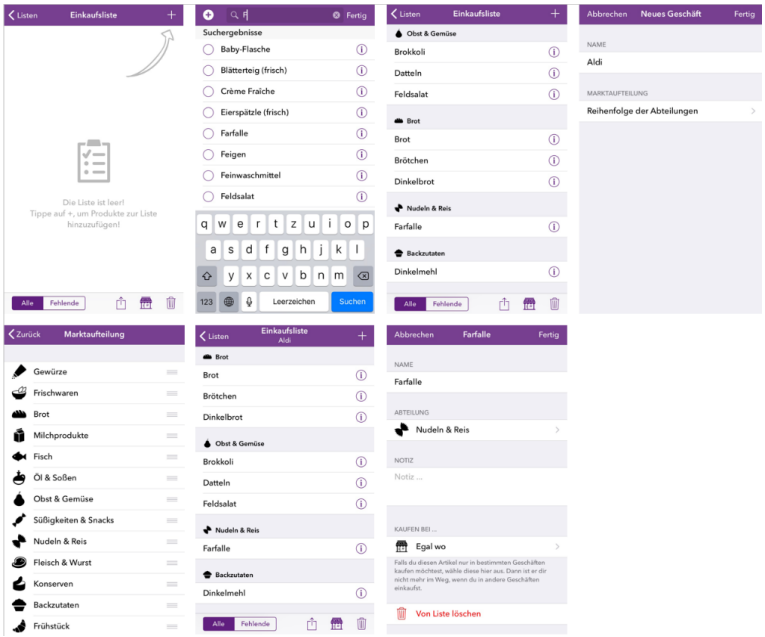


Abbildung 4: Milk for Us App

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 im Einkaufsladen 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. Man bekommt eine Mitteilung, wenn man 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 Handlungsreisenden auch genannt als *Travelling Salesman Problem* ist ein kombinatorisches Optimierungsproblem unter Anderem in der theoretischen Informatik. 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 dadurch 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 Städten beschreibt. Zur Vereinfachung kann der Graph als **vollständig** angenommen werden, d.h. zwischen zwei Knoten existiert immer eine Kante. Unter Umständen muss dann eine künstliche Kante eingefügt werden. 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

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

unterschiedliche Längen besitzen. Deswegen wird diese Situation dann mit einem gerichteten Graphen modelliert, um die Richtung ebenfalls angeben zu können.
Beim symmetrischen Travelling Salesman Problem die Kantenlängen in beiden Richtungen identisch. Deswegen wird meist ein ungerichteter Graph zur Modellierung verwendet[WI01][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]

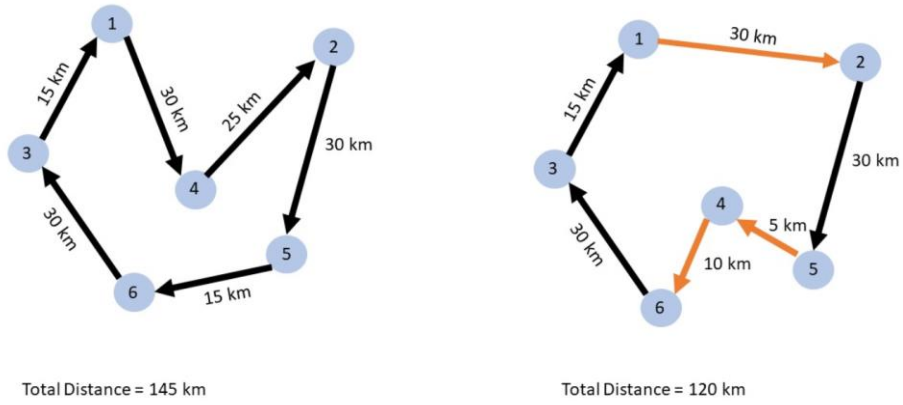


Abbildung 5: Travelling Salesman Problem

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 kodiertes Chromosom abgebildet. Dann initialisiert man eine Ausgangspopulation, indem eine Population von Individuen erzeugt und zufällig initialisiert wird. Man spricht hier von der Generation 0. Jedes Individuum wird daraufhin mit einer Fitnessfunktion bewertet, d.h. jedes einzelnem Chromosom wird eine reellwertige Zahl zugeordnet.
Daraufhin werden jeweils zwei Elternteile mittels einer Selektionsvariable selektiert und es entstehen mittels einer gewählten Kreuzungsvariante Nachkommen dieser Elternteile. Da die Allele der Nachkommen mutieren können, werden die Werte invertiert.

[2] Vgl. Ders. (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.

Dann wird die Population um den 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. Das Abbruchkriterium sollte geschickt gewählt werden. 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 zulassen. Um ein zeitliches Abbruchkriterium zu erhalten kann man außerdem festlegen, dass man nach einer maximalen Anzahl von Generationen aufhört. Dies bietet sich an, wenn der genetische Algorithmus sich dem Optimum nicht schnell genug nähert [KE04][4].

