Obsah

[3 Implementácia 2](#_Toc37239657)

[3.0 Technológie 2](#_Toc37239658)

[3.1 Dáta 2](#_Toc37239659)

[3.1.1 Parsovanie dát zo súboru 2](#_Toc37239660)

[3.1.2 Použitie databázy 2](#_Toc37239661)

[3.1.3 Testovacie dáta 3](#_Toc37239662)

[3.2 Dátová štruktúra 3](#_Toc37239663)

[3.2.1 Vytvorenie dátovej štruktúry 3](#_Toc37239664)

[3.2.2 Implementácia dátovej štruktúry 3](#_Toc37239665)

[3.2.3 Naplnenie dátovej štruktúry 4](#_Toc37239666)

[3.3 Čas a časový simulátor 4](#_Toc37239667)

[3.3.1 Trieda Time 4](#_Toc37239668)

[3.4 Algoritmus 4](#_Toc37239669)

[3.4.1 Označovanie zastávok medzi kolami 4](#_Toc37239670)

[3.4.3 Ako sme prisposobili algoritmus s dátovou štruktúrou 5](#_Toc37239671)

[3.4.4 Vyhľadávanie tripu, static vs real 5](#_Toc37239672)

[3.4.5 Filtrovanie 6](#_Toc37239673)

[3.4.6 Zapracovanie používateľských preferencií. 7](#_Toc37239674)

[3.4.7 Naledujúce cesty 8](#_Toc37239675)

[3.5 Frontend 8](#_Toc37239676)

[3.5.1 Implementácia frontendovej časti 8](#_Toc37239677)

[3.5.2 Zobrazovanie zastávok na mape 9](#_Toc37239678)

[3.5.3 Práca s časom 9](#_Toc37239679)

[3.5.4 Zobrazovanie časov ciest 9](#_Toc37239680)

# 3 Implementácia

V tejto kapitole popíšeme technológie, ktoré sme vybrali na implementáciu aplikácie. Ďalej popíšeme základné problémy, ktoré vznikli pri spracovaní dát, pri implementácii dátovej štruktúry a navrhovaného algoritmu. Ukážkami kódov priblížime, ako sme tieto problémy riešili.

## 3.0 Technológie

Serverovú časť aplikácie sme implementovali v jazyku Java, konkrétne vo verzii Java11 s využitím frameworku SpringBoot. Na mapovanie Java objektov na databázové entity sme použili ORM framework Hibernate. Pre komunikáciu s frontendom sme si vybrali REST architektúru. Táto architektúra je dokumentovaná pomocou technológie Swagger.

Na frontendovej strane sme si zvolili progresívnu webovú aplikáciu a na jej implementáciu sme použili moderný JavaScriptový framework Vue.js, konkrétne verziu Vue 2. Zároveň sme použili knižnicu Vuex na spravovanie stavu aplikácie.

## 3.1 Dáta

### 3.1.1 Parsovanie dát zo súboru

Náš projekt obsahuje dáta vo formáte GTFS, ktoré reprezentujú cestovné poriadky v bratislavskej MHD platné pre obdobie v minulosti.

Dáta zo súboru stops.txt sme zachytili v 3 rôznych objektoch. Polohu zastávky sme vyčlenili do zvlášť objektu Coords. Zastávky s rovnakým názvom sme zoskupili do objektu StopArea. Samotný objekt Stop zachytáva zónu a informáciu o tom, či je zastávka na znamenie.

Keďže v dátach nemáme informáciu o peších presunoch medzi jednotlivými zastávkami, implementovali sme FootPathsFileGenerator. Táto služba nám využitím Google Distance Matrix API zistí vzdialenosti medzi jednotlivými zastávkami a vytvorí súbor foot\_paths.txt. Súbor obsahuje len také dvojice zastávok, ktoré sme definovali v sekcii REFERENCIA NA PESIE PRESUNY. Dáta z tohto súboru sú v namapované na objekty FootPath.

