The user perceived performance of route planning APIs

Bert Marcelis

Supervisor: Prof. dr. ir. Ruben Verborgh
Counsellors: Pieter Colpaert, Julian Andres Rojas Melendez

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Information Engineering Technology

Department of Electronics and Information Systems
Chair: Prof. dr. ir. Koen De Bosschere
Faculty of Engineering and Architecture
Academic year 2017-2018

Dankwoord

Inhoudsopgave

Lijst van figuren **6**

Lijst van tabellen **7**

1 Inleiding **9**

1.1 Wat is user-perceived performance? 12

1.2 Probleemstelling en doel van de masterproef 13

1.3 Onderzoeksvraag 14

2 Implementatie **15**

2.1 Linked Connections formaat 15

2.1.1 Vraag- en antwoordformaat 16

2.2 Connection Scan Algoritme 18

2.2.1 Implementatie en aanpassingen 18

2.3 Vertrekken en aankomsten per station 30

2.4 Route van een voertuig 31

Bibliografie **32**

Bijlagen **34**

Lijst van figuren

1.1 GTFS structuur 10

1.2 RPC structuur 11

1.3 Routeplanning HTTP interfaces op de LDF as 12

Lijst van tabellen

Lijst van listings

"Inspirational quote"

~Source

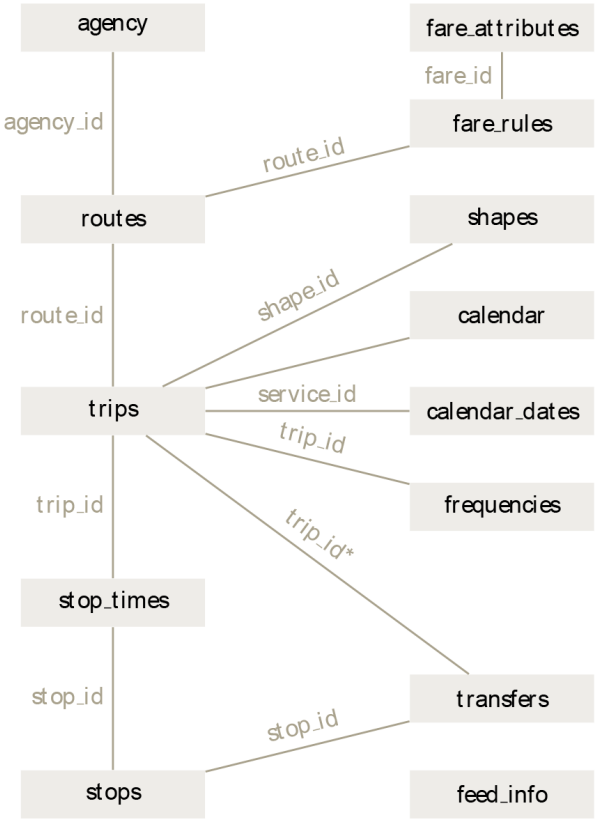
1 Inleiding

Openbaar vervoer is ook een essentiële dienst in elke stad[1]. Om vlot van dit openbaar vervoer gebruik te maken, zijn er tientallen websites en apps (user-agents) die gebruikers informatie verstrekken over vertrekken, aankomsten, ritten, routes en vertragingen. Voorbeelden hiervan in België zijn iRail.be, HyperRail en Railer, en CityMapper, TheTransitApp, Here WeGo en Google maps wereldwijd. Op dit moment zijn al deze user-agents echter toegewezen op het gebruik van data dumps of specifieke APIs om informatie met betrekking tot openbaar vervoer te publiceren, of een variant ervan.

Enerzijds zijn er volledige data dumps, in de vorm van General Transit Feed Specification (GTFS)[1] en General Transit Feed Specification Realtime (GTFS-RT)[2]. GTFS bestanden bevatten informatie over alle voertuigen van een dienstverlener, over een relatief grote tijdspanne, typisch enkele maanden tot een jaar. GTFS-RT bestanden bevatten realtime informatie over ritten in de komende dag. Om al deze data compact op te slaan en te versturen, worden deze opgeslagen in de vorm van regels. Deze regels omschrijven wanneer welk voertuig welke rit maakt. Om op basis van deze regels vragen te beantwoorden, dient deze set abstracte regels omgevormd te worden naar een gepast model waarin ritten en stopplaatsen opgevraagd kunnen worden, en routes berekend kunnen worden. Hiervoor zijn, afhankelijk van welke informatie gewenst is, zware berekeningen vereist, die afhankelijk van de grootte van het GTFS bestand vijf à tien

[1] https://developers.google.com/transit/gtfs/

[2] https://developers.google.com/transit/gtfs-realtime/

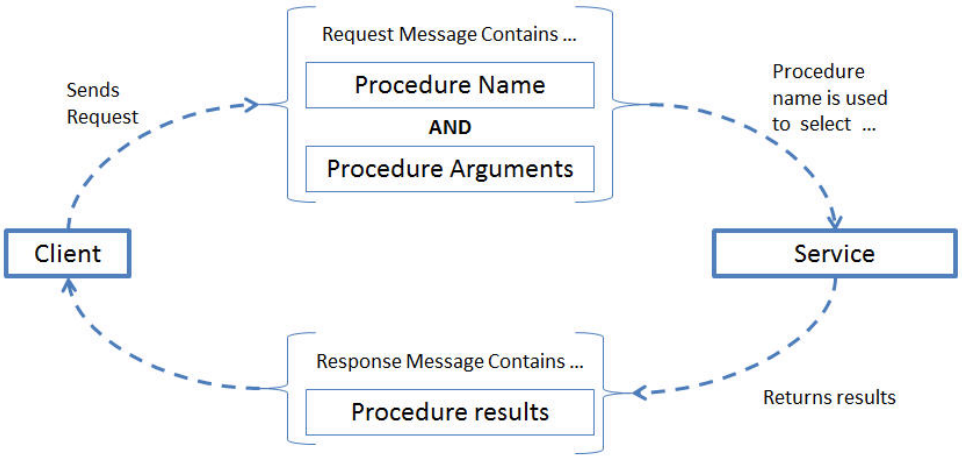


Figuur 1.1: De bestandsstructuur van GTFS data.

minuten kunnen duren op een moderne computer. Gebruikers kunnen geen 10 minuten wachten tot de data getransformeerd zijn, waardoor deze optie niet beschikbaar is op mobiele toestellen. Verder is dit formaat een mogelijke technologische restrictie op de vervoersdata: enkel gevorderde ontwikkelaars kunnen hiervan gebruik maken. Open data is slechts open als deze (onder andere) beschikbaar zijn in een begrijpbaar formaat [2]. GTFS is dus vooral geschikt om vervoersdata te delen met grote bedrijven, en in mindere mate voor individuele ontwikkelaars die vervoers data eenvoudig willen visualiseren (digital signage, routeplanner applicaties, websites, ...).