3.2.3 Parallele Genetische Algorithmen

Kommentiert [PS5]:

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, dass bei genetischen Algorithmen genutzte Prinzip der Populationen zu nutzen, um diese über mehrere Systeme hinweg zu berechnen [Punch]. Bei der Berechnung ist der Schritt der Evaluierung der Gene, also die Berechnung der Fitnessfunktion die zeitaufwendigste Operation [Chong].
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.
Der große Unterschied gängiger Methodiken ist, wie der Austausch unter den unterschiedlichen Populationen gestaltet ist. Folgend werden das Inselmodel, das Pollenmodel und die Nearest-neighbor migration in Kürze erläutert.

[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).

3.2.3.1 Inselmodel

Das Inselmodel ist die einfachste Form der Parallelisierung genetischer Algorithmen. Es werden unterschiedliche Populationen generiert, welche unabhängig voneinander bearbeitet werden [Riedel]. 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, indem nach einer festgelegten Anzahl an Generationen, Gene zwischen verschiedenen Populationen ausgetauscht werden [Nguyen], [Shrestha und Mahmood].

3.2.3.2 Pollenmodel

Das Pollenflugmodel orientiert sich am biologischen Vorbild des Pollenflugs durch 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. [Riedel]

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 Pollenmodel. Auch müssen keine zusätzlichen Berechnungen durchgeführt werden. Anzumerken sei, dass die Nachbarschaftpopulationen 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. [Riel], [Tongchim und Chongstitvatana]

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, 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 Abb. 3).
Die mögliche Mutation wird durch ein zufälliges Ändern der Ortreihenfolge hervorgerufen.

3.3 Architektur

3.3.1 Mögliche Architekturen bei verteilten Systemen und ihre Anwendung

(Quelle...) 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.

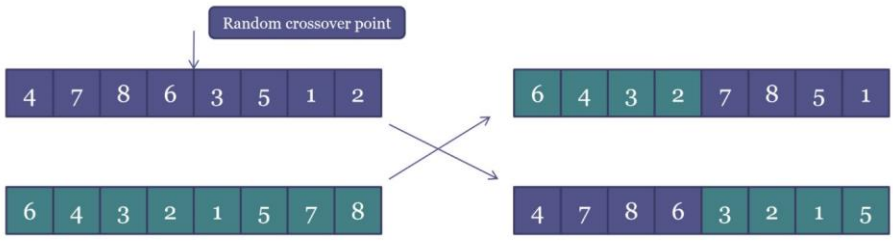


Abbildung 6: Kreuzung zweier Eltern

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 auf unaufgefordert 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-toPeer** Verbindung, wo 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 schaffen.
Eine dritte Möglichkeit ist die **N-Tier-Architektur**. Dies ist eine Schichtenarchitektur, in der Aspekte des Systems einer Schicht zugeordnet werden. Bei einer drei-SchichtenArchitektur 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 als deutlich verbessert 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 der Programmierkosten, 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 könnte zwar durchaus

abgrenzend zur SOA genutzt werden, aber hinsichtlich der verwendeten Algorithmen bietet sich eher 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.

Außerdem ist diese Architektur so bekannt und verbreitet, dass eine Implementierung in den meisten Programmiersprachen keinen großen Aufwand darstellt.

4 Entwurfsdiagramme

4.1 Architekturdiagramm

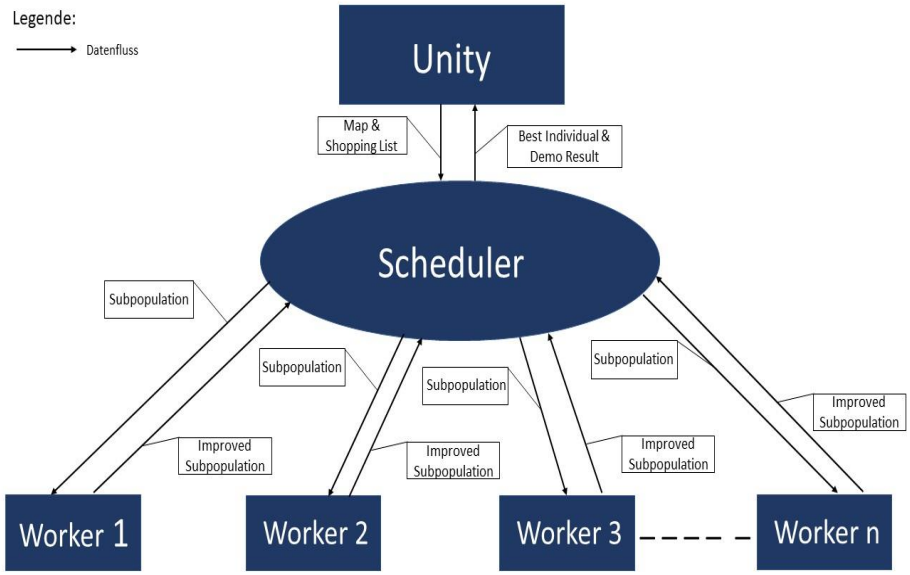


Abbildung 7: Architekturdiagramm

Die Abbildung beschreibt die Architektur der Anwendung. In Unity wird die Karte von „Dornseifer“ 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 Worker generieren (mittels Genetische Algorithmen und Travelling Salesman Problem) mehrere Generationen und schicken die verbesserte Subpopulation anschließend an den Scheduler weitergegeben, der die kürzeste Distanz, also den besten Individuum von allen Workern bestimmt und zusammen mit einem Demoergebnis an Unity übergibt, welche das Ergebnis visualisiert.

