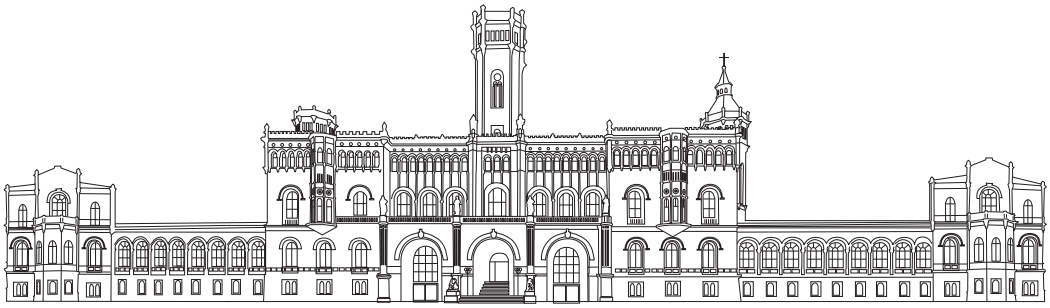


LEIBNIZ UNIVERSITÄT HANNOVER

FAKULTÄT FÜR ELEKTROTECHNIK UND INFORMATIK

FACHGEBIET MENSCH-COMPUTER-INTERAKTION

HEURISTISCHE OPTIMIERUNG VON SCHWARM-AGENTEN FÜR UNBEKANNTE LOGISTIK-PROBLEME

Abschlussarbeit zur Erlangung des
akademischen Grades Bachelor of Science

vorgelegt von

Ole Brenner

10043819

am 29. August 2024

Erstprüfer :	apl. Prof. Matthias Becker
Zweitprüfer :	Prof. Michael Rohs
Betreuer :	apl. Prof. Matthias Becker

ZUSAMMENFASSUNG

In dieser Arbeit werden zwei heuristische Verfahren zur Lösung von unbekannten logistischen Problemen vorgestellt und Implementiert. Zur Modellierung von Multi-Agent System (MAS) Problemen wird das AbstractSwarm Simulation System (ASSS) genutzt, welches unter anderem für die AbstractSwarm Multi Agent Competition (ASMAC) verwendet wird. Zur Lösung dieser Probleme wird eine heuristische Parameteroptimierung mit einfachen Kriterien zur Entscheidungsfindung vorgestellt und Implementiert. Zusätzlich wird diese Implementierung mit einem Genetischer Algorithmus (GA) erweitert, sodass die Anpassungsfähigkeit erhöht wird.

In der Auswertung der Implementierungen erzielen die Parameteroptimierung und der GA eine Effizienz von ≈ 34.87 und ≈ 33.97. Verglichen mit der besten eingereichten Implementierung (≈ 35.83) der ASMAC 2024 sind diese Verfahren effizienter in der Problemlösung. Eine eingereichte Zwischenversion der Parameteroptimierung belegt den 2. der ASMAC 2024. An getesteten unbekannten Problemen ist der GA effizienter als die Parameteroptimierung.

INHALTSVERZEICHNIS

1 EINLEITUNG

Logistische Probleme können verschiedene Szenarien aus dem realen Leben modellieren. Heuristische Optimierungsverfahren werden in diesem Kontext verwendet, um begrenzten Ressourcen effizient aufzuteilen. Diese Verteilung der Ressourcen kann mithilfe von Zeitplänen dargestellt werden. Ein effizienter Zeitplan enthält möglichst wenig Wartezeit, sodass ein Ablauf in möglichst geringer Zeit ausgeführt werden kann und der Nutzen von eingesetzten Gütern maximal ist.

Beispielhaft kann die Logistik im Gesundheitswesen betrachtet werden. Im Gesundheitswesen wird unter anderem die Verfügbarkeit von Personal, Patienten und Räumen berücksichtigt. [Simić et al. 2014] Erstellte Zeitpläne müssen räumliche und zeitliche Bedingungen einhalten, die durch das modellierte Problem definiert werden. Beispielsweise kann die Reihenfolge in der eine Person verschiedene Behandlungsräume aufsucht entscheidend sein oder ein verwendeter Raum muss vorbereitet werden bevor ein Patient in diesem behandelt werden kann.

Das Meal Delivery Routing Problem (MDRP) ist ein weiteres Beispiel für die Modellierung eines logistischen Problems. [Giraldo-Herrera und Álvarez Martínez 2024] Die Lieferung von Essen soll in geringerer Zeit geschehen, sodass die Wartezeit auf eine Bestellung möglichst gering ist. Dafür müssen die Wege der Lieferanten optimiert werden, um möglichst kurze Lieferwege zu erzielen.

Die Modellierung der betrachteten logistischen Probleme erfolgt mit dem Abstract-Swarm Simulation System (ASSS) von Apeldoorn et al. [2021]. Dieses wird im Rahmen der AbstractSwarm Multi Agent Competition (ASMAC) zur Analyse von Implementierungen an vorher unbekannten Problemen verwendet. Dies ermöglicht insbesondere die Modellierung von Multi-agent Problemen, in denen verschiedene Agenten mit der Umgebung interagieren, um definierte Aufgaben zu lösen. In dieser Arbeit werden zwei heuristische Optimierungen vorgestellt und implementiert: Eine Parameteroptimierung sowie ein Genetischer Algorithmus (GA) als Erweiterung.

2

RELATED WORK

In “Training agents for unknown logistics problems” von Schmidt und Becker [2023] werden bereits Verfahren zur Lösung von Problemen vorgestellt, die mit dem ASSS modelliert sind. Das Fazit dieser Arbeit ist, dass eine große Anzahl verschiedener Probleme effizient gelöst werden kann, wenn verschiedene Algorithmen kombiniert verwendet werden.

Bereits Wolpert und Macready [1997] beschrieben mit den “No Free Lunch Theoremen”, dass kein allgemeingültiger perfekter Optimierungsalgorithmus für die Klasse der Optimierungsprobleme existiert. Ein Optimierungsverfahren, welches auf eine Klasse von Problemen zugeschnitten ist im Durchschnitt die gleiche Effizienz erzielen wie jedes andere Verfahren wenn alle Probleme berücksichtigt werden. Dies bedeutet, dass die hier vorgestellten Implementierungen für die präsentierten Logistischen Problemen eine effiziente Lösung finden können. Wird aber die gesamte Klasse an Problemen betrachtet ist durchschnittliche Effizienz gleich.

Dennoch werden zwei heuristische Verfahren vorgestellt, die zur Lösung von Problemen, die mit AbstractSwarm modelliert werden, verwendet werden können. Mithilfe des GA soll eine Anpassung an die betrachteten Probleme erfolgen. Die Implementierungen verwenden die konkrete Funktionsweise des ASSS. Außerhalb dieses Anwendungsbereich ist der nutzen der Implementierungen eingeschränkt.

Für andere spezifische Problemstellungen existieren bereits verschiedene Implementierungen von Genetische Algorithmen: Liao et al. [2024] präsentiert beispielsweise eine Implementierung zum Erlernen von Ampelschaltungen. Die Ampelschaltungen von benachbarten Kreuzungen sollen dafür so geschaltet werden, dass alle Fahrzeuge eine möglichst geringe Wartezeit haben und ein ausreichender Verkehrsfluss hergestellt werden kann. Xu et al. [2024] beschreibt eine andere Verwendung von einem GA zur Optimierung von Arbeiterzeitplänen in logistischen Lagern. Das Ziel ist die Minimierung der benötigten Arbeitstage von Beschäftigten unter Berücksichtigung von notwendigen Bedingungen. Unter anderem ist die Arbeitszeit von einem Beschäftigten pro Tag beschränkt.

Beide Anwendung werden zur Lösung dieser spezifischen Problemstellungen verwendet. Das ASSS ermöglicht die Modellierung von verschiedenen Problemen, sodass die vorgestellten Implementierung in einer größeren Problemdomäne angewendet werden können.

3

ABSTRACTSWARM SIMULATION SYSTEM

Das AbstractSwarm Simulation System (ASSS) wurde von dem Institut für Medizinische Biometrie, Epidemiologie und Informatik (IMBEI) zur Modellierung von logistischen Problemen entwickelt. Die Software ist ein Multi-Agent System (MAS) und dient der Analyse von Lösungen für Probleme, die mit AbstractSwarm modelliert sind. [Apeldoorn 2013] In diesem Kontext sollen Zeitpläne in medizinischen Einrichtungen analysiert und optimiert werden. Die Software wird unter anderem für die ASMAC genutzt. [Apeldoorn et al. 2024] Das ASSS ermöglicht eine graphische Darstellung von modellierten MAS Problemen als Graphen. Zusätzlich stellt das ASSS Schnittstellen zur Verfügung, die zur Entscheidungsfindung der Agenten genutzt werden. Die Implementierungen werden dann genutzt, um eine Simulation durchzuführen und anschließend ein Zeitplan für das Problem zu erstellen. Dieser Zeitplan enthält die Informationen, zu welchem Zeitpunkt ein Agent eine Station besucht hat. Mithilfe des Zeitplans wird die Wartezeit aller Agenten total waiting Time (TWT) berechnet. Bei aufeinanderfolgenden Simulationen wird immer der Zeitplan mit der geringsten TWT betrachtet.

Zusätzlich stellt das ASSS eine Visualisierung des erstellten besten Zeitplan zur Verfügung. Diese Visualisierung und die Informationen, wann welcher Agent eine Station besucht hat, sind für die weiteren Auswertungen nicht relevant. Die Zeitpläne können aber genutzt werden, um das Verhalten von Agenten in der Simulation näher zu analysieren.

3.1 MODELLIERUNG

Das ASSS basiert auf "AbstractSwarm - Graphical Modelling Language for MAS" von Apeldoorn [2013]. In diesem wird die formale Grundlage für die Implementierung definiert. Ein Logistisches Problem wird mithilfe eines Graphen modelliert. Diese Modellierung wird von Apeldoorn [2013] ausführlich beschrieben. In der aktuellen Version des ASSS sind nicht alle Funktionen vollständig Implementiert und werden deshalb nicht berücksichtigt. Im folgenden werden die berücksichtigten Komponenten näher beschrieben:

Ein "AbstractSwarm" Graph wird definiert als Tupel:

$$G := (K, E)$$

Im folgenden wird ein Graph auch als Szenario bezeichnet. Ein Graph stellt immer ein konkret modelliertes Problem dar. K beschreibt die Knoten des Graphen G und ist definiert als

$$K := P_{T_A} \cup P_{T_S}$$

P_{T_A} beschreibt die Menge an Agententypen $T_A \in P_{T_A}$. T_A enthält eine Gruppe von Agenten $a \in A$. Mit $|T_A| \in \mathbb{N}$ wird die Anzahl von a beschrieben, die zu dem Typ T_A gehören. $T_S \in P_{T_S}$ beschreibt die Gruppierung von Stationen $s \in S$ mit Typ T_S. $|T_S| \in \mathbb{N}$ beschreibt die Anzahl von $s \in S$, die zum Typ T_S gehören. T_A und T_S verfügen über eine Bezeichnung zur Benennung des Typen. Alle Agenten und Stationen die zum gleichen Typ gehören verfügen über die gleichen Attribute mit gleichen Eigenschaften.

E beschreibt die Kanten von G die zwei verschiedene Komponenten $k_1, k_2 \in K$ miteinander verbinden. E ist definiert als $E := E_{place} \cup E_{time} \cup E_{visit}$. Mithilfe von E_{place} werden räumliche Abhängigkeiten beschrieben. Mit E_{time} werden zeitliche Abhängigkeiten in dem Graph dargestellt. Dadurch können räumliche und zeitliche Probleme in G modelliert werden.

3.1.1 *Visit Kanten*

Eine "visit" Kante $e \in E_{visit}$ verbindet $T_A \in P_{T_A}$ und $T_S \in P_{T_S}$. Jeder Agent $a \in A_1$ mit dem Typen T_{A_1} muss eine Station $s \in S_1$ vom Typ T_{S_1} entsprechend der Attribute von T_{A_1} und T_{S_1} besuchen. Agenten beginnen die Simulation an einer beliebigen Station, die durch eine "visit" Kante mit dem Agenten verbunden ist. Verfügt eine Kante $e \in E_{visit}$ über ein "Bold" Attribut, starten die Agenten an dem verbundenen Stationstypen. Sind mehrere Stationstypen mit einem Agenten Typen durch "visit" Kanten mit "Bold" Attribut verbunden wird für jeden Agenten ein zufälliger Stationstyp von diesen Ausgewählt.

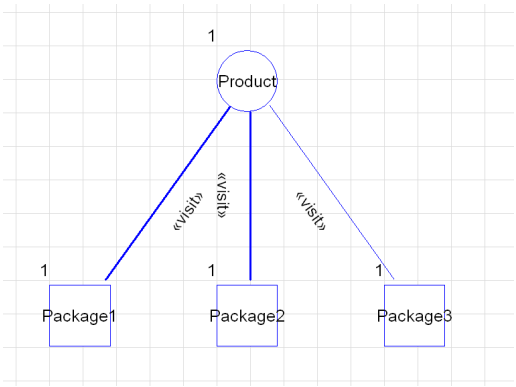


Abbildung 1: Szenario mit einem "Product" Agenten, der 3 Stationen besuchen kann. Die "visit" Kanten zu den Stationen "Package1" und "Package2" sind mit dem "bold" Attribut versehen. Der Agent beginnt die Simulation an einer dieser beiden Stationen. Erstellt mit dem ASSS

3.1.2 *Place Kanten*

Eine "place" Kante $e \in E_{place}$ verbindet zwei Stationstypen $T_S \in P_{T_S}$, sodass diese räumlich verbunden sind. Agenten können diese Kanten verwenden um zwischen zwei Verbundenen T_S hin und her zu gelangen. Jede "place" Kante verfügt über ein Gewicht $g \in N_0$. Seien $T_{S_1}, T_{S_2} \in P_{T_S}$ mit einer ungerichteten "place" Kante verbunden, dann benötigt ein Agent, um von einer Station $s_1 \in S_1$ zu $s_2 \in S_2$ zu gelangen g Zeiteinheiten. Wenn das Gewicht einer "place" Kante 0 entspricht, benötigt ein Agent keine Zeiteinheit um diese Kante zu verwenden. In einem Zeitschritt t kann ein Agent mehrere Kanten mit einem Gewicht von 0 verwenden.

Angenommen die Stationstypen T_{S_3}, T_{S_4} sind mit einer gerichteten Kante $e \in E_{place}$ von T_{S_3} nach T_{S_4} verbunden. Für einen Besuch einer Station von T_{S_4} ist ein vorheriger Besuch an T_{S_3} notwendig. Dadurch wird in einem Szenario die Reihenfolge in der Stationsbesuche erfolgen müssen festgelegt.

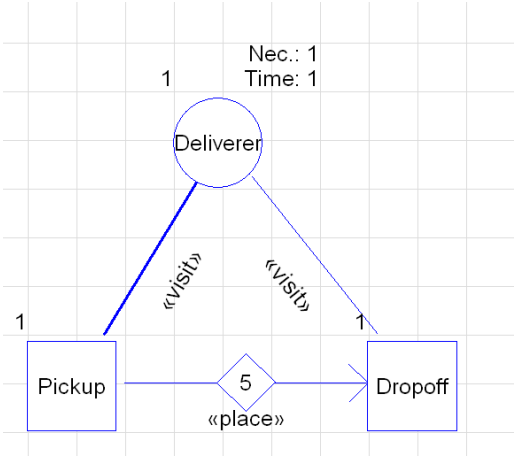


Abbildung 2: Einfaches Szenario mit einer gerichteten "place" Kante. Ein "Deliverer" Agent muss zuerst "Pickup" besuchen und gelangt anschließend über die "place" Kante in 5 Zeiteinheiten zu "Dropoff". Der Agent darf nicht zuerst "Dropoff" besuchen. Erstellt mit dem ASSS

3.1.3 *Time Kanten*

Sobald ein Agent den Besuch einer Station beginnt, wird ein "visit" Event ausgelöst. Seien zwei Typen T_x, T_y mit $x, y \in \{S, A\}$ mit einer ungerichteten Kante $e \in E_{time}$ verbunden. Dann muss bei x und y im gleichen Zeitschritt ein "visit" Event ausgelöst werden, damit ein Besuch beginnen kann. Für einen verbundenen Stationstypen gilt, dass an einer Station von diesem ein "visit" Event ausgelöst werden muss. Für einen verbundenen Agententypen gilt, dass ein Agent eine beliebige Station besuchen muss. Wenn bei x ein "visit" Event ausgelöst werden kann, bei y im gleichen Zeitschritt nicht, wird der Beginn von

Besuch bei x verzögert. Dies bedeutet, falls $x \in S$, dann kann ein Agent den Besuch einer Station nicht beginnen. Falls $x \in A$ kann ein Agent keine Station besuchen, bis bei y ein "visit" Event ausgelöst werden kann.

Sind T_x,T_y mit einer gerichteten Kante $e \in E_{time}$ von T_x nach T_y verbunden. Dann muss auf ein "visit" Event in x ein "visit" Event in y nach $t \in \mathbb{N}$ Zeiteinheiten erfolgen. Ein "visit" Event in y wird verzögert, bis das entsprechende Event bei x ausgeführt wurde und g Zeiteinheiten vergangen sind. g wird beschrieben durch das Gewicht der Kante e.

Wenn mehrere Kanten $e \in E_{time}$ mit einer Komponente verbunden sind, kann mithilfe des "and" Attributes bestimmt werden ob die Auslösung von "visit" Event an einer Komponenten gleichzeitig erfolgen muss oder die Auslösung an einer einzelnen Komponente ausreichend ist.