Anderzijds zijn er traditionele Remote Procedure Call (RPC) zoals iRail[3], die beschikken over verschillende endpoints die specifieke vragen kunnen beantwoorden. Achterliggend kunnen zware berekeningen uitvoeren of grote databases raadplegen zonder dat de gebruiker hier nadeel van ondervindt. Deze antwoorden zijn rechtstreeks bruikbaar voor de client toepassing, maar bieden enkel een antwoord op één specifieke vraag. Een andere vraag, al dan niet door dezelfde client, vereist een nieuwe request naar de server, en zal een ander antwoord tot gevolg hebben. Elk verzoek naar de server vraagt relatief veel processortijd langs de serverkant. Een continue internetverbinding is dus vereist, en server-side is een potentieel grote en dure infrastructuur

[3]https://irail.be

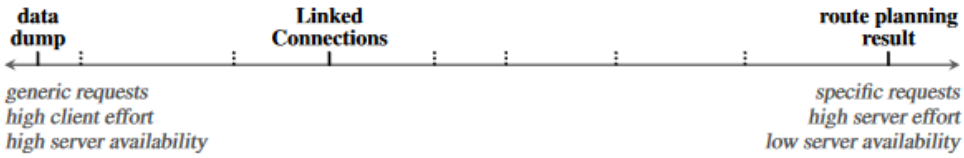


Figuur 1.2: De werkwijze van een RPC API.

nodig om aan alle vragen te voldoen. Een ander belangrijk nadeel bij deze techniek is dat deze data moeilijk te combineren zijn met andere datasets. Een route plannen die gebruik maakt van meerdere openbaar vervoer aanbieders is enkel mogelijk als iemand een API aanbiedt die achterliggend door meerdere datasets zoekt. Simpelweg twee API's combineren is niet mogelijk.

Deze twee methodes zijn elkaars tegengestelde. Ontwikkelaars moeten kiezen voor data die compact maar complex, en slechts indirect bruikbaar is, of voor een vraag-antwoord systeem wat voor elke nieuwe vraag een nieuw verzoek naar een server moet maken. Aan de IDLab onderzoeksgroep aan UGent is onderzoek gedaan naar Linked Connections (LC)[4], een nieuw formaat dat een nieuw evenwicht tracht te vinden. Alle vertrekken van alle voertuigen worden in één chronologische lijst verzameld, waarbij de lijst kan opgevraagd worden volgens vaste tijdsintervallen met een grootteorde van enkele minuten. Hierdoor hoeft de server enkel deze lijst in fragmenten aan te bieden, waarbij alle clients dezelfde informatie krijgen. De clients dienen zelf nog berekeningen te maken, maar deze zijn relatief eenvoudig vergeleken met de berekeningen die nodig zijn om een GTFS feed te verwerken. Data in het Linked Connections formaat kunnen eenvoudig toegankelijk gemaakt worden via een open-source serverapplicatie[5].

[4]https://linkedconnections.org
[5]https://github.com/julianrojas87/linked-connections-server/



Figuur 1.3: De Linked Data Fragments as illustreert dat alle HTTP interfaces data fragmenten aanbieden, maar verschillen in hoe specifiek de aangeboden data is, en dus de moeilijkheid om deze aan te maken [3]. In deze figuur is de as toegepast op HTTP interfaces voor routeplanning [4].

1.1 Wat is user-perceived performance?

Elke interface voor het ophalen van data heeft specifieke eigenschappen zoals latency, performance, cache hergebruik, ... [5]. Wanneer we verschillende technieken vergelijken door dezelfde user-agent, kunnen we de impact van de verschillende achterliggende technieken op de eindgebruiker onderzoeken. Hiervoor definiëren we de user-perceived performance. De user-perceived performance is de performance zoals de gebruiker deze ervaart, welke niet strikt gelijk hoeft te zijn aan de performance van technische component. De user-perceived latency werd gedefinieerd in 2000 door Roy T. Fielding gedefinieerd als de tijd tussen het selecteren van een link en het renderen van een bruikbaar resultaat [6]. Latency treedt op op verschillende punten:

1. de tijd die de client nodig heeft om actie te ondernemen
2. de tijd die nodig is voor voorbereidende acties
3. de tijd om een verzoek te verzenden
4. de tijd die de server nodig heeft om te antwoorden
5. de tijd die nodig is om het antwoord te verzenden
6. de tijd voor het antwoord te verwerken en weer te geven

Terwijl enkel stappen 3, 4 en 5 rechtstreeks afhankelijk zijn van het netwerk, kunnen al deze stappen beïnvloed worden door de gebruikte techniek [6].

Ondertussen zijn we geëvolueerd naar een wereld waarin data vaak mobiel geconsumeerd worden: 78% van de Vlamingen beschikt over een smartphone, 80,5% beschikt over een laptop. Slechts 41,8% beschikt over een desktop-computer [7]. Hierbij zijn er ook andere aspecten die meespelen in de user experience: batterijgebruik en offline toegang tot data vormen een aanzienlijke factor in de user experience. Een applicatie presteert beter wanneer deze dezelfde data kan weergeven met aanzienlijk minder energieverbruik, of wanneer deze consistent goed presteert, ook wanneer

netwerk slecht of niet beschikbaar is. Hoewel de user-perceived performance nog steeds gedomineerd wordt door de tijd tussen het selecteren van een link en het renderen van een bruikbaar resultaat, dienen we dus ook deze andere aspecten in rekening te brengen. Mobiele gebruikers hebben ook nog steeds angst om te veel data te verbruiken [8].

1.2 Probleemstelling en doel van de masterproef

Linked Connections werd ontwikkeld met de bedoeling een evenwicht te vinden tussen data dumps en RPC API's. In plaats van elke query op een server te beantwoorden, wordt een gelinkte lijst van connecties gepubliceerd. Linked Connections laat hierdoor toe om queries te beantwoorden door middel van een lineair groeiende lijst van connecties [4]. Bovendien gebruiken alle user-agents dezelfde lijst, waardoor deze zeer cachebaar is. Bij stijgende belasting daalt de tijd die nodig is per verzoek [9].

