Intelligentes Paketmanagement-System

Maximilian Brauchle, Nicolas Mahn, Leon Rottler

Fakultät Wirtschaftsinformatik

Hochschule Furtwangen | Furtwangen University

Wissenschaftliche Arbeit im Modul Industrial Analytics

betreut durch Prof. Dr.-Ing. Ulf Schreier

28. Januar 2022

Zusammenfassung

Das Einkaufsverhalten der Bevölkerung ändert sich zunehmend. Mehr und mehr Personen erwerben ihre Waren über Onlineshopping. Die Zahl der Online-Einkäufe steigt und Paketzulieferer stehen vermehrt vor der Herausforderung, Pakete in kürzester Zeit auszuliefern. Daher ist es notwendig zu verstehen, an welchen Stellen Zeit und Arbeit eingespart werden können, um dem immer größer werdenden Wettberwerbsdruck stand zu halten.

Das Ziel dieses Projektes ist es, ein System zu entwickeln, welches eine sortierte Reihenfolge von Paketen ermittelt, woraus schließlich eine optimale Route entsteht. Die Route soll unter anderem eine Priorisierung von beispielsweise sehr schweren oder sehr großen Paketen zulassen.

Um das Ziel zu erreichen, wurde zunächst untersucht, welche Daten für die Umsetzung des Projektes erforderlich sind und aus welchen Datenquellen, diese Informationen bezogen werden können. Des Weiteren wurden Nachforschungen erhoben, welche Machine Learning Algorithmen für die Routenberechnung und damit auch für die Priorierung der Pakete und dem Generieren der Daten benötigt werden. Anschließend wurde eine Software-Architektur entworfen, auf deren Grundlage die eigentliche Umsetzung des Projektes erfolgen konnte.

Die Projektarbeit zeigte, dass Pakete auf Basis von Adressdaten und synthetisch generierten Paketdaten mithilfe von Machine Learning priorisiert und sortiert werden können, sodass sich eine optimale Route ergibt. Der oder die Paketzusteller*in spart damit einiges an Zeit bei der Beladung und der Auslieferung der Pakete.

Dies zeigt, dass sich die Effizienz von Paketzusteller*innen durch die automatisierte Sortierung und Priorisierung der Pakete erheblich steigern lässt. Auf dieser Grundlage ist es empfehlenswert, Machine Learning im Bereich der Paketzustellung einzusetzen, um Arbeitszeiten und dadurch entstehende Kosten zu sparen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Projektbeschreibung

In dem Projekt „Intelligentes Paketmanagement-System“(IntPakMan) geht es um das automatisierte Sortieren von Paketen mithilfe von Machine Learning. Als Ausgangssituation wird angenommen, dass die Paketverteilung folgendermaßen organisiert ist (typischer Prozess bei der Deutschen Post):

1. Die Pakete werden morgens von einem Paketzentrum unsortiert an den Zustellstützpunkt („Basisstation“für einen Ort/Stadt) gebracht.
2. Postler*innen müssen diese Pakete auf die jeweiligen Bezirke verteilen (mehrere Bezirke pro Zustellstützpunkt).
3. Postler*innen müssen die Paketen für ihren Bezirk in die richtige Reihenfolge bringen und das Auto beladen.

Aus dieser klassischen Art und Weise der Paketverteilung ergeben sich folgende Probleme:

1. Die Pakete sind morgens unsortiert, der/die Postler*in muss die Pakete sortieren, was sehr zeitaufwendig ist. (Pakete müssen zuerst auf die Bezirke verteilt werden und dann pro Bezirk in die richtige Reihenfolge gebracht werden)
2. Die Pakete werden anhand einer festen, vorgegebenen Route ausgeliefert, dadurch kann es beispielsweise vorkommen, dass große Pakete erst am Ende der Route kommen und die gesamte Route über einen großen Teil des Laderaums belegen.
3. Der Laderaum der Autos ist an manchen Tagen wegen zu vieler (großer) Pakete zu gering, weshalb der/die Postler*in mehrmals zum Zustellstützpunkt fahren muss, um nachzuladen.

Diese Probleme versuchen wir mit unserem Projekt zu beheben. Die Vision ist, dass es ein System gibt, indem die Paketdaten insofern verarbeitet werden, dass eine Sortierung der Pakete möglich wird und eine Maschine auf diese Daten zugreifen kann. Dadurch wären die Pakete dann bereits richtig sortiert, sodass ein/e Postler*in nur noch an ein Förderband o. ä. gehen müsste und die Pakete in sortierter Reihenfolge erhalten würde.

1.2 Zielsetzung

Das Ziel ist es, ein solches System zu bauen, dass Paketdaten insofern verarbeitet, dass eine Sortierung der Pakete möglich wird. Konkret soll ein System entstehen, an das Paketdaten übermittelt werden und welches anschließend in der Lage ist, mittels Machine Learning Pakete zu priorisieren und anhand der Priorisierung eine optimale Route zu ermitteln. Die Ergebnisse sollen mittels einer API abrufbar sein und in einem Frontend dargestellt werden. Ebenfalls denkbar wäre, dass eine Maschine die Daten abruft und die Pakete sortiert.

1.3 Motivation

Die Motivation hinter diesem Projekt ist vielfältig. Betrachtet man den/die Postler*in, besteht die Motivation darin, den Arbeitsalltag zu verbessern und erleichtern, indem die Pakete zum einen morgens nicht mehr sortiert werden müssen, was Kraft für den restlichen Arbeitsalltag spart, und zum anderen, dass schwere und große Pakete tendenziell am Anfang der Route kommen, sodass gegen Ende der Route lediglich die angenehmer zu handhabenden Pakete auszuliefern sind.

Aus Unternehmenssicht könnten aus dem Projekt viele Vorteile resultieren. Beispielsweise würde die Mitarbeiterzufriedenheit aus den zuvor genannten Gründen steigen. Aber auch die Kundenzufriedenheit würde sich vermutlich verbessern, da Pakete seltener beschädigt werden sowie Pakete zuverlässiger ankommen, da es zu keinen/sehr wenigen Fehlern in der Sortierung kommt, anders als wenn Menschen die Pakete sortieren würden. Es würden sich sogar neue Einnahmequellen generieren lassen, beispielsweise indem man gewisse Vorteile durch die Priorisierung monetarisiert.

2 Anforderungen

In Kapitel 1.1, der Projektbeschreibung, wird die Ausgangssituation der Paketverteilung und die damit verbundenen Probleme erläutert. Hieraus lassen sich folgende Anforderungen an das System/Projekt ableiten:

1. Die Pakete sollen bei vorgegebener Route in der richtigen Reihenfolge sortiert werden. Dies würde den Postler*innen insofern helfen, als dass diese die Pakete nicht mehr selbst sortieren müssen, sondern diese bereits sortiert (beispielsweise mittels eines Förderbands) ankommen. Somit könnte etliches an Zeit eingespart sowie Fehler reduziert/vermieden werden.
2. Die Pakete sollen pro Bezirk nach ihrer optimalen Route sortiert werden und zusätzlich sollen die Pakete mittels Machine Learning priorisiert werden. Durch diesen Schritt lässt sich die Route optimieren und beispielsweise besonders schwere oder besonders große Pakete besser in die Route einplanen.
3. Es soll eine tagesabhängige Zuweisung von Postautos auf die Bezirke geben. Diese soll auf verschiedenen Faktoren basieren, wie beispielsweise dem Volumen der Pakete die zu geladen werden sollen, der insgesamt zu fahrende Strecke (relevant bei E-Autos), Restriktionen innerhalb eines Bezirkes usw ...
4. Sollte ein Bezirk an einem Tag mehr Pakete haben als ein anderer, soll eine bezirksübergreifende Paketauslieferung möglich sein, das heißt, dass ein Bezirk einen kleinen Teil der Pakete eines anderen Bezirks ausliefert, um diesen zu entlasten.

Zur erfolgreichen Umsetzung des Projektes müssen mindestens Punkt 1 und Punkt 2 erfolgreich umgesetzt werden.

3 Literaturreview

Es gibt gewisse Programme, welche eine schnellere Durchlaufzeit haben als andere. Wesentlicher ist aber, dass bei manchen Programmen die Durchlaufzeit immer höher wird, je mehr Daten das Programm bearbeiten muss. Ein Programm, welches zum Beispiel immer nur den ersten Eintrag einer Liste auslesen soll, hat immer die gleiche Durchlaufzeit egal wie lang die Liste ist. Dieses Programm hat somit eine Komplexität von $O(1)$. Ein anderes Programm, das einen gewissen Eintrag aus einer Liste ausgeben soll, ohne den Index oder Key zu kennen, hat eine Komplexität von $O(n)$. n steht hier für die Länge der Liste. Da der Eintrag im schlimmsten Fall, der letzte Eintrag in der Liste seien könnte, ist die Komplexität $O(n)$. Wenn man annimmt, dass die Liste vorher sortiert wird, gibt es Möglichkeiten, den Eintrag schneller ausgeben zu lassen. Ist die Liste zum Beispiel nach einem Binary Tree sortiert, ist die Komplexität der Suche nur $O(log_2(n))$. Allerdings lassen sich nicht alle Probleme mit so einer einfachen Komplexität (mit einem bekannten Algorithmus) lösen. Generell unterscheidet man in der Informatik zwischen zwei verschieden Typen von Algorithmen. Algorithmen, die sich in Polynom-Zeit lösen lassen (kurz P). Also Probleme die höchstens eine Komplexität von $O(n^x)$ ($x \in \mathbb{R}^+$) haben. Probleme die eine höhere Komplexität besitzen (z.B. $O(x^n)$), nennt man auch NP (nondetermenistic polynomial time). Ein bekanntes NP Problem ist das Travelling Sales Person (TSP) Problem. Bei diesem gibt es eine*n Händler*in welche*r von Ort zu Ort fährt und dabei versucht, die optimale Route zu nehmen. Dabei soll kein Ort zweimal besichtigt werden und die Route soll am gleichen Ort starten und aufhören. Dieses Problem ist somit im Zentrum einer effizienten Paketzustellung.

3.1 Bellman-Held-Karp Algorithmus

Der simpelste Algorithmus um das TSP Problem zu lösen ist einfach, alle möglichen Routen zu berechnen und dann die kürzeste auszuwählen. Ein solcher Algorithmus hat allerdings eine Komplexität von $O(n!)$. Bei 5 Knoten müsste man somit schon 120 mögliche Routen berechnen. Bei 10 Knoten wären es schon 3.628.800 Routen.

Ein schnellerer, genauer Algorithmus ist der Bellman-Held-Karp Algorithmus. Bei diesem wird die kürzeste Route mit der Hilfe von speziellen 1-Trees, ein wiederum spezieller Spannbaum, berechnet. Ein Spannbaum beschreibt einen Graphen, welcher aus K_n Knoten besteht. Ein Knoten ist im Bezug auf das TSP Problem ein Ort. Die Route zwischen zwei Orten ist in Bezug auf ein Spannbaum eine Kante, welche durch ein Gewicht c, in TSP oft die Distanz, klassifiziert wird. Ein Graph mit Knoten wird als Spannwald bezeichnet (siehe Fig. 1).

Ein Spannbaum zwischen allen Knoten ist ein Baum ohne Kreislauf, welcher allerdings alle Knoten miteinander verknüpft (siehe Fig. 2). Ein minimaler Spannbaum kann mit dem Algorithmus von Prim ausgerechnet werden. Ein 1-Tree hat im Gegensatz zu einem normalen Spannbaum genau einen Kreislauf (siehe Fig. 3). In dem Held-Karp Algorithmus werden spezielle 1-Trees benötigt, wo jeder Knoten genau 2 Grad hat, also wenn an einen Knoten 2 Kanten angrenzen (siehe Fig. 4). So ein 1-Tree beschreibt folglich eine Route. Im weiteren Verlauf des Dokuments wird bei einem 1-Tree immer von diesem speziellen 1-Tree ausgegangen. Ein minimaler 1-Tree beschreibt dabei die optimale Route. Da man im TSP Problem im Kreis fährt, kann man zum Lösen des Problems einen beliebigen Anfangsknoten auswählen. Diesen Knoten bezeichnen wir als Knoten 0. Nimmt man zusätzlich einen Knoten i bleiben nur noch k Knoten übrig. Nimmt man an, dass die optimale Route die Kante zwischen 0 und i beinhaltet, dann besteht der optimale 1-Tree aus einem Spannbaum $f(i; j_1, ...j_k)$ und der Kante von 0 nach i. Die Funktion $f(i; j_1, ...j_k)$ beschreibt dabei den Spannbaum von Knoten 0 nach Knoten i über die Knoten $(j_1, j_2, ...j_k)$. Wenn man den Spannbaum $f(0; j_1, ...j_n)$ $(n = \forall k, i)$ findet, also vom Knoten 0 zum Knoten 0, hat man den optimalen 1-Tree. Man kann also generell definieren, dass die Kosten eines generischen Spannbaums durch die Formel:

$$f(i; j_1, ...j_k) = min_k\{c_{ij_m} + f(i; j_1, ...j_{m-1}, j_{m+1}, ...j_k)\}$$

bestimmt werden können. Anhand dieser Formel kann man folglich auch $f(0; j_1, ...j_n)$ bestimmen.

