Projekt: Paketlieferservice

Pflichtenheft

Projektbezeichnung	Paketlieferservice
Projektleiter	Gabriel Goller
Erstellt am	17.12.2020
Letzte Änderung am	15.01.2021
Status	in Bearbeitung
Aktuelle Version	5

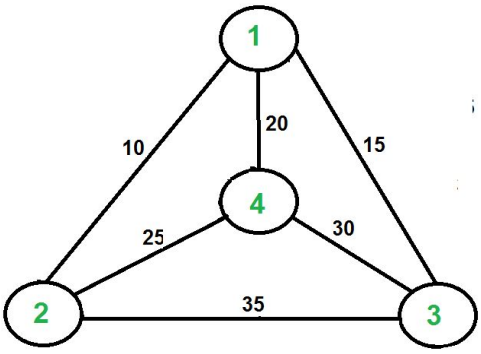
Änderungsverlauf

Nr.	Datum	Version	Änderung	Autor
1	17.12.2020	1	Algorithmen	Lukas S.
2	18.12.2020	2	Formatierung	Gabriel G.
3	14.01.2021	3	Softwarekomponente	Gabriel G
4	15.01.2021	4	Algorithmen	Lukas S.
5	15.01.2021	5	Gesamtüberblick vor Abgabe	Alle

Das Problem: “Problem des Handlungsreisenden” (Travelling Salesman Problem “TSP”)

Dabei hat man eine bestimmte Anzahl von Knoten, welche man jeweils einmal besuchen muss. Der Startpunkt ist gleichzeitig der Endpunkt und wird deshalb zwei mal besucht. Die gesamte Reisestrecke soll möglichst kurz sein. Um eine kurze Route möglichst schnell zu erhalten, kann man verschiedene Algorithmen benutzen. Erhöht man die Anzahl der Knoten, welche man besuchen will, steigt die Anzahl der möglichen Routen sehr stark.

Hat man 4 Knoten kann der Aufbau davon so aussehen:



Die kürzeste Route würde so verlaufen: 1-2-4-3-1.

Das Problem ist ein NP schweres Problem, das bedeutet, dass es nicht möglich ist, ein optimales Ergebnis in polynomialer Zeit zu berechnen. Deshalb werden bei vielen Knoten heuristische Verfahren angewendet.
Das Problem findet oft Anwendung in der Arbeitswelt, z.B. bei Paketlieferdiensten oder bei der Tourenplanung.

Organisation

Team

Rolle(n)	Name	Github	E-Mail
Projektleiter	Gabriel Goller	@kaffarell	stgolgab@bx.fallmerayer.it
Programmierer	Daniel Bosin	@Kurisu003	stbosdan@bx.fallmerayer.it
Programmierer	Lukas Schatzer	@zLuki	stschluk2@bx.fallmerayer.it
Programmierer	Philipp Olivotto	@zbaakez	stoliphi@bx.fallmerayer.it
Designer	Alex Unterleitner	@LaxeChef	stuntale@bx.fallmerayer.it

Software

Damit wir eine bessere Übersicht über die ganzen Anforderungen hatten, haben wir auf verschiedene Tools zurückgegriffen, mit denen wir unseren Fortschritt sichern, die Anforderungen festhalten und den Code speichern konnten. Anstatt verschiedene Tools für all diese Aufgaben zu verwenden, benutzen wir Github. Mit Github können wir den Code online speichern, worauf alle Teammitglieder zugreifen können. Auf Github haben wir eine Organisation erstellt und dort haben wir alle Mitglieder hinzugefügt und zwei Repositories erstellt. Ein Repository für die Website (frontend und backend) und das andere Repository für die Algorithmen. Bei der Organisation auf Github haben wir auch ein Projekt hinzugefügt, dort verwenden wir eine Kanban Tabelle, wo wir verschiedene Issues aus beiden Repositories sammeln können. Bei dem Repository mit den Algorithmen haben wir auch drei Milestones erstellt, wo wir dann verschiedene Issues mit den einzelnen Aufgaben hinzufügen. Für alle Milestones haben wir dann auch ein Enddatum eingetragen.