5 Komponentenaufbau

5.1 Unity-Frontend

Abbildung 8: Unity-Logo

Für die graphische Oberfläche wurde die Anwendung Unity verwendet. Mit Unity können neben eigene Spiele auch 3D-Modelle, Texturen, Animationen und viele andere komplexe Projekte realisieren.
Das Frontend bildet eine Unity-Instanz. In dieser 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 die Frontend in Unity umzusetzen, damit am Ende die Visualisierung des Ergebnisses in Unity erfolgt, wurde zuerst die folgende Skizze von dem Einkaufsladen „Dornseifer“ erstellt. Dabei wurden die einzelnen Artikeln zuerst in Kategorien zusammengefasst. Bei Kategorien, die mehrmals auftreten, wird später eine Spezifikation erfolgen. Diese Skizze hat die Umsetzung der Frontend in Unity vereinfacht und ebenfalls veranschaulicht, wie der Einkaufsladen in Unity aussehen soll.

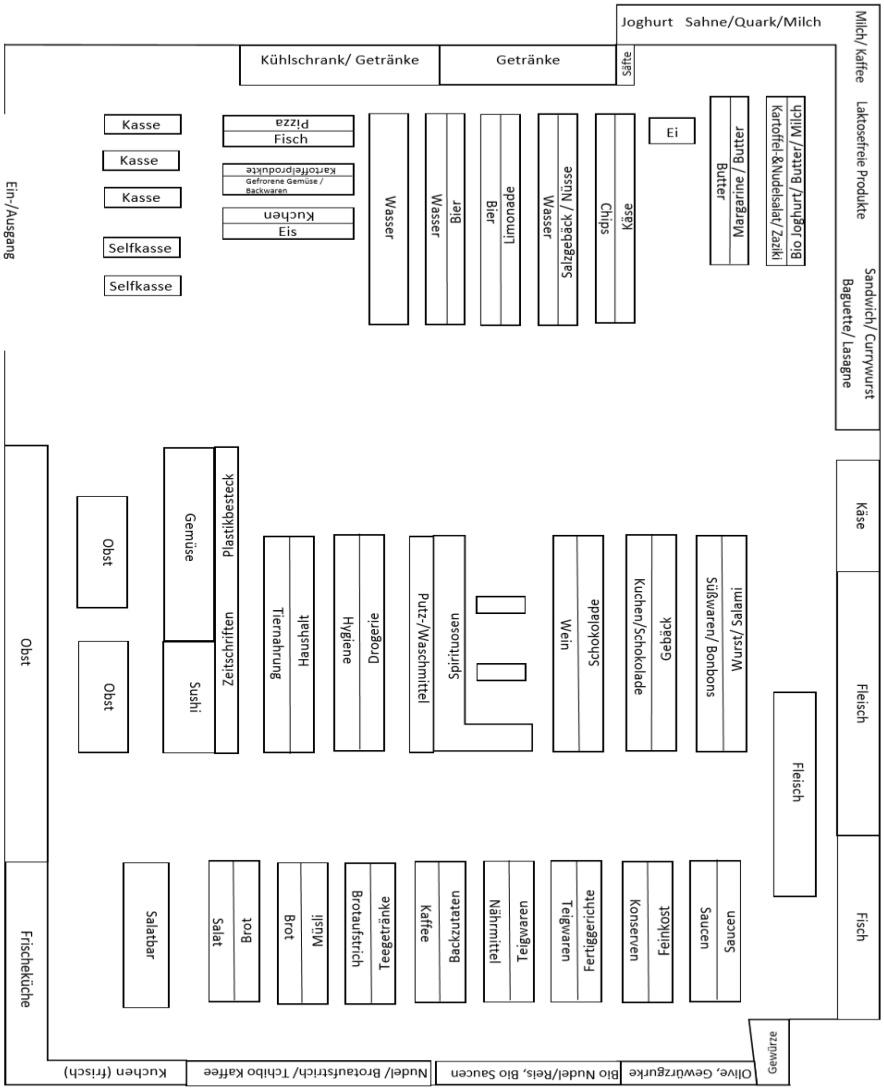


Abbildung 9: Skizze von Dornseifer

ÜBERARBEITEN (SVEN) Scheduler schickt temporär beste Ergebnisse und bekommt allerbestes Ergebnis erst später.