Každý dátum z rozmedzia dátumov platnosti daného cestovného poriadku potrebujeme zaradiť do pod nejaký typ dňa (pracovné dni, víkendy, školské prázdniny, štátne sviatky,...). Z dát uložených v  GTFS štandarde sa komplikovaným spôsobom zisťuje, ktoré typy dní prislúchajú konkrétnemu dátumu. Objekty CalendarDate budú uchovávať dátumy z rozmedzia dátumov platnosti cestovného poriadku. V objektoch ServiceDay budú uložené jednotlivé typy dní.

Dáta zo súborov trips.txt, routes.txt a stop\_times.txt len mapujeme na objekty Trip, Route a StopTime. Zmenou je, že informáciu o tom, či je zastávka na znamenie si neudržujeme v objekte StopTime, ale priamo v objekte Stop. V dátach chýbala informácia o tom, či je jazda linky vedená nízkopodlažným vozidlom. Parameter lowFloor náhodne generujeme pre jednotlivé jazdy do objektu Trip.

### 3.1.2 Použitie databázy

V projekte sme sa rozhodli použiť relačná SQL databázu na uchovanie všetkých dát a vzťahov, ktoré vytvárame a vkladáme pri prvotnom spustení aplikácie. Pôvodne sme plánovali z databázy vytvárať dátovú štruktúru pre algoritmus, keďže v databáze sú zachytené vzťahy medzi dátami.

Počas implementácie sme však zistili, že voľba ORM framework Hibernate nebola veľmi dobrou voľbou. Mysleli sme si, že Hibernate bude postačujúci pre jednorazové vloženie údajov do databázy a následne pre jednoduché čítanie údajov bez nutnosti použitia komplikovaných SQL príkazov. Vzťahy medzi údajmi sú však komplexné a údajov je veľké množstvo. Najväčšiu záťaž sme pocítili pri čítaní všetkých údajov z tabuľky trips, ktorá je prepojená so všetkými ostatnými tabuľkami. Tieto údaje však potrebujeme na vytvorenie dátovej štruktúry. Z toho dôvodu sme sa rozhodli, že z databázy budeme ťahať údaje pre frontend. Napríklad na zobrazenie všetkých liniek a zastávok, an ktorých stoja. Ďalej na zobrazenie všetkých zastávok spolu so súradnicami pre možnosť vyberania zastávky z mapy alebo všetky skupiny zastávok pre vyberanie zastávky zo zoznamu.

### 3.1.3 Testovacie dáta

Dátová sada vo formáte GTFS obsahuje veľa záznamov. Už pri implementácii bolo potrebné myslieť na to, ako budeme overovať správnosť mapovania dát do databázy, korektnosť vytvorenej dátovej štruktúry a v neposlednom rade vedieť ohodnotiť a upraviť algoritmus tak, aby počítal čo najoptimálnejšie cesty. Pri veľkej dátovej sade sa ťažko testuje správnosť implementácie a preto sme pred samotnou implementáciou navrhli zmenšenú testovaciu sadu dát.

Táto sada obsahuje 7 skupín zastávok (stopArea), 16 zastávok, 3 linky, 70 jázd a 185 záznamov stopTimes. Pre porovnanie reálna sada obsahuje až vyše 700 000 stopTimes. Okrem GTFS dát sme vytvorili aj pár testovacích záznamov o meškaní.

Pre jednoduché prepínanie medzi dvomi dátovými sadami aplikácie sme použili profily z frameworku Spring. Každý profil má definované premenné prostredia, ktoré pre náš projekt definujú napríklad údaje pre dátové pripojenie ako názov databázy, používateľa a heslo do databázy. Ďalej sú to cesty k GTFS súborom a dátam o meškaniach.

## 3.2 Dátová štruktúra

### 3.2.1 Vytvorenie dátovej štruktúry

Dátová štruktúra sa vytvára pri spustení serverovej aplikácie. Najskôr sme začali implementovať vytvorenie dátovej štruktúry podľa návrhu, teda ťahaním údajov z databázy. Z dôvodu pomalého ťahania údajov z databázy sme sa rozhodli obísť databázu a dátovú štruktúru vytvárame hneď pri parsovaní súboru. Tento proces trvá približne 15-20 minút v závislosti od zariadenia. Toto trvanie je prijateľné pri produkcii keďže aplikácia sa bude spúšťať len pri aktualizácii cestovných poriadkoch a v prípade nejakých výpadkov. V procese implementácie je však 20 minútové čakanie veľmi obmedzujúce pri každom spustení aplikácie. Vytvorený a naplnený model sme sa rozhodli serializovať do súboru. Proces deserializácia už trval do jednej minúty.