Terwijl de cost-efficiency van Linked Connections reeds is aangetoond, waarbij Linked Connections hetzelfde aantal verzoeken kan beantwoorden met slechts 25% van de rekencapacteit[9, 10], is er nog geen onderzoek gebeurt naar de user-perceived performance van een user-agent wanneer deze gebruik maakt van Linked Connections, vergeleken met wanneer deze zelfde user-agent gebruik maakt van een traditionele RPC API.

In deze studie richten we ons specifiek op routeplanning gebruik makend van mobiele toestellen. Deze toestellen hebben minder processorkracht en geheugen vergeleken met traditionele computers, maar ook bandbreedte en beschikbaarheid van internet zijn vaak beperkt. In het slechtste geval is er geen netwerkverbinding, waarbij enkel een cache beschikbaar is. Verder zullen we ons specifiek richten op het verschil tussen een RPC REST API gebaseerd op Linked Connections [9] en de originele Linked Connections webserver. Als user-agent zullen we een fork van de Android HyperRail[6] applicatie gebruiken, gemodificeerd om de genoemde API's te gebruiken. Door deze testopstelling zijn de oorspronkelijke data, de server hardware, de user-agent en de client hardware gelijk bij elke vergelijking. Enkel het formaat voor serverinteracties en transport van data zal verschillen.

Om routes te berekenen zullen we gebruik maken van het Connection Scan Algorithm (CSA) [11, 12, 13]. Dit algoritme vereist een op vertrektijd gesorteerde lijst van vertrekken. Dit is de exacte definitie van de LinkedConnections knowledge graph, waardoor dit algoritme zonder al te veel modificaties toegepast kan worden. Fragmenten kunnen hierbij geladen worden op het moment dat ze nodig zijn. We zullen dezelfde implementatie gebruiken zowel bij de client-side API als bij de server-side API om zo correct mogelijke resultaten te behalen.

[6] https://hyperrail.be

In eerste instantie zal een traditionele (RPC) API geschreven worden welke gebruik maakt van de Linked Connections fragmenten op de Solid State Disk (SSD) van de server. Deze API zal endpoints bevatten voor het tonen van vertrekken en aankomsten per station, het berekenen van routes, en voor het weergeven van het traject per trein. Vervolgens zal een API zonder specifieke server-side geïmplementeerd worden in de applicatie. Deze zal dezelfde informatie ter beschikking stellen in de applicatie, maar zal hiervoor enkel (delen van) de gelinkte lijst met vertrekken downloaden.

Eenmaal beide API's volledig geïmplementeerd zijn, zal de user-experienced performance onderzocht worden. Hiertoe worden begeleide user tests gehouden, waarbij een aantal testgebruikers afwisselend met beide API's hun dagelijkse opzoekingen zullen uitvoeren, waarna ze aan de hand van een vragenlijst bevraagd zullen worden naar hun ervaringen en voorkeuren. Het is essentieel om de subjectieve ervaringen van gebruikers te bevragen, gezien verschillende gebruikers mogelijk verschillende afwegingen maken. We verwachten dat sommige gebruikers offline toegang waardevol zullen vinden, terwijl anderen mogelijk geen belang hechten aan offline toegang. Ook zal er technische data verzameld worden, zoals geheugen- en processorgebruik, laadtijden, en batterijverbruik.

Deze masterproef zal gebruik maken van data afkomstig van de NMBS om routeplanning en realtime data over treinen in België weer te geven. Door de bron van de data te vervangen kan dit onderzoek ook toegepast worden op andere openbaar vervoer maatschappijen die gebruik maken van tijdsschema's, ongeacht het soort voertuig dat gebruikt wordt.

1.3 Onderzoeksvraag

Hypothese 1 De gebruiker ervaart de mogelijkheid voor offline zoekopdrachten als een meerwaarde.

Hypothese 2 De gebruiker ervaart de mogelijkheid om voorkeuren voor routes in te stellen (overstaptijd, toegankelijkheid, ...) als een meerwaarde.

Onderzoeksvraag Verbetert de user experience en user perceived perforance van een applicatie voor openbaar vervoer wanneer gebruik gemaakt wordt van Linked Connections in plaats van traditionele RPC API's?

"Inspirational quote"

~Source

2

Implementatie

Om een zo eerlijk mogelijke vergelijking te bekomen, zullen we zowel bij de client-side API als de server-side API dezelfde algoritmes toepassen. Gezien de jonge leeftijd van het Linked Connections framework zijn er nog geen algoritmes beschikbaar om deze data te verwerken. We zullen de ontwikkeling van deze algoritmen bespreken, met speciale aandacht voor het routeplanning algoritme vanwege de hogere complexiteit en de uitgebreide mogelijkheden.

2.1 Linked Connections formaat

In plaats van een dump van planningsdata of een volledige routeplanner te publiceren, publiceert Linked Connections zogenoemde connecties. Een connectie is het kleinste ondeelbaar stuk van een treinrit, en beschrijft het vertrek in een station en de aankomst in het volgence station op de route. Deze connecties worden gesorteerd volgens vertrektijd. Hierna wordt deze lijst gesplitst, om pagina's van gelijke grootte of gelijke tijdsduur te bekomen. Deze fragmenten kunnen gepubliceerd worden via HTTP als *JSON-LD*[1], waarbij user-agents kunnen kiezen welke pagina's ze opvragen. Links in de gepubliceerde documenten zorgen ervoor dat user-agents steeds weten welke pagina ze als volgende moeten laden [14].

[1] https://json-ld.org/

Om bovenstaande methode in de praktijk om te zetten, wordt gebruik gemaakt van de open source LC-Server[2]. Om GTFS om te zetten naar Linked Connections, wordt er achterliggend gebruik gemaakt van de gtfs2lc tool[3].

Deze data zijn publiek toegankelijk via https://graph.irail.be/.

2.1.1 Vraag- en antwoordformaat

Om een pagina met data op te halen, wordt een verzoek gemaakt naar de API, waarbij de vervoersmaatschappij en het gewenste tijdstip in ISO8601 formaat in de URL opgenomen worden. Codefragment **??** toont een ingekort resultaat voor de vertrekken bij de NMBS op 20 maart 2018, 12:30. De volledige specificatie kan teruggevonden worden op de LC website[4].