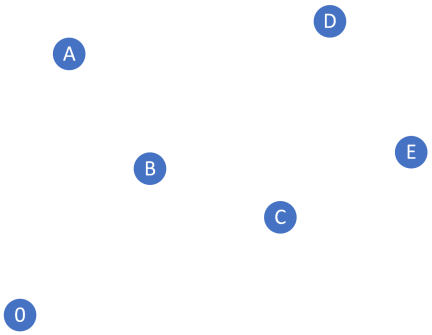

Abbildung 1: Spannwald

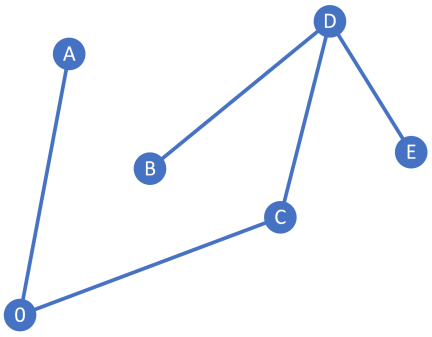

Abbildung 2: Spannbaum

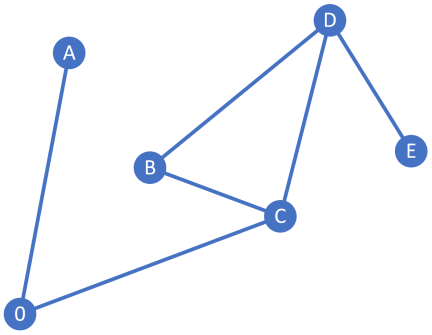

Abbildung 3: 1-Tree

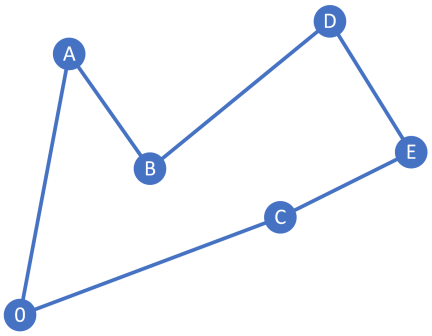

Abbildung 4: spezieller 1-Tree

Anschaulich wird dies anhand eines Beispiels. Eine optimale Route soll durch den Spannwald in Fig. 5 gefunden werden. Anfangs müssen die Gewichte von 0 nach i über einen weiteren Knoten bestimmt werden:

$f(A;B) = 6 + 5 = 11$
$f(A;C) = 9 + 7 = 16$
$f(A;D) = 13 + 8 = 21$
$f(A;E) = 13 + 11 = 24$
$f(B;A) = 8 + 5 = 13$
$f(B;C) = 9 + 4 = 13$
$f(B;D) = 13 + 7 = 20$
$f(B;E) = 13 + 7 = 20$
$f(C;A) = 8 + 7 = 15$
...

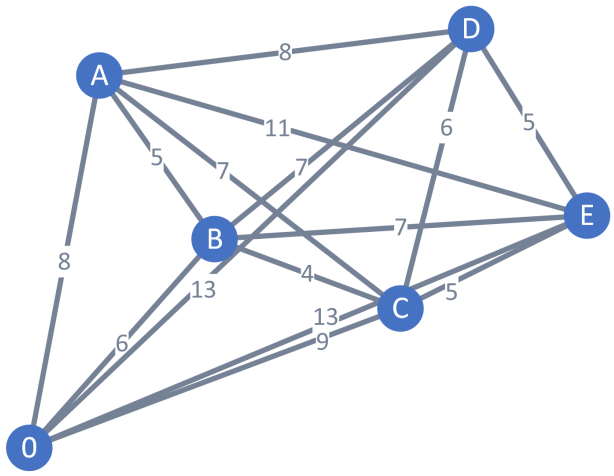

Abbildung 5: Spannwald Beispiel

Jetzt werden die Gewichte von 0 nach i über zwei weitere Knoten bestimmt:
$f(A;B,C) = min\{c_{BA} + f(B;C), c_{CA} + f(C;B)\} = min\{5 + 13, 7 + 10\} = 17$
$f(A;B,D) = min\{c_{BA} + f(B;D), c_{DA} + f(D;B)\} = min\{5 + 20, 8 + 13\} = 21$
$f(A;B,E) = min\{c_{BA} + f(B;E), c_{EA} + f(E;B)\} = min\{5 + 20, 11 + 13\} = 24$
$f(A;C,D) = min\{c_{CA} + f(C;D), c_{DA} + f(D;C)\} = min\{7 + 19, 8 + 15\} = 23$
...
$f(E;C,D) = min\{c_{CE} + f(C;D), c_{DE} + f(D;C)\} = min\{5 + 19, 5 + 15\} = 23$

So geht man weiter vor bis der Spannbaum alle Knoten beinhaltet:

$$\begin{aligned} f(A;B,C,D) &= min\{c_{BA} + f(B;C,D), c_{CA} + f(C;B,D), c_{DA} + f(D;B,C)\} \\ &= min\{5 + 22, 7 + 19, 8 + 16\} = 18 \\ f(A;B,C,E) &= min\{c_{BA} + f(B;C,E), c_{CA} + f(C;B,E), c_{EA} + f(E;B,C)\} \\ &= min\{5 + 21, 7 + 18, 11 + 15\} = 25 \end{aligned}$$

...

$$\begin{aligned} f(A;B,C,D,E) = min\{&c_{BA} + f(B;C,D,E), c_{CA} + f(C;B,D,E), c_{DA} + f(D;B,C,E), \\ &c_{EA} + f(E;B,C,D)\} = 28 \\ f(B;A,C,D,E) = min\{&c_{AB} + f(A;C,D,E), c_{CB} + f(C;A,D,E), c_{DB} + f(D;A,C,E), \\ &c_{EB} + f(E;A,C,D)\} = 30 \end{aligned}$$

...

$$
\begin{aligned}
f(0; A, B, C, D, E) = min\{&c_{A0} + f(A; B, C, D, E), c_{B0} + f(B; A, C, D, E),\\
&c_{C0} + f(C; A, B, D, E), c_{D0} + f(D; A, B, C, E),\\
&c_{E0} + f(E; A, B, C, D)\}\\
&= min\{8 + 28, 6 + 30, 9 + 29, 13 + 27, 13 + 30\}\\
&= 36
\end{aligned}
$$

Das Beispiel hat gezeigt, dass die schnellste Route durch den Spannwald aus Fig. 5 ein Gewicht von 36 hat. Die beste Route hätte man mit dem kleinsten Gewicht immer mit speichern können. Hier $(0, A, D, E, C, B)$ bzw. $(0, B, C, E, D, A)$. Die Komplexität des Bellman-Held-Karp Algorithmus ist $O(n^3) \times 2^n$, dass ist deutlich besser als $O(n!)$. Aufgrund der hohen Komplexität, für dieses Projekt, aber nicht ideal. Da der Bellman-Held-Karp Algorithmus allerdings der schnellste (bekannte) genaue Algorithmus ist, muss für dieses Projekt ein heuristischer oder Approximationsalgorithmus zum lösen des TSP Problems angewandt werden. Diese zeichnen sich dadurch aus, dass sie nicht immer zur perfekten Lösung kommen, jedoch die durchschnittliche Durchlaufzeit wesentlich geringer ist [4, 5].

3.2 Ameisenalgorithmus

Der Ameisenalgorithmus ist ein Optimierungsalgorithmus, welcher deshalb auch Ant-Colony Optimization (kurz ACO) genannt wird. Der Ameisenalgorithmus ist ein Approximationsalgorithmus. Das heißt, die ACO gibt eine ungefähre Antwort. Optimierungsalgorithmen, eine spezielle Form der Approximationsalgorithmen, sind iterierende Algorithmen, welche sich mit jeder Iteration dem Optimum annähern. Normalerweise wird entweder unendlich viel Zeit oder unendlich viel Platz benötigt, um definitiv zum optimalen Ergebnis zu kommen. Approximationsalgoritmen sind vor heuristischen zu bevorzugen, da sie ein ungefähres Ergebnis geben. Heuristische Algorithmen können im schlimmsten Fall auch auf das schlechteste Ergebnis kommen. In Bezug zur künstlichen Intelligenz zählt ACO zur Schwarmintelligenz. Der Algorithmus basiert auf dem tatsächlichen Verhalten von Ameisen bei der Suche nach Nahrung. Ameisen laufen über einen Graphen $G = (V, E)$ (siehe Fig. 6).

In unserem Beispiel besteht V aus zwei Knoten. v_s repräsentiert das Nest der Ameisen, während v_d die Nahrungsquelle ist. E besteht aus zwei Kanten zwischen v_s und v_d, e_1 und e_2. Wie man in Fig. 6 sehen kann, ist e_2 länger als e_1. Ameisen n_a gehen vom Nest auf

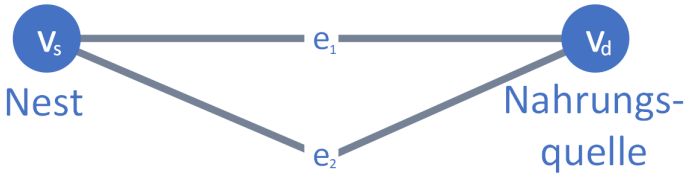

Abbildung 6: ACO Beispiel

Nahrungssuche und entscheiden sich mit einer 50% Wahrscheinlichkeit für e_1 oder e_2. Die Ameisen hinterlassen bei der Nahrungssuche Pheromone. Die Ameisen folgen den Pheromonen (wenn diese vorhanden sind). Wenn die ersten Ameisen, die die Kante e_1 gewählt haben, an der Nahrung angekommen sind, sind die Ameisen, die die andere Kante gewählt haben, noch nicht da. Deshalb entscheiden sich die Ameisen mit einer hohen Wahrscheinlichkeit für die Kante e_1 auf dem Rückweg. Sobald die ersten Ameisen von e_2 ankommt, wird e_1 mit einer Wahrscheinlichkeit p_i gewählt. Diese Wahrscheinlichkeit p_i wird anhand der Menge an Pheromonen auf einem Weg τ_i bestimmt. Da mehr Ameisen den kürzeren Weg genommen haben, haben sich auf diesem mehr Pheromone angesammelt. Deshalb entscheiden sich mehr Ameisen für e_1 auf dem Rückweg. Mit der Zeit verwenden alle Ameisen die kürzere Kante.

Im Vergleich zu den echten Ameisen haben die virtuellen Ameisen ein paar Besonderheiten. Echte Ameisen bewegen sich asynchron. Die virtuellen Ameisen hingegen bewegen sich synchron. Virtuelle Ameisen hinterlassen nur Pheromone auf dem Rückweg zum Nest, echte hinterlassen diese jedoch immer. Echte Ameisen können die Pheromonwerte schneller bzw. auf dem Rückweg implizit auswerten. Virtuelle Ameisen benötigen Rechenzeit dafür.

Wenn virtuelle Ameisen nach Essen suchen, entscheiden sie sich auch erst mit einer 50% Wahrscheinlichkeit für eine der beiden Kanten, kommen aber gleichzeitig am Ziel an. Virtuelle Ameisen nehmen die gleiche Kante zurück, wie die, von der sie gekommen sind, hinterlassen diesmal allerdings einen virtuellen Pheromon-Wert. Dieser wird anhand der Kosten c_i der Kante und einem positiven Parameter des Algorithmus Q wie folgt berechnet:

$$\tau_i \leftarrow \tau_i + \frac{Q}{c_i}.$$

Die Wahrscheinlichkeit, mit welcher die Ameise sich für eine der Kanten entscheidet, wird mit dieser Formel berechnet:

$$p_i = \frac{\tau_i}{\sum^i \tau_i}.$$

Hierbei beschreibt i die Menge der Kanten. Um eine frühe Annäherung an eine suboptimale Lösung zu verringern, wird auch das echte Phänomen der Pheromonverdunstung simuliert. Dies wird jede Runde gemacht, bevor sich die Ameisen für einen der Wege entscheiden. Die Verdunstung wird mit dieser Formel berechnet:

$$\tau_i \leftarrow (1 - \rho) \times \tau_i.$$

ρ ist ein Parameter des Algorithmus und darf zwischen 0 und 1 liegen ($\rho \in (0, 1]$). Je öfter die virtuellen Ameisen zur Nahrungsquelle laufen, um so mehr Ameisen wählen den kürzeren Weg, da die Pheromonwerte über die Iterationen für e_1, immer höher werden. Je mehr virtuelle Ameisen man auf Nahrungssuche schickt, umso vorhersagbarer nähern sie sich dem Optimum. Der Algorithmus wird durch eine Stop-Bedingung beendet.