Softwarekomponente

Berechnen der bestmöglichen Route:

Hier müssen wir unterscheiden, aus wie vielen Knoten die Route besteht:

Mit folgender Formel berechnet man die Anzahl der möglichen Routen:

$$\frac{(n-1)!}{2}$$

n steht dabei für die Anzahl der Knoten. Als Beispiel wählen wir 11 Knoten:

$$\frac{(11-1)!}{2}$$ = 1.814.400 mögliche Routen

Falls man die Anzahl der Knoten erhöht, steigen die möglichen Routen sehr stark, aufgrund des Ergebnisses der Fakultät. Bei 15 Knoten hätte man 43.589.145.600 mögliche Routen.

Wir haben einen Brute Force Algorithmus, um die bestmögliche Route zu finden.

Bei vielen Knoten würde ein Brute Force Algorithmus viel zu lange für die Berechnungen brauchen, deshalb verwenden wir verschiedene heuristische Verfahren.

Sie haben ihre Vor- und Nachteile, z.B. Nearest-Neighbor ist schnell und einfach zu implementieren, Convex Hull ist etwas langsamer, deutlicher schwerer zu implementieren, dafür liefert es aber etwas bessere Ergebnisse. Simulated Annealing liefert meist schnell gute Ergebnisse, kann aber bei bestimmten Routen total versagen.

Algorithmen

Wir haben mehrere Klassen für die Algorithmen erstellt:
- Bruteforce
- NearestNeighbor
- NearestInsertion
- SimulatedAnnealing
- ConvexHull

Brute Force Algorithm

Probiert alle möglichen Routen aus. So wird die bestmögliche Route gefunden. Es ist darauf zu achten, dass man nicht alle Routen doppelt ausprobiert. Bei drei Knoten a,b,c ist z.B. a->b->c->a dasselbe wie a->c->b->a.

Nearest Neighbor Algorithm

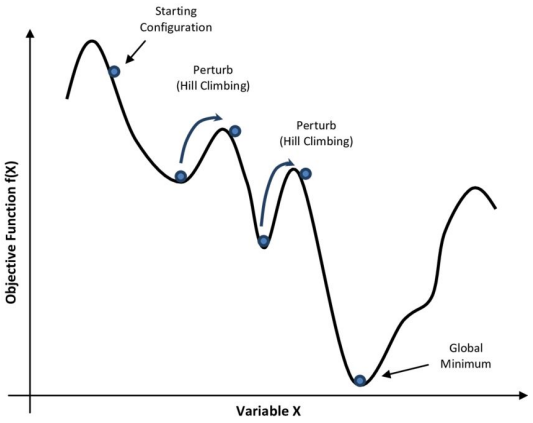
Vom fixen Startknoten aus wird der am wenigsten entfernte Knoten besucht. Von dem zweiten Knoten wird dann wieder der am geringsten entfernte Knoten besucht, solange bis alle Knoten besucht worden sind. Abschließend wird noch der Startpunkt besucht. So kommt es zu einem Hamiltonkreis. Dieses Verfahren wählt so meistens eine kurze Tour, jedoch fast nie die Optimale. Ein weiterer Vorteil ist, dass dieser Algorithmus sehr schnell ist.

Nearest Insertion Algorithm

Ziel ist es wieder eine relativ kurze Rundreise über alle Knoten zu erhalten. Der Algorithmus wählt bei jedem Schritt einen Knoten aus, welcher die geringste Entfernung zu einem Knoten hat, welcher auf der bereits vorhandenen Teilroute vorhanden ist. Dieser Punkt wird dann in die vorhandene Teilroute so eingebaut, dass diese Teilroute sich am geringsten verlängert. Die entstandene Route kann nicht länger sein, als die doppelte Strecke der optimalen Route.