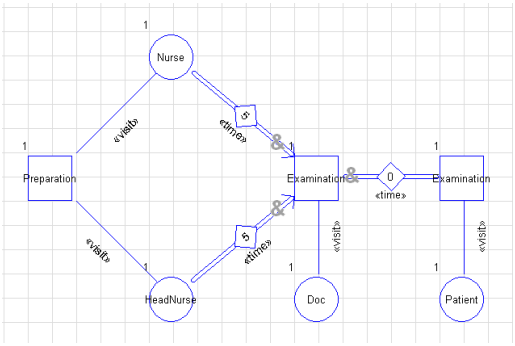


Abbildung 3: Szenario in dem ein Doktor einen Patienten untersuchen soll. Der Patient und Doktor müssen dafür die "Èxamination" Zeitgleich besuchen. Bevor der Doktor die Untersuchung beginnen kann müssen zwei Krankenschwestern gleichzeitig 5 Zeiteinheiten vorher beginnen "Preparation" zu besuchen. Erstellt mit dem ASSS

3.1.4 *Attribute*

T_A, T_S verfügen über Attribute, die die Eigenschaften jeder Komponente $a \in A$ und $s \in S$ beschreiben. Ein Attribut besteht aus einem Namen und einem dazugehörigen Wert, sodass

$$\{Y_x | Y \in \{T_A, T_S\}, x \in \{time, space, size, frequency, necessity\}\}$$

jeweils einen Wert zugeordnet wird. Jeder Agent $a \in A$ vom Typ T_A und jede Station $s \in S$ vom Typ T_S verfügt über die gleichen Attribute. Wird ein Attribut nicht gesetzt ist $Y_x = -1$, sodass für den Wert eines Attributes gilt: $Y_x \in \mathbb{N}_0 \cup \{-1\}$ Sofern nicht anders angegeben gilt $T_{S_x} = S_x$ und $T_{A_x} = A_x$.

Time A_{time} beschreibt die Zeiteinheiten, die ein Agent $a \in A$ benötigt, um den Besuch einer Station $s \in S$ zu beenden. S_{time} definiert die Zeiteinheiten die Agenten benötigen, um die Station $s \in S$ vollständig zu besuchen.

Wenn sowohl der Agent a als auch die Station s über ein "time" Attribut verfügen gilt $A_{time} \neq -1$ und $S_{time} \neq -1$. Für einen Besuch von a an der Station werden dann $min(A_{time}, S_{time})$ Zeiteinheiten benötigt. Gilt $A_{time} = -1$ und $S_{time} = -1$ ist ein Besuch von a an s nicht zeitlich begrenzt und dauert an bis die Simulation endet.

Space S_{space} beschreibt den aktuell vorhanden Platz in der Station $s \in S$. Falls $S_{space} = -1$ ist, dann ist die Station s in ihrem Platz nicht begrenzt, sodass beliebig viele Agenten diese Station gleichzeitig besuchen können. $T_{S_{space}}$ beschreibt die maximale Kapazität für alle Stationen $s \in S$. Es gilt immer $T_{S_{space}} \geqslant S_{space}$. Es gilt immer $A_{space} = -1$, da der Agent über kein $space$ Attribut verfügt. Das entsprechende Attribut für die Größe eines Agenten ist A_{size}.

Size A_{size} repräsentiert die Größe eines Agenten in einer Station. S_{size} beschreibt kein vorhandenes Attribut. Es gilt $S_{size} = -1$.

Frequency $A_{frequency}$ beschreibt die Anzahl an besuchen die jeder Agent $a \in A$ durchführen muss. Jeder Agent kann dabei beliebige Stationen wählen, die durch eine visit Kante mit T_A verbunden sind. $S_{frequency}$ beschreibt wie viele Besuche Agenten an der Station $s \in S$ durchführen müssen.

Necessity $A_{necessity}$ beschreibt wie oft jede einzelne Station $s \in S$ von jedem $a \in A$ besucht werden muss. Ist $T_S \neq -1$ muss jeder Agent $a \in A$, der mit T_S durch eine "visit'" Kante verbunden ist, jede Station $s \in S$ dem Attribut entsprechend $S_{necessity}$-mal besuchen.

Eine Simulation gilt als vollständig abgeschlossen, wenn alle Besuche entsprechend der Attribute $frequency$ und $necessity$ abgeschlossen wurden. Es wird davon ausgegangen, dass entweder ein Agenttyp oder ein Stationstyp über ein $frequency$ oder $necessity$ Attribut verfügen muss, sodass eine Begrenzung von Besuchen vorhanden ist. Agenten und Stationen die durch "visit" Kanten verbunden sind dürfen nicht gleichzeitig über $necessity$ und $frequency$ Attribute verfügen. Wenn eine Simulation nicht mehr abgeschlossen werden kann wird diese abgebrochen. Für diese Simulation wurde dann keine gültige Lösung gefunden.

3.2 SCHNITTSTELLEN

Das ASSS stellt zur Implementierung von Entscheidungsstrategien das "AbstractSwarmAgentInterface" zur Verfügung. Dieses enthält die drei Funktionen "evaluate", "communication" und "reward", die innerhalb einer Simulation für die Ausführung genutzt werden. Die Eingabeparameter von diesen sind teilweise identisch:

- **Agent me**
  Referenze auf den dazugehörigen Agenten

- **HashMap<Agent, Object> others**
  enthält alle anderen Agenten und ein dazugehöriges generiertes Kommunikationsobjekt
- **List<Station> stations**
  eine Liste aller im Szenario verfügbaren Stationen
- **long time**
  der aktuelle Zeitschritt

3.2.1 *evaluation-Funktion*

Die "evaluation" Funktion wird verwendet, um für einen Agenten und eine spezifische Station einen Wert zu ermitteln. Dadurch wird für einen Agenten eine Station ausgewählt.

Listing 1: Die "evaluation" Funktion ohne konkrete Implementierung aus dem "AbstractSwarmAgentInterface".

```
public static double evaluation(Agent me, HashMap<Agent, Object> others,
        List<Station> stations, long time, Station station ) {

}
```

Neben den in 3.2 beschriebenen Parametern bekommt die "evaluation" Funktion noch die auszuwertende Station als Eingabeparameter. Zurück wird der ermittelte Wert der Station gegeben.

Der Aufruf der "evalutaion" Funktion erfolgt für jeden verfügbaren Agenten und für jede Station die dieser Agent noch besuchen muss. Ein Agent wählt die Station als nächstes Ziel, die den höchsten Wert von der "evaluation" Funktion zugeordnet bekommen hat. Sollte der Wert von einem Agenten für mehrere Stationen identisch sein, wird eine zufällige Station mit höchstem Wert gewählt. Eine Station kann von mehreren Agenten gleichzeitig als Ziel ausgewählt werden. Der Aufruf von Agenten erfolgt in zufälliger Reihenfolge, jedoch werden für einen Agenten immer zuerst alle verfügbaren Stationen bewertet, bevor der nächste Agent gewählt wird. Ein Agent der keine Station mehr Besuchen kann wird nicht länger als Teil der Simulation gesehen.

In der Tabelle 1 werden zwei mögliche Aufrufszenarien von der Funktion "evaluation" beispielhaft dargestellt. Nachdem für einen Agenten alle Stationswerte berechnet wurden wird die entsprechende Station sofort ausgewählt, sodass bei der Auswertung des nächsten Agent, das Ziel des ersten Agenten bereits eingetragen ist.

Aufrufreihenfolge	Simulation 1	Simulation 2
1	Agent 1 Station A	Agent 2 Station A
2	Agent 1 Station B	Agent 2 Station B
3	Agent 2 Station B	Agent 1 Station A
4	Agent 2 Station A	Agent 1 Station B

Tabelle 1: Mögliche Aufrufszenarien der "evaluation" Funktion in dem ASSS für ein Szenario mit Agenten 1 und 2 und Stationen A und B

3.2.2 *communication-Funktion*

Die "communication" Funktion wird in jedem Zeitschritt $t \in \mathbb{N}$ für jeden Agenten aufgerufen, der im Szenario verfügbar ist. Sobald ein Agent keine Stationen mehr besuchen kann, wird die Funktion für diesen Agenten nicht länger aufgerufen.

Listing 2: Die "communication" Funktion ohne konkrete Implementierung aus "AbstractSwarmAgentInterface".

```
public static Object communication(Agent me, HashMap<Agent, Object> others,
        List<Station> stations, long time, Object[] defaultData ) {

}
```

Die Funktion verfügt als Eingabeparameter zusätzlich zu den in 3.2 beschriebenen Parametern, noch ein Array aus Objekten als Eingabe. Das Array besteht aus 3 Objekten: die von dem Agenten ausgewählte Station, die Zeiteinheit wann diese Station erreicht wird, sowie der Wert dieser Station. Die Rückgabe der Funktion ist ein einzelnes Objekt, welches in der HashMap für den Agenten gespeichert wird. Diese Objekte werden in den nachfolgenden "evaluation" und "reward" Funktionsaufrufen genutzt. Es ist möglich, dass eine Implementierung ein eigenes Objekt erstellt und dieses anstelle der initialen Daten verwendet.

3.2.3 *reward-Funktion*

Die "reward" Funktion wird für einen Agenten aufgerufen sobald ein Besuch bei einer Station beendet wird.

Listing 3: Die "reward" Funktion ohne konkrete Implementierung aus dem "AbstractSwarmAgentInterface".

```
public static void reward(Agent me, HashMap<Agent, Object> others,
        List<Station> stations, long time, double value ) {

}
```

Neben den Standard Parametern (beschrieben in 3.2), wird ein Wert $reward \in (0, 1]$ bereitgestellt. $reward$ wird mit der Arbeitszeit eines Agenten und dem aktuellen Zeitschritt berechnet. Der Wert ist größer je weniger ein Agent wartet. Die Berechnung erfolgt mit:

$$reward := \frac{t_{geleistet}}{Zeitschritt}$$

$t_{geleistet} \in \mathbb{N}$ beschreibt die Anzahl der Zeiteinheiten in der ein Agent eine Station besucht hat. Angenommen ein Agent $a \in A$ benötigt 5 Zeiteinheiten eine Station $s \in S$ zu erreichen und 5 Zeiteinheiten s vollständig zu besuchen. Die "reward" Funktion wird dann mit $reward = \frac{5}{10} = 0.5$ aufgerufen, wenn a in Zeitschritt 10 den Besuch von s abschließt.

3.3 SCHNITTSTELLEN-AUFRUF

Die Schnittstellen werden in der Reihenfolge aufgerufen, die in Algorithmus 1 dargestellt wird. Der Algorithmus stell nicht die tatsächliche Implementierung in dem ASSS dar. Es ist zu beachten, dass die verwendete Liste $agents$ (Zeile 1), die Agenten in zufälliger Reihenfolge enthält. Die für einen Agenten verfügbaren Stationen (Zeile 7) sind ebenfalls in zufälliger Reihenfolge. Dies entspricht der in 3.2.1 beschriebenen Funktionalität. Die Variable "time" beschreibt den aktuellen Zeitschritt in der Simulation. Sobald ein Agent keine Besuche mehr ausführen kann wird dieser aus der Liste "agents" entfernt.

Algorithm 1 Abfolge der Funktionsaufrufe

```
1:  agents ← list of all agents in the szenario
2:  time ← 0
3:  while agents is not empty do
4:      time ← time + 1
5:      for all agents do
6:          if current agent has no target then
7:              for all avaiable stations do
8:                  call evaluation function
9:              choose station with largest value as target for current agent
10:         call communication function
11:         if current agent finished current station then
12:             call reward function
13:         if current agent has no avaible station left then
14:             remove current agent
```

Sobald die aktuelle Simulation in ein Deadlock gerät, wird diese abgebrochen. Ein Deadlock wird erreicht, wenn in einer Simulation kein Zeitplan mehr erstellt werden kann ohne die Bedingungen des Szenarios zu verletzten. Näheres

zu Deadlocks wird in 5.2 beschrieben. Diese Funktionalität ist in Algorithmus 1 nicht modelliert.

3.4 BEISPIEL PROBLEME

Im folgenden werden 4 mit dem ASSS modellierte Probleme sowie Zeitpläne zur Lösung präsentiert. Diese sind Teil des später verwendeten Trainings-Set, bereitgestellt von der ASMAC. [Apeldoorn et al. 2024] Die Erstellung der Beispiellösungen erfolgte ebenfalls mit dem ASSS. Zur vereinfachten Zuordnung von Agent und Station in den Zeitplänen sind diese unterschiedlich eingefärbt. Die vorgestellten Szenarien modellieren verschiedene logistische Probleme.

- **CHANGEDManufacturingProcess**: Ein "Worker" Agent muss 4 verschiedene Aufgaben erfüllen, die unterschiedlich viele Zeiteinheiten benötigen. Die Aufgaben sind räumlich voneinander getrennt. Jede dieser 4 Aufgaben muss durch den "Preparer" Agent vorbereitet werden. Die Vorbereitung von jeder Aufgabe dauert 5 Zeiteinheiten. Eine Aufgabe kann von dem "Worker" begonnen wenn mindestens 5 Zeiteinheiten vergangen sind nachdem der "Preparer" Agent die dazugehörige Vorbereitungsaufgabe begonnen hat.

  Dieses Szenario modelliert ein Problem mit zeitlichen und räumlichen Abhängigkeiten. Zur Erstellung eines Zeitplans müssen diese Abhängigkeiten berücksichtigt werden.

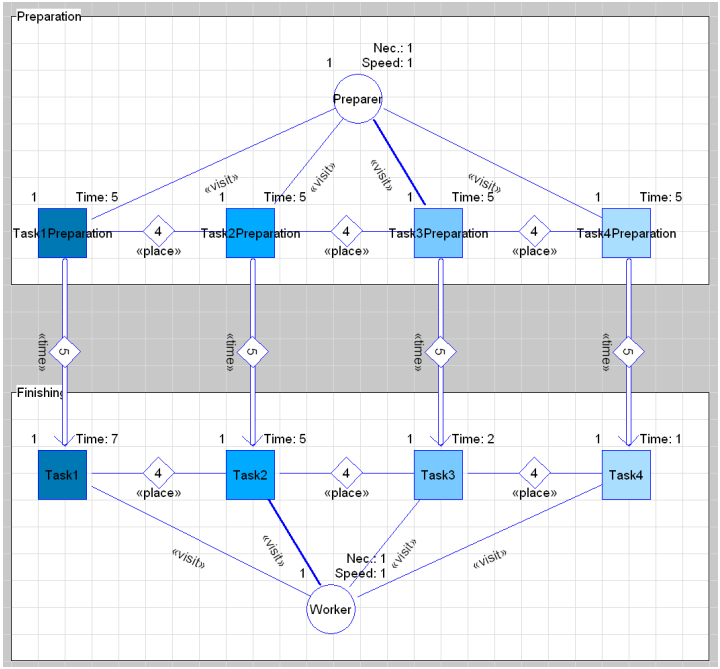


Abbildung 4: Modellierung des "CHANGEDManufactoringProcess" Szenarios aus dem ASSS. Quelle: Apeldoorn et al. [2024]

Der Agent "Worker" startet an der Station "Task2" und der Agent "Preparer" beginnt an der Station "Task3Preparation" die Simulation.

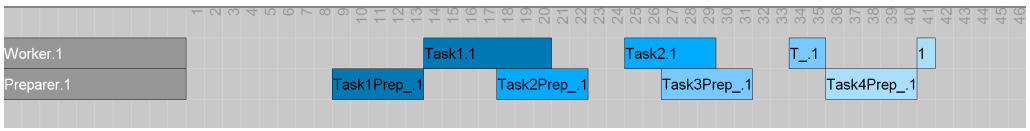


Abbildung 5: Ein möglicher Zeitplan des "CHANGEDManufactoringProcess" mit einer Optimalen TWT von 47.

In der Lösung in Abbildung 5 startet der "Preparer" an der Station "Task1-Preparation". Es werden 8 Zeitschritte benötigt, um diese Station zu erreichen. Anschließend werden die "Preparation" Stationen in der Reihenfolge $2, 3, 4$ besucht. Der "Worker" besucht die Stationen in der Reihenfolge $1, 2, 3, 4$. Die Station "Task4" benötigt nur 1 Zeiteinheit. Aus diesem Grund sollte diese möglichst am Ende besucht werden. Wenn beispielsweise Station "Task1" am Ende besucht wird ist die TWT der Lösung größer.

- **CHANGEDTreatments**: Zehn verschiedene Agenten müssen 4 verschiedene Stationen als Behandlung durchlaufen. Jede Behandlung bietet unterschiedlich viel Kapazität, benötigt unterschiedlich viele Zeiteinheiten und sind räumlich voneinander getrennt. Die Behandlungen $1, 2, 3$ sind untereinander mit einer Distanz von 5 verbunden. Die Behandlung 4 ist nur über Behandlung 2 erreichbar.

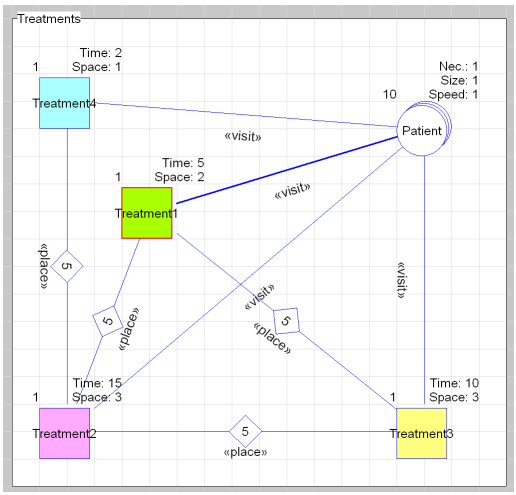


Abbildung 6: Modellierung des "CHANGEDTreatments" Szenarios. Quelle: Apeldoorn et al. [2024]

Alle Agenten beginnen die Simulation von "Treatment1" aus.

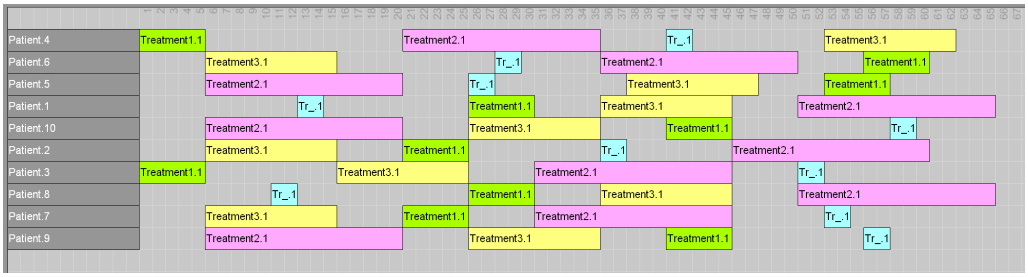


Abbildung 7: Eine mögliche Lösung des "CHANGEDTreatments" Szenarios. Die optimale TWT von diesem Zeitplan beträgt 330 Zeiteinheiten.