Um die ACO auf das TSP Problem anzuwenden muss das Ziel der Ameisen angepasst werden. Das Ziel der Ameisen ist es jetzt, einen 1-Tree zu vollenden. Dafür wird wieder ein zufälliger Knoten als Startpunkt, oder hier Nest, definiert. Die virtuelle Ameise hinterlässt bei jeder Kante, welche sie auf ihrer Reise durchquert hat, Pheromone. Dafür muss sich die Ameise ihren gelaufenen Weg s merken. Die Berechnung der Pheromonwerte muss dementsprechend auch angepasst werden:

$$\tau_i \leftarrow \tau_i + \frac{Q}{f(s)}.$$

$f(s)$ berechnet die Kosten der Kante e_i von s.

Um ACO zu verbessern wurden metatheoretische Vorgehensweisen entwickelt. Bei dieser Vorgehensweise werden bei jeder Iteration heuristische Klassifikationen hinzugefügt. Bei der Implementation von Metaheuristiken in der ACO wird jede Iteration in 3 Bereiche unterteilt:

```
while termination_conditions is False:
  AntBasedSolutionConstruction()
  PheromoneUpdate()
  DaemonActions() #optional
```

In der ersten Funktion *AntBasedSolutionConstruction*() wird eine zusätzliche Gewichtungsfunktion η definiert.

$$\eta_i = \frac{1}{c_i}$$

η wird in die Wahrscheinlichkeitsrechnung integriert:

$$p_i = \frac{(\tau_i)^\alpha \times (\eta_i)^\beta}{\sum^i (\tau_i)^\alpha \times (\eta_i)^\beta}.$$

α und β sind zwei Parameter des Algorithmus, die bestimmen wie stark die heuristischen Werte die Wahrscheinlichkeit beeinflussen sollen. In anderen Worten, wird hier definiert, wie stark die Länge der Kanten im Verhältnis zu den vorhandenen Pheromonwerten gewertet werden soll.

In *PheromoneUpdate*() wird eine heuristische Gewichtung in die Pheromoneverdunstung eingefügt:

$$\tau_i \leftarrow (1-\rho) \times \tau_i + \rho \times \sum_{\{s \in S_{upd}\}} w_s \times F(s)$$

$F(s)$ ist eine spezielle Form von $f(s)$, der die Qualität besser misst. w_s ist ein Parameter des Algorithmus, mit welchem man angibt, wie stark $F(s)$ gewichtet werden soll. s ist in dieser Formel ein Element von S_{upd}. S_{upd} wird aus Lösungen von vorherigen Iterationen zusammengestellt. Oft wird S_{upd} aus S_{iter}, den Lösungen aus der letzten Iteration, gebildet. Es gibt auch andere Variationen. Wie die Elitist Variation, welche nach Elitarismus benannt wurde. Nach ihr gilt: $S_{upd} \leftarrow S_{iter} \cup_{bs}$. Mit s_{bs} ist die bisher beste Lösung (best-so-far) gemeint. Bei einer anderen Varianten wie der Ranked-based Variante werden nur die

besten s aus S_{iter} zusammen mit s_{bs} in S_{upd} eingefügt. Eine häufig genutzte Variante ist die MMAS (Min-Max-Ant-System) Variante. Diese kann zusätzlich zu anderen Varianten verwendet werden. Hier wird eine τ_{max} und τ_{min} definiert.

$$\tau_{max} = \frac{1}{1-\rho} \times \frac{1}{F(s_l)}$$

$$\tau_{min} = \frac{\tau_{max} \times (1-\rho)}{avg(\tau_i) \times \rho}$$

s_l ist entweder s_bs oder der beste s aus S_{iter}. Diese sind die maximalen bzw. minimalen Pheromonwerte, die einer Kante zugewiesen werden können. Wenn die Maximal- bzw. Minimalwerte überschritten werden, werden sie durch τ_{max} bzw. τ_{min} ersetzt. Es gibt noch viele andere Varianten.

DaemonActions() ist optional. Hier kann man zusätzliche Pheromone verteilen. In der Umsetzung des Intelligenten Paket Management Systems könnte man möglicherweise hier die Pheromonwerte einzelner Kanten anpassen, um die Priorisierung einzelner Knoten zu gewährleisten. Probleme würden hier wahrscheinlich dennoch entstehen, da die priorisierten Knoten am Anfang der Route angefahren werden sollen, was durch solche Aktionen schwierig bevorzugt werden kann. Besser wäre es wahrscheinlich, eine spezielle Ranked-based Variante im vorherigen Schritt zu implementieren, welche Lösungen belohnt, die die priorisierten Adressen früher anfährt.

Der ACO Algorithmus wird nicht nur für das TSP Problem angewandt, sondern vor allem zum Lösen von Proteinfaltungen, welche zum Beispiel zur Impfstoffherstellung gebraucht werden. Aber auch bei Planungsproblemen sowie Grapheinfärbungen oder beim Daten-Mining kann ACO eingesetzt werden [1, 2].

3.3 Genetischer Algorithmus

Der genetische Algorithmus (kurz GA) ist genau wie die ACO ein Optimierungsalgorithmus. Auch der genetische Algorithmus basiert auf Beobachtungen der echten Welt. In diesem Fall auf den evolutionären Entdeckungen von Charles Darwin und den genetischen Entdeckungen von Gregor Mendel. Auch hier wird ein Graph, oder Spannwald, $G = (V; E)$ definiert. Statt τ für die Pheromonwerte werden Kosten c zur Gewichtung von Kanten eingesetzt. Die Ameisen n_a fallen hier auch weg und werden durch eine Bevölkerung P ersetzt. Die Bevölkerung hat eine DNA, welche eine Route und somit eine mögliche Lösung repräsentiert. In der ersten Iteration des GA bekommt jedes S_i ($S_i \in P$) eine zufällige Route zugewiesen. In der Iteration, für diesen Algorithmus, auch Generation genannt, reproduzieren sich die S_i. Hierfür finden sich zwei S_i und geben jeweils einen Teil ihrer DNA bzw. Route an ihr Kind S_i' weiter. Da in der echten Welt sich nur die Lebewesen fortpflanzen, die Fit genug sind um bis zu einem zeugungsfähigen Alter zu überleben, wird dies in dem Algorithmus simuliert. In Beziehung auf den TSP werden die Routen

bevorzugt, die geringere Kosten haben. Für diese Berechnung gibt es auch verschiedene Varianten, die denen aus der ACO stark ähneln. Bei der Reproduktion selbst wird die DNA der Eltern in kleiner Teile aufgeteilt und zu einem vollständigen und validen 1-Tree wieder zusammengefügt (siehe Fig. 7).

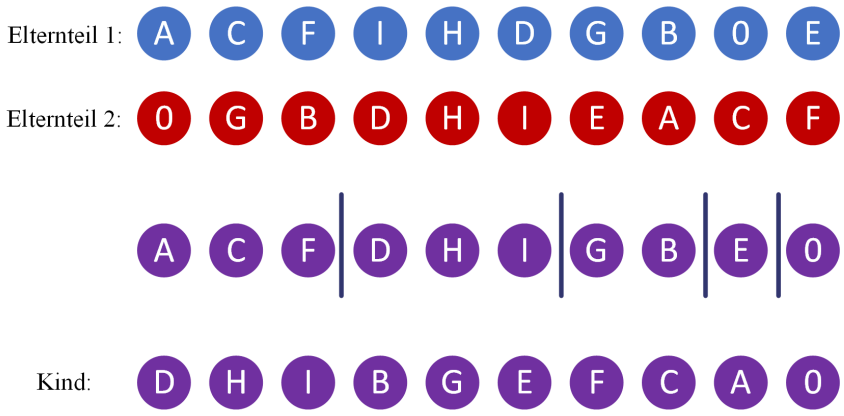

Abbildung 7: Fortpflanzung im GA

Die DNA der Eltern werden nur in die Teile aufgeteilt, die in der gleichen Reihenfolge bei beiden vorkommen. Die Teile werden dann in einer anderen Reihenfolge wieder zusammengefügt. Diese Reproduktion ist speziell für das TSP Problem. Bei anderen Problemen läuft die Reproduktion anders ab. Um die Wahrscheinlichkeit auf eine zu frühe Optimierung zu einem lokalen Minimum zu verringern, wird das auch in der Realität vorkommende Phänomen der Mutation genutzt. In Beziehung auf das TSP Problem wird ein Knoten in einem Kind mit einem anderen Knoten ausgetauscht. Standardmäßig wird die Wahrscheinlichkeit dafür als Parameter p des Algorithmus festgelegt.

Es gibt auch Vorschläge, wie in [6], diesen Parameter über die Generationen anzupassen.

$$p = p \times (1 - GenNum \times \frac{0.01}{maxGen})$$

$MaxGen$ bezieht sich damit auf die Maximale Anzahl der Generationen bis der TSP Algorithmus stoppt. $GenNum$ auf die momentane Generationsnummer. Die Wahrscheinlichkeit einer Mutation wird folglich immer unwahrscheinlicher. Die Argumentation ist, dass Mutation sehr gut am Anfang des Algorithmus ist, um ein lokales Minimum zu vermeiden. Im späteren Verlauf des Algorithmus ist die Mutation nicht mehr so relevant, da der Algorithmus sich auf ein approximatives Ergebnis einpendelt. Diese Vorgehensweise ist besonders erfolgreich für dynamische TSP Probleme, bei welchen sich manche Knoten im Verlauf der Berechnung ändern.

Es gibt auch Hybrid-Algorithmen, welche die Vorteile einer ACO und eines GA miteinander verknüpfen. Der GA kann für alle NP-Complete Probleme angewandt werden und wird auch in diesem Projekt und damit im Intelligenten Paket Management System verwendet [2, 6].

4 Architektur

Die Architektur des IntPakMan Systems wird in den folgenden Unterkapiteln anhand des C4 Modells visualisiert.

4.1 System-Kontext-Diagramm

Zuerst soll die externe Umgebung des IntPakMan System im Vordergrund stehen, weshalb das System im ersten Schritt als Black Box betrachtet wird, deren Funktion erst einmal nicht von Interesse ist. Deshalb liegt der Fokus bei der folgenden Abbildung auf der Interaktion mit externen Systemen und den Nutzer*innen.

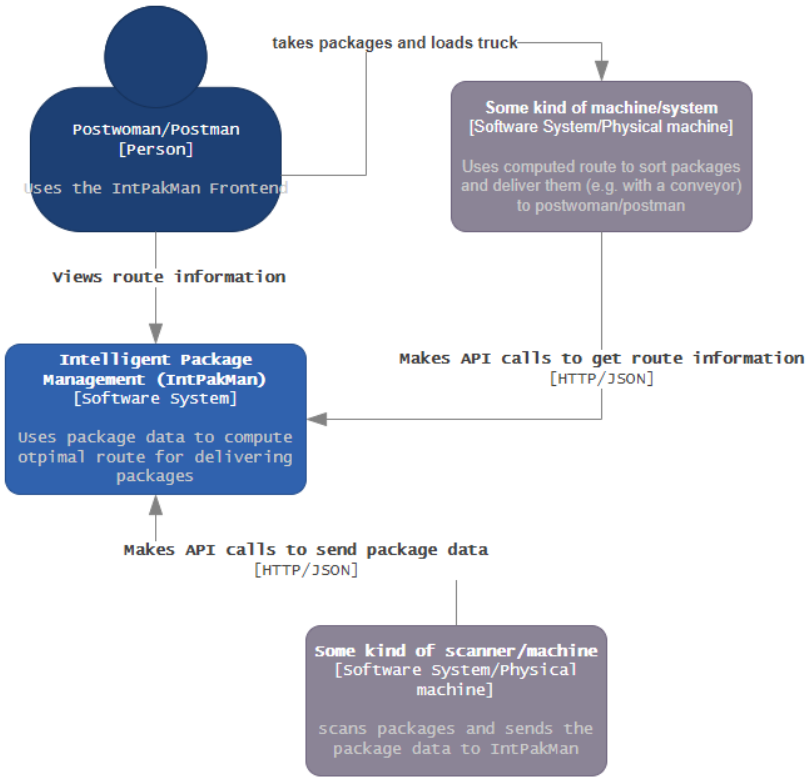

Abbildung 8: System Kontext Diagramm

Wie in der obigen Abbildung zu erkennen ist, gibt es zwei externe Systeme und eine Person als Nutzer*in, die mit dem System IntPakMan interagieren. Die Nutzer*innen sind in diesem Fall Postler*innen, könnten aber genereller gedacht auch Personen sein, die irgendeine Art von Ware nach einer bestimmten Route ausliefern. Diese Personen können sich mit Hilfe des in IntPakMan zur Verfügung gestellten Frontends Routeninformationen anzeigen lassen.