Verzoek: *https://graph.irail.be/sncb/connections?departureTime=2018-03-20T12:30:00.000Z*

Een voorbeeld van een LC pagina zien we in fragment 1, met volgende data:

1. *@context*: Deze lijst definieert de gebruikte namespaces en velden
2. *hydra:next* en *hydra:previous*: Links naar de pagina met respectievelijk de volgende en de voorgaande data
3. *hydra:search*: Informatie over de huidige pagina
4. *@graph*: Deze lijst bevat de eigenlijke data. Elk vertrek bevat de volgende informatie:
   (a) *departureStop*: De URI welke het station van vertrek uniek identificeert.
   (b) *arrivalStop*: De URI welke het station van aankomst uniek identificeert.
   (c) departureTime, arrivalTime: De geplande tijden, respectievelijk bij vertrek en aankomst.
   (d) *departureDelay*, *arrivalDelay*: De vertraging, respectievelijk bij vertrek en aakomst.
   (e) *direction*: De richting van dit voertuig, wat vaak ook op de lichtkrant van het voertuig weergegeven wordt.
   (f) *gtfs:trip*: Een URI welk de rit van het voertuig uniek identificeert
   (g) *gtfs:route*: Een URI welk de route van het voertuig uniek identificeert
   (h) *gtfs:pickupType* en *gtfs:dropOffType*: geeft aan of reizigers al dan niet kunnen op- of afstappen bij respectievelijk vertrek en aankomst

[2]https://github.com/julianrojas87/linked-connections-server/
[3]https://github.com/linkedconnections/gtfs2lc
[4]https://linkedconnections.org/specification/1-0

```
{
  "@context": {
   "xsd": "http://www.w3.org/2001/XMLSchema#",
  "lc": "http://semweb.mmlab.be/ns/linkedconnections#",
  "hydra": "http://www.w3.org/ns/hydra/core#",
  "gtfs": "http://vocab.gtfs.org/terms#",
  "Connection": "lc:Connection",
  "...": "..."
  },
  "@id": "https://graph.irail.be/sncb/connections?departureTime=2018-03-
  ↪ 20T12:30:00.000Z",
  "@type": "hydra:PagedCollection",
  "hydra:next":
  ↪ "https://graph.irail.be/sncb/connections?departureTime=2018-03-
  ↪ 20T12:40:00.000Z",
  "hydra:previous":
  ↪ "https://graph.irail.be/sncb/connections?departureTime=2018-03-
  ↪ 20T12:20:00.000Z",
  "hydra:search": {
    "..."
  },
  "@graph": [
    {
    "@id": "http://irail.be/connections/8822228/20180320/S11961",
    "@type": "Connection",
    "departureStop": "http://irail.be/stations/NMBS/008822228",
    "arrivalStop": "http://irail.be/stations/NMBS/008822210",
    "departureTime": "2018-03-20T12:30:00.000Z",
    "departureDelay": 60,
    "arrivalTime": "2018-03-20T12:32:00.000Z",
    "arrivalDelay": 0,
    "direction": "Anvers-Central",
    "gtfs:trip": "http://irail.be/vehicle/S11961/20180320",
    "gtfs:route": "http://irail.be/vehicle/S11961",
    "gtfs:pickupType": "gtfs:Regular",
    "gtfs:dropOffType": "gtfs:Regular"
    },
    {"...":"..."}
  ]
}
```

Code 1: Voorbeeld Linked Connections Formaat

2.2 Connection Scan Algoritme

Routeplanning wordt vaak opgelost met behulp van (een variant op) het algoritme van Dijkstra [11]. Toepassingen die gebruik maken van Dijkstra vereisen echter een graaf en een *priority queue*. Naast de impact op prestaties die deze eisen vormen, beperkt een graaf ook de flexibiliteit. De *open world assumption* stelt dat er steeds andere stopplaatsen wiens bestaan we (nog) niet kennen. Het opstellen van een graaf zou vereisen dat we alle gegevens eerst volledig moeten downloaden, terwijl Linked Connections net goed geschikt is voor streaming.

Het Connection Scan Algoritme (CSA) werd voor het eerst beschreven door Ben Strasser in 2013 [11]. Dit algoritme vereist van een lijst met vertrekken gesorteerd op vertrektijd. Hiermee worden alle routes in een tijdsinterval efficiënt berekend[12, 13]. In tegenstelling tot Dijkstra's algoritme is er geen graaf of *priority queue* benodigd. Waar andere algoritmen ofwel enkel op kleine netwerken performant zijn, ofwel niet altijd de best mogelijke route vinden, kan CSA de optimale route in grote netwerken toch efficiënt vinden [12]. In de praktijk is het vooral belangrijk om snel rekening te kunnen houden met vertragingen bij vertrek of aankomst[12, 13]. CSA berekent oorspronkelijk de snelste route, al is dit duidelijk niet altijd de route die de gebruiker wenst. Zo kan de snelste route nog steeds een station meermaals bezoeken, of kan men van een trein afstappen om later op deze zelfde trein weer op te stappen. Een oplossing hiervoor is om het aantal overstappen te beperken[12].

Naast de tijd van aankomst, zijn er nog een aantal andere criteria die vaak geoptimaliseerd worden. Het populairste tweede criterium is het aantal overstappen, gevolgd door de prijs [13]. Optimalisatie van de prijs is echter zeer complex vanwege de complexe tariefplannen bij openbaar vervoer[15]. Dit valt buiten de context van deze masterproef.

2.2.1 Implementatie en aanpassingen

De werking en implementatie van CSA worden behandeld in [13]. Dit algoritme kan zonder veel wijzigingen geïmplementeerd worden in zowel Java[5] als PHP[6].

Wanneer dit algoritme geïmplementeerd wordt merken we echter duidelijke verschillen met de voorgestelde routes door de NMBS. Deze verschillen manifesteren zich vooral in de keuze van het station waar er overgestapt moet worden tussen twee treinen, en de keuze van de tussenliggende treinen indien er meer dan één overstap is. Zo is het mogelijk dat er wordt aangeraden om een trein te nemen langs Brussel-Zuid en Brussel-Centraal tot Brussel-Noord, om van daar een andere

[5] `https://github.com/Bertware/linkedconnections-android-client/blob/master/Hyperrail/src/main/java/be/hyperrail/android/irail/implementation/linkedconnections/RouteResponseListener.java`

[6] `https://github.com/hyperrail/lc2irail/blob/master/app/Http/Repositories/ConnectionsRepository.php`

trein te nemen die op zijn beurt van Brussel-Noord langs Brussel-Centraal naar Brussel-Zuid rijdt. Verder zullen we ook nog aanpassingen doorvoeren om eenvoudig het aantal overstappen te beperken, en om op een betere manier aan *journey-extraction* te doen.