Nachdem man den optimalen Weg von dem Scheduler zurückbekommt, wird die Einkaufsliste sortiert ausgegeben. Um den optimalen Weg zu erhalten, sollte der Benutzer die Einkaufsliste in dieser 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 Scheduler Ressourcentabelle beschreibt, auf welche Daten die einzelnen Komponenten Zugriff haben und beschreibt deren Beziehung.

Ressource	Verb	URI	Semantik	Contenttype-Request	Contenttype-Response
Worker					
	Post	/worker	Erstellt einen Worker und gibt UUID und Population zurück	application/json	application/json
	Put	/worker?uuid={parameter}	Akutalisiert den entsprechenden Worker und liefert eine neue Population zurück	applicaiton/json	applicaiton/json
Map					
	Get	/map	Liefert die Karte der Unity Instanz		application/json
	Post	/map	Setzt die Karte der Unity Instanz	application/json	
Path					
	Get	/path	Liefert den aktuell besten Weg zurück.		application/json

Tabelle 3: Scheduler Ressourcentabelle

Kommentiert [PS6]: TODO: Sobald vorhanden, Tabellen durch Swagger Dokumentation ersetzen und zusätzlich Swagger kurz erläutern.

Die HTTP-Fehlercodes werden auf HTTP-Anfragen von einem Server als Antwort zurückgegeben, ob es zu einem Fehler beim Aufruf gekommen ist. Der Fehlercode gibt Informationen darüber, welcher Fehler bei der Ausführung aufgetreten ist.

Ressource	Verb	Fehlercode		Beschreibung
/worker	POST	503	Service Unavailable	Die Map ist nicht gesetzt.
				Die maximale Anzahl der Arbeiter wurde erreicht.
	PUT	400	Bad Request	Population ist nicht valide (enthält z.B. null-Werte).
				Das JSON konnte nicht gelesen werden.
				Die UUID fehlt.
				Der Nutzer ist nicht registriert.
/map	GET	204	No Content	Die Map ist nicht verfügbar.
	POST	400	Bad Request	Das JSON konnte nicht gelesen werden.
/path	GET	503	Service Unavailable	Es gibt momentan keine registrieren Worker, die an der Lösung arbeiten.
		204	No Content	Die Worker haben noch kein Ergebnis geliefert.

Tabelle 4: Scheduler HTTP-Fehlercodes

5.2.2 Technologie

Kommentiert [PS7]:

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 2011 ist Kotlin im Verhältnis zu Java oder C eine junge Sprache. Trotz dieser kurzen Lebenszeit, ist Kotlin bereits verbreitet und wird etwa in der Android Entwicklung als bevorzugte Sprache von Google empfohlen [Haase].
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

Kommentiert [PS8]: Ggf. Unterkapitel zu Komponenten und generellen Kotlin Erläuterung? Erläutert zwar Code, ist aber doch sehr allgemein gehalten

Kommentiert [PS9R8]: Hoch kopieren.

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 **Fehler! Verweisquelle konnte nicht gefunden werden.**), welche anstatt des Index, wie die 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 10: Extensionfunction für MutableList

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 **Fehler! Verweisquelle konnte nicht gefunden werden.** zu sehen ist.

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

Abbildung 11: Nutzung der Extensionfunction

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. Dabei wird dem Entwickler allerdings nicht die Möglichkeit genommen, diese Abhängigkeit selbst aufzulösen sofern nicht benötigt bzw. explizit nicht gewünscht. Ein Beispiel für die Nutzung von Coroutinenen kann im REST-Service gefunden werden. Alle Funktionen, welche Clientanfragen verarbeiten sind suspend (siehe **Fehler! Verweisquelle konnte nicht**

gefunden werden.) und können so asynchron und mehrfach vom Server ausgeführt werden.

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

Abbildung 12: Header der addWorker Funktion

5.3 Worker

Worker steht ständig in Verbindung mit dem Scheduler, um neue Individuen abfragen zu können. Jeder Worker berechnet mittels Genetische Algorithmen und Travelling Salesman Problem eine neue Sub Population und liefert diese an den Scheduler. Die durch den Worker ermittelten Distanz und Fitness Werte von den Individuen der Population, kann der Scheduler den besten Pfad ermitteln. Zurzeit werden 100 Generationen verwendet. Worker achtet auch ab und zu auf Mutation, der dann für den Fall, dass sich die Individuen nicht mehr "verbessern", die Pfade (Individuen) zufällig mutiert.

Kommentiert [PS10]: 100 Generationen […] ab und zu

Was heißt das, warum wurden die Werte so gewählt?

Der Worker hat keine eigene Schnittstelle. Die einzige Kommunikation, die vom Worker ausgeht, ist mit dem Scheduler. Es benutzt die Schnittstellen von 5.2.1, siehe Resource: "Worker".

5.3.1 Technologie

Kommentiert [PS11]: Nur ein Unterkapitel, macht man normalerweise nicht. Also am besten mit in den Rest des Textes eingliedern oder ein weiteres Kapitel finden und beschreiben.