Eines der externen Systeme scant die Pakete ein und schickt die Paketdaten an eine API, die vom IntPakMan System zur Verfügung gestellt wird. Die Daten werden anschließend innerhalb des Systems weiter verarbeitet (Im nächsten Abschnitt weitere Details zur

Verarbeitung). Das weitere externe System kann die Routeninformationen, die in einer Datenbank gespeichert werden, abrufen und diese verwenden um die Pakete in der richtigen Reihenfolge zu sortieren. Die sortierten Pakete könnten dann beispielsweise mittels eines Förderbands zur für diese Pakete zuständigen Person transportieren werden, die dann das Postauto (oder sonstiges Auslieferfahrzeug) belädt.

4.2 System-Container-Diagramm

Um nun die Funktionsweise des IntPakMan Systems genauer zu verstehen, wird im folgenden das Container-Diagramm abgebildet.

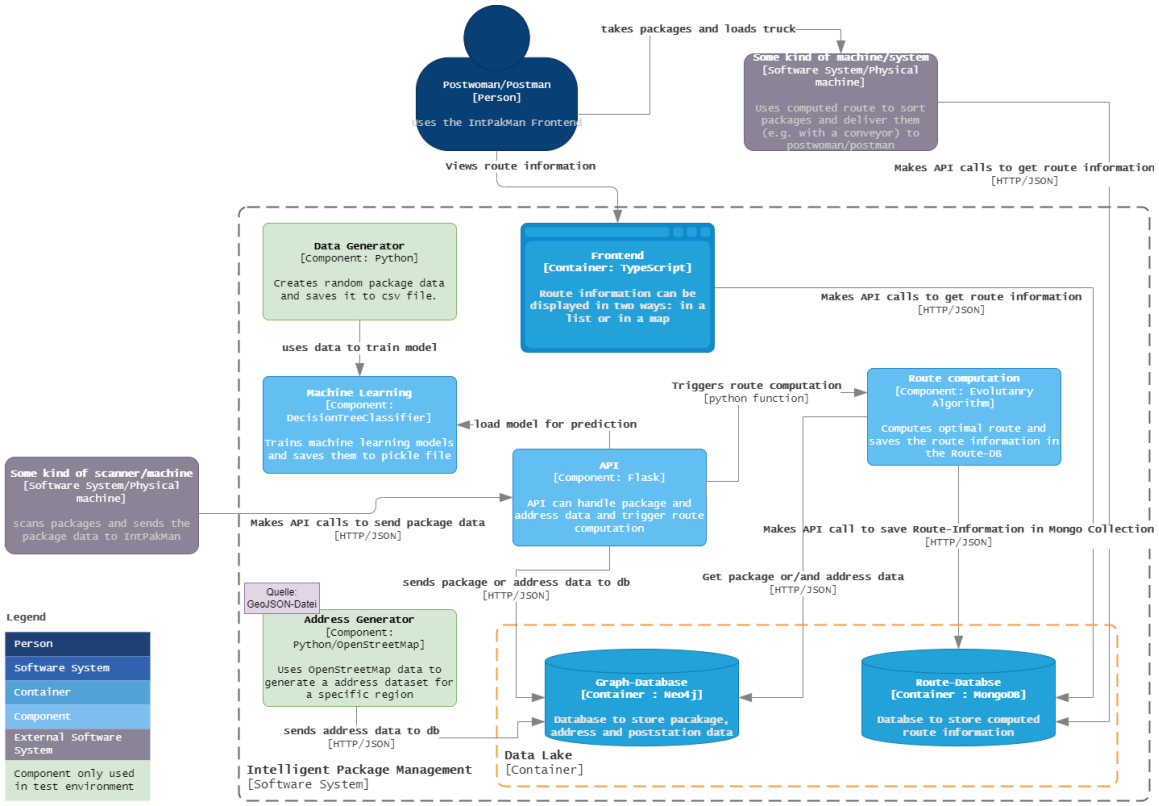

Abbildung 9: System Container Diagramm

Logischerweise sind die externen Systeme und Personen unverändert geblieben. Die einzelnen Container des IntPakMan Systems könnte man grundlegend in vier Rubriken aufteilen. Diese wäre zum einen das Frontend, die Business Logik, den Data Lake sowie Container, die lediglich in der Testumgebung benötigt werden.

Das Frontend wurde mit TypeScript umgesetzt und bietet die Möglichkeit, die Routeninformationen in einer Liste anzeigen zu lassen. Hierfür müssen Werte in die Felder „Zustellstation“, „Bezirk“und „Datum“eingegeben werden und dann wird im Data Lake, beziehungsweise genauer in der MongoDB, nach Routeninformationen gesucht, die auf diese Parameter passen.

Das Data Lake besteht aus zwei unterschiedlichen Datenbanken, zum einen der bereits erwähnten MongoDB und zum anderen aus der Graphdatenbank Neo4j. In der MongoDB werden die bereits berechneten Routeninformationen abgespeichert. Daten, die zur Berechnung der Route benötigt werden, werden in der Graphdatenbank Neo4j abgespeichert. Dies sind Daten zu den Paketen, den Adressen sowie der Zustellstation. Weitere Informationen zum Data Lake folgen in Kapitel 6.

Die Business-Logik findet in den restlichen drei Containern statt. Für die Kommunikation mit den externen Systemen ist die API verantwortlich. Ausführliche Informationen zur Implementierung der APIs finden Sie in Kapitel 6.2.2. Grundsätzlich bietet die API Schnittstellen um Paket- und Adressdaten an das IntPakMan-System senden zu können sowie eine Schnittstelle, um die Berechnung der Route für einen Bezirk steuern zu können.

Als letzte Rubrik kann man die Container ausmachen, die lediglich in der Testumgebung vorgesehen sind. Diese sind Datengeneratoren, die zum einen Paketdaten und zum anderen Adressdaten generieren, die zu Testzwecken verwendet werden können. Beide Datengeneratoren werden in Kapitel 5 genauer erläutert.

Ein typischer Prozessablauf würde also so aussehen, dass die Adressdaten bereits in der Graphdatenbank vorhanden sind und lediglich wenn notwendig (z. B. Neubau eines Hauses o. ä.) aktualisiert werden. Ein externes System würde viele Pakete einscannen und die Daten mittels der API an das IntPakMan-System senden. Dieses lädt das ML-Modell und Klassifiziert die Paketdaten und speichert diese anschließend in der Graphdatenbank. Sind alle Pakete für einen entsprechenden Bezirk eingescannt wird ebenfalls über die API die Routenberechnung gestartet und das Ergebnis dieser Berechnung in die MongoDB Datenbank gespeichert. Diese Daten können dann entweder von einem externen System, welches die Pakete sortiert, per API abgefragt werden oder die Informationen zur Route werden von einer Person mit Hilfe des Frontends eingesehen.

5 Datenquellen

Wie bereits deutlich wurde, sind Daten aus verschiedenen Quellen für die Umsetzung des Projekts unabdingbar. Da es uns nur in Teilen möglich war, an echte Daten heranzukommen, arbeiten wir im Rahmen dieses Projektes sowohl mit echten als auch mit synthetisch erzeugten Daten. Daten, die im Rahmen dieses Projekts betrachtet wurden, sind die Paketdaten und die Adressdaten.

Grundsätzlich ist es auch sehr gut denkbar, weitere Daten mit einzubeziehen und das System zu erweitern. Dies könnten beispielsweise Daten zu den Postautos, den Kunden oder Daten von der Zustellung sein. In den folgenden Unterkapiteln werden die im Rahmen dieses Projektes verwendeten Daten zu den Paketen und den Adressen erläutert.

5.1 Paketdaten

Es war leider nicht möglich, an echte Paketdaten von zum Beispiel der Deutschen Post/DHL oder Hermes zu kommen. Im Gegenteil, unsere Anfragen blieben leider unbeantwortet. Daher mussten die Daten zu den Paketen synthetisch erzeugt werden. Auf Grundlage von Domänen-Wissen besitzen Pakete in diesem Projekt folgende Attribute:

- *Sendungsnummer* (eine eindeutige ID)
- *length_cm* (Länge in Zentimeter)
- *width_cm* (Breite in Zentimer)
- *height_cm* (Höhe in Zentimeter)
- *weight_in_g* (Gewicht in Gramm)
- *fragile* (Zerbrechlichkeit)
- *perishable* (Verderblichkeit)
- *house_number* (Hausnummer)
- *street* (Straße)
- *post_code* (Postleitzahl)
- *city* (Stadt)
- *date* (Datum)

Wie in der Auflistung zu erkennen, sind typische Attribute eines Pakets vorhanden. Dazu zählt zum eine die Abmessung des Pakets mit den Attributen zur Länge, Breite und Höhe sowie das Gewicht. Weiterhin wurden domänenspezifische Attribute wie die Zerbrechlichkeit und die Verderblichkeit hinzugefügt. Aber auch Attribute zur Adresse sowie die Sendungsnummer zur eindeutigen Identifizierung sind vorhanden.

All diese Attribute wurden in dem programmierten Datengenerator berücksichtigt. Die durch den Datengenerator erstellten Paketdaten werden in einer CSV-Datei gespeichert.

Die Schwierigkeit bei der Programmierung des Datengenerators bestand darin, eine sinnvolle Gewichtung der Werte hinzubekommen. Der Hintergrund ist, dass ein*e Postler*in am Tag mehr kleine als große Pakete und auch deutlich mehr leichte als schwere Pakte ausliefert und dieses Phänomen in den Daten berücksichtigt werden musste. Gelöst wurde das Problem, indem Wertebereiche für *length_cm*, *width_cm*, *height_cm* und *weight_in_g* erstellt wurden. Folgender Code zeigt beispielhaft den Wertebereich für das Attribut *weight_in_g*. Die Wertebereiche der Attribute *length_cm*, *width_cm* und *height_cm* sind nach demselben Prinzip aufgebaut.

```
weight_low = np.arange(50, 2500, 10, int)
weight_middle = np.arange(2500, 5000, 10, int)
weight_high = np.arange(5000, 20000, 10, int)
weight_extremely_high = np.arange(20000, 31500, 10, int)
```

Bei der Ermittlung des konkreten Werts spielt die Funktion *random.choice*() aus dem numpy package eine wichtige Rolle. Diese wird in zwei verschiedenen Szenarien verwendet.

Zum einen um einen der Wertebereiche zu bestimmten und zum anderen um aus dem Wertebereich einen konkreten Wert zu ermitteln.

Die *choice*()-Methode ist deshalb gut geeignet, da man in dieser Methode pro angegebenem Wert über den Parameter p die Wahrscheinlichkeit definieren kann, dass dieser Wert gezogen wird.

```
x = np.random.choice([1, 2, 3, 4], 1, p=[0.6, 0.2, 0.16, 0.04])
```

In diesem Beispiel werden vier Werte $(1 - 4)$ angegeben, hierbei steht die 1 für den Wertebereich *weight_low*, die 2 für *weight_middle* usw. Über den zweiten Parameter kann angegeben werden, wie viele Ergebnisse es geben soll, in diesem Fall eins. Der dritte Parameter ist der bereits beschriebene Parameter für die Wahrscheinlichkeitsverteilung, dieser ist optional. In diesem Beispiel wäre also die Wahrscheinlichkeit, dass die 1 als Ergebnis zurück gegeben wird bei 60%. Somit kann man über die Wahrscheinlichkeit eine Gewichtung der Werte erzielen. Die oben abgebildete Zeile Code ist Teil des im folgenden abgebildeten Codeabschnitts, der beispielhaft die Methode für die Auswahl eines konkreten Wertes aus einem Wertebereich zeigt.

```
def get_weight():
    """
    First chooses between a value (1, 2, 3 or 4) for x, the value of x
                                        is responsible for choosing
                                        between the
    different range of values of weight.
    :return: return the weight
    """
    weight = 0
    x = np.random.choice([1, 2, 3, 4], 1, p=[0.6, 0.2, 0.16, 0.04])
    if x == 1:
        weight = np.random.choice(weight_low, 1)[0]
    elif x == 2:
        weight = np.random.choice(weight_middle, 1)[0]
    elif x == 3:
        weight = np.random.choice(weight_high, 1)[0]
    elif x == 4:
        weight = np.random.choice(weight_extremely_high, 1)[0]
    return weight
```

Der Variablen x wird hierbei zufällig (wie bereits beschrieben) ein Wert zwischen eins und vier zugeteilt. Anschließend wird überprüft, welcher Wert zugewiesen wurde und dementsprechend erneut zufällig aus dem entsprechenden Wertebereich ein Wert ausgewählt und zurück gegeben. Im folgenden Plot kann man erkennen, dass dadurch eine realitätsnahe Gewichtung der Werte möglich ist.