De implementatie en evolutie van het CSA algoritme worden uitgelegd aan de hand van code fragmenten in Java. Er wordt verondersteld dat sectie 4.2 van [13] gekend is. De code is asynchroon, waarbij na het laden van de eerste Linked Connections pagina een callback functie opgeroepen wordt om deze pagina te verwerken. Afhankelijk van het resultaat van deze verwerking, wordt er een nieuwe pagina opgevraagd, of wordt het resultaat doorgegeven aan de oproepende code door middel van callbacks.

Allereerst dienen we twee wijzigingen door te voeren aan de gegevensstructuren. De arrays S en T zijn vervangen door een Map, waardoor we onbeperkt nieuwe stations en trips kunnen toevoegen, en deze kunnen opvragen op basis van hun URI. Dit maakt het algoritme geschikt om te werken rekening houdend met de open world assumption. In de datastructuur voor een voertuig (fragment 2) houden we niet enkel bij wanneer we zouden aankomen, maar ook met hoeveel overstappen (beginnend na het opstappen op deze trein) we zouden aankomen, en waar we moeten afstappen van deze trein. Dit laatste is essentieel om niet enkel de aankomsttijd, maar ook de exacte route met alle overstappen te kunnen weergeven.

Ook de paren van vertrek en aankomsttijd per station, zogenoemde profielen, worden vervangen door een meer uitgebreide gegevensstructuur, zichtbaar in fragment 3. Naast de vertrek en aankomsttijd houden we nu ook de connectie bij waarmee we vertrekken in dit station op dit tijdstip, en de connectie waarmee we aankomen in het volgend station waar we moeten over- of afstappen. Ook het aantal overstappen, beginnend met tellen na het opstappen in dit station, wordt bijgehouden.

Wanneer we de ingeladen connecties willen verwerken, filteren we alle connecties uit de pagina die ofwel te vroeg, ofwel te laat vallen. Zoals te zien in fragment 4 stellen we een flag in wanneer we voorbij de vroegste vertrekdatum zijn. In dit geval zullen we na het overlopen van deze lijst geen nieuwe lijsten meer ophalen. Door deze methode toe te passen kunnen we nieuwe pagina's inladen wanneer deze nodig zijn, zonder te veel op voorhand in te moeten laden.

Het bepalen van T1 en T2 spreekt voor zich, en loopt vrijwel gelijk aan de implementatie uit [13]. In fragment 5 zien we hoe het aantal overstappen wordt bepaald. In het geval dat er geen aankomst mogelijk is (binnen de beperkte tijd) stellen we zowel de aankomsttijd als het aantal overstappen in op een onrealistisch hoog getal. Dit vereenvoudigt de code latere aanzienlijk, aangezien er geen rekening gehouden hoeft te worden met het mogelijk leeg zijn van variabelen.

Bij de bepaling van T3, terug te vinden in fragment 6, maken we de eerste grote afwijking van het

```
class TrainTriple {
  /**
  * The arrival time at the final destination
  */
  DateTime arrivalTime;

  /**
  * The number of transfers until the destination when hopping on to this
↪ train
  */
  int transfers;

  /**
  * The arrival connection for the next transfer or arrival
  */
  LinkedConnection arrivalConnection;
}
```

Code 2: In tegenstelling tot [13] wordt niet enkel de aankomsttijd, maar ook de afstaphalte en het aantal overstappen bijgehouden per trip.

```
class StationQuadruple {
/**
* The departure time in this stop
*/
DateTime departureTime;

/**
* The arrival time at the final destination
*/
DateTime arrivalTime;

/**
* The departure connection in this stop
*/
LinkedConnection departureConnection;

/**
* The arrival connection for the next transfer or arrival
*/
LinkedConnection arrivalConnection;

/**
* The number of transfers between standing in this station and the
↪  destination
*/
int transfers;
}
```

Code 3: In tegenstelling tot [13] wordt niet enkel de vertrek- en aankomsttijd, maar ook het aantal overstappen en de afstaphalte van de volgende trein bijgehouden.

```
if (data.connections.length == 0) {
  mLinkedConnectionsProvider.getLinkedConnectionByUrl(data.previous, this,
  ↪  this, null);
  return;
}

boolean hasPassedDepartureLimit = false;
for (int i = data.connections.length - 1; i >= 0; i--) {
  LinkedConnection connection = data.connections[i];

  if (connection.departureTime.isAfter(mArrivalLimit)) {
    continue;
  }
  if (connection.departureTime.isBefore(mDepartureLimit)) {
    hasPassedDepartureLimit = true;
    continue;
  }

  ...
}
```

Code 4: Connecties worden overlopen volgens dalende vertrektijd. Er worden beperkingen gesteld op vertrek- en aankomsttijd.

```
 if (Objects.equals(connection.arrivalStationUri,
 ↪  mRoutesRequest.getDestination().getSemanticId())) {
  T1_walkingArrivalTime = connection.arrivalTime;
  T1_transfers = 0;
} else {
  T1_walkingArrivalTime = infinite;
  T1_transfers = 999;
}

// Determine T2, the first possible time of arrival when remaining seated
if (T.containsKey(connection.trip)) {
  T2_stayOnTripArrivalTime = T.get(connection.trip).arrivalTime;
  T2_transfers = T.get(connection.trip).transfers;
} else {
  T2_stayOnTripArrivalTime = infinite;
  T2_transfers = 999;
}
```

Code 5: Het aantal overstappen wordt bepaald bij het bepalen van minimale aankomsttijden

oorspronkelijk algoritme. Om te bepalen of een overstap mogelijk is, moeten er reeds profielen voor dit station bekend zijn. Indien dit het geval is, gaan we op zoek naar het profiel waarbij er genoeg tijd is om over te stappen, maar waarbij het aantal overstappen het maximum aantal niet overschrijdt. Wanneer we een overstap vinden die aan deze voorwaarden voldoet, verhogen we het aantal overstappen ook met één. Deze aanpak is eenvoudiger dan de array-gebaseerde aanpak omschreven in [13]. Het voordeel van deze aanpak is dat automatisch alle snelste opties worden bijgehouden, zolang hun aantal overstappen onder het maximum blijft.