Ak by sme premýšľali o vyhľadávaní offline potrebovali by sme vyhľadávať na klientskej strane a dátovú štruktúru tiež udržiavať na klientskej strane. Otázkou je kedy by sa dátová štruktúra vytvárala. Ak by sa vytvárala len pri prvotnej inštalácii alebo po aktualizácii cestovných poriadkov dátová štruktúra by zaberala veľa pamäte. Serializovaný model dátovej štruktúry má takmer 0,5 GB. Druhá možnosť je načítavať dátovú štruktúru pri každom spustení aplikácie, čo by mohlo trvať neprimerane dlho. Síce sú frontendové jazyky rýchlejšie a objekty sú menšie, stále by mohol byť problém s vytvorením dátovej štruktúry a pamäťou, ktorá aplikácia zaberá. Zostávame teda pri vyhľadávaní spojov a udržiavaní dátovej štruktúry na serverovej strane ako aj mnohé existujúce aplikácie.

### 3.2.2 Implementácia dátovej štruktúry

Pre celý beh aplikácie bude potrebná len jedna inštancia dátovej štruktúry. Pri spustení aplikácie sa vytvorí aplikačný kontext, ktorý reprezentuje množinu prepojených komponentov. Tieto komponenty manažuje IoC kontainer. Objekt DataStructure je označený anotáciou @Component, ktorý pri autowirovaní vytvorí DataStructureModel načítaním z GTFS dát v produkčnom prostredí alebo deserializovaním zo súboru v implementáčnej fáze. @Component je vlastne @Bean a ten sa v Springu správa ako Singleton.

Trieda DataStrctureModel definuje dátovú štruktúru pre RAPTOR algoritmus, ktorá je tvorená šiestimi dátovými štruktúrami.

public class DataStructureModel implements Serializable {

private Map<Route, List<Subroute>> routeSubroutes = new HashMap<>();

private Map<Long, List<Transfer>> stopTransfers = new HashMap<>();

private Map<Stop, List<Subroute>> stopSubroutes = new HashMap<>();

private Map<String, Map<Long, Integer>> stopIndexInSubroute = new HashMap<>();

private List<CalendarDate> calendarDates = new ArrayList<>();

...

}

Štruktúra routeSubroutes je definovaná mapou, kde kľúčom je linka r a hodnota je pole úsekov, ktoré patria linke r. Kľúčom v stopTransfers je identifikačné číslo zastávky p a hodnotou je pole peších prestupov, pričom začiatočná zastávka je zastávka p. Mapa stopSubroutes ma zastávku p ako kľúč a hodnotou je pole úsekov linky, ktoré stoja na zastávke p. Štruktúra stopIndexInSubroute poskytuje poradové číslo zastávky v rámci úseku linky. Posledné je pole objektov CalendarDate, ktoré poskytuje všetky dátumy z rozsahu platnosti cestovných poriadkov a ku každému dátumu d je priradená množina typov dní.

### 3.2.3 Naplnenie dátovej štruktúry

...

## 3.3 Časový simulátor

Keďže máme k dispozícii historické dáta o meškaní vozidiel a platnosť cestovných poriadkov je tiež v minulosti, potrebujeme v aplikácii simulovať čas v minulosti. V konfigurácii nastavujeme čas, ktorý, aplikácia začína spustením. Na prelome dňa sa čas vynuluje a navýši sa dátum o jeden deň.

Pri spustení klienta sa klient dopytuje na aktuálny čas servera. Klientská strana pokračuje v navyšovaní času po sekunde rovnako ako serverová strana. Klient potrebuje mať stále aktuálny čas, aby vedel obmedziť vyhľadávanie do minulosti. Keďže pracujeme s reálnymi dátam, pri vyhľadávaní do minulosti by mohol nastať problém.