Der Worker wurde in Java implementiert. Die Java-Technologie ist eine ursprünglich von Sun entwickelte Sammlung von Spezifikationen, die zum einen die Programmiersprache Java und zum anderen verschiedene Laufzeitumgebungen für Computerprogramme definieren. Diese Computerprogramme werden meistens in Java geschrieben. Da sich der Worker mit komplexen Algorithmen beschäftigt, wurde Java verwendet, da unser Team die höchsten Kenntnisse und Erfahrungen in dieser Sprache hatte.

Die "Requests" werden mittels JSON-Objects versendet. Als JSON Parser wurde die Bibliothek GSON von Google verwendet, da es eine relativ vereinfachte Variante ist, um JSON Objekte parsen zu können.

6 Implementierung

6.1 Unity-Frontend

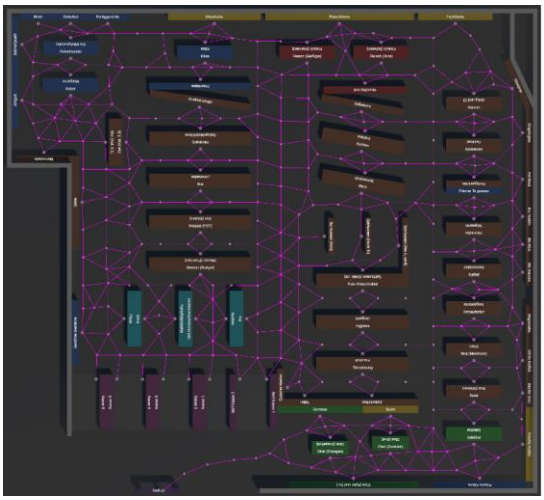
6.1.1 Erstellung des Geschäfts

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 Geschäftes 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 wiederspiegeln. Dabei ist ein echter Meter in Unity ebenfalls auf eine Längeneinheit übersetzt. Die nebenstehende Abbildung 10 zeigt das für das Projekt modellierte Geschäft „Dornseifer“ in der 3D-Welt. Wenn über das UI, mehr dazu im Kapitel 6.1.2, die Berechnung gestartet wird, transformiert das Unity-Fronend 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.

Abbildung 13: Modelliertes Geschäft in Unity

Allerdings gehen bei der Transformation der 3D-Welt in das JSON-Format einige Eigenschaften dieser verloren. Bei den meisten ist dies nicht von Relevanz, da die Berechnung ohnehin nicht von diesen 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. 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. Die folgende Abbildung 14 zeigt die Übersicht des Graphen in Unity (links im Bild) und das daraus resultierende JSON (rechts im Bild)



```
{
    "NavMesh": [
        {
            "id": "-4114",
            "position": {
                "x": -29,
                "y": -54
            },
            "nextNodes": [
                "-32780"
            ]
        },
        ...
        {
            "id": "-32780",
            "position": {
                "x": -24.469999313354492,
                "y": -50.36000061035156
            },
            "nextNodes": [
                "-4114",
                "-67150"
            ]
        }
    ]
}
```

Abbildung 14: 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 15). 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.

Kommentiert [ST12]: Future

Abbildung 15: Übersichts-Ansicht des Benutzerinterfaces

Bei einem Klick auf den Button „Einkauf planen“ wird das Menü zur Erstellung einer Einkaufsliste aufgerufen (vgl. Abbildung 16). In diesem hat der Benutzer die Möglichkeit, aus einer Liste an verfügbaren Produkten alle jene auszuwählen, welcher er einkaufen möchte. Dazu klickt dieser auf das gewünschte Produkt, um es in die jeweils andere Liste zu transferieren. Zusätzlich kann der Benutzer noch die Kasse wählen, an welcher er seinen Einkauf beenden möchte. Auch wenn diese Option in dem ~~M~~modellierten Geschäft nur minimale Auswirkungen auf das berechnete Ergebnis hat, so kann dies bei Geschäften mit größeren oder mehreren Kassenbereichen durchaus einen Einfluss auf die Berechnung haben. Neben der Auswahl der Kassen befinden~~t~~ sich noch Buttons, um zuvor definierte Einkaufslisten erneut zu laden. Dies ist vor allem für Testzwecke nützlich, um eventuelle Fehler reproduzieren zu können, als auch ein Anfang einer Funktion, um den Benutzer in Zukunft ein Speichern und Laden seiner Einkaufsliste zu ermöglichen.

Abbildung 16: Einkaufslisten-Planer

Bei Öffnen des in Abbildung 16 gezeigten Planungs-UI wird die link~~t~~e 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 17). 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 „GetComponentsInChildren“-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 ~~bei~~ 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 Buttons 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.

Kommentiert [ST13]: In 5.1 erklären?