Simulated Annealing (“Simuliertes Abkühlen”)

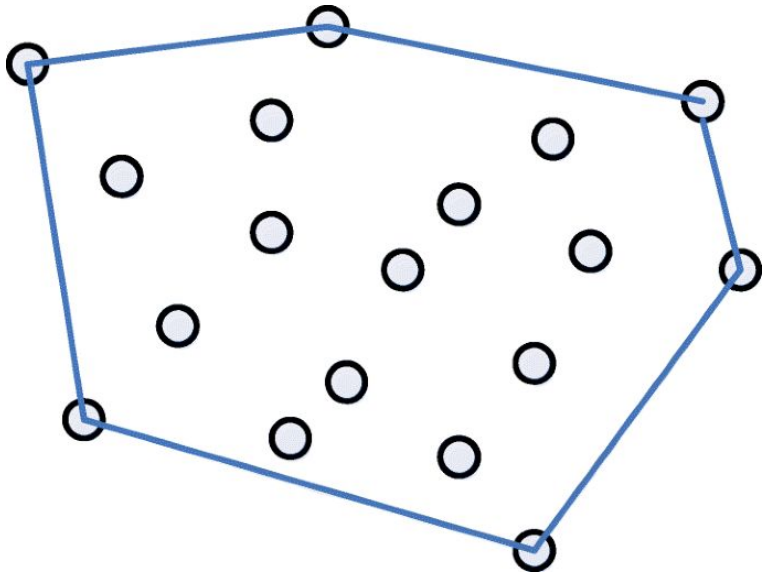
Man startet zuerst mit der Nearest Neighbor Route, dann beginnt man zufällig zwei Punkte in der Route zu tauschen. Dann schaut man, ob sich die Route dadurch verbessert hat oder nicht. Man würde zu einem Punkt kommen, an dem sich die Route nicht mehr verbessern kann, aber noch nicht die beste Route ist. Um das so lange wie möglich zu verhindern, wird die Route auch kurzzeitig verschlechtert, um dann wieder ein besseres Ergebnis zu erreichen. Es kann jedoch vorkommen, dass einige während der Laufzeit berechneten Routen kürzer sind, als die final ausgegebene Route, deshalb wird das beste Ergebnis immer gespeichert.
Das folgende Diagramm sollte den Algorithmus recht gut beschreiben:



Ob die neue Route akzeptiert wird oder nicht, wird wie folgt berechnet:
Wenn die neue Route kürzer als die alte Route ist, dann wird sie klarerweise akzeptiert, ansonsten verwendet man folgende Formel:
Math.exp(-delta-sigma) > Math.random().
Dabei ist -delta die negative Differenz der Länge der neuen Route und der Länge der alten Route und sigma ein Toleranzwert, der sich bei jedem Durchgang verringert.

Convex Hull

Dieser Algorithmus eignet sich gut, um eine kurze Rundreise zu bestimmen, wenn man die Koordinaten der Punkte gegeben hat. Mit einer Adjazenzmatrix ist es eher schwieriger, deshalb eignet sich dieser Algorithmus am besten, um auf unserer Webseite ausgeführt zu werden.
Bei diesem Algorithmus werden zuerst die äußersten Punkte miteinander verbunden, sodass man einen Kreis um die restlichen inneren Punkte erhält:



Danach wird der Kreis “zusammengedrückt”, die Punkte, die weiter außen liegen, werden zum Äußeren Kreis hinzugefügt, bis am Ende alle Punkte dem Kreis hinzugefügt wurden. Ein großer Vorteil dieses Algorithmus ist, dass es durch das Vorgehen von Außen nach Innen zu einem art Kreis kommt, die Route wird sich nie überkreuzen. Da es bewiesen ist,

dass sich die optimale Route niemals kreuzt und auch Convex Hull sich niemals kreuzt, schafft es dieser heuristische Algorithmus sehr nahe an die optimale Lösung zu kommen.

Desktop App

Für die Desktop App (Konsolen App) haben wir einfach eine Klasse Console erstellt, die die ganzen I/O Operationen ausführt. In dieser Klasse haben wir Methoden, um Input vom User durch die Konsole einzulesen und die entsprechenden Ergebnisse in der Konsole auszugeben.

CSV Reader

Zum Einlesen der Adjazenzmatrix aus einer CSV-Datei, benutzen wir die Klasse CsvReader. Dabei wird die CSV-Datei geöffnet und die Matrix wird in einem zweidimensionalen Array gespeichert. Die Städte werden getrennt in einer ArrayList gespeichert.

Weitere Helper Klassen, Enums und Interfaces kurz beschrieben

City
Simple Klasse zum Speichern einer Stadt bzw. eines Knotens. Speichert einfach den Namen der Stadt und eine ID. Hat einen passenden Konstruktor, Getter und Setter Methoden, sowie eine überschriebene .equals() und .hashcode() zum Vergleichen von Objekten.