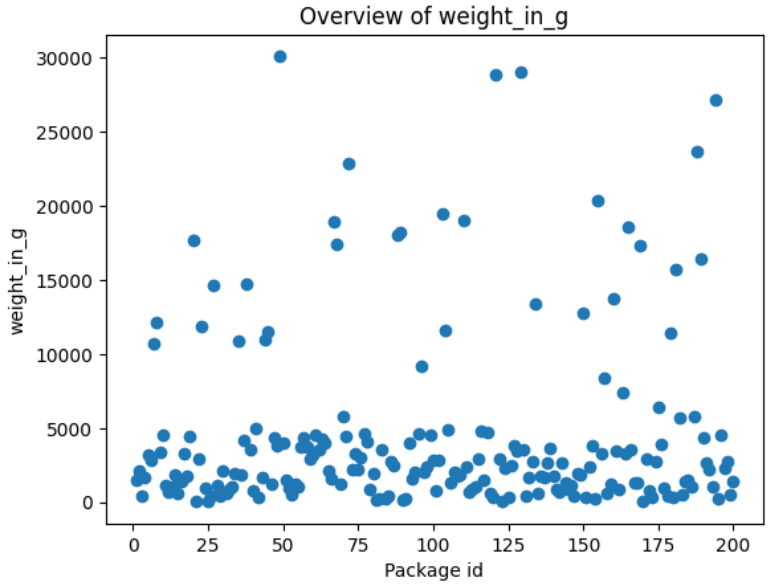

Abbildung 10: Plot zur Gewichtung des Attributs *weight_in_g*

Diese Logik wird auch auf die Ermittlung der Werte der Attribute *length_cm*, *width_cm* und *height_cm* angewendet. Für die Attribute fragile und perishable wird eine etwas andere Logik verwendet, da diese lediglich die Werte 0 oder 1 annehmen können wird folgende Codezeile verwendet:

```
np.random.choice([0, 1], 1, p=[0.85, 0.15])[0]
```

Die bisher genannten Attribute werden ergänzt durch die *Sendungsnummer* (Integer der hochgezählt wird) sowie den Adressdaten, die aus einer CSV-Datei eingeladen werden und aus denen zufällig ein Eintrag entnommen wird. Hierbei wurde auch berücksichtigt, dass eine Adresse mehrere Pakete am Tag erhalten kann.

5.2 Adressdaten

Die Adressdaten wurden von OpenStreetMap exportiert. Die OpenStreetMap-Daten haben alle Straßen, Adressen und andere Objekte in Furtwangen beinhaltet. Aus den Daten von OpenStreetMap wurden die GeoJSON-Objekte heraus gefiltert, die Adressen beinhalten. Die ausgelesenen Adressen wurden in eine Liste abgespeichert. Zu den Attributen, *house_number*, *street*, *post_code*, *city* und *geojson_geometry*, dem wichtigsten Teil des GeoJSON-Objekts aus den OpenStreetMap-Daten, wurden außerdem die Attribute *post_station_id*, dem wir für Furtwangen den Wert 1 gegeben haben, ein vorläufiges *district* Attribut und eine iteratives *Id* Attribut hinzugefügt. Es wurde darauf geachtet, dass nur einzigartige Adressen in die Liste gespeichert werden. Die Adressen wurden nun anhand folgender Koordinaten in 4 Distrikte unterteilt:

```
MAX_LO = 8.2302962
MAX_LA = 48.066499
MID_LO = 8.2033782
MID_LA = 48.056221
MIN_LO = 8.1764602
MIN_LA = 48.041943
```

LO steht für Längengrad und *LA* steht für Breitengrad. Da manche Adressen, keine einzelne Koordinate haben sondern in OpenStreetMaps durch ein Multipoligon repräsentiert sind, musste noch der Mittelpunkt für diese bestimmt werden. Dies wurde anhand der Formel:

$$center_{lo} = \frac{\sum_{\{\forall g_i \in lo\}} g_i}{i}, \quad center_{la} = \frac{\sum_{\{\forall g_i \in la\}} g_i}{i}$$

berechnet. Die Punkte wurden dann einem Distrikt zugewiesen. Schließlich wurde noch ein zufälliger Distrikt erstellt, welcher sich zu Demonstrationszwecken eignet, da er größere Distanzen zwischen den Adressen hat, aber keine reelle Anwendung, der Paketverteilung, repräsentiert. Die Distrikte sind nicht alle gleich groß, um das Programm mit unterschiedlichen Rechenkapazitäten testen zu können. Die Distrikt-Größen sind folgendermaßen verteilt:

District 1:	47	3.1%
District 2:	50	3.3%
District 3:	832	54.85%
District 4:	588	38.76%
Random District:	46	3.03%

5.3 Integration weiterer Daten

Die Architektur des Systems ermöglicht es, relativ einfach weitere Daten hinzuzufügen. Denkbar wären beispielsweise Daten zu Postautos (bzw. im generelleren Daten zu Auslieferfahrzeugen). Mit diesen Daten könnte beispielsweise eine tagesaktuelle Zuweisung von Autos zu Bezirken stattfinden, basierend auf den jeweiligen Auslastungen und zu fahrenden Kilometern.

Des Weiteren könnte man auch Daten über Kunden sowie Daten aus der Zustellung verwenden, um die Routenberechnung weiter zu optimieren. Ein mögliches Szenario wäre, dass beispielsweise die Bewohner*innen einer Adresse von Montag bis Freitag erst ab 14 Uhr zu Hause sind. Dann könnte die Route (falls an dem Tag sinnvoll) so angepasst werden, dass die Auslieferung des Pakets Richtung Ende der Route erfolgt und die Bewohner*innen sehr wahrscheinlich zu Hause sind. Dies sind drei beispielhafte Datenquellen, wie das System erweitert werden könnte.

6 Data Warehouse

Als Data Warehouse wird die Graphdatenbank Neo4j verwendet. Mittels einer ETL-Pipeline sollen die Daten aus den jeweiligen Datenquellen extrahieret, transformiert und dann in die Datenbank geladen werden. Im Rahmen unseres Projekts arbeiten wir jedoch mit synthetischen Daten, die direkt in Neo4j persistiert werden können. Die persistierten Daten können dann direkt für unsere Machine Learning Algorithmen zur optimalen Berechnung der Route verwendet werden. Im Anschluss an die Routenberechnung werden die Informationen zur optimalen Route in einer MongoDB-NoSQL-Datenbank abgespeichert. Die optimale Gangfolge kann somit direkt aus der Datenbank abgerufen und im Frontend angezeigt werden. So kann der Nutzer jederzeit auf die optimale Route eines bestimmten Bezirks an einem bestimmten Datum zugreifen, ohne dass diese bei jedem Abruf neu berechnet werden muss.

6.1 Neo4j

Da es sich hierbei um eine Graphdatenbank handelt, sind die Daten in Form von Knoten organisiert, die über Kanten miteinander in Beziehung stehen. Damit die Knoten unterschieden werden können, werden diese beschriftet. So wird zwischen Adressknoten (blau), Paketknoten (orange) und dem Startpunkt der Route, dem Poststationknoten (rot), differenziert.

Die Knoten haben je nach Beschriftung auch andere Eigenschaften. Eine Adresse wird beispielsweise mit Attributen wie einer Id, Bezirk, Stadt, Straße, Hausnummer und über geojson-Informationen definiert. Bei den Paketdaten spielen die Maße, das Gewicht, die Addresse aber auch das Datum eine wichtige Rolle. Daneben sind die Beziehungen zwischen den eigentlichen Entitäten von zentraler Bedeutung, da im Zuge dessen die tatsächliche Route berechnet wird. In der folgenden Abbildung sind die Adressdaten innerhalb eines Bezirks abgebildet. Zwischen allen

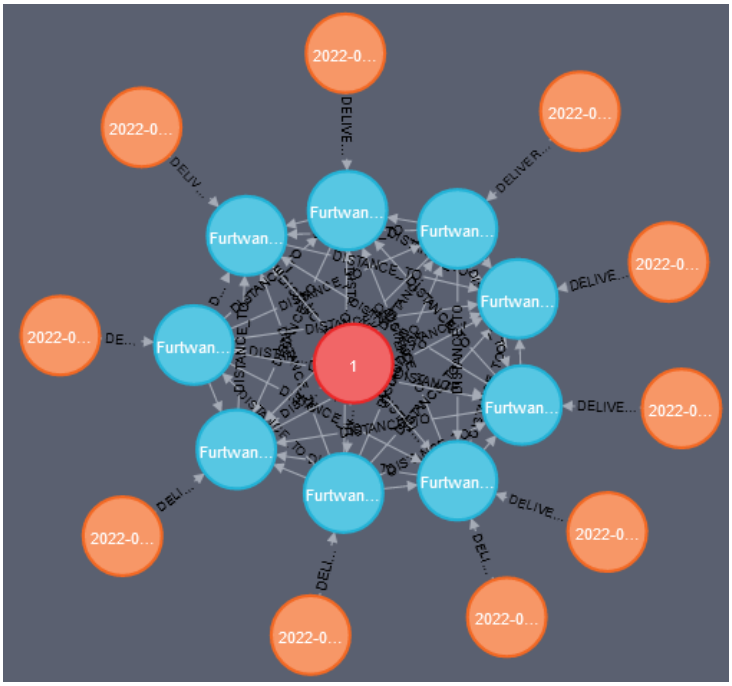

Abbildung 11: Strutkur des Neo4j-Graphen

Adressen besteht eine *DISTANCE_TO*-Beziehung, die wiederum mit Eigenschaften wie der Distanz in Metern und der Dauer in Sekunden definiert wird. Außerdem gibt es noch *DELIVERED_TO*-Beziehungen, um darzustellen, welche Pakete an welche Adresse geliefert werden muss.

6.2 APIs

6.2.1 Google Distance Matrix API

Wie bereits erwähnt werden die Distanzen zwischen allen Adressen eines Bezirks berechnet, dazu wird die Google-Distance-Matrix API verwendet. Damit stehen uns monatlich über 5.000 API-Anfragen zur Verfügung. Der Vorteil von Google zu vielen anderen Anbietern (beispielsweise OpenStreetMap API oder MapBox API) besteht darin, dass neben Koordinaten und vielen anderen Parametern auch einfach Adressen verwendet werden können, um die Distanz dazwischen zu berechnen. Außerdem gilt die API von Google als eine der effizientesten, da sie unter anderem nur sehr wenige Einschränkungen, beispielsweise hinsichtlich der Anzahl an Anfragen pro Minute, beinhaltet.

```
    @staticmethod
    def _add_distance(tx, results):
        distance_list = []
        for x in range(len(results)):
            entry = results[0]
            id_a = entry["n"]["id"]
            results.pop(0)
            distance_list = results
            for y in range(len(distance_list)):
                distance, duration = calculate_distance(entry["n"],
                                                        distance_list[y]["n"])
                id_b = distance_list[y]["n"]["id"]
                query = """MATCH(a:Address),(b:Address)
                           WHERE a.id = $id_a AND b.id = $id_b
                           CREATE (a)-[r:DISTANCE_TO {distance:
                           $distance, duration: $duration}]->(b)"""
                tx.run(query, id_a=id_a, id_b=id_b, distance=distance,
                                                    duration=duration)
```

Über ein Python Skript wird mithile eines API-Keys auf die Schnittstelle zugegriffen. Nach der Authentifizierung werden die Postleitzahl, die Straße, die Hausnummer und die Stadt der Start- und Zieladresse an die API über eine GET-Anfrage geschickt.Diese liefert uns Informationen über die Route dieser zwei Adressen zurück, darunter auch die Distanz und die Dauer, die für nachfolgende Anwendungsfälle gebraucht werden. Aufgrund dessen, dass nicht nur die Distanz zwischen zwei bestimmten Adressen bedeutend ist, sondern die Distanz zwischen allen Adressen innerhalb eines Bezirks, werden diese in Form einer Liste

gespeichert und in einer for-Schleife durchlaufen. Dabei wird die Distanz zwischen den, im Schleifendurchlauf betrachteten Adressen, berechnet und eine Datenbank-Anweisung ausgeführt, die die Daten in Neo4j speichert. Der Vorgang wird hierbei für alle Adressen wiederholt.

6.2.2 Flask API

Flask ist ein in Python geschriebenes WSGI Web-application-Framework. Es ermöglicht sehr schnell und relativ einfach Webapplikationen aufzubauen mit der Möglichkeit, sehr gut auf komplexere Anwendungsfälle skalieren zu können. Deshalb ist Flask aktuell eines der beliebtesten Python Web-application Frameworks.