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 shelfs = new List<Shelf>(shelfParent.GetComponentsInChildren<Shelf>());
31          shelfs.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 shelfs) {
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 17: Initialisierung des Einkaufslisten-Planers

6.1.3 Scheduler-Aufruf

Nachdem der Nutzer die Einkaufsliste zusammengestellt ~~hat~~ und die Berechnung gestartet hat, wird über die UI-Komponente die Transformation der Liste in ein JSON-Objekt, das Hinzufügen des Wege-Graphen und das Senden der Daten an den Scheduler in der Methode in Abbildung 18 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 wiederspiegelt kopiert. Das Senden der Daten an den Scheduler geschieht dann in Zeile 35. Die Klasse „SchedulerRestClient“ ist nach einem Singelton-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 auf statischen Klassen verfügbar ist. Mitgegeben wird neben der kopierten Einkaufsliste und der Scheduler Adresse noch zwei Delegates (ein Verweis auf eine Methode), welche sowohl während der Berechnung als auch nach dem Erhalt der endgültigen Rechenergebnisses aufgerufen werden.

Kommentiert [ST14]: Glossar

```
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 shoppping list...");
33          var hostUrl = schedulerIpField.text;
34
35          SchedulerRestClient.Instance.StartCalculationForShoppinglist(nodes, hostUrl, ProcessIntermediateResult, ProcessCalculationResult);
36          statisticsUI.ResetAndStartTimer();
37      }
```

Abbildung 18: 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 19. 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“ übergeben wurde. Als Zwischenaktion wird beispielsweise das aktuelle Zwischenergebnis im UI angezeigt. Wenn in dem 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($"Cant 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 19: Abfragen des aktuellen Rechenergebnisses

6.1.4 Visualisierung

Sobald das 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 20 stellt einen Beispielhaften Überblick über diese genannten Funktionalitäten.

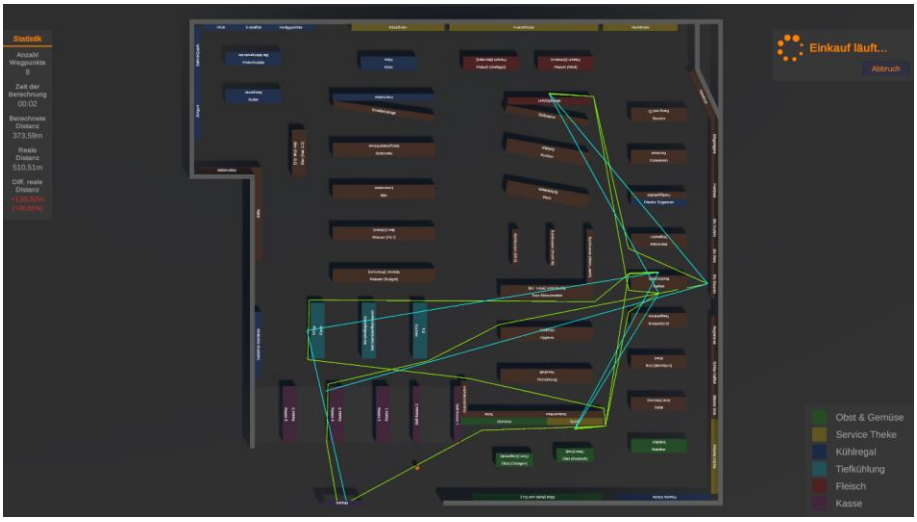


Abbildung 20: Visualisierung des Rechenergebnisses

Kommentiert [ST15]: TODO: Bild vom echten Ergebnis

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“-Komonente ein Array an Positionen gegeben, über die dieser dann die Linie aufspannt. Die Zusammenstellung des Positions-Array geschieht in der Methode in Abbildung 21. 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 wiederspiegeln. Aus diesem Grund werden auch zwei verschiedene Linien in Abbildung 20 angezeigt.
Die türkise Linie ist aus exakt den Wegpunkten, die aus der Berechnung stammen erstellt worden (nicht in der Methode in Abbildung 21 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 21 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 Navigationsgraphen, 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 angezeigt Linie nicht im Boden, und damit nicht mehr 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 Line 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
52                  foreach(var point in path.corners) {
53                      linePoints.Add(new Vector3(point.x, point.y + hightOffsetPath, point.z));
54                  }
55              }
56          }
57
58          rendererPath.positionCount = linePoints.Count;
59          rendererPath.SetPositions(linePoints.ToArray());
60
61          rendererPath.Simplify(1);
62      }
```

Abbildung 21: Berechnung der Visualisierung

6.2 Scheduler

6.2.1 Der Ktor-Server

Kommentiert [PS16]:

Wie bereits im Kapitel 5.2.2 Technologie 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 die erste 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 abdeckt.
Nach der Einbindung kann der Server u a. mittels der embeddedServer-Funktion erstellt und konfiguriert werden, wie in Abbildung 22 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 22: 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 23 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 23: GET /worker Definition

Über den weitergereichten call-Parameter vom Typ ApplicationCall besteht die Möglichkeit auf alle benötigten Informationen wie Requestbody oder sonstige zuzugreifen. Abbildung 24 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 24: Umsetzung der logRequest-Funktion

6.2.2 OpenAPI 3.0 Dokumentation

Kommentiert [PS17]: TODO: Dokumentieren, sobald umgesetzt.

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. Sprint 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-swagger[5] oder Ktor-OpenAPI-Generator[6] 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 (Sheila)

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. Die Struktur kann der **Abbildung (???)** entnommen werden. 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 Navigationsgraph 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.

[5] Implementierung durch nielsfalk vgl. https://github.com/nielsfalk/ktor-swagger
[6] 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, HttpStatusCode.OK)
224         } ?: respondJsonError(call)
225     }
226
```