Cities
Simple Klasse zum Speichern einer Route. Speichert die Distanz der Route, die benötigte Zeit zum Berechnen und die Namen der Städte in richtiger Reihenfolge in einer ArrayList. Hat einen passenden Konstruktor, Getter und Setter Methoden, sowie eine überschriebene .equals() und .hashcode() zum Vergleichen von Objekten.

Algorithm
Interface, das jeder Algorithmus implementieren muss.
Beinhaltet diesen Methodenkopf: Cities getResult();

AlgorithmName
Enum, das die Namen der Algorithmen beinhaltet. Erleichtert I/O des Users.

Webapp

Die Webapp ist in einem eigenen Repository auf Github angesiedelt. Das Repository beinhaltet frontend und backend. Das backend ist mit Java programmiert und nutzt das Framework Spring Boot. Der Java Code hostet die statischen Seiten und hat eine einzige

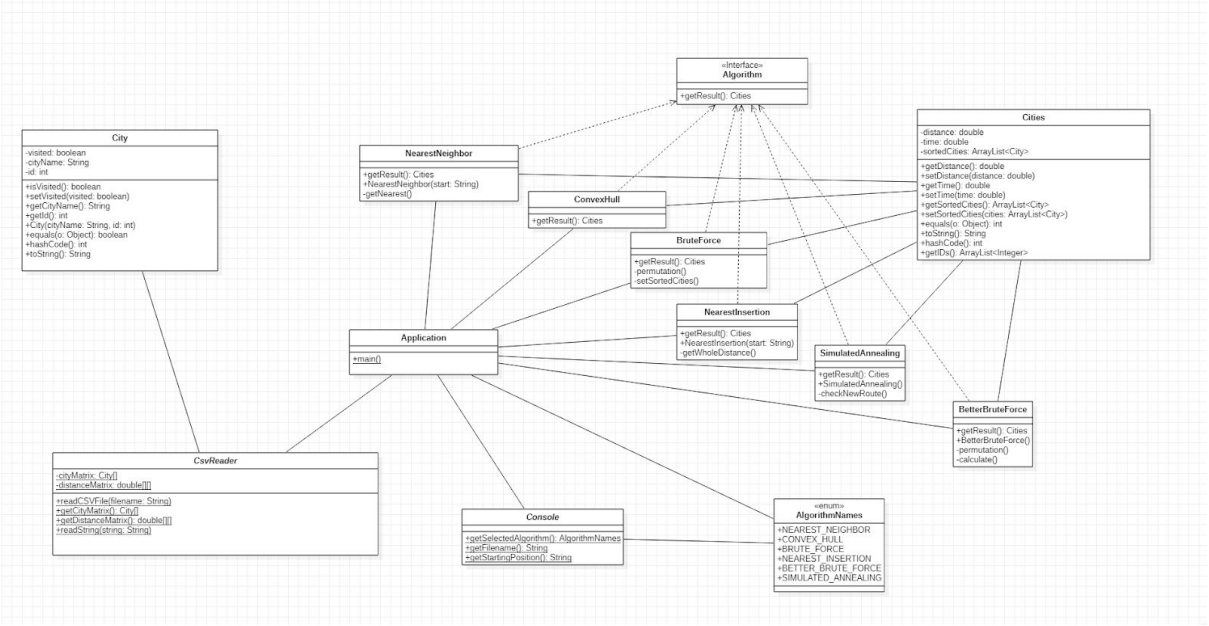
POST Route, welche die Adjazenzmatrix als String entgegennimmt und die kürzeste Route zurückgibt. Das frontend besteht aus einer Reihe von statischen HTML Dateien, die mit SCSS gestyled sind. Außerdem gibt es auch noch eine Javascript Datei, welche die Mapbox API einbettet und eine Landkarte zeichnet. Auf dieser Landkarte kann man Punkte auswählen, die in einer Adjazenzmatrix zum Java-Server gesendet werden. Wenn die kürzeste Route ermittelt wurde, wird diese zurückgesendet und in der Landkarte dargestellt. Das Übermittelte Paket enthaltet nicht nur die Adjazenzmatrix sondern auch andere Infos:

```
0ni,0,1,2,3,4,5,
0,0,1435.145609969578,2269.731794560078,2112.4219507681078,1385.0959229878117,1221.5430530269782,
1,1435.145609969578,0,1554.4568034702547,2249.3668648564367,2115.35860619595119,1059.2154807524123,
2,2269.731794560078,1554.4568034702547,0,1253.3103892950367,1834.7180703815754,1052.4606436141228,
3,2112.4219507681078,2249.3668648564367,1253.3103892950367,0,951.379918821631,1218.7931836682121,
4,1385.0959229878117,2115.358606195119,1834.7180703815754,951.379918821631,0,1129.8963547848548,
5,1221.5430530269782,1059.2154807524123,1052.4606436141228,1218.7931836682121,1129.8963547848548,0
```