Pri prvotnom spustení serverovej strany zároveň aktualizuje e dátovú štruktúru dátami o meškaní. Podľa aktuálneho dňa vyhľadáme súbor, ktorý obsahuje meškania. Zo súboru vytiahneme všetky záznamy od začiatku dňa po aktuálny čas. Podľa získaných záznamov nastavíme meškajúcim linkám hodnotu meškania v dátovej štruktúre. S každou nasledujúcou pribudnutou minútou hľadáme už len také záznamy, ktoré vznikli v predchádzajúcej minúte.

### 3.3.1 Vlastná trieda Time

Na serverovej strane sme si vytvorili vlastnú triedu Time, keďže s časom potrebujeme často pracovať nezávisle od dátumu. Jej parametre sú: hours, minutes, seconds, nextDay a prevDay. Trieda umožňuje skonštruovať čas z textového reťazca. Takto definovaný čas prichádza z GTFS dát, ktorý určuje kedy konkrétna jazda linky stojí na zastávke. Triedu čas vieme skonštruovať aj priamo kombináciou vymenovaných parametrov. Rovnako vieme skonštruovať maximálny čas: 23:59:59, ktorý potrebujeme definovať pri inicializácii výsledkov RAPTOR algoritmu. Časy vieme medzi sebou porovnávať alebo k nim odpočítavať a pripočítavať minúty.

Parameter nextDay slúži na porovnávanie časov, keďže v dátach sa vyskytujú aj časy nad 23:59. Metóda porovnávania časov prihliada aj na parameter nextDay. Tento parameter navyšujeme aj vtedy, keď pri pripočítavaní minút k času presiahneme maximálny čas.

Parameter prevDay používame pri hľadaní jazdy v dátovej štruktúre podľa záznamov o meškaní vozidiel. Ak máme záznam v čase t=0:00 a vozidlo mešká 1 minútu, od času t je potrebné odpočítať 1 minútu. Dostávame sa tak do predchádzajúceho dňa, kde je potrebné hľadať prislúchajúcu cestu.

## 3.4 Algoritmus

### 3.4.1 Označovanie zastávok medzi kolami

Algoritmus nepočíta s tým, že existuje možnosť na zo začiatočnej zastávky riešiť pešie presuny, keďže pole označených zastávok sa vyprázdni už pri inicializácii množiny Q a pridávajú ich do množiny len ak zlepšíme nejaký čas. Ten chalan to má dobre. Iniciálne marked stops načíta zo serchmarams – nulté kolo. Potom v prvom kole marked stops nevymazáva pri vytváraní Q. A tým pádom sa pri trasovaní peších presunov hľadajú a tie z pôvodných zastávok. Množinu marked stops meníme až po prvom kole a sú to tie zastávky, ktoré majú v tom kole iný čas ako nekonečno. Toto môže byť ale oveľa viac zastávok ako keď sa išlo pôvodným štýlom. Bolo by dobré držať si nejakú množinu markedStopsInNewRound a do nej pridávať tie zastávky pri každom zlepšení a na záver tieto zastávky vložiť do markedStops. Tak budem mať tie pôvodné markedStops aj pri traverzovaní peších presunov a zároveň markedStops budú len tie, ktoré boli v predchádzajúcom kole zlepšené. Tomu šuhajkovi to funguje asi preto lebo on nekopíruje pri inicializácii nového kola všetko z predchádzajúceho kola. Čo je možno aj dobre lebo vie v ktorom kole čo vzniklo.

### 3.4.2 Výsledky RAPTOR algoritmu

RAPTOR algoritmus operuje v kolách a v každom kole vylepšuje časy zastávkam. Potrebujeme si pre každé kolo držať pole zastávok. Pre každú zastávku a každé kolo existuje jeden najlepší čas. Tento čas si budeme držať v slovníku roundArrivals, ktoré je definované:

Map<Integer, List<StopArrivalTime>> roundArrivals;

Kľučom je kolo a jeho hodnotou je pole objektov StopArrivalTime. Tento objekt zjednocuje zastávku a čas príchodu na zastávku.