Im Rahmen dieses Projektes wird Flask in einer sehr simplen Form genutzt, um drei verschiedene APIs zur Verfügung zu stellen, über die mit dem IntPakMan-System interagiert werden kann. Folgend werden die drei APIs kurz beschrieben:

Paketdaten-API:

- Diese Schnittstelle ermöglicht es, Paketdaten an das IntPakMan-System senden zu können.
- Die Paketdaten werden als JSON-Objekt beim Aufruf an die Schnittstelle übergeben.
- Die API prüft ob alle notwendigen Felder im JSON-Objekt enthalten sind und ist dies der Fall, wird ein Skript aufgerufen, das die Daten in die Graphdatenbank speichert. Dieses Skript verarbeitet die Daten zuerst insofern weiter, als dass ein Machine Learning Modell geladen wird und das Paket klassifiziert wird, ob es sich um ein zu priorisierendes Paket handelt oder nicht (Weitere Informationen hierzu in Kapitel 7). Anschließend werden die Daten dann in der Graphdatenbank persistent gespeichert.

Adressdaten-API:

- Diese Schnittstelle ermöglicht es, Adressdaten an das IntPakMan-System senden zu können.
- Die Adressdaten werden als JSON-Objekt beim Aufruf an die Schnittstelle übergeben.
- Die API prüft, ob alle notwendigen Felder im JSON-Objekt enthalten sind und, ist dies der Fall, wird ein Skript aufgerufen, das die Daten in der Graphdatenbank persistent speichert.

Routenberechnungs-API:

- Diese Schnittstelle ermöglicht es, die Berechnung der Route für einen Bezirk zu starten. Hierfür sind verschiedene Parameter notwendig. Diese Parameter sind teilweise zwingend notwendig und teilweise optional.

- Notwendige Parameter:
  - „post_station_id“ : Die ID der Post-Station, in der sich der Bezirk befindet, für den die Route berechnet werden soll.
  - „district“ : Die ID/Nummer des Bezirks, für den die Route berechnet werden soll.
  - „date“ : Das Datum im Format YYYY-MM-DD, für das die Route berechnet werden soll.
- Optionale Parameter:
  Alle optionalen Parameter können mittels einer booleschen Variablen gesteuert werden. Hierbei kann für „True“ entweder True, Yes, T oder 1 sowie alles auch in Kleinbuchstaben eingegeben werden. Alle anderen Eingaben werden als „False“ interpretiert.
  - „distance“ : Über diesen Parameter kann gesteuert werden, ob die Distanz oder die Dauer zur Ermittlung des kürzesten Wegs zwischen zwei Adressen im TSP-Algorithmus verwendet werden soll. Standardmäßig wird die Distanz verwendet (distance=True).
  - „prio“ : Dieser Parameter steuert, ob bei der Berechnung der optimalen Route die Priorisierung berücksichtigt wird oder nicht. Standardmäßig wird die Priorisierung berücksichtigt (prio=True).
  - „evaluate“ : Über diesen Parameter kann gesteuert werden, ob nach der Berechnung der Route, Kennzahlen über die Route berechnet und die Route somit evaluiert werden soll. Standardmäßig ist findet keine Evaluierung statt (evaluate=False).
  - „curve“ : Über diesen Parameter lässt sich steuern, ob nach der Berechnung der Route die sogenannte fitness_curve des TSP-Algorithmus in der Konsole ausgegeben werden soll (weitere Infos hierzu in Kapitel 8). Standardmäßig wird diese fitness_curve nicht ausgegeben (curve=False).
- Mit diesen Parameter wird anschließend ein Python Skript gestartet, das die Routenberechnung durchführt und die Ergebnisse in die MongoDB-Datenbank speichert.

6.3 MongoDB

Die berechneten Routen werden in einer MongoDB Cloud-Datenbank abgespeichert, auf die das User-Interface mit einem Verbindungsstring zugreifen kann. Die Routen liegen alle in derselben Sammlung in Form von JSON-Dokumenten vor. Die Dokumente enthalten zum einen Metadaten, wie das Datum der erstellten Route, dem Bezirk und der Zustellstation. Zum anderen enthält das Dokument eine Gangfolge von Adressen, die in einem Array gespeichert sind, die angibt in welcher Reihenfolge die Route abgefahren werden soll.

```
_id: ObjectId("61d2d0d28ac648ad61edfbad")
post_station: 1
district: 1
date: "2022-01-03"
route_data: Array
  0: Object
      house_number: "1"
      id: "1"
      city: "Furtwangen im Schwarzwald"
      street: "Robert-Gerwig-Platz"
      geojson_geometry: "{'type':'Point', 'coordinates':[48.05104904201273, 8.208381086475308]}"
      post_code: "78120"
  1: Object
  2: Object
  3: Object
  4: Object
```


Abbildung 12: Berechnete Route als JSON Dokument

7 Machine Learning

Eines der Kernelemente des System ist die Integration von Machine Learning in die Gesamtarchitektur. Wie bereits erläutert, findet vor jedem Hinzufügen eines Pakets in die Datenbank eine Vorhersage statt, ob es sich um ein Paket handelt, das priorisiert werden soll, oder nicht. Dies funktioniert so, dass in dem Skript, das die Paketdaten in die Datenbank speichert, ein Pfad zum ML-Modell angegeben ist. Somit entsteht eine sehr lose Kopplung zwischen System und ML-Modell, was ermöglicht, dass bei Bedarf sehr leicht ein neues ML-Modell verwendet werden kann.

Wie das in diesem Projekt verwendete Modell erstellt und worauf hierbei geachtet wurde, wird in den folgenden Abschnitten erläutert.

7.1 Architektur

In diesem Fall wurde eine klassische Machine Learning Architektur verwendet. Die Daten werden zuerst aus einer CSV-Datei geladen (generiert durch den Datengenerator). Anschließend findet die Datenvorverarbeitung statt. Sind die Daten soweit vorbereitet, werden sie in Test- und Trainingsdaten getrennt und das Modell trainiert. Das Modell wird daraufhin getestet und evaluiert und es besteht die Option, das Modell als Pickle-Datei abzuspeichern.

7.2 Modell

Das Modell soll in diesem einfachen Anwendungsfall zwei Klassen vorhersagen können, „Prio“ und „NoPrio“. Die Klasse „Prio“ steht für alle priorisierten Pakete und die Klasse „NoPrio“ steht für alle Pakete, die nicht priorisiert werden.

Wie eventuell schon deutlich wurde, handelt es sich in unserem Fall um ein Problem, das am besten mit einer supervised-learning Methode angegangen wird. Da zwei Klassen

vorhergesagt werden sollen, ist es sinnvoll ein Klassifizierungsmodell zu verwenden.

Da die Komplexität der Vorhersage in diesem Fall nicht wirklich hoch ist, wird im Rahmen dieses Projekts mit einem einfachen Entscheidungsbaum gearbeitet. Konkret wird mit dem Decision Tree Classifier aus dem scikit-learn Python Paket gearbeitet (sklearn.tree.DecisionTreeClassifier). Dieser Entscheidungsbaum wird in der Standardkonfiguration genutzt. Sollte durch verwenden von realen Daten die Komplexität der Daten/Vorhersage zunehmen und ein einfaches Modell wie der Decision Tree Classifier aus dem sklearn Paket keine guten Ergebnisse mehr erzielen, wäre es wie bereits beschrieben einfach umsetzbar, ein anderes Modell, wie beispielsweise den XGBoost-Classifier, zu verwenden.

7.3 Datenvorverarbeitung

Die Datenvorbereitung ist in diesem theoretischen Fall nicht sonderlich aufwendig. Der Grund hierfür ist, dass die Daten synthetisch erstellt wurden und keine Fehler enthalten. Dadurch müssen die Daten nur insofern vorbereitet werden, als dass ein Label, nachdem klassifiziert werden soll, hinzugefügt werden muss. Außerdem wird aus den Attributen Länge, Breite und Höhe ein weiteres, neues Attribut Volumen berechnet.

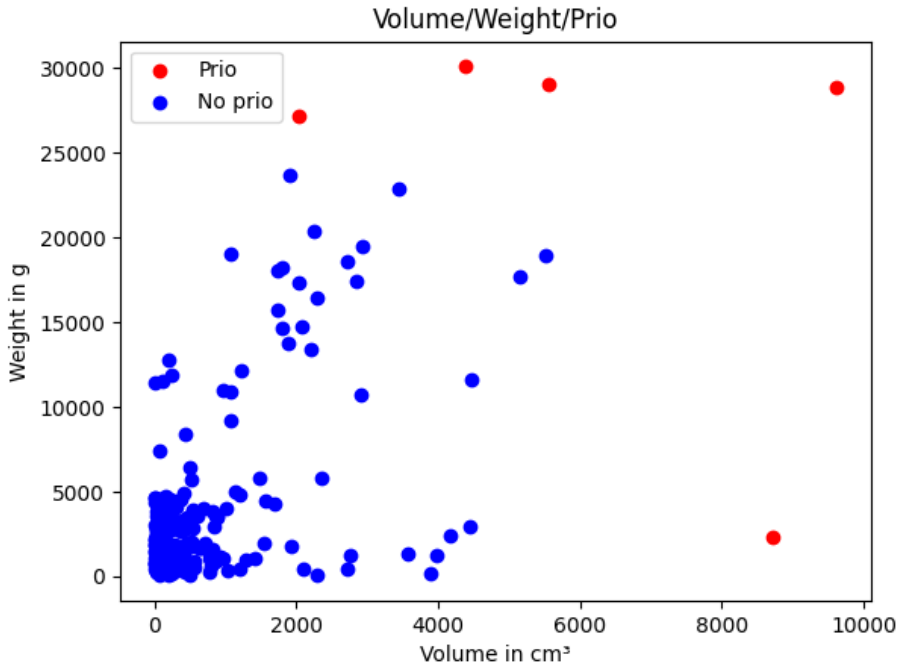

Abbildung 13: Plot zur Datenvorbereitung

Für die Erstellung des Prio-Lalebs werden klare Regeln verwendet. So erhält jedes Paket, dessen Volumen größer $8000cm^3$ oder dessen Gewicht größer $25KG$ ist das Label 1 (= Prio). Die Pakete, auf die das nicht zutrifft, erhalten das Label 0 (= NoPrio). Nach

der Datenvorbereitung lassen sich die Abhängigkeiten zwischen den Daten durch den in Fig. 13 visualisierten Plot darstellen. Die Datenvorbereitung wurde im Machine Learning Skript in eine eigene Methode geschrieben und kann somit sehr leicht auf sich ändernde Daten angepasst werden.

7.4 Label/Feature

Zum Trainieren des Modells wird in diesem Fall mit dem Label „Prio“ gearbeitet. Dieses kann wie bereits erläutert die Werte 0 oder 1 haben. Nach mehreren Tests hat sich herauskristallisiert, dass in diesem Fall für die Priorisierung von Paketen die Eigenschaften Gewicht und Volumen am besten geeignet sind. Daher werden alle Features, also Werte, aus denen die Zusammenhänge für die Vorhersage gelernt werden sollen, das Volumen und das Gewicht verwendet.

Die Daten werden in Test- und Trainingsdaten getrennt, wobei 80% der Daten zum Trainieren und 20% zum Testen verwendet werden. Der synthetisch erzeugte Datensatz zum Erstellen des Modells enthielt insgesamt Daten zu 20.000 Pakten, somit gab es 16.000 Paketdaten zum Trainieren und 4.000 Paketdaten zum Testen.

7.5 Evaluation des Modells

Zur Evaluation des Klassifikationsmodells werden die Metriken „Confusion Matrix“ und „ROC-Curve“ genutzt. Folgend ist die Confusion-Matrix als Tabelle abgebildet und die ROC-Kurve als Plot visualisiert:

	Prio	No Prio
Prio	85	0
No Prio	0	3915

Tabelle 1: Confusion-Matrix des Machine Learning Modells

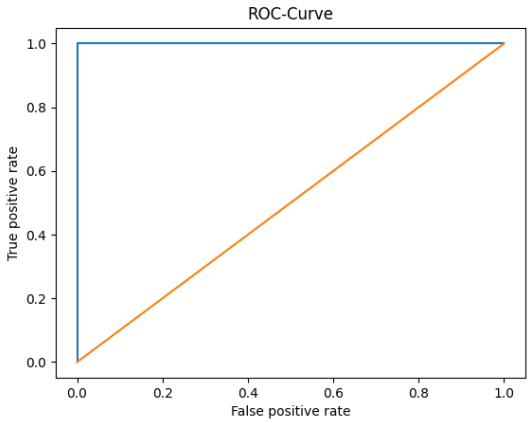

Abbildung 14: ROC-Kurve des Machine Learning Modells

Vor allem anhand der Confusion Matrix ist sehr gut zu erkennen, dass das Modell sehr gute Ergebnisse liefert. So wurden alle Werte korrekt vorhergesagt. Es wurde 85 mal „Prio“, somit ein „True Positiv“ Ergebnis und 3915 mal „No Prio“, ein „True Negativ“ Ergebnis vorhergesagt. Somit gab es keine „False Negative“ und keine „False Postive“ Vorhersagen. Natürlich muss hierzu auch gesagt werden, dass die Abhängigkeiten in den

Daten sehr gut zu Erlernen waren, denn wie bereits bei der Datenvorbereitung erklärt wurde, werden die Labels, nach denen gelernt wird, mit klaren Regeln erstellt und diese Regeln können vom ML-Modell sehr gut gelernt werden. Würde man anstelle von synthetischen reale Daten nehmen, könnte dies natürlich wieder ganz anders aussehen.