Der erste Charakter ist entweder 0 oder 1, also visualisierung aus oder an. Das dritte und vierte Zeichen ist der Algorithmus also in diesem Fall ni (Nearest Insertion) (ni = Nearest Insertion, nn = Nearest Neighbor, sa = Simulated Annealing, bf = Brute Force, ch = Convex Hull).

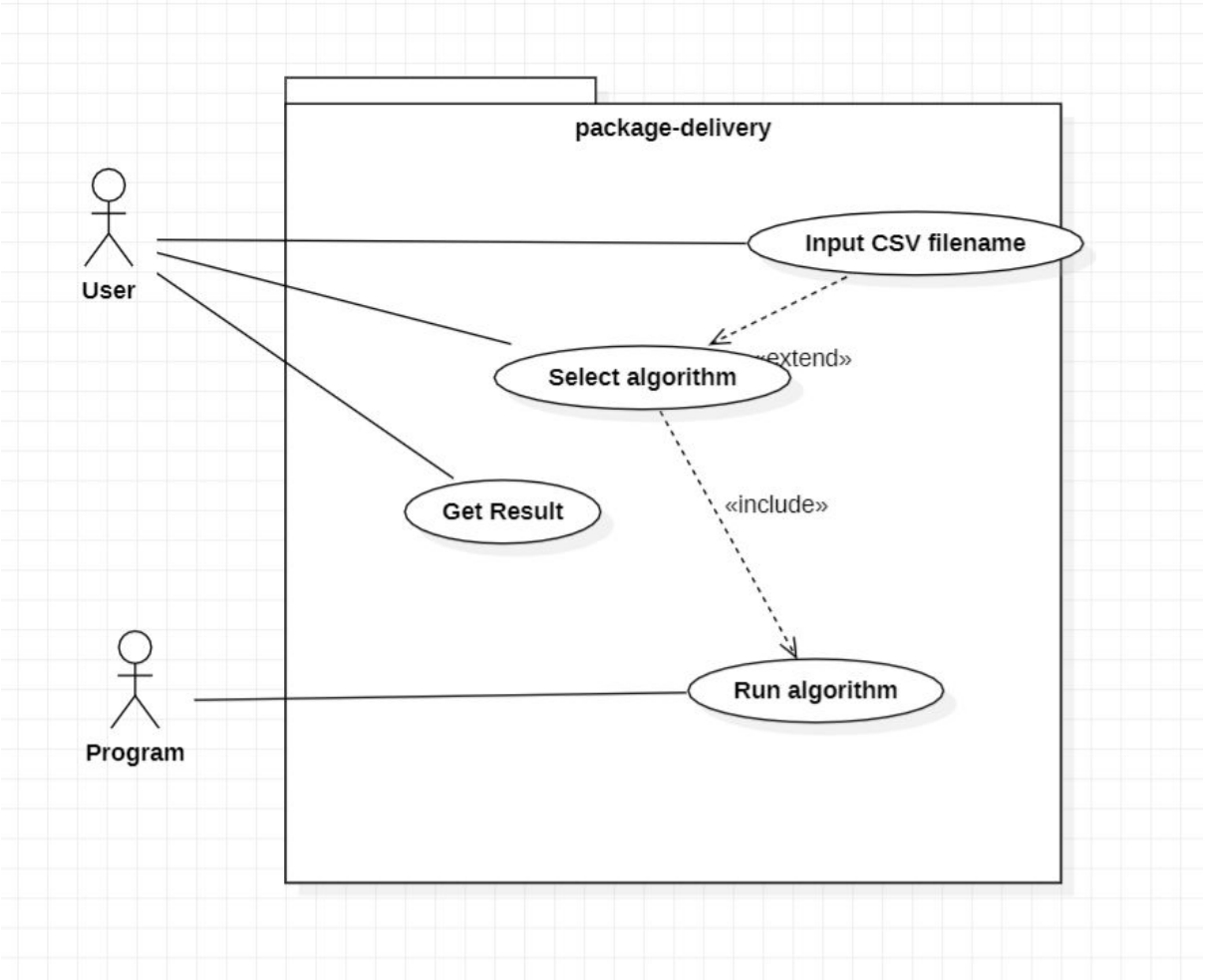
Diagramme

Klassendiagramm



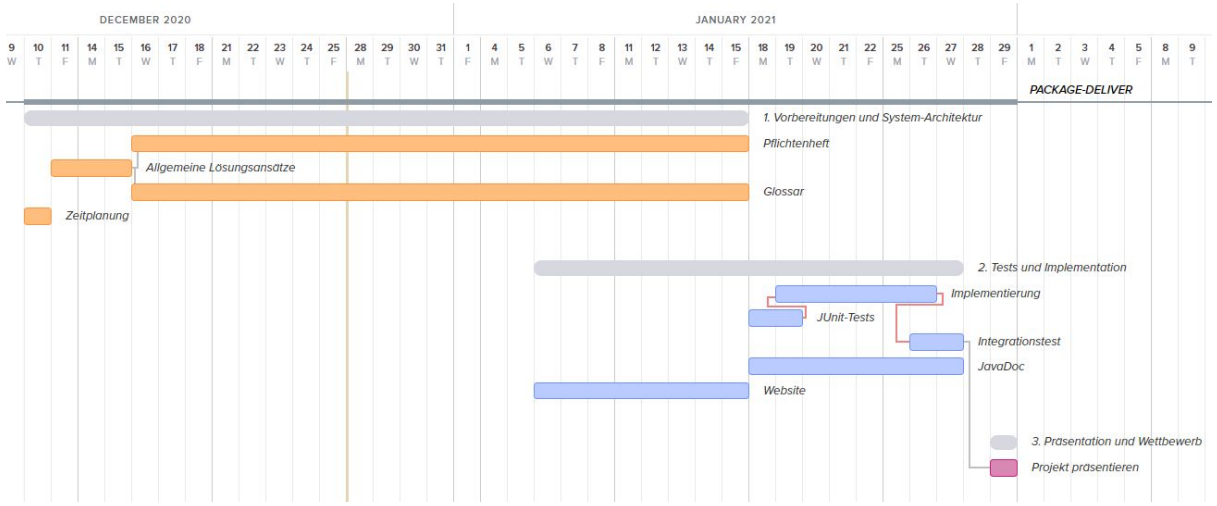
Komplettes Klassendiagramm auf Staruml erstellt und verfügbar auf GitHub: https://github.com/package-delivery/package-delivery/blob/master/documentation/classdiagram.mdj

Use-Case Diagramm



Use-Case Diagramm auf Staruml erstellt und hier verfügbar auf GitHub: https://github.com/package-delivery/package-delivery/blob/master/documentation/usecase.mdj

Zeitplanung



Gantt Diagramm erstellt mit TeamGantt.

Quellen:

https://en.wikipedia.org/wiki/Travelling_salesman_problem

https://tspvis.com/

https://stackoverflow.com/questions/11703827/brute-force-algorithm-for-the-traveling-salesman-problem-in-java

https://de.wikipedia.org/wiki/Nearest-Neighbor-Heuristik

https://cs.stackexchange.com/questions/88933/how-does-the-nearest-insertion-heuristic-for-tsp-work

https://de.wikipedia.org/wiki/Nearest-Insertion-Heuristik

https://de.wikipedia.org/wiki/Hamiltonkreisproblem

https://www.sciencedirect.com/topics/engineering/simulated-annealing-algorithm#:~:text=The%20simulated%20annealing%20algorithm%20is,state%20is%20reached%20%5B143%5D.

https://en.wikipedia.org/wiki/Convex_hull_algorithms