Pri vyhodnocovaní však budeme potrebovať hľadať podľa zastávky najlepší čas a preto si udržujeme ešte jeden slovník, ktorý je definovaný:

Map<Stop, TimeRound> bestArrivals;

Aby sme po zbehnutí algoritmu vedeli vytvoriť cesty z výsledkov, potrebujeme si pamätať, ako sme pre zastávku získali  vylepšený čas v každom kole. Preto sme definovali slovník roundActions.

Map<Stop, List<RoundTimeAction>> roundActions;

Slovník pre každú zastávku udržuje pole objektov RoundTimeAction. Tento objekt je tvorený indexom kola, časom a objektom Action. Objekt Action je určený dvomi zástavkami a typom akcie, čo môže byť buď peší presun alebo úsek jazdy. V prípade pešieho presunu si pamätáme, aké je jeho trvanie a v prípade úseku jazdy si pamätáme identifikačné číslo jazdy. Pre konkrétnu zastávku a kolo môže existovať viac akcií, keďže najlepší čas môžeme dosiahnuť dvomi rôznymi akciami. Pred vytváraním ciest tieto akcie ešte prefiltrujeme, pretože môže sa stať, že pre zastávku sme dosiahli čas 6:00 v kole 2 ale zároveň sme tento čas dosiahli v kole 3. Druhý záznam je nevýhodnejší takže neho môžeme odfiltrovať ešte pred vytváraním ciest.

Z takto vyfiltrovaných akcií tvoríme rekurzívne cesty s tým, že začíname v konečnej zastávke a končíme, ak sme dosiahli niektorú zo začiatočných zastávok. V prípade, že vyhľadávanie začínalo v aktuálnej lokalite, pripojíme k ceste na začiatok ešte prvotný peší presun z aktuálnej lokality do začiatočnej zastávky.

#### 3.4.2.1 Target prunning

Do výsledkov RAPTOR algoritmu sme pridali ešte jeden slovník, ktorý uchováva pre každú skupinu zastávok (stopArea) najlepší čas pre každé kolo.

Map<StopArea, TimeRound> stopAreaBestArrivals;

Týmto znížime počet označených zastávok a rovnako aj výpočtový čas. Keďže my hľadáme cesty z jednej stopArea do druhej stopArea a nie medzi zastávkami, vieme obmedziť prehľadávanie. Ak sme sa už do niektorej zo zastávok v spoločnej stopArea dostali skorej, čas nevylepšíme a zastávku neoznačíme ako prestupné miesto na ďalšie prehľadávanie.

### 3.4.3 Ako sme prisposobili algoritmus s dátovou štruktúrou

subroutes, serviceDays

### 3.4.4 Hľadanie najbližšej vyhovujúcej jazdy

RAPTOR algoritmus funguje tak, že hľadáme najskoršiu vyhovujúcu jazdu po čase $\tau$ ktorá stojí na zastávke p a jazdí v niektorý zo zadaných typov dní (serviceDay). Aby sme nemuseli prechádzať všetky jazdy v linke hľadanie jazdy funguje tak, že začíname od poslednej jazdy v utriedenom poli. Spoliehame sa, že jazdy sú utriedené podľa času príchodu na prvú zastávku vzostupne. Kým nie je čas menší ako čas $\tau$, iterujeme jazdy smerom na začiatok poľa a do premennej lastTrip si ukladáme vždy aktuálnu jazdu. V momente keď narazíme na jazdu, ktorá na zastávke p stojí skôr ako je čas $\tau$, máme v premennej lastTrip vyhovujúcu jazdu. Môže nastať prípad, že všetky jazdy stoja na zastávke p skôr ako v čase $\tau$, preto sme museli pridať podmienku, ktorá overí, či príchod poslednej jazdy na zastávku p je neskôr ako čas $\tau$.