Abbildung 25: Abspeichern der Einkaufslistenelemente und des Navigationsgitters

Wie man der Abbildung 25 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 (Abb. 25 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 Abhängigkeitsproblem. Das Erstellen dieser Population geschieht analog zu den anderen Populationen im Worker (siehe Kapitel 6.3 Worker).

Kommentiert [PS18]: Klingt komisch, die Worker erzeugen keine Populationen mehr, sondern bearbeiten diese nur noch.

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 26: 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 27).

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

Abbildung 27: Initialisierung des Workers beim Scheduler

Wurde der Worker erfolgreich erstellt, wird der *HttpStatus Created* zurückgegeben.

Fragt derselbe Worker im nächsten Schritt mit seiner UUID als Parameter an und kann dieser im Scheduler identifiziert werden, so erhält er eine Subpopulation und den Navigationsgraph, damit dieser arbeiten kann. Sind alle Berechnungen auf Seiten des Workers abgeschlossen, wird die Population dem Scheduler mitgeteilt.

Kommentiert [PS19]: Hast recht der steht noch drin, sollten wir denke ich aber noch in OK umbauen, oder ist es http konform bei einem Get Created zurück zu bekommen? Meines wissens nach nicht.

Kommentiert [PS20]: Daten bekommt der Worker schon beim Initialen Get auf /worker

Sowohl die einzelnen Individuuen der mitgeschickten Population als auch das beste Individuum werden aktualisiert und abgespeichert (siehe Abbildung 28).

```
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 28: Aktualisieren des Workers beim Scheduler

Bei der Abspeicherung des besten Individuums musste ein Kriterium gefunden werden, sodass 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). 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 rausgeschmissen 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.
Jedes Ergebnis von den Workern sendet der Scheduler an das Unity-Frontend. 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 das Unity-Frontend mitten in der Berechnung einer Navigation 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 29). Danach kann der Scheduler eine neue Einkaufsliste in Form von Einkaufslistenelementen und Navigationsgraph wie es am Anfang dieses Kapitels beschrieben worden ist, abspeichern und den Workern eine neue Aufgabe geben.

Kommentiert [PS21]: Direkt seden können wir an Unity momentan nicht. Unity fragt immer an.

```
/**
 * 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 29: Löschen einer alten Einkaufsliste

6.3 Worker

6.3.1 Genetischer Algorithmus in Zusammenhang mit “Travelling Salesman Problem”

Die Implementierung der genetischen Algorithmen fand im Worker statt.

Da der Worker eine Sub Population vom Scheduler bekommt, wendet es darauf 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 (Anfangs ist diese Population leer aber enthält dieselbe Größe wie die übergebene Population).

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

Abbildung 30: 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.

Durch das Crossover entsteht ein “child” der dann ~~eben~~ in die neue Population hinzugefügt wird. Dieser Vorgang wird solange ausgeführt, bis die ~~schleife~~ 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 31: Crossover

Wichtig: Die Crossover Funktion, die in Abbildung 31 gezeigt wird, rekombiniert alle~~n~~ Gene~~n~~ (Produkte) außer den ersten und letzten. Denn die erste Gene repräsentiert den Eingang des Ladens und die letzte Gene den Ausgang (Kasse).

Im nächsten Schritt erfolgt die Mutation der neuen 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”.