Auch die oben abgebildete ROC-Kurve (ROC: englisch für receiver operating characteristics) verdeutlicht, dass das ML-Modell eine sehr gute Performance besitzt. Man kann erkennen, dass die Trefferquote bei 100 Prozent (1) und die Fehlerquote bei 0 Prozent (0) liegen. Somit liegt in diesem Fall kein Zufallsprozess vor, dieser würde vorliegen, wenn die Werte sehr nahe an der Diagonalen liegen würden, sondern das Modell konnte die Zusammenhänge in den Daten korrekt lernen und kann korrekte Vorhersagen zur Priorisierung treffen.

7.6 Implementierung der Klassifizierung

Wie bereits einleitend für dieses Kapitel angesprochen, wurde bei der Implementierung der Klassifizierung in das IntPakMan-System darauf geachtet, dass die Machine Learning Komponente sehr lose an den Rest des Systems gekoppelt ist, sodass diese bei Bedarf sehr einfach austauschbar ist.

Grundsätzlich wird so vorgegangen, dass das ML-Modell mit einem Link zum Speicherort geladen wird. Das geladene Modell wird anschließend der Methode übergeben, die über die Paketdaten iteriert um diese in die Datenbank zu speichern. Wie im folgenden Code-Beispiel erkennbar ist, werden von den übergebenen Paketdaten die Attribute *length_cm*, *width_cm*, *height_cm* und *weight_in_g* ausgelesen und in einen pandas DataFrame geschrieben. Grund hierfür ist, dass in diesem Beispiel aus Testzwecken ein anderes Modell als das eben beschriebene verwendet wurde. In diesem Fall findet die Klassifizierung nicht anhand von Gewicht und Volumen statt, sondern mit den Eigenschaften Gewicht, Länge, Breite und Höhe.

```
def _create_package(tx, packages, model):
      for i in range(len(packages)):
          pred_data = pd.DataFrame(columns=["length_cm", "width_cm",
                                            "height_cm", "weight_in_g"])
          pred_data.loc[0] = [
              packages.loc[i, "length_cm"],
              packages.loc[i, "width_cm"],
              packages.loc[i, "height_cm"],
              packages.loc[i, "weight_in_g"]
          ]
          prio = str(model.predict(pred_data)[0])
```

Der nun erstellte DataFrame wird dem Modell zur Klassifizierung gegeben und das Ergebnis dieser Klassifizierung wird als neues Attribut *prio* gespeichert. Die restlichen

Attribute des Paketdatensaztes werden weiterhin auch ausgelesen und anschließend in der Neo4j Graphdatenbank persistiert. Der Travelling-Salesman-Problem-Algorithmus berechnet nun anhand der zuvor generierten Priorisierung die optimale Route.

8 Travelling Sales Person

8.1 Implementierung

Wie schon im Literaturreview erwähnt, steht das Travelling Sales Person (kurz TSP) Problem im Zentrum eines Intelligenten Paketmanagementsystems. Beim TSP werden klassisch die Koordinaten der Orte (Knoten) genommen, um daraus die Luftlinie (Kante) zu jedem anderen Ort zu berechnen. Da Postboten allerdings die Verkehrsregeln einhalten müssen, ist es besser, die zu fahrenden Meter oder die Fahrtdauer als Entfernung zwischen den zu beliefernden Häusern zu nehmen. Die Distanz zwischen jedem Haus wird mit der Hilfe einer API von Google berechnet (siehe Kapitel 6.2.1) und dann in einer Graphdatenbank abgespeichert (siehe 6.1).

Für eine einfache Umsetzung des TSP Problems wurde das Python Paket mlrose verwendet. Dies ist eine große Machine Learning Library. Um mlrose für das Intelligente Paket Management System zu Nutzen müssen allerdings Anpassungen gemacht werden, da zum Beispiel keine Priorisierung von Knoten im TSP Algorithmus von mlrose implementiert ist. Hierfür wurde ein neues package mltulip erstellt, welches zuerst eine Kopie von mlrose war.

Ein Problem bei mlrose ist, dass die *mate_probability* für unsere Anwendung schlecht berechnet wurde. Standardmäßig wird dafür in mlrose die Fitness, also die Kosten (Strecke) einer Route, durch die Summe der Fitness aller Routen in dieser Generation geteilt.

```
self.mate_probs =pop_fitness/np.sum(pop_fitness)
```

Da wir nach der kürzesten Route suchen, wird in mlrose der Fitness wert negativ beschrieben. Die Rechnung oben kommt somit zu einem suboptimalen Ergebnis. Der Code wurde in mltulip angepasst um diesem Problem aus dem Weg zu gehen.

```
sum_fitness = np.sum(pop_fitness)
pop_fitness = [0 if x == 0 else sum_fitness /x for x in pop_fitness]
self.mate_probs =pop_fitness / np.sum(pop_fitness)
```

Um zu verifizieren, dass das Ergebnis besser ist, wurden Test durchgeführt aus welchen sowohl eine klare Verbesserung des durchschnittlichen Fitnesswerts als auch eine Verbesserung in der Durchlaufzeit zu erkennen ist. Die Tests wurden anhand der Paketdaten vom 02.01.22 im Distrikt 2 mit 10 unterschiedlichen Seeds durchgeführt. Es wurde eine Bevölkerungsgröße von 300 gewählt, da sich in [3] diese Größe als optimal herausgestellt

hat. Für die Mutationswahrscheinlichkeit wurde für diesen Test $0,2$ gewählt. Die maximale Anzahl der Versuche wurde auf 200 gesetzt. Dieser Wert beschreibt nicht die maximale Anzahl der Iterationen bis der Algorithmus abbricht, sondern die maximale Anzahl der Iterationen, in welchen sich die beste Fitness nicht verbessert. Die Anzahl der Iterationen an sich wurden nicht limitiert. Die Ergebnisse des Tests sieht man als Durchschnitt der Ergebnisse in Tabelle 2. Die erste Änderung ist mit „mltulip v1“betitelt.

Um eine Priorisierung einzelner Knoten zu ermöglichen, muss es einen klaren Anfangsknoten geben. Denn die priorisierten Knoten sollen am Anfang der Route sein, wenn man die Paketstation als Startpunkt betrachtet. Deshalb wurde der Knoten 0, welcher die Paketstation repräsentiert, immer als Anfangspunkt definiert. Um dies zu realisieren wurde die Funktion, die die erste Bevölkerung zufällig generiert, verändert, sodass 0 nicht Teil der Generierung ist, sondern nachträglich in die Bevölkerungsmitglieder injiziert wird. Zusätzlich wurde die Reproduktion angepasst, sodass 0 zuerst aus beiden Eltern entfernt wurde und in das Kind direkt an erster bzw. nullter Stelle rein gesetzt wird. Auch die Mutation des Kindes wurde angepasst werden, sodass 0 nicht an eine andere Stelle mutiert wird. Diese Änderungen wurden auch getestet. Da die Bevölkerung nicht mehr komplett zufällig generiert wurde, ist der Algorithmus aufgrund dieser Änderungen schlechter geworden (siehe Tabelle 2, Zeile „mltulip v2“). Die Änderungen sind allerdings notwendig, deshalb muss die Performance Verschlechterung in Kauf genommen werden. Aus diesem Grund wäre eine ACO, welche oft einen festen Startpunkt besitzt, möglicherweise besser gewesen.

Hinsichtlich dieser Änderungen wurde auch eine andere Form der Reproduktion getestet. In mlrose besteht ein Kind aus der ersten Hälfte der Route des ersten Elternteils und den Knoten, die nicht in der ersten Hälfte vom ersten Elternteil sind. Diese zweite Hälfte ist in der gleichen Reihenfolge in der die Knoten auch im zweiten Elternteil vorkommen. Im Literaturreview wird allerdings ein anderer Reproduktionsprozess beschrieben. Dieser neue Reproduktionsverlauf wurde auch in mltulip getestet, hatte jedoch schlechtere Testergebnisse als mit der Reproduktion mit mltulip v2 (siehe Tabelle 2, Zeile„mltulip v3“). Deshalb wird im Intelligenten Paketmanagementsystem der Reproduktionsprozess nach mlrose verwendet.

Algorithmus	Insgesamte Distanz (m)	Durchlaufzeit (s)	Durchlaufzeit/Iteration (s)
mlrose	$14830,0$	$167,87$	$0,36$
mltulip v1	$14343,7$	$158,98$	$0,38$
mltulip v2	$14414,0$	$194,24$	$0,52$
mltulip v3	$14594,6$	$145,00$	$0,42$
mltulip v4	$10764,1$	$240,82$	$0,41$

Tabelle 2: Testergebnis des genetischen Algorithmus

Da auch eine spezielle Berechnung für die Mutationswahrscheinlichkeit im Literaturreview erwähnt wurde, wurde auch überlegt, diese in das Intelligente Paketmanagement System zu implementieren. Allerdings ist mlrose nicht auf eine maximale Anzahl an Iterationen begrenzt, weshalb die Umsetzung dieser Änderung unmöglich ist. Die Mutationswahrscheinlichkeit wurde jedoch auf $0,02$, wie in [6] angepasst. Diese Änderung hat wesentlich bessere Ergebnisse ergeben, auch wenn die Durchlaufzeit deutlich schlechter geworden ist (siehe Tabelle 2, Zeile „mltulip v4“).

Im letzten und wahrscheinlich wichtigsten Schritt wurde die Priorisierung hinzugefügt. Hierfür wurde mltulip um den Parameter *weights* erweitert. *weights* kann entweder 1 sein (priorisiert) oder 0 (nicht priorisiert). Dies beschreibt die Gewichtung jedes Knotens. In mlrose wird die Fitness für jede Kante nach diesem Code Schema berechnet:

```
path = (min(state[i], state[i + 1]), max(state[i], state[i + 1]))
fitness += self.dist_list[self.path_list.index(path)]
```

Diese Formel wurde erweitert, sodass die Position des Knotens im Verhältnis zum Knoten 0 auch eine Rolle spielt. Eine gute Formel dafür zu finden war schwierig. Würde man einen Gewichtungswert einfach mit der *dist_list* verrechnen, also subtrahieren, würde man die priorisierten Knoten immer mit einem besseren Fitnesswert belegen. Da priorisierte Knoten aber nur mit einem besseren Wert besetzt werden sollen, wenn sie früher in der Route sind, ist die Rechnung nicht ausreichend. Deshalb wird das Gewicht mit der Länge der Route durch die Distanz des Knotens zu 0 multipliziert. Aber auch diese Formel gibt noch kein zufriedenstellendes Ergebnis, da sich die Fitnesswerte mit unterschiedlichen Distanzen der gesamten Route stark ändern können. So würde eine Gewichtung bei einer durchschnittlichen Route von $100m$ mehr ins Gewicht fallen als bei einer Route von $100.000m$. Deshalb wird das Gewicht zusätzlich mit der durchschnittlichen Kantenlänge multipliziert. Die durchschnittliche Distanz ist allerdings viel zu hoch, sodass das Gewicht zu stark gewichtet werden würde. Deshalb wird es mit einem heuristischen Wert, der *prio_importance*, klein gehalten. Je größer die *prio_importance* desto weniger wird die Priorität gewertet. Der Standardwert für die *prio_importance* ist 25. Das Produkt wird dann von der $fitness$ abgezogen, da eine geringere Fitness im Algorithmus besser ist. Die Operratorreihenfolge wurde entsprechend folgender Formel für ein perfektes Ergebnis angepasst.

```
path = (min(state[i], state[i + 1]), max(state[i], state[i + 1]))
fitness += self.dist_list[self.path_list.index(path)]
fitness -= ((self.weights[state[i]]  *
            (sum(self.dist_list)/
            (len(self.dist_list)* self.prio_importance))*
            (len(state) - 1)) / (i + 1))
```

Man kann in den Fig. 15 und 16 die Fitnesskurve von mltulipv4 mit Priorisierung und ohne Priorisierung, anhand vier verschiedenen Seeds, sehen. Anhand der Graphen ist zu erkennen, dass der Algorithmus deutlich schneller zu einem Ergebnis kommt, wenn die Priorität berücksichtigt wird. Hier ist es wichtig zu erwähnen das die Fitness nicht mehr der Distanz entspricht. Im nächsten Abschnitt wird die Effektivität dieser Änderungen weiter bewertet.