#### 3.4.4.1 Zoradenie jázd v rámci linky

Už pri vytváraní dátovej štruktúry sme riešili ako správne zotriediť jazdy v linke. Najskôr sme sa rozhodli využiť atribút sequence order, ktorý je priradený ku každému objektu StopTime. StopTime okrem iného definuje v akom čase stojí jazda linky na zastávke. Atribút sequence order by mal určiť poradie zastávky v rámci jazdy a zároveň aj v rámci celého dňa. Prvá jazda linky na prvej zastávke má sequence order 1 a toto číslo sa navyšuje až po poslednú zastávku poslednej jazdy linky. Vieme teda vypočítať poradie jazdy v linke. Po hlbšom skúmaní sme zistili, že atribút sequence order nezoraďuje jazdy linky tak, ako by sme to potrebovali pre náš algoritmus. Zároveň sme zistili, že časy príchodu jazdy na zastávku sú v niektorých prípadoch väčšie ako 23:59:59 viď obrázok. Dokonca existujú také jazdy, ktoré prichádzajú na všetky zastávky linky až na druhý deň. Tieto výnimky v dátach nám po menšej analýze nedávali zmysel a keďže sme boli už v pokročilej fáze vývoja algoritmu, rozhodli sme sa takéto dáta odignorovať. Z toho dôvodu vyhľadávanie na prelome dní nemusí fungovať správne.

V každom prípade potrebujeme mať zoradené jazdy linky tak, aby vyhovovali nášmu algoritmu a teda podľa času príchodu na jednotlivé zastávky jazdy linky. Jazdy sa udržujú v štruktúre stopSubroutes, kde pre každú zastávku p existuje pole subroutes, ktoré na zastávke p stoja. Subroute je zoskupenie jázd s rovnakou postupnosťou zastávok v rámci linky. Subroute si udržuje pole jázd, ktoré sú indexované pomocou kľúča, ktorým je identifikačné číslo serviceDay, v ktorý táto jazda operuje.

Toto pole jázd sme zoradili podľa času príchodu jazdy na zastávku.

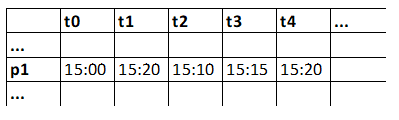
#### 3.4.4.2 Prihliadanie na meškanie vozidiel

Dostávame sa k jadru našej práce a prispôsobeniu algoritmu, aby prihliadal na dáta o meškaní. Ako sme už spomínali, na pozadí beží služba, ktorá zabezpečuje, že v dátovej štruktúre je pre každý StopTimeObject správna hodnota parametra delay pre aktuálny deň a aktuálny čas.

Táto služba sa spúšťa každú minútu. Zisťuje aký je dnes deň a podľa toho vyberá súbor s informáciami o meškaní. Hľadá v ňom také riadky, ktoré vznikli za predchádzajúcu minútu. Každý riadok nám poskytne informácie o meškaní: kedy meškanie vzniklo, na akej zastávke, na akej linke a aká je jeho hodnota. Po získaní zoznamu meškaní, zapracovávame takéto dáta do dátovej štruktúry. Konkrétne dáta vkladáme do štruktúry stopSubroutes, obsahujúcej aj StopTimeObject-y, ktoré si pre každú zastávku na každej jazde držia čas, kedy jazdy linky stojí na zastávke (time) a zároveň, s akým predpokladaným omeškaním príde jazdy linky na zastávku (delay). Použili sme springovskú anotáciu @Scheduled s parametrom fixedRate=60000, ktorá zaručí, že každú minútu sa bude vykonávať funkcia, ktorá s súbore hľadá údaje o meškaní z predchádzajúcej minúty.

Ako vyhľadáme jazdu?