Für die Lösung dieses Szenarios müssen sowohl die räumlichen Abhängigkeiten und die Auslastungen der Stationen berücksichtigt werden. Die Auslastung der Station mit der längsten Besuchsdauer ist entscheidend für die Wartezeit im Zeitplan. Ohne die Pfad Kosten zu berücksichtigen benötigt die Simulation mindestens 60 Zeiteinheiten, wenn alle 10 Agenten Behandlung 2 besuchen, da nur 3 Agenten die Station gleichzeitig besuchen können.

- **SCALEDPackaging**: Zwei Produkte mit unterschiedlicher Größe müssen in zwei Paketen mit unterschiedlich viel Platz verstaut werden. Die Produkte müssen insgesamt nur eine der beiden Stationen einmalig besuchen.

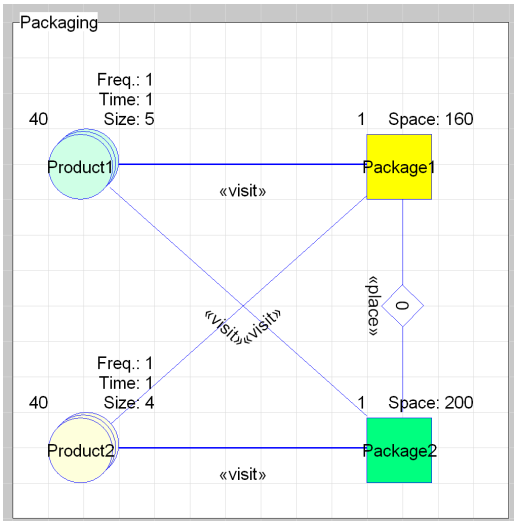


Abbildung 8: Das "SCALEDPackaging" Szenario mit 2 Produkten, die in 2 Paketen verstaut werden müssen. Jedes der Produkte existiert jeweils 40-mal. Quelle: Apeldoorn et al. [2024]

"Product1" beginnt die Simulation bei "Package1" und "Product2" beginnt die Simulation bei "Package2".

Abbildung 9: Ausschnitt des Zeitplans für das "SCALEDPackaging" Szenario. Die optimale TWT ist 0.

Für die Lösung dieses Szenarios (Abbildung 9) muss die Größe der Stationen berücksichtigt werden. Die 40 Agenten von "Product1" haben eine Gesamtgröße von 200. Die 40 Agenten von "Product2" haben eine Gesamtgröße von 160. Entsprechend kann eine Zuweisung von "Product1" zu "Package2" und "Product2" zu "Package1" erfolgen. Eine andere Zuweisung resultiert in einer höheren TWT. Es handelt sich um ein modelliertes räumlichen Problem.

- **SCALEDWardRound**: Ein Doktor muss 4 verschiedene Räume jeweils 2-mal besuchen. Dafür wird pro Station 10 Zeiteinheiten benötigt. Ein Doktor kann einen Raum nur besuchen, wenn die Krankenschwester gleichzeitig beginnt die Station "Doc" besucht. Die "Nurse" muss die Station "Doc" 8-mal besuchen und zusätzlich die Station "Office" 2-mal besuchen.

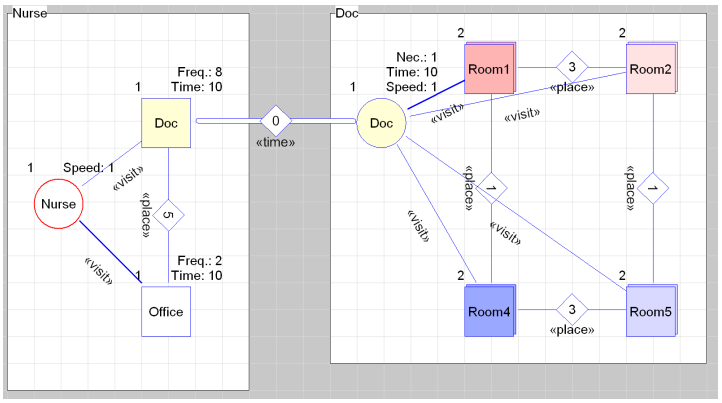


Abbildung 10: Das "SCALEDWardRound" Szenario mit einer Krankenschwester und einem Doktor. Quelle: Apeldoorn et al. [2024]

Ein Doktor startet die Simulation bei "Room1" und die Krankenschwester startet bei der "Office" Station. Eine mögliche Lösung von diesem

Szenario wurde aufgrund von Lesbarkeit im Zeitschritt 68 in zwei Teile unterteilt.

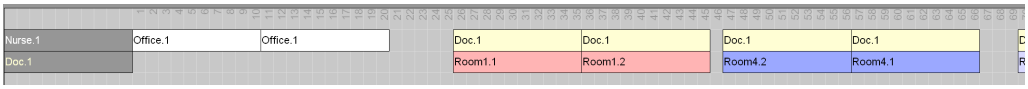


Abbildung 11: Erster Teil der vorgestellten Lösung von “SCALEDWardRound”. Die optimale TWT des Szenarios beträgt 40 Zeiteinheiten.

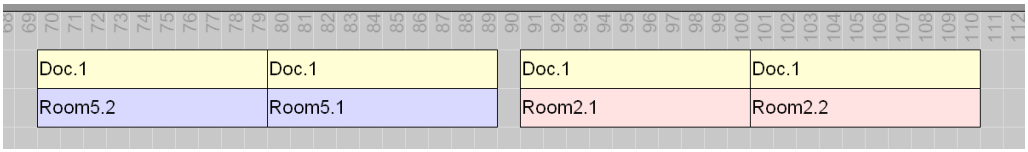


Abbildung 12: Zweiter Teil der vorgestellten Lösung von “SCALEDWardRound”.

Für die Lösung dieses Szenarios müssen die zeitlichen und räumlichen Komponenten berücksichtigen werden. Ein Doktor kann nur eine Station besuchen wenn die Krankenschwester die “Doc” Station gleichzeitig besucht. Aufgrund der räumlichen Trennung der Stationen ist es effizienter wenn die Krankenschwester zuerst die “Office” Station besucht.

4

HEURISTISCHE OPTIMIERUNG

Die Optimierung der erstellten Zeitpläne erfolgt durch die Minimierung der TWT. Die TWT ist die Summe der Wartezeit von jedem Agenten in einem Zeitplan. Mithilfe der Parameteroptimierung 4.1 und einem GA 4.2 soll ein möglichst effizienter Zeitplan für verschiedene modellierte Probleme gefunden werden.

Bei der Parameteroptimierung wird in jedem Zeitschritt der Wert einer Station für einen Agenten durch eine Kombination von Basis-Kriterien bestimmt. Die Idee dahinter ist, dass eine große Anzahl verschiedener Probleme aus einer Kombination von einfachen Kriterien effizient gelöst werden kann. Die Kombination verschiedener Kriterien basiert auf Wahrscheinlichkeiten. Ziel ist es, die Wahrscheinlichkeiten währen deiner Simulation anzupassen, sodass Kombinationen gewählt werden, die für Agenten die besten Stationen auswählen. Der Vorteil von einem solchen Verfahren ist, dass dieses ohne vorheriges Training auf ein Szenario angewandt werden kann. Ein Problem ist, dass keine neue Kombinationen oder neue Kriterien während den Simulationen erstellt werden können. Wenn eine Kombination aus Kriterien ein gegebenes Problem nicht effizient lösen kann, wird dieses verfahren keine effiziente Lösung für dieses finden. Als Erweiterung wird ein GA in 4.2 verwendet, um mehr Abwandlungen und neue Kombinationen zu ermöglichen. Eine Anpassung der verwendeten Kombinationen während den Simulationen soll eine größere Anzahl von Problemen effizient lösen können. Für diese Anpassungen ist eine größere Anzahl von Iterationen nötigt als bei der reinen Optimierung mit Basis Parametern. Es ist also zu erwarten, dass die Effizienz einer genetischen Implementierung auf wenige Iterationen geringer ist. Für unbekannte Szenarios, die durch Basis-Funktionen nicht optimal abgedeckt werden, könnte dieser GA eine bessere Lösung finden.

4.1 PARAMETEROPTIMIERUNG

Zur Bestimmung des Wertes einer Station $s \in S$ für einen Agenten $a \in A$ im Zeitschritt $t \in \mathbb{N}$ wird eine Kombination von Parametern T verwendet. Sei X_p die Wahrscheinlichkeit für einen Parameter $p \in P$. P beschreibt alle verfügbaren Kriterien. Dann ist die Wahrscheinlichkeit von T:

$$P(T) = \prod_{p \in P} X_p$$

$Wert(p) \in \mathbb{R}$ beschreibt den ausgewerteten Wert des Parameters durch eine entsprechende Funktion. Sei $\lambda \in (0, 1]$ ein zufälliger Wert. Der Wert von T wird beschrieben von:

$$Wert(T) := \sum_{p \in P} w_p * wert(p) \text{ mit } w_p = \begin{cases} 1 & \text{falls } \lambda \leqslant X_p \\ 0 & \text{sonst} \end{cases}$$

Für die Wahrscheinlichkeit $P(X_p)$ gilt: $0.05 \leqslant X_p < 1.00$. Dadurch wird verhindert, dass ein Parameter ausgeschlossen wird, der nicht deaktiviert wird. Ein Parameter kann deaktiviert werden, wenn der Parameter ein Kriterium beschreibt, welches nicht im Szenario vorhanden ist. Als konkretes Beispiel: Verfügt ein Szenario über keine ungerichteten "time" Kanten wird dieses Kriterium für die ungerichteten "time" Kanten deaktiviert und bei der Auswertung nicht berücksichtigt.

Am Beginn einer Simulation $i \in \mathbb{N}$ wird X für jeden Parameter $p \in P$ auf einen Initialwert gesetzt. Für diesen Wert gilt:

$$X_{P_i} = \begin{cases} 0.35 & falls\, i = 1 \\ \frac{S(X_{p_1}),\ldots,S(X_{p_{i-1}})}{i-1} & sonst \end{cases} \tag{1}$$

$S(X_{p_i})$ ist der durch die Simulation i veränderte Wert von X_p. Also die Wahrscheinlichkeit X vom Parameter $p \in P$ nach Beendigung der Simulation S_i. Zusätzlich wird die höchste Wahrscheinlichkeit X_p nach einer Simulation i auf 0.05 gesetzt. Dies soll ermöglichen, dass in nachfolgenden Simulationen andere Kombinationen ausgetestet werden. Die Anpassung von X_p während einer Simulation ist wie folgt:

Sobald ein Agent den Besuch einer Station beendet wird die Wahrscheinlichkeit der verwendeten Parameter um $0.2 * (reward\ Wert)$ erhöht. Der $reward\ Wert$ entspricht dem in 3.2.3 beschriebenen Wert. Bei einem Agenten mit weniger Wartezeit ist die Erhöhung von X_p größer als bei Agenten mit mehr Wartezeit. Alle Wahrscheinlichkeiten X_p werden anschließend normalisiert mit:

$$X_p = \frac{X_p}{\sum_{m \in P} X_m}$$

Jedoch ist zu beachten, dass alle $X_p < 0.05$ auf 0.05 gesetzt werden. Dies verhindert, dass ein Parameter p ausgeschlossen wird. Aus dem Ausschluss eines Parameters resultiert, dass die Wahrscheinlichkeit von diesem nicht mehr erhöht wird, da der Parameter nicht ausgewählt wird. Aus diesem Grund wird eine Wahrscheinlichkeit von unter 5% ausgeschlossen.

In der Berechnung 1 werden alle durchgeführten Simulationen berücksichtigt. Simulationen, die nicht abgeschlossen werden und deren berechnetet Wartezeit nicht $(current\ TWT) \leqslant 1,6 * (best\ TWT)$ entspricht, sollen nicht berücksichtigt

werden. Die current TWT entspricht der Wartezeit der Simulation S und die best TWT ist die minimale Wartezeit der bisherigen durchgeführten Simulationen. Im folgenden beschreibt $S(X_{p_n})$ mit $n \in i$ alle Simulationen die diesen Kriterien entsprechen. Die Berechnung der initialen Wahrscheinlichkeit von einem Parameter p wird durch die Gleichung 2 beschrieben.

$$X_{p_i} = \begin{cases} 0.35 & falls\, i = 1 \\ \frac{S(X_{p_1}),\ldots,S(X_{p_n})}{n} & sonst \end{cases} \tag{2}$$

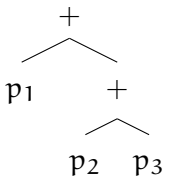
Für eine Kombination T ist dann $P(T)$ erhöht wenn die Wahrscheinlichkeit der Parameter in T erhöht wird. Das Ziel ist möglichst viele verschiedene Kombinationen zu testen. Eine hohe Abdeckung des Suchbereiches wird erreicht, wenn ein Parameter der sehr häufig verwendet wird, in der nachfolgenden Simulation weniger verwendet wird. Die Wahrscheinlichkeit der anderen Parametern wird beibehalten. Dadurch wird diese Kombination ohne den zurückgesetzten Parameter verwendet.

Zusätzlich definieren wir für ein Szenario eine Lösung Solution mit $y \in t$ als Liste von verwendeten Kombinationen T:

$$Solution_{szenario} := T_1, \ldots, T_y$$

Die TWT einer Solution wird mit erfolgreichen Abschluss der Simulation bestimmt, ist aber nicht eindeutig. Für eine Solution kann in mehreren Simulationen unterschiedliche TWT berechnet werden. Aus diesem Grund wird immer die von dem ASSS berechnete TWT von einer Lösung als Bewertung von dieser Verwendet. Die TWT wird als Heuristik verwendet, sodass keine komplexe Analyse einer Lösung notwendig ist. Dafür werden die Informationen wann ein Agent eine Station besucht nicht benötigt.

Die Darstellung von einer Kombination T erfolgt mit einem Binär-Baum. Seien $p_1, p_2, p_3 \in P$ drei für T ausgewählte Parameter. Dann kann diese Kombination durch folgenden binären Baum dargestellt werden:



Eine Auswertung dieses Baumes repräsentiert den *Wert* von T. Der Wert dieses Baums T wird berechnet: $Wert(T) = p_1 + (p_2 + p_3)$ Diese Visualisierung ermöglicht insbesondere eine einfachere Implementierung der Verfahren, die für den GA benötigt werden.

4.2 GENETISCHER ALGORITHMUS

Ein Genetischer Algorithmus verwendet die in der Natur vorhandenen Prozesse der Natürlichen Selektion und Genetik, um eine Lösung für ein Problem zu erzeugen [Goldberg 2002]. Dadurch sollen neue Kombinationen T zur Auswertung von Stationswerten für Agenten möglich werden, die initial nicht berücksichtigt sind.

Die hier verwendete Abfolge von dem GA wird von Sastry et al. [2014] beschrieben. Es erfolgt eine Aufteilung der Abfolge in 7 Teilschritte.

1. Initialisierung: Eine Population wird als Lösung "Solution" für ein Szenario betrachtet. Die Erstellung einer Population erfolgt mit dem in 4.1 beschriebenen Verfahren. Als initiale Population wird die zu diesem Zeitpunkt beste Lösung betrachtet. Zur Erstellung der ersten Ausgangspopulation werden zuerst $i \in N$ Simulationen durchgeführt.

2. Auswertung der Fitness: Die Bewertung der Fitness einer Population kann nur erfolgen, wenn die TWT durch eine Simulation bestimmt worden ist. Die Fitness wird einer $\mathtt{Solution}$ wird berechnet mit:

   $$\mathrm{Fitness}_{\mathtt{Solution}} = \frac{\mathtt{best}\ \mathrm{TWT}}{\mathtt{current}\ \mathrm{TWT}}$$

   Es gilt stets: $\mathtt{best}\ \mathrm{TWT} \leqslant \mathtt{current}\ \mathrm{TWT}$, sodass $\mathrm{Fitness}_{\mathtt{Solution}} \in [0, 1]$. Die Lösung einer nicht abgeschlossenen Simulation wird nicht bewertet. Es ist nicht möglich mehrere Populationen gleichzeitig zu bewerten.

3. Auswahl: Es wird ei Die Ausgangspopulation ist immer die aktuell beste Lösung $\mathtt{Solution}$. Wenn eine nachfolgende Lösung eine Identische TWT in der Bewertung erreicht ersetzt diese nicht die Ausgangspopulation. Zusätzlich wird eine Liste aus $n \in \mathbb{N}$ Lösungen verwendet, in der die besten vergangene Populationen gespeichert werden.

4. Rekombination: Bei der Rekombination wird eine neue Population durch die Kombination von zwei anderen Populationen erstellt. Eine neue Population wird mit der aktuell besten Population und einer gespeicherten Population bestimmt. Die Fitness der n gespeicherten Populationen wird um 10% verringert, sobald eine bessere Lösung gefunden wird.

5. Mutation: Die Mutation stellt eine zufällige Veränderung einer Kombination T aus einer Population dar. Die genauen Veränderungen werden in 4.2.2 beschrieben. Eine Mutation ist nur eine minimale Veränderung von einer einzelnen Kombination.

6. Auswechselung: Eine Ausgangspopulation wird ersetzt, wenn die berechnete TWT einer anderen Population geringer ist, als die der Ausgangspopulation. Zusätzlich wird die neue Population in die Liste der gespeicherten Populationen aufgenommen.

7. Iteration: Eine Iteration entspricht einer Wiederholung der Schritte $2-6$ mit einer einzelnen Population, da wir nur mittels einer Simulation die Fitness einer Population bestimmen können.