In plaats van de door [13] voorgestelde verhoging van de aankomsttijd met één, om zo routes met een gelijke aankomsttijd maar minder overstappen voorkeur te geven, verhogen we hier de aankomsttijd met een vooraf gedefinieerd aantal seconden. Dit aantal geeft aan hoeveel seconden we langer op een trein wensen te zitten, in plaats van over te stappen. Door dit in te stellen op 240, wordt aangegeven dat een route die er tot 4 minuten langer over doet, met een overstap minder, toch de voorkeur krijgt over de snellere route met meer overstappen. Dit is een eerste veld dat door gebruikers ingesteld kan worden om de routes te personaliseren.

Bij het bepalen van de vroegste aankomsttijd (fragment 6), wordt nu ook het aantal overstappen dat bij deze aankomsttijd hoort bepaald, en de connectie waar van de trein afgestapt wordt. We geven bij gelijke aankomsttijden de voorkeur aan overstappen: aangezien de vertrekkende

```
// Determine T3, the time of arrival when taking the best possible transfer
↪  in this station
if (S.containsKey(connection.arrivalStationUri)) {
  int position = S.get(connection.arrivalStationUri).size() - 1;
  StationQuadruple quadruple =
  ↪  S.get(connection.arrivalStationUri).get(position);

  while (
    (quadruple.departureTime.minusSeconds(transferSeconds).getMillis() <=
    ↪  connection.arrivalTime.getMillis() ||
    quadruple.transfers >= maxTransfers) &&
    position > 0
  ) {
    position--;
    quadruple = S.get(connection.arrivalStationUri).get(position);
  }
  if (quadruple.departureTime.minusSeconds(transferSeconds)
      .isAfter(connection.arrivalTime) &&
      quadruple.transfers <= maxTransfers) {
    T3_transferArrivalTime =
    ↪  quadruple.arrivalTime.plusSeconds(extraTimeInsteadOfTransfer);
    // Using this transfer will increase the number of transfers with 1
    T3_transfers = quadruple.transfers + 1;
  } else {
    // When there isn't a reachable connection, transferring isn't an
    ↪  option
    T3_transferArrivalTime = infinite;
    T3_transfers = 999;
  }
} else {
  // When there isn't a reachable connection, transferring isn't an option
  T3_transferArrivalTime = infinite;
  T3_transfers = 999;
}
```

Code 6: Bij een eventuele overstap worden ook extra factoren in rekeningen gebracht.

voertuigen volgens dalende vertrektijd overlopen worden, geven we dus de voorkeur aan zo vroeg mogelijk overstappen. Dit geeft extra marge binnen de trip. Door de extra toevoegingen voor

```
DateTime Tmin;
LinkedConnection exitTrainConnection;
int numberOfTransfers;

if (T3_transferArrivalTime.getMillis() <= T2_stayOnTripArrivalTime.getMillis())
↪  {
  Tmin = T3_transferArrivalTime;
  exitTrainConnection = connection;
  numberOfTransfers = T3_transfers;
} else {
  Tmin = T2_stayOnTripArrivalTime;
  if (T2_stayOnTripArrivalTime.isBefore(infinite)) {
    exitTrainConnection = T.get(connection.trip).arrivalConnection;
  } else {
    exitTrainConnection = null;
  }
  numberOfTransfers = T2_transfers;
}
// For equal times, we prefer just arriving.
if (T1_walkingArrivalTime.getMillis() <= Tmin.getMillis()) {
  Tmin = T1_walkingArrivalTime;
  exitTrainConnection = connection;
  numberOfTransfers = T1_transfers;
}

if (Tmin.isEqual(infinite)) {
  continue;
}
```

Code 7: Bepalen van de vroegste aankomsttijd

journey extraction en het optimaliseren van de routes, is het bijwerken van de gegevenstructuren aanzienlijk ingewikkelder vergeleken met de originele implementatie. Voor voertuigen houden we niet langer enkel de aankomsttijd, maar ook de afstap halte bij. Hierbij verkiezen we de halte waarlangs we zo snel mogelijk aankomen, maar bij gelijke aankomsttijd wensen we een zo lang mogelijke periode voor de overstap. Wanneer de aankomsttijd gelijk is, onderzoeken we of de nieuwe afstap halte (de connectie die op dit moment onderzocht wordt) meer tijd voor een

overstap geeft. Indien dit het geval is, werken we de afstap halte bij. Het bijwerken van een bestaande trip is zichtbaar in fragment 8.

Het bijwerken van de stopprofielen, zichtbaar in fragment 9, is lichtjes aangepast om de efficiëntie te verhogen. De vroegste vertrekken worden nu achteraan toegevoegd. Door deze aanpassing, en het gegeven dat de vertrektijd van de huidige connectie gelijk of kleiner dan de vertrektijd van alle vorige connecties is, hoeven we nu enkel het laatste profiel in de lijst te evalueren. Als de vertrektijd kleiner of gelijk is, moet de aankomsttijd kleiner zijn. we controleren dus enkel of de aankomsttijd kleiner is, en zo ja, of de vertrektijd kleiner of gelijk is. Afhankelijk van deze laatste controle voegen we een nieuw item toe aan de lijst, of vervangen we het laatste. Aangezien we telkens enkel toevoegen wanneer de aankomsttijd vroeger ligt, zal deze lijst altijd gesorteerd zijn volgens dalende aankomsttijd. Hiermee is bewezen dat deze optimalisatie correct is, en een beter alternatief voor het overlopen van de volledige lijst.

De lijst met volledige routes reconstrueren (fragment 10) is relatief eenvoudig. Voor elk profiel horend bij de stoplocatie van waar de reiziger vertrekt, volgen we de vertrek- en aankomst-connecties. Om bij elke tussenstop de juiste connectie te vinden waarmee de reis verder zal gezet worden, vergelijken we de aankomsttijd uit het stopprofiel waaruit we vertrokken, met de aankomsttijden uit de stopprofielen van de tussenstop (fragment 11). Wanneer deze gelijk zijn, hebben we het volgende deel van de reis gevonden.

```
if (Tmin.isEqual(T.get(connection.trip).arrivalTime) &&
  T3_transferArrivalTime.isEqual(T2_stayOnTripArrivalTime) &&
  S.containsKey(T.get(connection.trip).arrivalConnection.arrivalStationUri)
  ↪ &&
  S.containsKey(connection.arrivalStationUri)
) {
  LinkedConnection currentTrainExit =
  ↪ T.get(connection.trip).arrivalConnection;

  StationQuadruple quad = new StationQuadruple();
  quad.departureTime = connection.departureTime;
  quad.departureConnection = connection;
  quad.arrivalTime = Tmin;

  // Current situation
  quad.arrivalConnection = currentTrainExit;
  Duration currentTransfer = new Duration(currentTrainExit.arrivalTime,
  ↪ getFirstReachableConnection(quad).departureTime);
  // New situation
  quad.arrivalConnection = exitTrainConnection;
  Duration newTransfer = new Duration(exitTrainConnection.arrivalTime,
  ↪ getFirstReachableConnection(quad).departureTime);

  if (newTransfer.isLongerThan(currentTransfer)) {
    TrainTriple triple = new TrainTriple();
    triple.arrivalTime = Tmin;
    triple.arrivalConnection = exitTrainConnection;
    triple.transfers = numberOfTransfers;
    T.put(connection.trip, triple);
  }
}

if (Tmin.isBefore(T.get(connection.trip).arrivalTime)) {
  TrainTriple triple = new TrainTriple();
  triple.arrivalTime = Tmin;
  triple.arrivalConnection = exitTrainConnection;
  triple.transfers = numberOfTransfers;
  T.put(connection.trip, triple);
}
```