Tuto môže nastať jeden problém: a teda v dátovej štruktúre sa spoliehame že jazdy sú utriedené podľa času príchodu na zastávku. Pri vyhľadávaní v aktuálny deň budeme prihliadať aj na meškania. Čo znemená, že čas príchodu na zastávku predstavuje čas príchodu jazdy na zastávku podľa statických cestovných poriadkov + predpokladané meškanie jazdy na zastávku. Keďže každá jazda môže mať iné meškanie a môže nastať taká situácia, že jazda t1 ma príchod na zastávku p1 v čase 15:05 ale má meškanie 15 minút. Takže predpokladaný príchod jazdy t1 na zastávku p1 je 15:20. Za ňou nasledujúca jazda t2 má na zastávku p1 prísť 15:10 a nemá zistené žiadne meškanie. V prípade, že by používateľ zadal vyhľadávanie v aktuálny deň napríklad o 14:59 zo zastávky p1, algorimus by vybral jazdu t2 ako najskoršiu jazdu, ktorú vie používateľ chytiť na zastávke p1. Správnym riešením by však bolo vybrať jazdu t0. OBRÁZOK



Získanú hodnotu meškania v minútach sme priradili k získanej zastávke a ku všetkým zastávkam nasledujúcim za ňou.

Tento postup sa vykonáva každú minútu. Dátová štruktúra je teda každú minútu aktuálna.

Pri počítaní optimálnej cesty sme prihliadali na dve rôzne situácie:

- kedy používateľ zadal, pre vyhľadanie spoja dnešný deň

- kedy používateľ zadal pre vyhľadanie iný ako dnešný deň.

V prvom prípade prihliadame aj na dáta o meškaní v druhom prípade nás zaujímali len časy zo statických cestovných poriadkov.

### 3.4.5 Filtrovanie

#### 3.4.5.1 V algoritme

1. Pod prestupom v našom prípade rozumieme zmenu medzi jednotlivými akciami. Pod akciou rozumieme peší presun FP, alebo nejaký úsek jazdy. TS tvorí postupnosť akcií. FP, TS je cesta, ktorá obsahuje jeden prestup. TS-FP-TS-TS je cesta, ktorá obsahuje 3 prestupy. V algoritme je obmedzenie, aby mohli vzniknúť 2 pešie prestupy bezprostredne za sebou. Cesta, ktorá je vyhľadávaná z aktuálnej lokality bude vždy začínať peším presunom.

Môže sa stať, že algoritmus vyhľadá takú cestu, ktorá obsahuje dvakrát po sebe vyhľadá peší presun. Pešie presuny slúžia na prestup medzi dvoma linkami, preto nechceme umožniť, aby ponúklo dva pešie presuny po sebe. Pri hľadaní peších presunov, prechádzame označené zastávky (tie, ktoré boli v predchádzajúcom kole vylepšené). Ak nájdeme zastávku, do ktorej sa dá dostať v najlepšom čase len peším presunom, z takejto zastávky nehľadáme ďalšie pešie presuny. Čo znamená, že pešie presuny hľadáme len z takých označených zastávok, do ktorých sme sa v predchádzajúcom kole dostali v najlepšom čase nejakou jazdou.

public boolean previousRoundBestArrivalTimeReachedBySomeTrip(Stop stop){

if(!roundActions.containsKey(stop)){

return true;

}

TimeRound bestTimeRound = bestArrivals.get(stop);

return roundActions.get(stop).stream()

.anyMatch(ra -> ra.getRound().equals(ROUND -1)

&& ra.getArrivalTime().equals(bestTimeRound.getTime())

&& ra.getAction().getActionType().equals(ActionType.TRIP));

}

private void traverseTransfers(RaptorResults raptorResults, Set<Stop> improvedStops, int maxTimeOfWalking){

for(Stop stop: markedStops) {

if (**raptorResults.previousRoundBestArrivalTimeReachedBySomeTrip(stop)**) {

for (Transfer transfer :dataStructure.getDataStructureModel().getStopTransfers().get(stop)){

...

}

}

}s

}

Ak používateľ zadá začiatočný bod ako aktuálnu lokalitu pravá akcia, ktorú vykoná je peší presun. Tu je tiež potrebné ošetriť, aby sme v 1. kole nevyhľadávali pešie presuny.