Entscheidend für die Effizient eines GA ist das Verhältnis von Exploration und Exploitation. [Črepinšek et al. 2013] Exploitation beschreibt die Suche von Lösungen in dem bekannten Suchbereich. Die Mutation einer Lösung erzeugt eine Nachbarlösung in einem bekannten Bereich. Exploration beschreibt die Suche nach neuen Lösungen in einem neuen bisher nicht betrachteten Bereich. Die Rekombination von bekannten Lösungen erzeugt eine neue Lösung.

Mutation und Rekombination müssen demnach im Verhältnis stehen, sodass die Suche ausreichend neue Bereiche erforscht und gleichzeitig vorhandene Lösungen weiter optimiert werden. Ist die Wahrscheinlichkeit der Rekombination zu gering besteht die Gefahr, das kein globales Optimum gefunden wird. In diesem Fall wird möglicherweise nur ein lokales Optimum gefunden. Ist Wharscheinlichkeit für Mutation zu gering wird ein lokales Optimum möglicherweise nicht gefunden, sodass eine nicht gefundene Nachbarlösung noch effizienter wäre.

Die Wharscheinlichkeit von Mutation und Rekombination wird zu beginn festgelegt und während den Simulationen nicht angepasst. Es erfolgt keine Überprüfung der Diversität von den gespeicherten Lösungen. Dadurch entsteht die Möglichkeit, dass die für die Rekombination vorhandenen Populationen nicht ausreichend sind, um ein globales Optimum zu finden. Die Effektivität dieses Suchalgorithmus ist dadurch eingeschränkt.

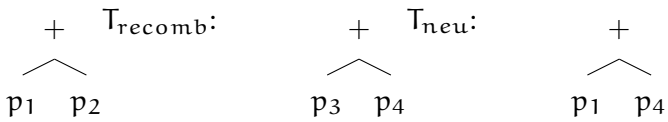
4.2.1 *Rekombination*

Für die Rekombination werden die n besten Populationen verwendet. Für die Implementierung wird $n := 20$ definiert. Es werden zwei verschiedene Varianten der Rekombination näher betrachtet:

1. Die Rekombination von einer Kombination T_{inital} mit einer anderen Kombination T_{recomb} von einer anderen Population. Dadurch ist es möglich, dass in der neu erstellten Kombination T_{neu} Parameter mehrmals vorkommen. Sei $T_{initial}$ eine Kombination von Parametern $P_1, \ldots, P_n$ mit $n \in \mathbb{N}$ und T_{recomb} eine Kombination von $P_1, \ldots, P_m$ mit $m \in \mathbb{N}$. Sei zusätzlich $T'_{initial} \in T_{initial}$ und $T'_{recomb} \in T_{recomb}$ jeweils eine beliebige Auswahl von Parametern. Dann ist:

   $$T_{neu} = (T_{initial} \backslash T'_{initial}) \cup (T_{recomb} \backslash T'_{recomb})$$

   Die Rekombination kann Beispielsweise dargestellt werden mit: Seien $p_1, p_2, p_3, p_4 \in P$ mit $p_1, p_2 \in T_{initial}$ und $p_3, p_4 \in T_{recomb}$, sowie $p_1, p_4 \in T_{neu}$. Die entsprechende Darstellung der Kombinationen:



An dieser Stelle repräsentiert T_{neu} eine neue Kombination von $T_{initial}$ mit dem rechten Teil von T_{recomb}.

2. Die Rekombination von der Ausgangspopulation mit einer anderen Population. Sei $S_{initial} = T_1, \ldots, T_y$ die Ausgangspopulation und $S_{recomb} = T_1, \ldots, T_x$ die zur Rekombination ausgewählte Population. Es wird ein zufälliger Wert $1 \leqslant cut \leqslant min(y, x)$ gewählt

   Es wird eine Stelle $1 \leqslant cut \leqslant min(y, x)$ gewählt und die Populationen $S_{initial}$ und S_{recomb} an dieser

   Dabei wird die Ausgangspopulation $S_{initial} = T_1, \ldots, T_y$ an einer Stelle $1 \leqslant cut \leqslant y$ abgeschnitten und durch einen Teil der anderen Population S_{recomb} ersetzt, sodass

$$S_{neu} = T_{initial_1}, \ldots, T_{initial_{cut}}, T_{recomb_{cut+1}} \ldots, T_{recomb_y}$$

Die Auswahl einer Population für die Rekombination ist nicht abhängig von der Fitness dieser Population. Eine Population wird zufällig aus den gespeicherten Populationen ausgewählt.

4.2.2 *Mutation*

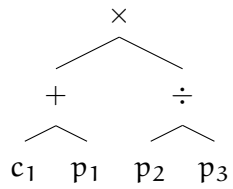
Die Mutation einer vorhandenen Lösung erzeugt eine neue Nachbarlösung von dieser. Diese neue Lösung kann eine verbesserte Lösung mit geringerer TWT sein. Goldberg [2002] beschreibt diese Verbesserung als Form von "Hillclimbing"-Algorithmus. Mit diesem wird eine vorhandene Lösung lokal Optimiert, indem Nachbarlösungen von einer existierenden Lösung generiert werden. Sobald eine bessere Nachbarlösung gefunden wurde wird ausgehend von dieser nach weiteren Lösungen gesucht.

Für die Mutation sind einige Modifikationen von T notwendig: Die Kombination von Parametern T wird erweitert, sodass ein Parameter $p \in P$ mit Operatoren kombiniert werden können. Ein Operator Op wird definiert als:

$$Op \in \{Addition, Subtraktion, Division, Multiplikation, Modulo\}$$

Zusätzlich wird T erweitert um Konstanten $c_1, \ldots, c_n \in \mathbb{R}$ mit $n \in \mathbb{N}$. Es sind mehr verschiedene Kombinationen T möglich, sodass zusätzliche Werte erzeugt werden können. Ein Parameter kann nicht nur mit Parametern kombiniert werden, sondern auch mit Konstanten.

$T_{initial}$ kann als Baum mit den Parametern $p_1, p_2, p_3 \in P$ und der Konstanten c_1 dargestellt werden als:



Der Wert von $T_{initial}$ berechnet sich in diesem Fall:

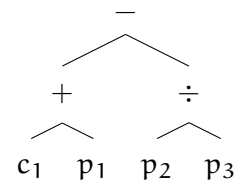
$$Wert(T) = (c_1 + p_1) \times (\frac{p_2}{p_3})$$

Die Mutation von $T_{initial}$ kann in zwei Variationen auftreten:

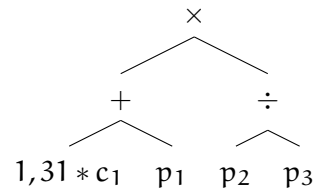
1. Die Mutation eines Operators: Wir verändern einen in T vorhandenen Operator Op_{alt}, sodass

   $$Op_{neu} \in \{+, -, \div, \times, mod\} \setminus Op_{alt}$$

   Die Wahrscheinlichkeit für einen spezifischen Operator ist 25%, da $Op_{neu} \neq Op_{alt}$ gilt. Als Beispiel kann eine Mutation der Kombination $T_{initial}$ den Multiplikations-Operator zu einem Subtraktions-Operator verändern:

2. Die Mutation einer Konstanten: Wir verändern einen Konstanten Wert $c_{alt} \in \mathbb{R}$ mit einem zufälligen Wert $\lambda \in (-2, 2]$. Die neue Konstante c_{neu} wird berechnet mit $c_{neu} = \lambda \times c_{alt}$. Sei $\lambda = 1,35$ zufällig gewählt. Die Mutation von c_1 in dem Baum $T_{initial}$ ergibt dann:

5

IMPLEMENTIERUNG

Die Konkrete Implementierung der in 4.1 und 4.2 beschriebenen Verfahren erfolgt mit einigen Anpassung. Diese Anpassungen sind aufgrund der speziellen Funktionsweise vom ASSS notwendig. Die Implementierung der Parameteroptimierung erfolgt mit "DecisionTreeAgent". Der Genetischer Algorithmus ist mit "MutationTreeAgent" implementiert. Der vollständigen Quellcode ist im Repository unter [Brenner 2024] einsehbar. Im Folgenden werden einige Implementierungen und Besonderheiten vorgestellt.

5.1 BASIS KRITERIEN

In beiden Optimierungsverfahren werden folgende Kriterien zur Entscheidungsfindung verwendet: Pfad Kosten, Stations Kapazität, Maximale Verteilung, Frequenz, Time Kanten. $value$ beschreibt den Ausgabewert der Funktionen basierend auf dem auszuwertenden Agenten, allen anderen Agenten und einer Station.

5.1.1 *Pfad Kosten*

Zur Berechnung der Pfad Kosten zwischen zwei Stationen wird ein einfacher Dijkstra Algorithmus verwendet.[Dijkstra 1959] Für alle "place" Kanten im Szenario muss gelten:

$$\forall e \in E_{place} : e_g \geqslant 0$$

Falls kein Pfad zwischen zwei T_S existiert, wird als Pfadkosten -1 zurück gegeben.

Listing 4: Implementierung des Dijsktra Algorithmus zum Finden der geringsten Pfadkosten zu einer Station.

```
private static int pathCost(StationType start, StationType target,
    Predicate<PlaceEdge> predicate) {
    PriorityQueue<Pair> queue = new PriorityQueue<>();
    List<StationType> used = new ArrayList<>();
    queue.add(new Pair(start, 0));
    while (!queue.isEmpty()) {
        Pair current = queue.poll();
        if (used.contains(current.station())) continue;
        used.add(current.station());
```

```
        if (current.station() == target) {
            return current.cost();
        }

        for (PlaceEdge edge : current.station().placeEdges) {
            if (predicate.test(edge)) continue;
            queue.add(new Pair((StationType) edge.connectedType, current.
                cost() + edge.weight));
        }
    }
    return -1;
}
```

Die Funktion "pathCost" (Code 4) bekommt neben dem Start und Ziel noch ein "Predicate" übergeben. Dieses kann genutzt werden um spezielle "place" Kanten auszuschließen. In der Prioritätswarteschlange werden die Stationstypen und die Kosten zum Erreichen verwaltet. Dafür wird die Klasse "Pair" verwendet:

Listing 5: "Pair" Klasse bestehend aus einem Stationstypen und den Kosten zu diesem Typen als Integer.

```
record Pair(StationType station, Integer cost) implements Comparable<Pair>
    {
    @Override
    public int compareTo(Pair other) {
        return Integer.compare(this.cost, other.cost);
    }
};
```

Ein Paar ist definiert als Tupel:

$$pair := (S_T, c) \text{ mit } c \in \mathbb{N}_0$$

c beschreibt die Kosten vom Start bis zum Stationtyp S_T. Die Klasse "Pair" verwendet das "Comparable" Interface, sodass die Paare in der Prioritätswarteschlange aufsteigend nach c angeordnet werden. Durch die Verwendung der Prioritätswarteschlange wird zuerst die Lösung mit den geringsten Kosten gefunden, falls ein Pfad vorhanden ist. Die Pfadkosten fallen negativ ins Gewicht. Für $value$ gilt:

$$value = -1 \times pathCost$$

5.1.2 *Stations Kapazität*

Die Bewertung der Kapazität von einer Station erfolgt nur über die Anzahl von Agenten, die in eine Station passen, unabhängig davon wie viele Agenten eine Station bereits als Ziel ausgewählt haben.

$$value = \frac{T_{S_{space}}}{A_{size}}$$

Die Implementierung erfolgt über die Verwendung des ATD-Baum 5.3.1. $T_{S_{space}}$ und A_{size} werden als Kindknoten zu einem Operator Knoten mit Divisions-Operator hinzugefügt. Die Auswertung des Baums repräsentiert dann die Berechnung.

5.1.3 *Maximale Verteilung*

Die maximale Verteilung soll eine möglichst hohe Verteilung von Agenten unter Berücksichtigung der Stationskapazitäten erzeugen. Sei $ziel_S \in \mathbb{N}_0$ die Anzahl an Agenten, die die Station $s \in S$ als Ziel haben. Der Wert einer Station wird bestimmt mit:

$$value = \begin{cases} T_{S_{space} - ziel_S} & \text{falls } T_{S_{space}} \neq -1 \\ \frac{2}{ziel_S + 1} & \text{sonst} \end{cases} \tag{3}$$

Falls die Station in der Kapazität begrenzt ist wird die Station mit der größten Kapazität bevorzugt. Die Größe der Agenten die s als Ziel haben wird nicht berücksichtigt. Wenn die Station in der Kapazität nicht begrenzt ist wird nur die Anzahl der Agenten mit s als Ziel berücksichtigt.

In der Implementierung der "maxDistribution" Funktion 6 wird überprüft, ob die übergebene Station über ein "space" Attribut verfügt. Gemäß Formel 3 wird der Wert der Station berechnet.

Listing 6: Implementierung der "maxDistribution" Funktion.

```
private static double maxDistribution(Agent me, HashMap<Agent, Object>
    others, Station station) {
    if (stationSpace(me, others, station) != Integer.MAX_VALUE) {
        return -1.0 * stationTargeted(me, others, station) + 1.0 *
            stationSpace(me, others, station);
    }
    return 2 / (stationTargeted(me,others,station) + 1);
}
```

Die Berechnung von $ziel_S$ erfolgt durch Code 7. Die übergebenen "comunication"-Objekte werden verwendet um die aktuellen Ziele aller anderen Agenten zu

extrahieren. Diese Ziele werden dann mit der übergebenen Station verglichen und die Anzahl an Übereinstimmungen gezählt.

Listing 7: Die Hilfsfunktion "stationTargeted" ermittelt die Anzahl an Agenten, die die übergebene Station als Ziel ausgewählt haben.

```
private static int stationTargeted(Agent me, HashMap<Agent, Object> others,
     Station station) {
    int counter = 0;
    for (Object object : others.values()) {
        if (object == null) continue;
        Object[] communication = (Object[]) object;
        if (((Station) communication[0]) == station) {
            counter += 1;
        }
    }
    return counter;
}
```

Die Funktion "stationSpace" (Code 8) ermittelt $T_{S_{space}}$. Falls $T_{S_{space}} = -1$ gilt, wird von der Funktion der maximal mögliche Integer Wert zurück gegeben.

Listing 8: Die "stationSpace" Funktion ermittelt die Kapazität einer Station. Wenn die Station keine Begrenzung hat wird eine Größe von Integer.MAX_VALUE angenommen.

```
private static int stationSpace(StationType station) {
    if (station.space == -1) return Integer.MAX_VALUE;
    return station.space;
}
```

Diese Basis Funktion wird verwendet, wenn keine der anderen Basis Funktionen verwendet wurde oder das Szenario über kein "frequency" Attribut verfügt.

5.1.4 *Frequenz*

Bei der Berechnung von dem Wert einer Station mit dem "frequency" Attribut wird unterschieden zwischen $S_{frequency}$ und $A_{frequency}$. Der Unterschied wird in 3.1 erläutert.

- Station mit "frequency": Verfügt die Station S über kein "frequency" Attribut wird ein Wert von 0 zurück gegeben. Falls $S_{T_{space}} \neq -1$ wird der Wert analog zur Berechnung der maximalen Verteilung 5.1.3 um $-1 \times ziel_S + T_{S_{space}}$ erhöht. Zusätzlich wird der aktuell vorhandene Platz in der Station relativ zur Agenten Größe mit ein berechnet: $\frac{S_{space}}{A_{size}}$. Wenn die vom Agenten vorherig besuchte Station vom gleichen Typ T_S ist wird der Wert um 1 angehoben. Anschließend wird die Zeit, die für den Besuch

an der Station benötigt wird von dem Wert abgezogen. Dadurch werden Stationen mit geringerer Besuchsdauer von Agenten bevorzugt.

Listing 9: Implementierung der Frequency Berechnung für Stationen

```
private static double stationFrequency(Agent me, HashMap<Agent,
    Object> others, Station station) {
    if (station.frequency == -1) return 0.0;
    double result = 0.0;

    if (stationSpace(me, others, station) != Integer.MAX_VALUE) {
        result = -1.0 * stationTargeted(me, others, station) + 1.0 *
            stationSpace(me, others, station);
    }
    if (station.space != -1) result += station.space / (double)
        agentSize(me, others, station);
    if (me.previousTarget.type == station.type) result += 1.0;
    result -= timeAtStation(me, station.type) * 1.0;
    return result;
}
```

Die Implementierung 9 verwendet die "stationTargeted"-Funktion (Code 7). Die benötigte Zeit $t \in \mathbb{N}$ für einen Agenten $a \in A$ an $s \in S$ wird berechnet mit:

$$t_{AS} = \begin{cases} A_{time} & \text{falls } S_{time} = -1 \\ S_{time} & \text{falls } A_{time} = -1 \\ \min(A_{time}, S_{time}) & \text{sonst} \end{cases} \tag{4}$$

Listing 10: Die TimeAtStation-Fuktion berechnet die Besuchszeit von einem Agenten an einer Station.

```
private static int timeAtStation(Agent me, StationType stationType) {
    if (me.type.time == -1 && stationType.time == -1) return 1;
    if (me.type.time == -1) return stationType.time;
    if (stationType.time == -1) return me.type.time;
    return Math.min(me.type.time, stationType.time);
}
```

Der Fall $A_{time} = -1$ und $S_{time} = -1$ sollte niemals eintreten. Dennoch wurde dies in der Implementierung 10 mit abgedeckt.

- Agent mit "frequency": Der Wert wird durch 4 verschiedene Komponenten beeinflusst. Das Ziel ist für den Agenten eine Station auszuwählen, deren Kapazität groß genug ist alle Agenten von einem Typen gleichzeitig aufzunehmen. Die Funktion "computeAgentFrequency" berechnet den Wert der Station mit diesen 4 Komponenten. Die Funktion gibt den Wert 0 zurück, Wenn der Agent über kein "frequency" Attribut verfügt.