Code 8: Bijwerken van de trips gegevensstructuur.

```
StationQuadruple quad = new StationQuadruple();
quad.departureTime = connection.departureTime;
quad.arrivalTime = Tmin;

// Additional data for journey extraction
quad.departureConnection = connection;
quad.arrivalConnection = T.get(connection.trip).arrivalConnection;
quad.transfers = numberOfTransfers;

if (S.containsKey(connection.departureStationUri)) {
  int numberOfPairs = S.get(connection.departureStationUri).size();
  StationQuadruple existingQuad =
  ↪  S.get(connection.departureStationUri).get(numberOfPairs - 1);
  if (quad.arrivalTime.isBefore(existingQuad.arrivalTime)) {
    if (quad.departureTime.isEqual(existingQuad.departureTime)) {
      S.get(connection.departureStationUri).remove(numberOfPairs - 1);
      S.get(connection.departureStationUri).add(numberOfPairs - 1, quad);
    } else {
      S.get(connection.departureStationUri).add(quad);
    }
  }
} else {
  S.put(connection.departureStationUri, new ArrayList<StationQuadruple>());
  S.get(connection.departureStationUri).add(quad);
}
```

Code 9: Bijwerken van de stops gegevensstructuur.

```
// Results? Return data
Route[] routes = new
↪  Route[S.get(mRoutesRequest.getOrigin().getSemanticId()).size()];

int i = 0;
for (StationQuadruple quad :
↪  S.get(mRoutesRequest.getOrigin().getSemanticId())
) {
  // it will iterate over all legs
  StationQuadruple it = quad;
  List<RouteLeg> legs = new ArrayList<>();

  while (!Objects.equals(it.arrivalConnection.arrivalStationUri,
  ↪  mRoutesRequest.getDestination().getSemanticId())) {
    // use it.departureConnection and it.arrivalConnection to construct
    ↪  legs of this journey
    legs.add(...);
    it = getFirstReachableConnection(it);
  }

  routes[i++] = new Route(legs);
}
```

Code 10: Journey Extraction door middel van post-processing

```
private StationQuadruple getFirstReachableConnection(StationQuadruple
↪  arrivalQuad) {
  List<StationQuadruple> it_options =
  ↪  S.get(arrivalQuad.arrivalConnection.arrivalStationUri);
  int i = it_options.size() - 1;
    while (i >= 0 && it_options.get(i).arrivalTime.getMillis() !=
    ↪  arrivalQuad.arrivalTime.getMillis() - 240 * 1000) {
    i--;
  }
  return it_options.get(i);
}
```

Code 11: Vinden van volgende vertrek bij tussenstop

2.3 Vertrekken en aankomsten per station

Om de zogenoemde *liveboards* te berekenen, gebruiken we een eenvoudig algorithme. We overlopen alle pagina's, en houden enkel de stops bij die betrekking hebben op het gezochte station. Hiervoor gebruiken we twee lijsten voor respectievelijk de vertrekkende en aankomende connecties. We blijven pagina's overlopen tot de stopvoorwaarde is bereikt. Deze verschilt afhankelijk van de locatie waar we het algoritme implementeren:

- Op een webserver blijven we extra pagina's van de schijf laden tot het gewenst aantal resultaten is bereikt, of het gewenste tijdsinterval overlopen is.
- Bij een lokale implementatie stoppen we zodra we de pagina hebben afgewerkt waarin de eerste stop beschreven wordt. We maken gebruik van incrementele resultaten, waarbij telkens zo snel mogelijk een klein resultaat wordt berekend. Hierdoor krijgt de gebruiker (in theorie) sneller resultaten te zien.

Wanneer de stopvoorwaarde is bereikt, splitsen we de lijsten in drie soorten stops:

- Vertrekhalte: dit zijn de stops van voertuigen die deze stopplaats als begin van hun traject hebben en dus geen informatie over een aankomst bevatten
- Eindhalte: dit zijn de stops van voertuigen die deze stopplaats als laatste op hun traject hebben en dus niet meer vertrekken
- Stops: dit zijn de stops van voertuigen die een tussenstop maken

Om te bepalen tot welke categorie een connectie behoort, dienen we voor elke aankomende connectie in het station te bepalen of er een vertrekkende connectie mee overeenkomt. De connecties die vertrekken vanuit het station, en waarmee geen aankomst overeenkomt, zijn dan de vertrekhaltes. Als we m nemen voor het aantal aankomende, en n voor het aantal vertrekkende voertuigen, zou dit algoritme een efficiëntie hebben van $O(mn)$. We kunnen hier echter gebruik maken van de chronologische volgorde van alle connecties: een vertrekkende connectie zal altijd na een eventuele aankomst volgen. We houden bij elk ontdekte aankomst dus bij hoeveel vertrekken er al ontdekt zijn. Vervolgens zullen we enkel de vertrekken na elke aankomst overlopen.