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

return newPopulation;
```

Abbildung 32: Mutation

Kommentiert [PS22]: Heißt es nicht, „das erste Gen“?

Kommentiert [PS23]: Klingt so als würde dies nur mit Individuen geschehen, die vorher selektiert und mittels Chrossover verändert wurden. Meines Verständnisses nach wird die Mutation auf jedes Individuum angewandt.

Kommentiert [PS24]: Siehe Kommentar in 5.3, was heißt ab und zu. Am besten oben erläutern und hier nochmal referenzieren.

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 33: Selektion der Überlebenden für nächste Generation

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

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

Abbildung 34: Berechnung bester Pfad

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

Zusammenfassung der Population:

Population => besteht aus mehreren Individuen (Mehrere Pfade).

Ein Individuum => Ein Pfad das aus mehreren Produkten besteht.

Gene => Das eigentliche Produkt bzw. Regal.

StartGene eines Individuums => Eingang des Ladens.

EndGene eines Individuums => Ausgang (Kasse) des Ladens.

6.3.2 A*

Kommentiert [PS25]: Glossareinträge?

Kommentiert [PS26]: Beschreibung endet mit der Rückgabe einer Selektierung. Weiterer Prozess bis zurück zum Scheduler sollte zumindest kurz beschrieben werden.

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-Rechner besitzen, war der Umgang mit Mac für alle gewöhnungsbedürftig. Insbesondere waren die Tastenkombinationen, die nicht identisch sind mit den Tastenkombinationen von Windows am Anfang problematisch, 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 logischerweise darauf geeinigt wurde, dass die Komponente Unity das Format vorgibt und eine Demo hochgeladen wurde, die für alle verfügbar war, war die Fehlersuche aufgrund nicht lesbarer JSONs 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, die jedoch von der anderen Komponente nicht abgefangen und bearbeitet wurden. 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

Abhängigkeitsproblem

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, da sonst …. ? (Phillip fragen/Code)

(Worker_count +1)² <= Populationsize < product.size² (Was ist hiermit gemeint)

Kommentiert [PS27]: Es sollten immer mehr Subpopulationen als Worker vorhanden sein, damit alle registrierten Worker jederzeit Zugriff auf eine Subpopulation haben. Auch eine Implementierung mit mehr Workern wäre denkbar würde allerdings zu Wartezeiten seitens der Worker führen.

Es mussten zwei Gleichungen erfüllt sein:

I. Workersize +1 > Subpopulation

II. Subpoulation = Populationsgröße/(Workersize +1)

➔ Workersize +1 > Populationsgröße /(Workersize +1)

➔ (Workersize +1)² > Populationsgröße

Kommentiert [PS28]: Wird nach Sheilas Plan bereits in der Implementierung der Worker erwähnt. Entsteht ein Mehrwert durch die doppelte Erwähnung. Könnte man an einer Stelle die Logik nur kurz Erläutern und hier das zu Grunde liegende Problem?

Somit ergib sich für 2 Worker beispielsweise eine Populationsgröße von max. 8.

Festlegung weiterer Konstanten (Sheila)

-> Worker_response_time

-> Demo_individual_size: wie viele individuen sollten abgewartet werden, bis der Scheduler überprüft ob min delta erreicht ist.

-> Min_delta: wie groß soll der Unterschied zwischen zwei ergebnisssen sein?

Verweis auf Kapitel messung

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 die Kasse laufen sollte, 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.
Außerdem stellt sich die Figur beim Abschließen des Einkaufs an der falschen Seite der Kasse an.
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. Die Figur muss allerdings bei der Darstellung um das Regal herumlaufen und somit ist dies kein guter Ansatz.

Lösung: Ein zusätzliches “Gitter” von Punkten wird übergeben.

8 Messungen

8.1 Verwendete Tools

1. Vorschlag: Pakete vom Scheduler mittels Wireshark sniffen und auswerten.

2.Vorschlag: lasttest mit JMeter

3.Vorschlag einfache Diagramme für die Konstanten des Scheduler mit Chart.js (evtl. Auch zur Visualisierung der Messungen) (**Implementierung im Worker: Volkan, Ayhan Auswertung: Sheila)**

- Für **jeden** Testfall benötigen wir zwei Diagramme
    - Distanzverbesserung über der Zeit (x-Achse Zeit in millisekunden, y-achse distanz in meter)
    - Summe der Ergebnisse über der Zeit (x-achse Zeit in millisekunden, y-achse Summe der Ergebnisse)
- Demo_size -> Ab wie vielen Ergebnissen konvergiert das Ergebnis (wenn es konvergiert)
- Min_Delta: Wenn es konvergiert ->???
    - Wenn es nicht konvergiert: gibt es trotzdem signifikante Punkte, um die es evtl. Umherspringt.

8.2 Testszenarien

8.2.1 Ein Scheduler - ein Worker

8.2.2 Ein Scheduler – mehrere Worker

8.3 Messergebnisse

- Traffic
- Stresstest
- Flaschenhals
- Lokales Maxima
- Vergleich mit Messungen der Luft und Reale Messungen

9 Fazit

9.1 Zeitmessung

erwähnen.

9.2 Erfahrungsbericht

- zusammenfassen, falls besondere Tendenzen oder Einzelfälle auftauchen, diese gern hervorheben.
- Tabelle Berührungspunkte hierhin

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 Chemitz

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)

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 asyncronous migration, Artif Life Robotics

Feldfunktion geändert

11 Anhang

11.1 Ergebnisprotokolle

10.1.2. Tabelle Berührungspunkte während des Projekts

Berührungspunkte während des Projekts

	Ayhan	Philipp	Sheila	Sümeyye	Sven	Volkan
Apple						
Unity						
C++						
Kotlin						
Java						
Genetische Algorithmen						
Traveling Salesman						
Verteilte Architekturen						

Legende	
Dies war mir völlig unbekannt	
Ich habe davon gehört, bin aber nie/selten in Berührung damit gekommen	
Ich habe mich schon einmal damit auseinandergesetzt, aber bin kein Experte	
Ich bin Experte in diesem Bereich	

10.1.3. Erfahrungsbericht von allen

10.1.4. Zusammenfassung

11.2 Zeitprotokolle