1. Erhöhe den Wert um 1 falls:

$$A_{size} \times |A_T| \leqslant T_{S_{space}}$$

Listing 11: Erster Teil der "computeAgentFrequency"-Funktion.

```
double result = 0.0;
if (agentSize(me, others, station) * me.type.components.size() <=
     stationSpace(me, others, station)) {
    result += 1.0;
}
```

2. Verringere den Wert für jeden anderen Stationstypen den A besuchen könnte und $A_{size} \times |A_T| \leqslant T_{S_{space}}$ gilt um 0.5

Listing 12: Zweiter Teil der "computeAgentFrequency"-Funktion.

```
List<StationType> used = new ArrayList<>();
for (VisitEdge edge : me.type.visitEdges) {
    StationType stationType = (StationType) edge.connectedType;
    if (stationType == station.type) continue;
    if (used.contains(stationType)) continue;
    used.add(stationType);
    if (agentSize(me, others, station) * me.type.components.size
        () <= stationSpace(stationType)) {
        result -= 0.5;
    }
}
```

In Code 12 wird jeder Stationstyp nur einmal berücksichtigt.

3. Erhöhe den Wert um 0.5 für jeden Agententypen der s besuchen könnte und für den gilt: $A_{size} \times |A_T| > T_{S_{space}}$. In Code 13 wird überprüft, ob eine visit Kante zwischen einem Agenten und dem Stationstypen besteht, wenn dies nicht der Fall ist wird der Agent übersprungen. Jeder Agententyp wird nur einmal verwendet.

Listing 13: Dritter Teil der "computeAgentFrequency"-Funktion.

```
List<AgentType> usedAgent = new ArrayList<>();
usedAgent.add(me.type);
for (Agent agent : others.keySet()) {
    if (usedAgent.contains(agent.type)) continue;
    usedAgent.add(agent.type);

    boolean check = false;
    for (VisitEdge edge: agent.type.visitEdges) {
        StationType stationType = (StationType) edge.
            connectedType;
        if (stationType == station.type) {
```

```
            check = true;
            break;
        }
    }
    if (!check) continue;

    if (agentSize(agent, others, station) * agent.type.components
        .size() > stationSpace(me, others, station)) {
        result += 0.5;
    }
}
```

4. Der Wert von s wird um 0.5 erhöht, falls $S_{space} \geqslant A_{size}$.

Listing 14: Vierter Teil der "computeAgentFrequency"-Funktion.

```
if (station.space >= agentSize(me, others, station)) {
    result += 0.5;
}
```

5.1.5 *Time Kanten*

In der Berechnung des Wertes einer Station $s \in S$ für einen Agenten $a \in A$ mit dem Kriterium der "time" Kanten wird unterschieden zwischen gerichteten und ungerichteten Kanten. Zusätzlich wird zwischen den in 3.1.3 beschriebenen verbundenen Komponenten unterschieden.

Die Richtung von Verbundenen "time" Kanten wird festgestellt, indem die Richtung der Kante an den verbundenen Komponenten überprüft wird. Für eine Kante $e \in E_{time}$ gilt:

$$e \text{ ist } \begin{cases} \text{ungerichtet} & \text{falls: } \neg\text{eingehend} \wedge \neg\text{ausgehend} \\ \text{eingehend} & \text{falls: } \text{eingehend} \wedge \neg\text{ausgehend} \\ \text{ausgehend} & \text{falls: } \neg\text{eingehend} \wedge \text{ausgehend} \end{cases}$$

In der Implementierung werden "predicates" verwendet, um Kanten heraus zu filtern, die nicht den gesuchten Kanten entsprechen. Aus diesem Grund müssen die Aussagen noch negiert werden:

$$e \text{ ist } \textbf{nicht} \begin{cases} \text{ungerichtet} & \text{falls: } \text{eingehend} \vee \text{ausgehend} \\ \text{eingehend} & \text{falls: } \neg\text{eingehend} \vee \text{ausgehend} \\ \text{ausgehend} & \text{falls: } \text{eingehend} \vee \neg\text{ausgehend} \end{cases}$$

Listing 15: Predicates die beim Iterieren der "time" Kanten von einer Station überprüft werden, sodass gefiltert werden kann.

```
private static Predicate<TimeEdge> undirectedPredicate = edge -> (edge.
    incoming || edge.outgoing);
private static Predicate<TimeEdge> outgoingDirectedPredicate = edge -> (!
    edge.outgoing || edge.incoming);
private static Predicate<TimeEdge> incomingDirectedPredicate = edge -> (
    edge.outgoing || !edge.incoming);
```

Mithilfe der gefilterten Kanten wird dann die Berechnung durchgeführt: Für alle Agenten A_{all}, die durch eine "time" Kante mit S_T verbunden sind wird der Wert von S erhöht um:

$$\sum_{a \in A_{all}} ArbeitsZeit_a$$

$ArbeitsZeit_a$ beschreibt die verbleibende Arbeitszeit von einem Agenten a.

Listing 16: Berechnung des Stationswertes mit einer übergegbenen Liste von Agenten, die mit eine "time" kante verbunden sind.

```
private static double computeTimeConnectedAgents(Agent me, HashMap<Agent,
    Object> others, Station station, List<Agent> connectedAgents) {
    double result = 0.0;
    for (Agent agent : connectedAgents) {
        result += estimatedWorkTimeLeft(agent, others, station);
    }
    return result;
}
```

Für alle Stationen S_{all}, die durch eine "time" Kante $e \in E_{time}$ mit S_T verbunden sind wird der Wert von S erhöht um:

$$\sum_{s \in S_{all}} (t_{as} + e_g + 3 \times ziel_s)$$

Die Berechnung von t_{as} wird mit Gleichung 4 erläutert. Für ungerichtete Kanten $e \in E_{time}$ gilt $e_g = 0$. Die Gewichtung von $ziel_s$ ist erhöht, sodass die Priorität der Station s erhöht ist, wenn eine Verbundene Komponente als Ziel ausgewählt ist.

Listing 17: Berechnung des Stationswertes mit einer übergebenen Liste von verbundenen Stationen.

```
private static double computeTimeConnectedStations(Agent me, HashMap<Agent,
     Object> others, List<ResultPair> connectedStations) {
    double result = 0.0;
    for (ResultPair pair : connectedStations) {
        result += timeAtStation(me, others, pair.station);
        result += pair.cost;
        result += stationTargeted(me, others, pair.station);
```

```
    }
    return result;
}
```

Verfügt ein Szenario über keine ungerichteten oder gerichteten "time" Kanten werden die entsprechenden Parameter deaktiviert.

5.2 DEADLOCKS

Bei der Entscheidungsfindung welche Station als nächstes ausgewählt wird, muss berücksichtigt werden, dass es Szenarien gibt in denen Stationen nicht ausgewählt werden dürfen. Dies ist der Fall wenn der Besuch einer Station die Bedingungen des Szenarios verletzt. Sobald eine Bedingung Verletzt ist oder die Simulation nicht mehr abgeschlossen werden kann wird die Simulation abgebrochen und es kann kein Zeitplan erstellt werden. Aus diesem Grund werden zwei verschiedene Failsafes verwendet, um zu verhindern das ein Deadlock auftritt. Im Fall, dass eine Bedingung für einen Failsafe erfüllt ist, wird sofort -100 als Wert für diese Station zurück gegeben, ohne noch weitere Berechnungen durchzuführen. Es ist dennoch möglich, dass eine Simulation nicht zum Erfolgt führt.

5.2.1 *Räumliche Abhängigkeit*

Seien $s_1, s_2 \in S$ die mit einer gerichteten Kante e_{space} von s_1 nach s_2 verbunden sind. Dann darf ein Besuch von s_2 erst nach s_1 erfolgen. Andernfalls ist die räumliche Abhängigkeit verletzt und die Simulation kann nicht abgeschlossen werden. Durch gerichtete place Kanten wird eine Reihenfolge vorgegeben in der ein Agent Stationen besuchen muss.

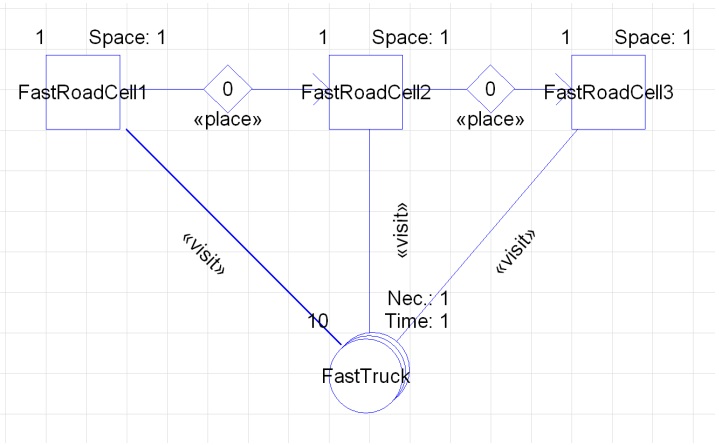


Abbildung 13: Auszug aus dem "Crossroad" Szenario für einen Graphen mit gerichteten "place" Kanten. Die Agenten müssen die Stationen in der Reihenfolge FastRoadCell1, FastRoadCell2, FastRoadCell3 jeweils 1-mal besuchen.

Exemplarisch wird dies in Abbildung 13 gezeigt. Die Agenten müssen jede Station genau einmal besuchen und zum Zeitpunkt 0 starten diese an der Station "FastRoadCell1". Die räumliche Abhängigkeit des Szenarios wird verletzt, wenn "FastRoadCell2" besucht wird bevor "FastRoadCell1" besucht worden ist. Unabhängig von der Implementierung der Entscheidungsfindung darf eine solche Station niemals gewählt werden.

Zunächst wird eine Pfadsuche von der aktuellen Station zur Zielstation durchgeführt, um herauszufinden, ob s_2 in Abhängigkeit zu s_1 steht. Dabei werden eingehende "place" Kanten ignoriert, sodass kein Pfad von s_2 nach s_1 gefunden wird. In diesem Fall dürfen wir s_2 nicht wählen. Es wird jedoch ein Pfad von s_1 nach s_2 gefunden. Aus diesem Grund kann s_1 potentiell ausgewählt werden. Dies wird für jede weitere Station die Besucht werden muss wiederholt. Zusätzlich werden dadurch nicht erreichbare Stationen nicht ausgewählt. Eine Station ist nicht erreichbar, wenn kein Pfad zu dieser von der aktuellen Position aus führt.

Listing 18: Die "otherStationReachable" Funktion überprüft, ob die übergebene Station nicht erreichbar ist oder eine räumliche Abhängigkeit existiert.

```
private static boolean otherStationsReachable(Agent me, Station station) {
        if (pathCost(me.previousTarget.type, station.type, edge -> (false))
             == -1) return false;
        for (Map.Entry<Station, Integer> entry : me.necessities.entrySet())
             {
                if (entry.getValue() <= 0) continue;
                if (pathCost(station.type, entry.getKey().type, edge -> (
                    edge.incoming)) == -1) return false;
        }

        return true;
```

```
}
```

In der Implementierung 18 wird nur über Stationen iteriert, die besucht werden müssen. Die Implementierung kann verbessert werden, indem alle eingehenden Kanten von Stationen betrachtet werden unabhängig von deren Attributen. Dafür ist es notwendig alle Agenten und deren Besuchte Stationen zu speichern. Für die Berechnung werden die Pfadkosten mit "pathCost" 4 ermittelt. Die "pathCost" Funktion gibt -1 zurück, falls kein Pfad zu einer Station gefunden wird. Dafür wird ein "Predicate" verwendet, um bei der Berechnung eingehende place Kanten nicht zu berücksichtigen.

5.2.2 *Platz in einer Station*

Eine Station $s \in S$ darf ebenfalls von einem Agenten $a \in A$ nicht gewählt werden, wenn $S_{T_{space}} < A_{size}$ gilt. In der Abbildung 14 sind die Agenten "Product" jeweils 2 Einheiten groß. Die Station "Entrance" hat eine initiale Kapazität von 1. Wenn "Entrance" von einem Agenten ausgewählt wird kann diese Station nicht besucht werden. Für einen Besuch einer Station von einem Agenten muss $S_{T_{space}} \geqslant A_{size}$ gelten.

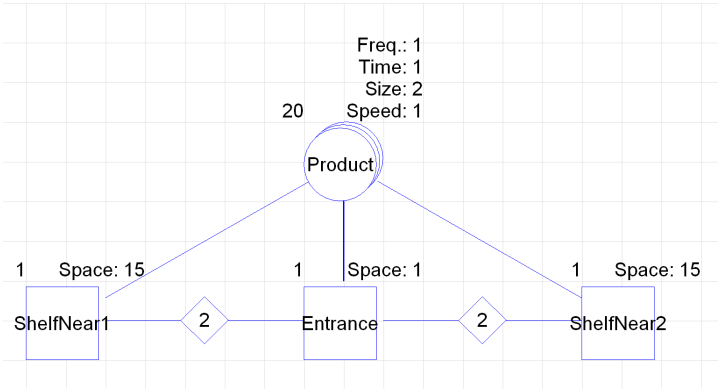


Abbildung 14: Auszug aus dem Szenario "NewStorage2" für ein Beispielszenario mit zu kleinen Stationen. Die Agenten müssen eine beliebige Station besuchen und starten bei der Station "Entrance". Diese hat eine Kapazität von 1. Die Agenten haben eine Größe von 2.

5.3 HILFSSYSTEME

Für die Implementierungen der beiden Verfahren werden verschiedenste Hilfssysteme verwendet. Diese stellen Informationen und Funktionalitäten bereit, sodass erfolgte Berechnungen nicht erneut durchgeführt werden müssen. An dieser Stelle wird die Implementierung der Kombination T als Binärer Baum,

die Verwaltung von Informationen über vergangene und aktuelle Simulationen, sowie die Berechnung der TWT von einer Simulation vorgestellt.

5.3.1 *ADT-Baum*

Ein binärer ADT-Baum wird zur Kombination und Auswertung verschiedener Basis Parametern verwendet. Ein ADT-Baum beschreibt einen Baum mit Knoten die Abstrakter Datentyp (ADT) enthalten. Die Implementierung des Baumes für die Parameteroptimierung 4.1 unterscheidet sich von der Implementierung des Baumes für den GA 4.2. Beide Implementierungen stellen für das jeweilige Verfahren die nötigen Funktionen zur Verfügung. An dieser Stelle wird der Baum für den GA vorgestellt.

Der ADT-Baum ermöglicht einfaches hinzufügen von neuen Knoten zum aktuellen Baum. Die Bäume werden verwendet um Knoten zu Mutieren und den Baum mit anderen zu Rekombinieren. Ein Baum besteht immer aus einem Wurzelknoten. Ein Knoten ist definiert als:

$$\mathtt{Knoten} := \mathtt{Knoten}_i \cup \mathtt{Knoten}_v \cup \mathtt{Knoten}_c$$

Die inneren Knoten $\mathtt{Knoten}_i$ des Baums bestehen aus dem Tupel $\mathtt{Knoten}_i = (L, Op, R)$ mit $L, R \in \mathtt{Knoten}$ und dem Operator $Op \in \{+, -, \div, \times, \mathrm{mod}\}$. Die $\mathtt{Knoten}_v$ werden "value" Knoten genannt und enthalten einen konstanten Wert $value \in \mathbb{R}$.

Die "consumer" Knoten $\mathtt{Knoten}_c$ sind ebenfalls Blätter und bestehen aus einer Funktionsreferenz zu einer der in 5.1 beschriebenen Basisfunktionen. Der Wert dieses Knoten ist abhängig von dem Ergebnis der Funktion. $value(consumer) \in \mathbb{R}$ ist definiert als der ermittelte Wert der referenzierten Basis Funktion.

Insgesamt kann der Wert des $\mathtt{Knoten}\,K$ beschrieben werden mit:

$$Wert_K = \begin{cases} L\ Op\ R & \text{Falls } K \in \mathtt{Knoten}_i \\ value & \text{Falls } K \in \mathtt{Knoten}_v \\ value(consumer) & \text{Falls } K \in \mathtt{Knoten}_c \end{cases} \tag{5}$$

In dem Fall, dass $Op = \div$ oder $Op = \mathrm{mod}$ und $R = 0$ ist, erfolgt die Auswertung dieses Knotens mit $R = 1$. Dies verhindert die Randfälle: $\frac{Wert_L}{0}$ und $Wert_L \bmod 0$. Der Wert eines Knoten wird mit der "evaluate" Funktion ermittelt.

Listing 19: Definition der "evaluate" Funktion zur Bestimmung des Wertes eines Knoten

```
public interface Node {
        double evaluate(Agent me, HashMap<Agent, Object> others, Station
            station);
```

```
}
```

Die Funktionsparameter der "evaluate" Funktion (Code 19) müssen mindestens die Informationen der "consumer" Schnittstelle (24) bereitstellen. Die Konkrete Implementierung von 19 ist für die verschiedenen Knoten unterschiedlich.

- Knoten$_i$: Die inneren Knoten des Baums verfügen über eine Referenz zum linken und rechten Kindknoten, sowie über einen Operator Op. Weder Op noch L oder R dürfen dabei leer sein.

Listing 20: Teil der Implementierung eines inneren Knotens: "OperatorNode"

```
public class OperatorNode implements Node {
        private Operator op;
        private Node left;
        private Node right;

        public OperatorNode(Operator op, Node left, Node right) {
                this.op = op;
                this.left = left;
                this.right = right;
        }

        @Override
        public double evaluate(Agent me, HashMap<Agent, Object>
            others, Station station) {
                return op.evaluate(left.evaluate(me, others, station)
                    , right.evaluate(me, others, station));
        }
}
```

Die überschriebene "evaluate" Funktion in 20 verwendet die für die Operator definierte "evaluate" Funktion 21. Die "evaluate" Funktion in der Klasse "Operator" nimmt zwei Werte entgegen und wendet den entsprechenden Operator auf diese beiden Werte an. Die Evaluierung des Knoten$_i$ erfolgt mit der Auswertung des linken und rechten Teilbaums.