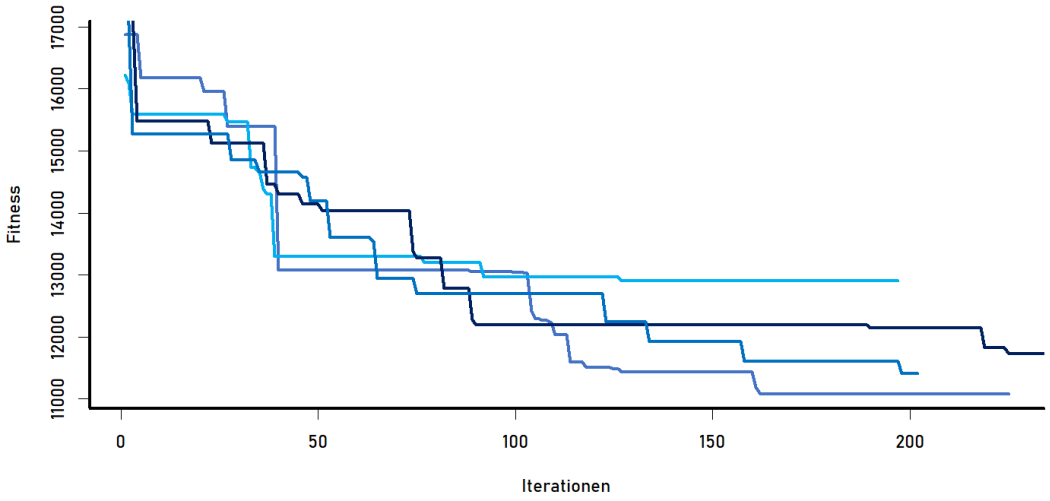

Abbildung 15: Fitnesskurve mit Priorität

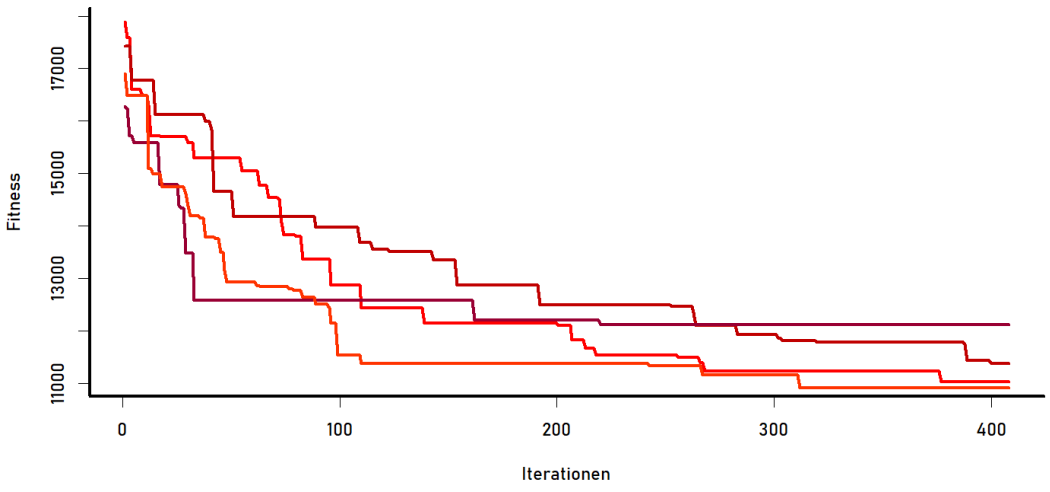

Abbildung 16: Fitnesskurve ohne Priorität

8.2 Evaluation der Implementierung

Um den TSP-Algorithmus evaluieren zu können wurden für 5 verschiedene Tage und für 2 Bezirke Paketdaten in die Datenbank gespeichert, sodass insgesamt 10 Paketdatensätze für die Evaluation genutzt werden konnten. Es soll untersucht werden, inwieweit Unterschiede in der insgesamten Dauer und der insgesamten Distanz der berechneten Route erkannt werden können, wenn die Routenberechnung mit unterschiedlichen Parameter für „prio“ und „distance“ durchgeführt wird. Konkret wurde die Routenberechnung mit folgenden Kombinationen für die Parameter „prio“ und „distance“ durchgeführt:

	prio	distance
1	True	True
2	False	True
3	True	False
4	False	False

Tabelle 3: Konfiguration der Parameter zur Evaluation des TSP-Algorithmus 1

	prio	distance
1	Priorisierung	Distanz
2	Keine Priorisierung	Distanz
3	Priorisierung	Dauer
4	Keine Priorisierung	Dauer

Tabelle 4: Konfiguration der Parameter zur Evaluation des TSP-Algorithmus 2

Die beiden oberen Tabellen zeigen den selben Inhalt, nur anders ausgedrückt. Die linke Tabelle zeigt den Inhalt programmatischer, die rechte Tabelle beschreibt die Kombinationen sprachlich.

Wie den oberen Tabellen zu entnehmen ist, wird die Berechnung der Route mit Parametern aus der Kombination Prio/Keine Prio und Dauer/Distanz durchgeführt. In der folgenden Tabelle wird aus der Berechnung der Route der 10 Paketdatensätze für die eben beschriebenen Kombinationen der Parameter, der Durchschnitt der Ergebnisse für die Werte insgesamte Distanz und insgesamte Dauer abgebildet:

	prio	distance		Insgesamte Distanz (m)	Insgesamte Dauer (Minuten)
1	True	True		$13740,7$	$31,76$
2	False	True		$13277,5$	$29,92$
3	True	False		$14338,9$	$29,75$
4	False	False		$14065,5$	$29,97$

Tabelle 5: Ergebnisse der Evaluation des TSP-Algorithmus

Diese Ergebnisse zeigen sehr deutlich den Unterschied, ob die Distanz oder die Dauer zur Ermittlung des kürzesten Weges zwischen zwei Adressen verwendet wird. Werden Zeile 1 und 2 (Distanz) im Vergleich zu Zeile 3 und 4 (Dauer) betrachtet, so ist zu erkennen, dass in Zeile 1 und 2 die insgesamte Distanz geringer ist als in Zeile 3 und 4. Dies war auch so zu erwarten, da in diesen Fällen die Distanz zur Ermittlung des kürzesten Wegs verwendet wurde. Ebenso ist aber auch in Zeile drei die durchschnittlich geringste insgesamte Dauer

zu erkennen, was ebenfalls zu erwarten war. Vergleicht man beispielsweise Zeile 1 und 3 so erkennt man, dass die mit Priorisierung berechnete Route in Zeile 1 zwar ca. 500 Meter kürzer ist, als die Route in Zeile 3, jedoch auch ca. 2 Minuten länger dauert.

Grundsätzlich lässt sich erkennen, dass die Route etwas länger ist, wenn die Priorisierung verwendet wird. Dies ist so auch zu erwarten gewesen, da Adressen, die priorisiert werden, die Reihenfolge der Route verändern und diese dann nur noch angepasst optimal ist. Ebenfalls ist zu erkennen, dass die Differenzierung zwischen der Distanz und der Dauer zur Ermittlung des kürzesten Weges zwischen zwei Adressen, einen Unterschied macht.

Betrachtet man die Ergebnisse, die die Implementation der Änderungen am TSP-Algorithmus bezüglich des Problems der Priorisierung erzeugt haben, so lässt sich ein sehr zufriedenstellendes Ergebnis feststellen. Die Adressen, die Pakete erhalten, die priorisiert werden sollen, sind auch am Anfang der berechneten Route zu finden. Es ist nicht so, dass alle zu priorisierenden Pakete direkt nacheinander am Anfang der Route angesiedelt werden, sondern diese werden sinnvoll mit den anderen Paketen kombiniert. So kann es beispielsweise sein, dass es insgesamt drei zu priorisierende Adressen gibt. Diese Adressen werden dann beispielsweise innerhalb der ersten zehn Adressen der Route angesiedelt.

9 Mögliche Erweiterungen

Es gibt noch einige Probleme mit der momentanen Implementierung des TSP Problems. Generell könnte man sich überlegen zu einem asynchronen TSP zu wechseln, welches Einbahnstraßen oder bei großen Straßen auch die Straßenseite berücksichtigt. Hierfür könnte man die Kosten zwischen den Knoten anpassen, sodass diese aus Distanz, Dauer, Höhenunterschied, Witterungsbedingungen und anhand vieler andere Inputdaten berechnet werden. Beispiele für Inputdaten wurden bereits ins Kapitel 5.3 erläutert. Man könnte dafür ein Neuronales Netz einsetzen, dieses könnte vielleicht auch schon im Vorhinein Routen miteinander verknüpfen. Beispielsweise ist es sinnvoll, wenn man von Ulm nach Stuttgart fährt, immer die Autobahn zu nehmen. Dieses Neuronale Netz könnte vielleicht auch die Priorisierung durchführen und dabei auch zusätzliche Daten, wie Kunden oder Postbotenfeedback, auswerten. Diese Erweiterung wären recht aufwändig.

Leider konnten die dritte und vierte Anforderung nicht mehr erfüllt werden. Diese könnten zusammen mit dem neuronalen Netzen in einem zukünftigen Projekt weiter angegangen werden.

10 Fazit und Aussichten

Mit diesem Projekt sollte ein System entwickelt werden, dass Paketdaten verarbeitet, sodass Pakete in sortierter Reihenfolge vorliegen. Dabei sollten diese nach festgelegten Eigenschaften priorisiert und anhand dessen die ermittelte Route angepasst werden.

Abschließend lässt sich sagen, dass sich der Verlauf des Projektes sehr positiv gestaltete. Sowohl das Zeitmanagement als auch die Zusammenarbeit im Team verlief reibungslos.

Die zu Beginn definierten Anforderungen wurden jedoch nicht alle umgesetzt. So ist bislang noch keine tagesabhängige Zuweisung von Autos zu Bezirken möglich. Weiterhin wurde die bezirksübergreifende Paketauslieferung ebenfalls nicht realisiert. Grund dafür war die Komplexität hinter dem Projekt und damit auch die fehlende Zeit, alle Anforderungen in einer kurzen Spanne umzusetzen.

Das Projektziel dagegen wurde erfolgreich erreicht. Dazu zählt, dass die Pakete automatisiert sortiert und weiterführend auch nach festgelegten Eigenschaften wie dem Volumen oder dem Gewicht priorisiert werden. Zusätzlich orientiert sich das System nicht mehr an einer vorgegebenen Route, wie in den Anforderungen beschrieben wurde, sondern ermittelt anhand von Adressdaten eine eigene optimierte Route, die bei bevorzugten Paketen gegebenenfalls angepasst wird.

Zusammenfassend lässt sich sagen, dass das entwickelte System sich als Prototyp eignet und zeigt, dass eine reale Implementierung möglich und auch sinnvoll ist. In diesem Zusammenhang wäre es außerdem spannend, das System anstelle von, synthetisch erzeugten Daten mit echten Daten zu testen.

11 Literatur

[1] Blum, Christian (2005): Ant colony optimization: Introduction and recent trends. In: Physics of Life Reviews 2 (4), S. 353–373. DOI: 10.1016/j.plrev.2005.10.001.

[2] Christopher M White, Gary G Yen (2022): A Hybrid Evolutionary Algorithm for Traveling Salesman Problem. IEEE Xplore. Online verfügbar unter https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1331070, zuletzt aktualisiert am 21.01.2022, zuletzt geprüft am 21.01.2022.

[3] M.O. Odetayo (2002): Optimal population size for genetic algorithms: an investigation. Hg. v. IEEE Xplore. Online verfügbar unter https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=257670, zuletzt aktualisiert am 22.01.2022, zuletzt geprüft am 22.01.2022.

[4] Michael Held, Richard Karp (1969): The Traveling-Salesman Problem and Minimum Spanning Trees. IBM Systems Research. Online verfügbar unter https://web.p.ebscohost.com/ehost/pdfviewer/pdfviewer?vid=0&sid=5d54b5c3-ceb2-46aa-a19b-deb1f0cc1d5a%40redis, zuletzt aktualisiert am 06.01.2022, zuletzt geprüft am 06.01.2022.

[5] Richard Bellman: Dynamic Programming Treatment of the Travelling Salesman Problem. Online verfügbar unter https://dl.acm.org/doi/pdf/10.1145/321105.321111?casa_token=ebjrGxpRPEwAAAAA%3ANRNrtkObPCQfPiK1tMOQkycKwUXh9HDHuGoaJ4oqRA3bQyJB9YtG3VSfGcKlywVYRkrFDCKEHBU, zuletzt geprüft am 17.01.2022.

[6] Xue-song Yan, Han-min Liu, Jia Yan and Qing-hua Wu (2022): A Fast Evolutionary Algorithm for Traveling Salesman Problem. IEEE Xplore. Online verfügbar unter https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4344648, zuletzt aktualisiert am 21.01.2022, zuletzt geprüft am 21.01.2022.