Intuïtief is duidelijk dat deze aanpak efficiënter is. Dit kan ook bewezen worden: aankomsten en vertrekken zijn ongeveer gelijk gespreid. Bij een stijgende index i in de lijst met aankomende connecties, zal ook de index j stijgen vanaf waar de lijst met vertrekken doorzocht worden. In de praktijk zijn i en j ongeveer gelijk aan elkaar: het verschil is bij een normale spreiding van de

aankomsten en vertrekken verwaarloosbaar. De benodigde tijd voor het bepalen van de types is nu

$$\Sigma_{i=0}^{i=m}(n-j) = m(n-i) = \frac{m*n}{2} = O(mn)$$

Terwijl de asymptotische efficiëntie gelijk blijft, zien we wel dat de verborgen constante gehalveerd wordt. Terwijl deze verbetering geen merkbaar verschil in uitvoeringsduur oplevert bij enkele honderden connecties, kan dit wel voor betere prestaties zorgen op grote schaal, zoals wanneer data in een groot tijdsinterval verwerkt worden. Nu zijn stations niet oneindig groot, en hebben treinen in de meeste gevallen meer dan 10 haltes. De meeste stops zullen dus geen begin of einde van de rit voorstellen, en een trein moet tijdig vertrekken om plaats te maken voor de volgende trein in het station. We kunnen dus veronderstellen dat het vertrek kort op de aankomst volgt: dit is in de meeste gevallen zo. Zodra we het bijhorend vertrek gevonden hebben, kunnen we stoppen met de lijst te doorzoeken. Nu moeten we slechts k connecties overlopen tot bijhorend vertrek gevonden is, met k element van N en k ongeveer gelijk aan het aantal perrons in het station. $m(n-i)$ wordt nu $m(k)$, wat asymptotisch gezien gelijk is aan $O(m)$.

2.4 Route van een voertuig

De route van een voertuig af te leiden uit Linked Connections data is relatief eenvoudig. Zowel de stops van een trein, als de connecties, zijn chronologisch geordend. We overlopen dus alle connecties, waarbij we telkens de laatst gebruikte connectie p en de huidige connectie c in variabelen bijhouden.

- Wanneer we de eerste connectie ontdekken die betrekking heeft op deze trein, is dit de vertrekhalte. We verwerken deze data en voegen dit toe aan de lijst met stops. We kopiëren c naar p.
- Wanneer p een connectie bevat, en c de hierop volgende connectie bevat, beschikken we over alle informatie om de tussenstop tussen p en c toe te voegen aan de lijst met stops.
- Wanneer alle connecties overlopen zijn, was c de laatste connectie van dit trein. Deze connectie duidt dus de eindhalte aan. We voegen deze toe aan de lijst met stops, en geven een antwoord terug aan de oproepende code.

Het bepalen van de route van een trein legt een eerste mogelijk pijnpunt bloot: hiervoor moeten alle connecties voor een volledige dag opgehaald worden. Er is immers geen enkele manier om te bepalen wanneer een trein vertrekt, of wat de eindhalte is. Hierdoor is er een relatief lange laadtijd om alle fragmenten op te halen, wat enigszins beperkt kan worden door deze asynchroon te laden. Het effect hiervan op de gebruikerservaring zullen we later onderzoeken.

Bibliografie

[1] M. Boyd. (2014) How smart cities are using APIs: Public transport APIs. [Online]. Available: https://www.programmableweb.com/news/how-smart-cities-are-using-apis-public-transport-apis/2014/05/22

[2] O. K. International. (2018) What is open? Open Knowledge Foundation. [Online]. Available: https://okfn.org/opendata/

[3] R. Verborgh, O. Hartig, B. De Meester, G. Haesendonck, L. De Vocht, M. Vander Sande, R. Cyganiak, P. Colpaert, E. Mannens, and R. Van de Walle, "Querying datasets on the web with high availability," in *Lecture Notes in Computer Science*, vol. 8796. Springer, 2014, pp. 180–196. [Online]. Available: http://linkeddatafragments.org/publications/iswc2014.pdf

[4] P. Colpaert, A. Llaves, R. Verborgh, O. Corcho, E. Mannens, and R. V. D. Walle, "Intermodal public transit routing using linked connections," vol. 1486, 2015. [Online]. Available: http://ceur-ws.org/Vol-1486/paper_28.pdf

[5] R. Verborgh, M. Vander Sande, O. Hartig, J. Van Herwegen, L. De Vocht, B. De Meester, G. Haesendonck, and P. Colpaert, "Triple pattern fragments: a low-cost knowledge graph interface for the web," *Journal of web semantics*, vol. 37-38, pp. 184–206, 2016. [Online]. Available: http://dx.doi.org/10.1016/j.websem.2016.03.003

[6] R. T. Fielding and R. N. Taylor, "Principled design of the modern web architecture," 1999. [Online]. Available: https://www.ics.uci.edu/~fielding/pubs/fse99_webarch.pdf

[7] B. Vanhaelewyn and L. D. Marez, "Imec.digimeter 2017." [Online]. Available: https://www.imec.be/digimeter

[8] P. van Ammelrooy. Onbeperkt smartphone-abbonement steeds meer in trek onder nederlanders. [Online]. Available: https://www.volkskrant.nl/economie/onbeperkt-smartphone-abonnement-steeds-meer-in-trek-onder-nederlanders~a4532349/

[9] P. Colpaert, "Publishing transport data for maximum reuse," Ph.D. dissertation, Ghent University, 2017. [Online]. Available: https://phd.pietercolpaert.be

[10] J. A. Rojas Melendez, D. Chaves, P. Colpaert, R. Verborgh, and E. Mannens, "Providing reliable access to real-time and historic public transport data using linked connections," vol. 1931, 2017, pp. 1–4. [Online]. Available: https://biblio.ugent.be/publication/8540883/file/8540885.pdf

[11] J. Dibbelt, T. Pajor, B. Strasser, and D. Wagner, "Intriguingly simple and fast transit routing," in *Experimental Algorithms*, V. Bonifaci, C. Demetrescu, and A. Marchetti-Spaccamela, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 43–54. [Online]. Available: https://pdfs.semanticscholar.org/a892/a54b02ce112e1302931231141a8b676b873b.pdf

[12] B. Strasser and D. Wagner, "Connection Scan Accelerated," in *2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2014, pp. 125–137. [Online]. Available: https://epubs.siam.org/doi/pdf/10.1137/1.9781611973198.12

[13] J. Dibbelt, T. Pajor, B. Strasser, and D. Wagner, "Connection Scan Algorithm," *CoRR*, vol. abs/1703.05997, 2017. [Online]. Available: http://arxiv.org/abs/1703.05997

[14] P. Colpaert. (2018) Linked connections: What is it? [Online]. Available: https://linkedconnections.org/#what

[15] M. Müller-Hannemann and M. Schnee, "Paying less for train connections with MOTIS," in *OASIcs-OpenAccess Series in Informatics*, vol. 2. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2006.

Bijlagen