Listing 21: Implementierung der "evaluate" Funktion in der enum Klasse "Operator".

```
public double evaluate(double first, double second) {
    if (this == ADDITION) return first + second;
    if (this == SUBTRACTION) return first - second;
    if (this == MULTIPLICATION) return first * second;
    if (this == DIVISION) {
        if (second == 0.0) return first / 1.0;
        return first / second;
    }
    if (this == MODULO) {
        if (second == 0.0) return first % 1;
        return first % second;
```

```
    }
    return 0;
}
```

- Knoten_v:

Listing 22: Teil-Implementierung des "value" Knoten. Die Implementierung der "evaluate" Funktion gibt den Konstanten Wert zurück.

```
public class ValueNode implements Node {
        private double value;

        public ValueNode(double value) {
                this.value = value;
        }

        @Override
        public double evaluate(Agent me, HashMap<Agent, Object>
            others, Station station) {
                return value;
        }
}
```

- Knoten_c: Die "consumer" Knoten enthalten eine Referenz auf das "Consumer" Interface und eine textuelle Beschreibung "title". Dieser "title" wird zur Identifizierung verwendet, sodass Bäumer anhand der Struktur und Werte vergleichbar sind. Es ist nicht möglich die gespeicherten Funktionsreferenzen eindeutig zu vergleichen.

Listing 23: Teil-Implementierung von dem "consumer" Knoten. Die Implementierung von "evaluate" verwendet die Funktionsreferenz, um den Wert der Basis-Funktion zu ermitteln.

```
public class ConsumerNode implements Node{
        private OwnConsumer consumer;
        private String title;

        public ConsumerNode(OwnConsumer consumer, String title) {
                this.consumer = consumer;
                this.title = title;
        }

        @Override
        public double evaluate(Agent me, HashMap<Agent, Object>
            others, Station station) {
                return consumer.compute(me, others, station);
        }
}
```

Listing 24: Definition des "Consumer" Interfaces zur weiterreichung der Funktionsreferenz.

```
interface OwnConsumer {
        double compute(Agent me, HashMap<Agent, Object> others,
            Station station);
}
```

Alle Basis Funktionen entsprechen der Signatur von (Code 24), sodass der "consumer" Knoten (Code 23) diese Referenz entgegennehmen kann und die "evaluate" Funktion (Code 19) die Basis Funktion auswertet. Aus diesem Grund verfügen die Basisfunktionen auch über diese Parameter, wenn diese nicht verwendet werden.

Zusätzlich verfügt der ADT-Baum über einige nicht genannte Hilfsfunktionen, die später verwendet werden um Änderungen am Baum vorzunehmen.

5.3.2 *Zeitmessung*

Die "TimeStatistics" Klasse (Code 25) wird zum Speichern verschiedener Daten verwendet, sodass Zeitstatistiken nur einmalig berechnet werden müssen. Die Klasse enthält nur Datenfelder, die in "AbstractSwarmAgentInterface" berechnet und gesetzt werden:

Listing 25: Die "TimeStatistics" Klasse, die für die weitere Berechnungen nötige Informationen über die Durchläufe bereit stellt.

```
public class TimeStatistics {
        public boolean newRun = true;
        public boolean newBestRun = false;
        public boolean lastRunCompleted = false;

        public int numberOfRuns = 0;
        public int numberOfCompletedRuns = 0;
        public int runsSinceCurrentBest = 0;

        public long time = 0L;

        public long roundTimeUnit = 0L;
        public long lowestTimeUnit = Long.MAX_VALUE;

        public long lastValue = 0L;

        public long lowestTwT = Long.MAX_VALUE;
        public long currentTwT = Long.MAX_VALUE;
}
```

Die in Code 25 vorhanden Attribute sind:

- **newRun**: ist auf wahr gesetzt für den ersten "evaluation" Aufruf in einer neuen Simulation

- **newBestRun** ist auf wahr gesetzt, falls die letzte Simulation die beste bisherige Simulationen gewesen ist
- **lastRunCompleted** ist auf wahr gesetzt, falls die letzte Simulation abgeschlossen worden ist
- **numberOfRuns** Anzahl an Simulationen
- **numberOfCompletedRuns** Anzahl von abgeschlossenen Simulationen
- **runsSinceCurrentBest** Anzahl von Simulationen seit der aktuell besten Simulation
- **time** der aktuelle Zeitschritt in der Simulation
- **roundTimeUnit** die höchste Zeiteinheit im letzten Durchlauf
- **lowestTimeUnit** die bisherig niedrigste Zeiteinheit von allen abgeschlossenen Durchläufen
- **lastValue** Zeitschritt von letzten Aufruf der "evaluation" Funktion
- **lowestTwT** die niedrigste TWT von allen Simulationen
- **currentTwT** die TWT von der letzten Simulation

5.3.3 *Berechnung der TWT*

Die Schnittstellen (Kapitel 3.2) des ASSS ermöglichen die Bestimmung folgender Ereignisse:

- Zeitpunkt an dem ein Agent eine Station als Ziel auswählt
- Zeitpunkt an dem ein Agent eine ausgewählte Station beginnt zu besuchen
- Zeitpunkt an dem ein Agent den Besuch abgeschlossen hat

Mithilfe dieser Ereignisse wird ein "PlanEntry" erstellt. Die "PlanEntry" Klasse 26 enthält unter anderem folgende Informationen: der spezifische Agent, die ausgewählte Station, der Zeitpunkt zu dem die Station ausgewählt wird, der Zeitpunkt an dem der Agent die Station erreicht und der Zeitpunkt an dem der Besuch abgeschlossen wurde.

Listing 26: Implementierung der "PlanEntry" Klasse, welche Informationen über den Besuch eines Agenten an einer Station speichert.

```
public record PlanEntry(Agent agent, Station station, Long predictionTime,
    Long predictedArrivalTime, Long arrivalTime, Long finishTime, Double
    stationValue) implements Comparable<PlanEntry>{
};
```

Dadurch wird der Zeitplan der vorangegangenen Simulation bestimmt. Wodurch die TWT zur Bewertung einer Solution berechnet wird: Sei $a \in A$ ein beliebiger Agent in der Simulation und S beschreibt eine Liste der Länge $i \in \mathbb{N}$ mit Stationen, die a besuchen muss. Dann gibt es für die Station S_i drei Zeitpunkte:

- $a_{S_i t_1}$: S_i wird ausgewählt.
- $a_{S_i t_2}$: Agent beginnt S_i zu besuchen.
- $a_{S_i t_3}$: Agent beendet den Besuch von S_i.

Für alle Agenten gilt: $A_{s_0 t_3} = 1$. Die TWT von einem Agenten A wird dann berechnet mit:

$$TWT_A = \sum_i (A_{S_i t_2} - A_{S_{i-1} t_3})$$

Die Berechnung der TWT erfolgt dann mit TWT_a für jeden Agenten der mindestens eine Station besucht hat. Der späteste Zeitpunkt an dem der Besuch einer Station beendet wurde unter allen Agenten wird beschrieben mit: $\max_A(A_{st_3})$. Der maximalen Zeitpunkt muss mit dem spätesten Zeitpunkt von jedem Agenten verrechnet werden. Seien a_1, a_2 zwei Agenten mit $\max(a_{1 S t_3}) > \max(a_{2 S t_3})$. Dann muss a_2 $\max(a_{1_{S t_3}}) - \max(a_{2_{S t_3}})$ Zeiteinheiten nach Beendigung aller Besuche Warten bevor die Simulation abgeschlossen ist. Agenten beschreibt alle Agenten, die mindestens eine Station besucht haben. Daraus folgt:

$$TWT = \sum_{A \in Agenten} \Big(\sum_{i \in |A_S|} (A_{S_i t_2} - A_{S_{i-1} t_3}) + \max_A(A_{S t_3}) - \max(A_{S t_3}) \Big) \quad (6)$$

Die "FinishedSolution" Klasse (Code 27) repräsentiert eine abgeschlossene Simulation. Die Klasse enthält eine sortierte Liste aus "PlanEntry" (Code 26) Einträgen. Die "calculateTWT" Methode wird verwendet, um die TWT der gespeicherten Lösung zu berechnen. Gespeichert wird immer der aktuell höchste Zeitschritt $A_{S_i t_3}$ für jeden Agenten. Dieser Eintrag wird mit dem nächsten Eintrag $A_{S_{i+1} t_2}$ verrechnet, falls $A_{S_{i+1}}$ existiert.

Listing 27: Auszug aus der "FinishedSolution" Klasse und der Methode zur Berechnung der TWT aus einer Liste von "PlanEntrys" gemäß Berechnung 6.

```
public class FinishedSolution {

        private List<PlanEntry> solution;

        public Long calculateTWT() {
                HashMap<Agent, Long> lastFinishPerAgent = new HashMap<>();
                Long result = 0L;
                Long maxTime = 0L;
                for (PlanEntry entry : solution) {
```

```
                        if (maxTime < entry.finishTime()) maxTime = entry.
                            finishTime();
                        if (lastFinishPerAgent.containsKey(entry.agent()))
                            {
                                result += entry.arrivalTime() - (
                                    lastFinishPerAgent.get(entry.agent()) +
                                     1);
                                lastFinishPerAgent.put(entry.agent(), entry
                                    .finishTime());
                        } else {
                                lastFinishPerAgent.put(entry.agent(), entry
                                    .finishTime());
                                result += entry.arrivalTime() - 1L;
                        }
                }

                for (Long value : lastFinishPerAgent.values()) {
                        result += maxTime - value;
                }

                return result;
        }
}
```

5.4 PARAMETEROPTIMIERUNG

Die Verwaltung der Wahrscheinlichkeiten X_p erfolgt durch die Klasse "ProbabilityStatistics". (Code 28) Diese Verfügt über eine Zuordnung von einer Repräsentation von X_p zu dem aktuellen Wert von X_{p_i}. Zusätzlich werden die vergangenen Werte $S_i(X_p)$ gespeichert. Der aktuelle Wert von X_p wird als "Pair" abgespeichert. Ein "Pair" (29) besteht aus einem zufälligen Wert $\lambda \in [0, 1)$ und dem gespeicherten Grenzwert.

Listing 28: Ausschnitt aus der "ProbabilityStatistic" Klasse mit den aktuellen und vergangenen Werten von X_p.

```
class ProbabilityStatistic {

        HashMap<String , Pair> map = new HashMap<>();

        HashMap<String, List<Double>> pastValues = new HashMap<>();

        public boolean compare(String name) {
                if (map.containsKey(name)) {
                        return map.get(name).compare();
                }
                return false;
        }
}
```

Listing 29: Ausschnitt der "Pair" Klasse mit den Methoden "checktreshold", "newRandom" und "compare".

```
private class Pair {
    private double randomValue;
    private double threshold;

    private void checkThreshold() {
        this.threshold = Math.max(this.threshold, 0.05);
    }

    public void newRandom() {
        this.randomValue = random.nextDouble();
    }

    public boolean compare() {
        return randomValue <= threshold;
    }
}
```

Die "checkTreshold" Methode stellt sicher, dass nachdem ein Grenzwert geändert wurde, dieser mindestens 0.05 beträgt. Die "newRandom" Methode wird verwendet, um einen neuen Zufallswert zwischen 0 und 1 zu erzeugen. Die "compare" Methode vergleicht den aktuell gespeicherten Zufallswert mit dem aktuellen Grenzwert.

Die "compare" Methode von ProbabilityStatistic (28) verwendet das gespeicherte "Pair" zum Vergleich. Sollte für einen Parameter kein "Pair" vorhanden sein wird diese Methode immer "falsch" zurückgeben. Ein solcher Parameter wird niemals verwendet.

Listing 30: Implementierung der "triggerCompare" Methode, die den Grenzwert um um $0.2 \times reward$ erhöht.

```
        public void triggerCompare(List<String> parameters, double reward)
            {
                for (String parameter : parameters) {
                        if (map.containsKey(parameter)) {
                                map.get(parameter).increaseThreshold(0.2 *
                                    reward);
                        }
                }
        }
```

Die "triggerCompare" Methode (30) nimmt eine Liste an Parametern und einen "reward" Wert entgegen. Der Grenzwert von dem Parameter p wird dann wie in 4.1 beschrieben erhöht.

5.4.1 *Erstellung des ADT-Baum*

Die Erstellung für die Auswertung verwendeten Baums erfolgt in jedem Aufruf der "evaluation" (Code 1) Funktion. In jedem Aufruf wird jeder Parameter mit dem dazugehörigen Grenzwert verglichen. (Code 45) Für einen Zeitschritt t ist das Ergebnis der "compare" Methode für einen Parameter p immer gleich, da der Zufallswert von X_p erst in einem neuen Zeitschritt einen neuen Wert zugewiesen bekommt.

Die Implementierung ist in Code 45 dargestellt. Die Initialisierung des ADT-Baums erfolgt mit einem Blattknoten mit Wert 0. Anschließend werden die Knoten hinzugefügt, für die der Vergleich der Parameter "wahr" ergibt. Sollte keine Station und kein Agent über ein "frequency" Attribut verfügen oder kein Vergleich von anderen Parametern Wahr ergeben wird die maximale Verteilung verwendet 5.1.3. Zurückgegeben wird dann die Auswertung des erzeugten Baums.

5.4.2 *Anpassung von Wahrscheinlichkeiten*

Am Beginn jeder Simulation werden die initialen Wahrscheinlichkeiten gemäß Kapitel 4.1 angepasst. Wenn die vorherige Simulation nicht vollständig Abgeschlossen wurde oder die Bedingung der TWT nicht erfüllt ist, werden die Werte auf den letzten gespeicherten Wert $S(X_p)$ gesetzt. Die initialen Werte $S(X_{p_1}$ werden verwendet, wenn keine gespeicherten Werte vorhanden sind.

Listing 31: Anpassung der Wahrscheinlichkeiten bei Start einer Simulation

```
if (timeStatistic.newRun) {
    probabilityStatistic.newRandom();
    if (timeStatistic.newBestRun || (timeStatistic.lastRunCompleted &&
        timeStatistic.currentTwT <= Math.round(timeStatistic.lowestTwT *
        1.6))) {
        probabilityStatistic.reset();
    } else {
        probabilityStatistic.recover();
    }

    decision.clear();
}
```

In der "reset" und "recover" Methode wird diese Funktionalität implementiert.

Listing 32: Die "reset"-Methode berechnet den nächsten Startwert mit dem aktuellen Wert und den vorangeganenen gespeicherten Werten.

```
public void reset() {
    addToPast();
```

```
    Pair highest = null;
    for (Pair pair : map.values()) {
        if (highest == null || highest.getThreshold() < pair.getThreshold()
            ) {
            highest = pair;
        }
    }
    for (Map.Entry<String, Pair> entry : map.entrySet()) {
        if (entry.getValue() != highest) {
            entry.getValue().setThreshold(computeAverage(entry.getKey()));
        } else {
            entry.getValue().setThreshold(0);
        }
    }
}
```

Die "reset"-Methode (Code 32) speichert die aktuellen Parameter Grenzwert ab. Der Grenzwert des Parameters mit dem aktuell höchsten Grenzwert wird auf 0 gesetzt. Aufgrund der Begrenzung der Wahrscheinlichkeit (Code 29) ist der Grenzwert von p anschließend 0.05. Dadurch wird dieser Parameter in der nächsten Simulation weniger Wahrscheinlich verwendet und andere Kombinationen werden mit höherer Wahrscheinlichkeit verwendet. Die "recover"-Methode (Code 33) verwendet den zuletzt Gespeicherten Wert $S(X_{p_i})$ als neuen Grenzwert. Für den Fall das keine vergangenen Werte gespeichert sind wird der initiale Grenzwert verwendet.

Listing 33: Die "recover"-Methode zur Wiederherstellung der vorherigen Werte.

```
public void recover() {
    for (Map.Entry<String, List<Double>> entry : pastValues.entrySet()) {
        if (map.containsKey(entry.getKey())) {
            if (entry.getValue().size() > 0) {
                map.get(entry.getKey()).setThreshold(entry.getValue().get(
                    entry.getValue().size() - 1));
            } else {
                map.get(entry.getKey()).setThreshold(0.35);
            }
        }
    }
}
```

Die Kombination T, die zur Auswahl verwendet wurde, wird für jeden Agenten einzeln in einer Map abgespeichert. Am beginn einer Simulation wird diese Map geleert, um zu verhindern das Einträge von der vorherigen Simulation vorhanden sind.

Listing 34: Speichern der Auswahl von Parametern für jeden Agenten in der "evalutate"-Funktion.

```
private static HashMap<Agent, List<String>> decision = new HashMap<>();
```

```
public static double evaluate(Agent me, HashMap<Agent, Object> others, List
    <Station> stations, long time, Station station, TimeStatistics
    timeStatistic) {
    if (timeStatistic.newRun) {

        decision.clear();
    }

    if (!decision.containsKey(me)) {
        decision.put(me, probabilityStatistic.getCurrentComparison());
    }

    return currentNode.evaluate(me, others, station);
}
```

Anschließend erhöhen wir den Grenzwert der verwendeten Parameter in der "reward"-Funktion mit der "triggerCompare" Methode (Code 30). Der Eintrag von T für den Agenten wird aus der Map entfernt. Durch die Verwaltung der Kombinationen T mit der Map ist es unerheblich wie groß der zeitliche Abstand zwischen Auswahl und Abschluss der Station ist. Zusätzlich werden die Wahrscheinlichkeiten nach der Erhöhung normalisiert und neue Zufallswerte werden erzeugt, wenn der Aufruf der "reward"-Funktion in einem neuen Zeitschritt erfolgt.

Listing 35: Anpassung der Wahrscheinlichkeiten innerhalb der "reward"-Funktion.

```
public static void reward(Agent me, HashMap<Agent, Object> others, List<
    Station> stations, long time, double value,TimeStatistics timeStatistic
    ) {
    if (decision.containsKey(me)) {
        probabilityStatistic.triggerCompare(decision.get(me), value);
        decision.remove(me);
    }

    probabilityStatistic.normalize();

    if (lastValue != time) {
        lastValue = time;

        probabilityStatistic.newRandom();
    }
}
```

5.5 GENETISCHER ALGORITHMUS

Für die Implementierung des Genetischen Algorithmus ist es notwendig entsprechende Funktionen für die in 4.2 beschriebene Rekombination und Muta-

tion für den ADT-Baum (5.3.1) bereitzustellen. Die Implementierung der 4 verschiedenen Varianten erfolgt durch die Funktionen: "crossover", "largeCrossover", "mutateOperator" und "valueMutation". Diese erzeugen und verarbeiten eine Kopie der übergebenen Bäume, sodass die übergebenen Bäume nicht verändert werden. Die initialen Bäume dürfen nicht verändert werden, da diese für die Rekombination gespeichert und genutzt werden. Die Implementierungen für die Mutation und Rekombination sind wie folgt:

- Rekombination von $T_{initial}$:

Listing 36: Rekombination von zwei gegebenen ADT-Bäumen

```
public static Tree crossover(Tree first, Tree second) {
    Tree firstCopy = first.copy();
    Tree secondCopy = second.copy();
    if (first.isEmpty() && second.isEmpty()) return firstCopy;
    if (first.isEmpty()) return secondCopy;
    if (second.isEmpty()) return firstCopy;

    List<OperatorNode> firstTreeNodes = firstCopy.getOperatorNodes();
    List<OperatorNode> secondTreeNodes = secondCopy.getOperatorNodes
        ();

    if (firstTreeNodes.size() < 1) return secondCopy;
    if (secondTreeNodes.size() < 1) return firstCopy;

    OperatorNode firstRandomNode = firstTreeNodes.get(random.nextInt(
        firstTreeNodes.size()));
    OperatorNode secondRamdomNode = secondTreeNodes.get(random.
        nextInt(secondTreeNodes.size()));

    if (random.nextDouble() >= 0.5) {
        firstRandomNode.setLeft(secondRamdomNode);
    } else {
        firstRandomNode.setRight(secondRamdomNode);
    }

    return firstCopy;
}
```

Zunächst erfolgt eine Überprüfung, ob die beiden kopierten Bäume leer sind. Wenn mindestens einer der Beiden Bäume leer ist kann keine Rekombination durchgeführt werden. Anschließend wird in den beiden Bäumen jeweils ein zufälliger Knoten ausgewählt. Es gibt für den Eltern Knoten vom ersten Baum genau 2 Möglichkeiten:

1. Wenn kein Eltern Knoten existiert, ist der Ausgewählte Knoten die Wurzel des Baumes. In diesem Fall ersetzen wird die Wurzel des ersten Baumes mit dem Ausgewählten Knoten des zweiten Baums.

2. Wenn ein Eltern Knoten existiert muss dieser Eltern Knoten vom Typ "Operator" sein. Denn nur "Operator" Knoten können Eltern Knoten sein. Wir ersetzten dann das entsprechende Kind mit dem ausgewählten Knoten vom zweiten Baum.

- Rekombination von $S_{initial}$:

Listing 37: Rekombination von $S_{initial}$ mit einer anderen Übergebenen Lösung.

```
public static List<Tree> largeCrossover(List<Tree> first, List<Tree>
    second) {
    if (first.size() == 0) {
        return second;
    }

    if (second.size() == 0) {
        return first;
    }

    int randomIndex = random.nextInt(Math.min(first.size(), second.
        size()));

    List<Tree> firstCopy = new ArrayList<>(first);
    List<Tree> secondCopy = new ArrayList<>(second);

    firstCopy.subList(randomIndex, firstCopy.size()).clear();
    secondCopy.subList(0, randomIndex).clear();
    firstCopy.addAll(secondCopy);

    return firstCopy;
}
```

Die Rekombination darf nicht mit leeren Bäumen durchgeführt werden. Die Listen werden an einem zufälligen Index abgeschnitten und kombiniert.

- Mutation eines Operators Op:

Listing 38: Implementierung der "mutateOperator" Funktion zur Mutation eines Operators von einem übergebenen Baum.

```
public static Tree mutateOperator(MutationStatistic statistic, Tree
    tree) {
    Tree copy = tree.copy();
    if (copy.isEmpty()) return copy;

    List<OperatorNode> nodes = copy.getOperatorNodes();
    if (nodes.size() < 1) return copy;
    OperatorNode randomNode = nodes.get(random.nextInt(nodes.size()))
        ;
```

```
    double randomValue = random.nextDouble(statistic.sum() -
        statistic.get(randomNode.getOperator()));
    for (Map.Entry<Operator, Double> entry : statistic.operatorWeight
        .entrySet()) {
        if (entry.getKey() == randomNode.getOperator()) continue;
        if (randomValue <= entry.getValue()) {
            randomNode.setOperator(entry.getKey());
            break;
        }
        randomValue -= entry.getValue();
    }

    return copy;

}
```

Die Implementierung 38 verwendet ein Objekt zur Verwaltung der Wahrscheinlichkeit, dass ein Operator ausgewählt wird. Dies ermöglicht eine dynamische Anpassung von diesen. Dadurch kann jedem Operator eine individuelle Wahrscheinlichkeit zugewiesen werden. In der aktuellen Implementierung werden diese Wahrscheinlichkeiten allerdings nicht dynamisch verwaltet.

- Mutation einer Konstanten c:

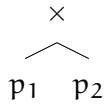
Listing 39: implementierung der "valueMutation" Funktion, die eine zufällige Konstante im übergebenen Baum verändert.

```
public static Tree valueMutation(Tree tree) {
    Tree copy = tree.copy();
    if (tree.isEmpty()) return copy;


    List<ValueNode> valueNodes = new ArrayList<>();
    for (Node node : copy.getLeafNodes()) {
        if (node instanceof ValueNode valueNode) valueNodes.add(
            valueNode);
    }
    if (valueNodes.size() < 1) {
        return consumerWeightMutation(tree);
    }
    ValueNode randomNode = valueNodes.get(random.nextInt(valueNodes.
        size()));
    randomNode.setValue(randomNode.getValue() * random.nextDouble
        (-2.0,2.0));

    return copy;
}
```

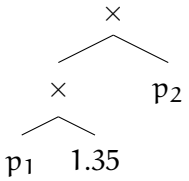
Für den Fall, dass der Baum über keinen konstanten Wert verfügt, wird für eine Basis Funktion ein Gewicht hinzugefügt. Dafür wird ein zufäl-

liger "consumer" Knoten ausgewählt. Dieser Knoten bekommt als neuen Elternknoten einen neuen Operator Knoten. Zusätzlich wird für diesen Operator Knoten ein "value" Knoten mit einem zufälligen Wert $\lambda \in (-2, 2]$ erzeugt. Dieser "operator" Knoten wird dann an die Position des ehemaligen "consumer" Knoten gesetzt.

Sei T_1 eine Kombination von zwei Parametern:



T_1 kann dann nach der Mutation mit $\lambda = 1.35$ visualisiert werden:



Mithilfe der aktuell besten Solution und der Funktionen wird in einem Zeitschritt t ein neuer Baum T_{neu} generiert. T_{neu} wird für den gesamten Zeitschritt t und für die Solution benötigt. Alle in der Simulation generierten Bäume werden in einer Liste zwischengespeichert. Zusätzlich wird die aktuelle beste Solution sowie ein Index benötigt, der auf den aktuellen Baum zeigt.

Listing 40: Speicherung von dem aktuellen und dem besten Bäumen als Liste.

```
private static List<Tree> currentTrees = new ArrayList<>();
private static int currentTreeIndex = 0;
private static List<Tree> bestTrees = new ArrayList<>();
```

Die Variable "generateTree" bestimmt, ob ein Baum mittels Parameteroptimierung (5.4) oder mit dem GA erfolgt. In den ersten 30 Simulationen wird mithilfe der Parameteroptimierung Passende Lösungen für ein Szenario erstellt. Anschließend wird mit dem GA begonnen. Ist zu diesem Zeitpunkt keine Lösung vorhanden wird die Parameteroptimierung solange fortgesetzt, bis ein Solution für das Szenario vorliegt.

Mit der Liste "treeFitness" werden generierte Lösungen und die dazugehörige Fitness als Paar verwaltet. (Code 41)

Listing 41: Definierung der Liste "treeFitness" zum Speichern von vorangegangenen Lösungen und deren Fitness.

```
private static List<FitnessPair> treeFitness = new ArrayList<>();
```

Ein ´´FittensPair" ist ein Tupel mit einer Solution als Liste von Auswertungsbäumen und der dazugehörigen Fitness. (Code 42)

Listing 42: Implementierung von "FitnessPair" als Paar zum Speichern von einer Lösung und deren dazugehörigen Fitness.

```
class FitnessPair {

        double fitness;

        List<Tree> trees;

        public FitnessPair(double fitness, List<Tree> trees) {
                this.fitness = fitness;
                this.trees = new ArrayList<>(trees);
        }
}
```

In Code 46 wird das vorgehen zur Verwaltung der gespeicherten Populationen aus Kapitel 4.2 implementiert. Die Fitness von jeder vorhandenen Lösung wird um 10% verringert, wenn eine neue beste Lösung gefunden wurde. Eine Optimale Lösung ist gefunden, falls TWT $= 0$ gilt. Die Fitness dieser Lösung ist 1. Eine gefundene Lösung wird zur Liste "treeFitness" hinzugefügt. Wenn die Liste anschließend mehr als 20 Elemente aufweist, wird das Element mit der geringsten Fitness entfernt.

Am Beginn einer Simulation wird überprüft, ob eine Rekombination von $S_{initial}$ durchgeführt werden soll. Wenn eine Rekombination $S_{initial}$ durchgeführt wird werden alle anderen Veränderungen mit "deactivateMutation" für die Simulation ausgeschaltet. (Code 43)

Listing 43: Auschnitt der "evaluate"-Funktion zum durchführen der Rekombination mit $S_{initial}$

```
if (timeStatistic.newRun && !generateTree) {
    mutationProbability.newRandom();
    if (mutationProbability.compare("largeCrossover") && !treeFitness.
        isEmpty()) {
        deactivateMutation = true;

        crossoverTree = TreeMutation.largeCrossover(currentTrees,
            treeFitness.get(random.nextInt(treeFitness.size())).trees);

    } else {
        deactivateMutation = false;
    }
}
```

Die Verwaltung der Wahrscheinlichkeiten für Rekombination und Mutation erfolgt mithilfe der "ProbabilityStatistics" Klasse. (Code 28) Die Wahrscheinlichkeiten für Mutation und Rekombination werden in Tabelle 2 dargestellt.

Operation	Wahrscheinlichkeit
Rekombination $S_{initial}$	0.3
Rekombination $T_{initial}$	0.5
Mutation gesamt	0.6
Mutation c	0.4
Mutation Op	0.6

Tabelle 2: Festgelegte einzelne Wahrscheinlichkeiten für Mutation und Rekombination.

Die tatsächlichen Wahrscheinlichkeiten in der Implementierung unterscheiden sich von den angegebenen. Wird die Rekombination von $S_{initial}$ durchgeführt werden Mutation und Rekombination für diese Simulation deaktiviert. Dadurch erfolgen keine weiteren Veränderungen von T_{neu}. Eine Rekombination kann nur durchgeführt werden, wenn keine Mutation durchgeführt wurde und andere Bäume für die Rekombination vorhanden sind. Angenommen es wird ein Zeitschritt t in einer Simulation betrachtet, in der Mutation und Rekombination nicht deaktiviert ist. Die Wahrscheinlichkeit für die Rekombination von $T_{initial}$ in einem Zeitschritt t ist: $(1 - 0.6) \times 0.5 = 0.2$. Dann ist die WAhrscheinlichkeit, dass keine Veränderung durchgeführt wird, $(1 - 0.6) \times (1 - 0.5) = 0.2$. Falls eine Mutation durchgeführt wird, beträgt die Wahrscheinlichkeit von der Mutation Op 0.6 und Mutation c 0.4. Mutation Op wird in diesem Zeitschritt dann mit einer Wahrscheinlichkeit von $0.6 \times 0.6 = 0.36$ durchgeführt.

Der Ablauf für die Erstellung von T_{neu} in Code 47 ist dann folgender: Der in "bestTrees" am Index gespeicherte Baum wird kopiert. Ein neuer Baum wird mit der Parameteroptimierung generiert, wenn der aktuelle Index größer ist als die Anzahl von gespeicherten Bäumen. Der kopierte Baum wird anschließend den Wahrscheinlichkeiten entsprechend verändert und "currentTrees" hinzugefügt. Sind Mutation in der Simulation deaktiviert, wird der Baum in "crossoverTree" zur Auswertung der Station verwendet.

Listing 44: Extrahierung des gespeicherten aktuellen Baum aus der Liste "currentTrees".

```
if (currentTrees.size() > 0) {
    evaluation = currentTrees.get(currentTrees.size() - 1);
}
```

Der abgespeicherte Baum wird verwendet, um den aktuellen Wert von der Station für einen Agenten zu bestimmen. In Code 44 wird die Implementierung gezeigt. Im Fall, dass kein neuer Zeitschritt erfolgt ist und kein neuer Baum generiert werden muss, wird der gespeicherte Baum wiederverwendet.

6

VERSUCHSAUFBAU

Mithilfe des PerformanceEvaluation Tool (PET) von Apeldoorn et al. [2023] wird die Effizienz einer Implementierung an erstellten Szenarien getestet. Dadurch können Implementierungen hinsichtlich ihrer Effizienz bewertet und verglichen werden können. Es ist zu beachten, dass die aktuelle Version des PET keine Szenarien auswerten kann, wenn zwischen einem Agenten und einer Station, die mit einer "visit" Kante verbunden sind, weder der Agent noch die Station über ein "frequency" oder "necessity" Attribut verfügt oder der Agent und die Station gleichzeitig über ein "frequency" oder "necessity" Attribut verfügen.

Die Entwicklung der Basis-Funktionen und Kriterien erfolgt mithilfe von 10 verschiedenen Szenarien. 4 von diesen Szenarien wurden bereits in 3.4 beschrieben. Die 10 Szenarien wurden in der ASMAC 2023 [Apeldoorn et al. 2024] zur Auswertung eingereichter Implementierungen verwendet. Im folgenden werden diese Szenarien als Trainings-Set bezeichnet. Pro Szenario werden 20-mal 100 aufeinanderfolgende Simulationen durchgeführt. Der daraus berechnete normalisierte Durchschnittswert wird in 6.1 näher erläutert. Zusätzlich wurde für diese 10 konkreten Szenarien die optimale TWT berechnet, sodass die berechnete Effizienz mit der optimalen Lösung vergleichbar ist. Dazu stellt das PET die berechnete maximale Wartezeit sowie die berechnete Wartezeit von jeder Simulation zur Verfügung. Daraus kann dann die optimale Effizienz gemäß 6.1 berechnet werden. Die Tabelle 5 enthält die normalisierte optimale TWT für jedes Szenario aus dem Trainings-Set. Die optimale normalisierte TWT des Trainings-Set wird mit dem Durchschnitt der verwendeten Szenarien berechnet. Für das Trainings-Set ergibt das: ≈ 0.24095

Die Auswertung der Implementierungen an unbekannten Problemen erfolgt an 10 Szenarien, die für die ASMAC 2024 [Apeldoorn et al. 2024] verwendet wurden. Diese Szenarien werden als Auswertungs-Set bezeichnet. Für die Entwicklung ist das Auswertungs-Set nicht betrachtet worden. Pro Szenario werden 20-mal 100 aufeinanderfolgende Simulationen durchgeführt.

Die Effizienz vom Trainings-Set und dem Auswertungs-Set wird nach 100 betrachtet. Zur weiteren Einordnung der Effektivität werden die Implementierungen mit den in der ASMAC eingereichten Lösungen verglichen. Für die ASMAC 2023 ist "AgentLearning" die beste eingereichte Lösung von Schmidt und Becker [2023]. Der Ansatz basiert darauf aus verschiedenen Kriterien auszuwählen, die zur Bewertung einer Station verwendet werden. Die beste Eingereichte Lösung der ASMAC 2024 ist "ABMetaLearnerMod" von Leonard Halstenberg. Für diese beiden Lösungen wurde eine neue Auswertung mit 20-mal 100 Simu-

lationen für die entsprechenden Szenarien durchgeführt. Dadurch unterscheiden sich die Ergebnisse von denen der ASMAC. Alle eingereichten Implementierungen können unter [Apeldoorn et al. 2024] eingesehen werden.

6.1 BERECHNUNG DER EFFIZIENZ

Die Effizienz wird berechnet indem die ermittelte TWT einer Simulation j normalisiert wird:

$$effizienz_j = \frac{\text{ermittelte TWT}_j}{\text{voraussichtlich maximale TWT}}$$

Dazu wird für jedes Szenario die voraussichtlich maximale TWT berechnet. Für die $effizienz$ gilt: $0 \leqslant effizienz \leqslant \infty$. Insbesondere ist es möglich, dass ein Fall eintritt, in dem ermittelte TWT $>$ maximale TWT gilt. Die maximale TWT ist nur geschätzt und muss nicht der tatsächlichen maximalen TWT entsprechen. Die $ermitteltTWT_j$ beschreibt die berechnete Wartezeit für $Simulation_j$. Die $effizienz_j$ ist optimal, wenn $effizienz_j = 0$ gilt. Nicht in jedem Szenario kann eine $effizienz$ von 0 erreicht werden. Wenn für eine Simulation keine Lösung gefunden wird gilt: $effizienz = 1$. Für einen $Datenpunkt$ folgt daraus:

Sei $n \in \mathbb{N}^+$ die Anzahl an verschiedenen Szenarios in der Auswertung, $j \in \mathbb{N}^+$ beschreibt die Anzahl von Simulationen für jedes Szenario und $y \in \mathbb{N}^+$ beschreibt wie oft jedes Szenario ausgewertet werden soll. Die $ermitteltTWT_{ij}$ beschreibt die berechnetet TWT für das Szenario i in Simulation j. Folgend wird davon ausgegangen, dass $ermitteltTWT_{ij} \neq -1$ gilt:

$$datenpunkt_j = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{y}\sum_{x=1}^{y}\frac{\min(ermitteltTWT_{i1}, \ldots, ermitteltTWT_{ij})}{maximaleTWT_i}$$

Es wird für jedes Szenario zunächst der Durchschnitt von y mal j Simulationen bestimmt, um anschließend den Durchschnitt von allen Szenarien zu berechnen.

Für den Vergleich der Effizienz von den Implementierungen werden zwei Faktoren berücksichtigt:

1. Die $avg\ best\ TWT_j = datenpunkt_j$, also die durchschnittlich geringste Wartezeit im Durchlauf j.
2. Die durch die Datenpunkte eingeschlossene Fläche. Diese beschreibt wie schnell eine Implementierung eine möglichst geringe Wartezeit erzielt hat.

In beiden Fällen wird der berechnete Wert nach 100 Simulationen betrachtet.

7

AUSWERTUNGSERGEBNISSE

Die Auswertung erfolgt anhand der in Kapitel 6 beschriebenen Trainings- und Auswertungs-Sets. In Tabelle 3 sind die Ergebnisse der Auswertung des Trainings-Set für beide vorgestellten Verfahren, "AgentLearning" [Schmidt und Becker 2023] und dem Optimum dargestellt. Die Implementierung der Parameteroptimierung "DecisionTree" erzielt nach 100 Simulationen eine geringere durchschnittliche TWT als der Genetische Algorithmus "MutationTree" und "AgentLearning" [Schmidt und Becker 2023]. Die Anpassung der Basis Kriterien für die Parameteroptimierung ist mit dem Trainings-Set erfolgt, dementsprechend ist es zu erwarten, dass dieses Verfahren eine höhere Effizienz aufweist. Die durchschnittliche beste TWT von "DecisionTree" erreicht dennoch nicht die ausgerechnete optimale TWT von ≈ 0.24095. In Abbildung 15 sind die gesamt Ergebnisse der Auswertung für jede der 100 Iterationen dargestellt. Mit diesen Werten sind die Ergebnisse von Tabelle 3 berechnet worden.

AGENT	DECISIONTREE	MUTATIONTREE	AGENTLEARNING	OPTIMUM
avg. best TWT	≈ 0.2487	≈ 0.2517	≈ 0.2613	≈ 0.24095
Area	≈ 26.44	≈ 26.74	≈ 28.44	≈ 24.095

Tabelle 3: Auswertungsergebnisse vom Trainings-Set mit der besten eingereichten Lösung nach 100 Simulationen von der ASMAC 2023 und dem Optimum. Alle eingereichten Lösungen sind einsehbar unter [Apeldoorn et al. 2024].

Die Ergebnisse der Auswertung von dem Auswertungs-Set sind in Tabelle 4 dargestellt. Die Implementierung des GA erzeugt eine bessere Lösung als "DecisionTree" und "ABMetaLearnerMod". Sowohl in der besten durchschnittlichen TWT als auch in der eingeschlossenen Fläche stellt diese Implementierung eine Verbesserung da. Die Ergebnisse legen nahe, dass der GA neue Kombinationen von Parametern erzeugt hat die für das Auswertungs-Set besser geeignet sind, um eine Effiziente Lösung zu finden. Die Abbildung 16 beinhaltet die visualisierte Darstellung der durchgeführten Simulationen analog zu Abbildung 15.

AGENT	DECISIONTREE	MUTATIONTREE	ABMETALEARNERMOD
avg. best TWT	≈ 0.3230	≈ 0.3166	≈ 0.3297
Area	≈ 34.87	≈ 33.97	≈ 35.83

Tabelle 4: Auswertungsergebnisse vom Auswertungs-Set im Vergleich mit der besten eingereichten Lösung "ABMetaLearnerMod" von der ASMAC 2024 nach 100 Simulationen.

Die Ergebnisse aller durchlaufender Simulationen sind unter [Brenner 2024] einsehbar. Die Effizienz von "DecisionTree" und "MutationTree" ist in den untersuchten Szenarien höher als die bisherigen in der ASMAC eingereichten Implementierungen. Eine Verbesserung der Parameteroptimierung mithilfe des Genetischen Algorithmus konnte nur für das Auswertungs-Set gezeigt werden. Die Anpassungsfähigkeit des GA an die vorher unbekannten Probleme ist für diese Szenarien höher.

8

FUTURE WORK

Es bedarf weiterer Analysen, um die Anpassungsfähigkeit des GA an unbekannte Probleme im Vergleich zur Parameteroptimierung zu untersuchen. Dazu können weitere komplexere Szenarien erstellt werden, an denen beide Verfahren getestet werden.

Für die Parameteroptimierung kann eine weitere Analyse der Basis-Funktionen erfolgen um diese zu Verbessern. Es ist möglich das bei der Implementierung Randfälle nicht beachtet wurden, die ergänzt werden können. Dafür sind weitere Auswertungen an neuen Szenarien notwendig. Es ist abzuwarten, ob die fehlenden Funktionalitäten zum ASSS hinzugefügt werden. In diesem Fall müssen beide Implementierungen erneut angepasst werden. Insbesondere an den Basis-Kriterien sind weitere Anpassungen notwendig.

Der vorgestellteGA kann durch Anpassung der Explorations- und Exploitationsraten zusätzlich verbessert werden. Es sind weitere Untersuchungen nötig, um festzustellen ob eine dynamische Anpassung der Wahrscheinlichkeiten die Effizienz noch verbessern kann. Zusätzlich kann der GA zum Ende der Auswertung mit anderen Optimierungsverfahren kombiniert werden. Beispielsweise könnte ein ”Hill-Climbing” Algorithmus verwendet werden um die gefundene Lösung lokal zu optimieren. Die Effektivität von einem hybriden Algorithmus muss analysiert werden.

Zukünftig können neuronale Algorithmen für das ASSS implementiert werden, sodass dieser Ansatz mit den vorgestellten Implementierungen verglichen werden kann.

A

ANHANG

Listing 45: Erstellung des ADT-Baums in der "evaluation" Funktion. Auszug aus "DecisionTree".

```
private static HashMap<Agent, List<String>> decision = new HashMap<>();

public static double evaluate(Agent me, HashMap<Agent, Object> others, List
    <Station> stations, long time, Station station, TimeStatistics
    timeStatistic) {
    Node currentNode = new Node(0);

    if (probabilityStatistic.compare("path")) {
        currentNode.addNode(attributeNodes.get(Attribute.PATH_COST));
    }
    if (probabilityStatistic.compare("space")) {
        currentNode.addNode(attributeNodes.get(Attribute.STATION_SPACE));
    }
    if (probabilityStatistic.compare("distribution")) {

        if (stationFrequency) {
            currentNode.addNode(attributeNodes.get(Attribute.
                STATION_FREQUENCY));
        }
        if (agentFrequency) {
            currentNode.addNode(attributeNodes.get(Attribute.
                AGENT_FREQUENCY));

        }
    }
    if (probabilityStatistic.compare("directedTime")) {
        currentNode.addNode(attributeNodes.get(Attribute.
            INCOMING_TIME_CONNECTION));
        currentNode.addNode(attributeNodes.get(Attribute.
            OUTGOING_TIME_CONNECTION));
    }
    if (probabilityStatistic.compare("undirectedTime")) {
        currentNode.addNode(attributeNodes.get(Attribute.
            UNDIRECTED_TIME_CONNECTION));
    }
    if ((!stationFrequency && !agentFrequency || !probabilityStatistic.
        compare("distribution"))
            && !probabilityStatistic.compare("space") && !
                probabilityStatistic.compare("path") && !
```

```
                probabilityStatistic.compare("directedTime") && !
                probabilityStatistic.compare("undirectedTime")) {
            currentNode.addNode(attributeNodes.get(Attribute.MAX_DISTRIBUTION))
                ;
        }
        if (!decision.containsKey(me)) {
            decision.put(me, probabilityStatistic.getCurrentComparison());
        }
        return currentNode.evaluate(me, others, station);
    }
```

Listing 46: Auszug aus der "evaluate" Funktion in der gespeicherte Populationen verwaltet werden.

```
public static double evaluate(Agent me, HashMap<Agent, Object> others, List
    <Station> stations, long time, Station station, TimeStatistics
    timeStatistic) {
    double currentFitness = 0.0;

    if (timeStatistic.newRun && timeStatistic.lastRunCompleted) {
        if (timeStatistic.newBestRun) {

            for (FitnessPair pair : treeFitness) {
                pair.fitness *= 0.9;
            }
        }

        if (timeStatistic.currentTwT == 0L) {
            currentFitness = 1.0;
        } else {
            currentFitness = timeStatistic.lowestTwT / timeStatistic.
                currentTwT;
        }

        if (currentFitness > 0.0) {
            treeFitness.add(new FitnessPair(currentFitness, currentTrees));

            if (treeFitness.size() > 20) {

                FitnessPair toRemove = null;
                for (FitnessPair pair : treeFitness) {
                    if (toRemove == null || toRemove.fitness > pair.fitness
                        ) {
                        toRemove = pair;
                    }
                }

                if (toRemove != null) treeFitness.remove(toRemove);
            }
        }
    }
    return evaluation.evaluate(me, others, station);
```

```
}
```

Listing 47: Auschnitt der “evaluate” Funktion zur genetischen Veränderung des aktuellen Baums.

```
if (!deactivateMutation) {
    evaluation = bestTrees.get(currentTreeIndex).copy();

    if (timeStatistic.newRun) {
        mutationProbability.newRandom();
        basicMutationProbability.newRandom();
    }

    if (timeStatistic.lastValue != timeStatistic.time || currentTrees.size
        () < 1) {
        if (mutationProbability.compare("mutation")) {

            if (basicMutationProbability.compare("value")) {
                evaluation = TreeMutation.valueMutation(evaluation);
            } else {
                evaluation = TreeMutation.mutateOperator(mutationStatistic,
                     evaluation);
            }
        }

        if (!mutationProbability.compare("mutation") && mutationProbability
            .compare("crossover") && !treeFitness.isEmpty()) {

            List<Tree> crossover = treeFitness.get(random.nextInt(
                treeFitness.size())).trees;

            if (crossover.size() > currentTreeIndex) {
                evaluation = TreeMutation.crossover(evaluation, crossover.
                    get(currentTreeIndex));
            } else {
                evaluation = TreeMutation.crossover(evaluation, crossover.
                    get(crossover.size() - 1));
            }
        }



    } else {
        evaluation = currentTrees.get(currentTrees.size() - 1);
    }
} else {
    if (crossoverTree.size() > currentTreeIndex) {
        evaluation = crossoverTree.get(currentTreeIndex).copy();
    }
}
```

A.1 AUSWERTUNGSERGEBNISSE

SZENARIO	WORST CASE TWT	BEST CASE TWT	NORMALISIERTE TWT
CHANGED Manufactoring Process	171	47	0.2749
CHANGED Supermarket	2015	1395	0.6923
CHANGED Treatments	1763.33	330	0.1871
NEW Elevator	112	84	0.75
NEW Patient Transportation	900	46	0.0511
NEW Starting Jobs	2310	154	0.0667
NEW Storage 2	446.6667	60	0.1343
SCALED Packaging	81.999	0	0
SCALED Production Lines	7280	852	0.1170
SCALED Ward Round	294	40	0.1361

Tabelle 5: Berechnung der normalisierten optimalen TWT für jedes Szenario des Trainings-Set.

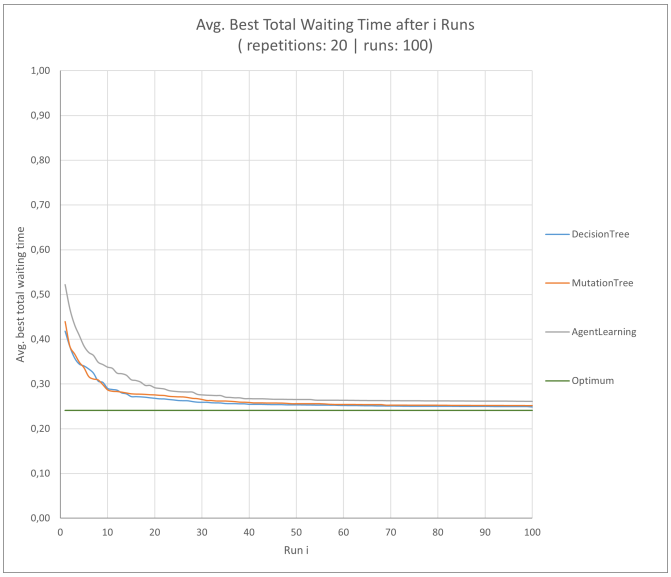


Abbildung 15: Die normalisierte TWT von den Agenten für das Trainings-Set über 100 Simulationen

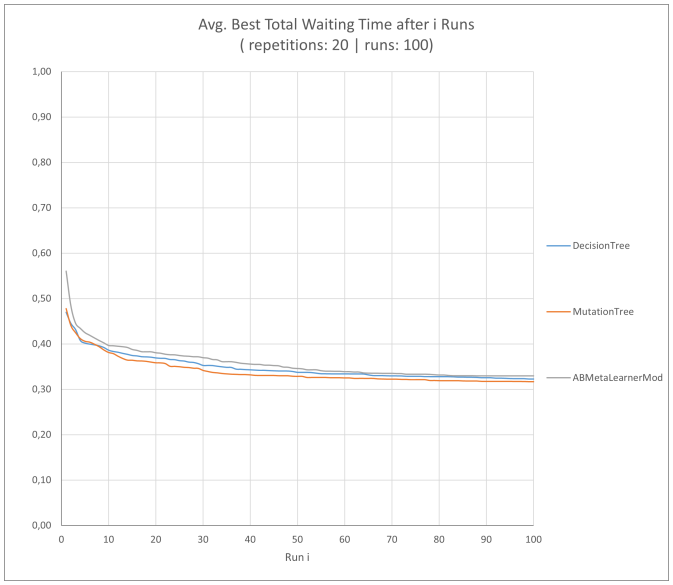


Abbildung 16: Die normalisierte TWT von den Agenten für das Auswertungs-Set über 100 Simulationen

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

LISTINGS

ABKÜRZUNGSVERZEICHNIS

TWT total waiting Time
ASSS AbstractSwarm Simulation System
MAS Multi-Agent System
IMBEI Institut für Medizinische Biometrie, Epidemiologie und Informatik
ASMAC AbstractSwarm Multi Agent Competition
PET PerformanceEvaluation Tool
MDRP Meal Delivery Routing Problem
GA Genetischer Algorithmus
ADT Abstrakter Datentyp

LITERATURVERZEICHNIS

Daan Apeldoorn. AbstractSwarm – A Generic Graphical Modeling Language for Multi-Agent Systems. In Matthias Klusch, Matthias Thimm und Marcin Paprzycki (Hrsg.), *Multiagent System Technologies*, S. 180–192, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-40776-5.

Daan Apeldoorn, Alexander Dockhorn und Torsten Panholzer. AbstractSwarm Simulation System. https://gitlab.com/abstractswarm/abstractswarm, 2021. Accessed: 12.05.2024.

Daan Apeldoorn, Alexander Dockhorn und Torsten Panholzer. Performance Evaluation Tool. https://gitlab.com/abstractswarm/performanceevaluation, 2023. Accessed: 12.05.2024.

Daan Apeldoorn, Alexander Dockhorn und Torsten Panholzer. AbstractSwarm Multi-Agent Logistics Competition. https://abstractswarm.gitlab.io/abstractswarm_competition/, 2024. Accessed: 14.08.2024.

Ole Brenner. AbstractSwarm Implementierungen und Auswertungsergebnisse. https://git.hci.uni-hannover.de/oleb/PublicAbstractSwarm, 2024.

E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, Dec 1959. ISSN 0945-3245. doi: 10.1007/BF01386390. URL https://doi.org/10.1007/BF01386390.

Daniel Giraldo-Herrera und David Álvarez Martínez. *A GRASP Algorithm for the Meal Delivery Routing Problem*, S. 306–320. Springer Nature Switzerland, 2024. ISBN 9783031629228. doi: 10.1007/978-3-031-62922-8_21. URL http://dx.doi.org/10.1007/978-3-031-62922-8_21.

David E. Goldberg. *Genetic Algorithms and Innovation*, S. 1–9. Springer US, Boston, MA, 2002. ISBN 978-1-4757-3643-4. doi: 10.1007/978-1-4757-3643-4_1. URL https://doi.org/10.1007/978-1-4757-3643-4_1.

Xiao-Cheng Liao, Yi Mei und Mengjie Zhang. Learning Traffic Signal Control via Genetic Programming, 2024. URL https://arxiv.org/abs/2403.17328.

Kumara Sastry, David E. Goldberg und Graham Kendall. Genetic Algorithms. In Edmund K. Burke und Graham Kendall (Hrsg.), *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, S. 93–117. Springer US, Boston, MA, 2014. ISBN 978-1-4614-6940-7. doi: 10.1007/978-1-4614-6940-7_4. URL https://doi.org/10.1007/978-1-4614-6940-7_4.

Elisa Schmidt und Matthias Becker. Training Agents for Unknown Logistics Problems. In *Proceedings of the Companion Conference on Genetic and Evolutionary Computation*, GECCO '23 Companion, S. 243–246, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701207. doi: 10.1145/3583133.3590724. URL https://doi.org/10.1145/3583133.3590724.

D Simić, S Simić, D Milutinović und J Djordjević. Challenges for nurse rostering problem and opportunities in hospital logistics. *Journal of Medical Informatics & Technologies*, 23:195–202, 2014.

Matej Črepinšek, Shih-Hsi Liu und Marjan Mernik. Exploration and exploitation in evolutionary algorithms: A survey. *ACM Comput. Surv.*, 45(3), jul 2013. ISSN 0360-0300. doi: 10.1145/2480741.2480752. URL https://doi.org/10.1145/2480741.2480752.

D.H. Wolpert und W.G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, April 1997. ISSN 1941-0026. doi: 10.1109/4235.585893.

Jinxin Xu, Haixin Wu, Yu Cheng, Liyang Wang, Xin Yang, Xintong Fu und Yuelong Su. Optimization of Worker Scheduling at Logistics Depots Using Genetic Algorithms and Simulated Annealing, 2024. URL https://arxiv.org/abs/2405.11729.

EIDESSTATTLICHE ERKLÄRUNG

Hiermit versichere ich, die vorliegende Arbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die wörtlich oder inhaltlich aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Hannover, 29. August 2024

Ole Brenner