### 3.4.6 Zapracovanie používateľských preferencií.

#### 3.4.6.1 Maximálny počet prestupov

RAPTOR algoritmus končí vtedy, ak už neexistujú žiadne ďalšie označené zastávky, teda ak sa v predchádzajúcom kole nevylepšil žiadny z časov. RAPTOR algoritmus beží v kolách. 1. kolo znamená 0 prestupov, 2. kolo 1 prestup, atď. Aby bol algoritmus obmedzený počtom prestupov, stačí pridať podmienku a obmedziť algoritmus navyše maximálnym počtom kôl.

public RaptorResults search(SearchParams searchParams) {

initializeMarkedStops(searchParams);

RaptorResults raptorResults = initializeRaptorResults(searchParams);

while(markedStops.size() > 0 && **raptorResults.getRound() <= searchParams.getMaxNumberOfTransfers()**){

...

}

return raptorResults;

}

#### 3.4.6.2 Maximálny čas pešieho presunu

Pri prechádzaní zastávok, ktoré boli v predchádzajúcom kole vylepšené hľadáme pešie presuny do okolitých zastávok. Okolité zastávky boli pre každú zastávku radiálne vyhľadané v okolí 800 metrov pri importe dát do databázy. V algoritme hľadáme pre zastávku len také pešie presuny do iných zastávok, ktorých trvanie je menšie alebo rovné ako používateľom definovaný maximálny čas pešieho presunu.

private void traverseTransfers(RaptorResults raptorResults, Set<Stop> improvedStops, int maxTimeOfWalking){

for(Stop stop: markedStops){

for(Transfer transfer: dataStructure.getDataStructureModel().getStopTransfers().get(stop)){

if(**transfer.getDuration() <= maxTimeOfWalking**) {

...

}

}

}

}

V algoritme však môže nastať prípad, vo viacerých kolách po sebe bude vybratý peší presun a v súčte trvanie týchto peších presunov môže trvať dlhšie ako používateľom definovaný maximálny čas pešieho presunu.

Takúto cestu neponúkneme používateľovi a vyfiltrujeme ju pri tvorení zoznamu výsledných ciest.

#### 3.4.6.3 Minimálny čas na prestup

Nech nastane prípad, že na zastávke A vystúpime v čase 6:00 a hodnota minimálneho času na prestup je 2 (minúty). Hľadáme najskoršiu jazdu linky r, ktorú vieme stihnúť na zastávke A. Nevyberieme jazdu t2 ale jazdu t3, keďže minimálny čas na prestup sú 2 minúty.

Môže nastať ale prípad, že sme sa na zastávku A dostali v čase 6:00 peším presunom. V tomto prípade by sme nechceli prirátavať ešte minimálny čas na prestup, ten chceme prirátavať len v prípade, že prestupujeme medzi dvoma jazdami. Tento prípad však budeme ignorovať, pretože najrýchlejší čas, ktorým sa dostaneme na nejakú zastávku môžeme dosiahnuť viacerými spôsobmi a my si uchovávame všetky. Teda ak sa na zastávku A vieme dostať v čase 6:00 peším presunom aj nejakou jazdou, nebudeme vedieť či máme brať minimálny čas na prestup do úvahy alebo nie. Preto ho budeme brať do úvahy vždy.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Linka r | t1 | t2 | t3 | t4 |
| A | 5:55 | 6:00 | 6:05 | 6:10 |
| B | ... | ... | ... | ... |

Tu sa zároveň vieme ukázať ako funguje mechanizmus hľadania vhodnej jazdy. V dátovej štruktúre sú jazdy linky zoradené podľa času príchodu na zastávku pre každý typ dňa kedy linka premáva. Pri hľadaní jazdy hľadáme smerom od konca poľa. Ak jazda vyhovuje podmienke, označíme ju ako vyhovujúcu a pokračujeme ďalej v prehľadávaní skorších jázd. Ak nájdeme jazdu, ktorá nevyhovuje podmienke ukončíme cyklus. Nájdená jazda zostane tá naposledy vyhovujúca. Index tejto jazdy si uložíme do poľa lastFoundTripsIndices v prípade, že by sme pre túto linku potrebovali nájsť ešte nejakú skoršiu možnú jazdu. Jazdy, ktoré prídu na zastávku neskôr ako označená jazda nás už nezaujímajú.

private SubrouteTrip getEarliestTripOperatingInServiceDay(Subroute subroute, int stopIndex, Time time, ServiceDay serviceDay, int minTimeForTransfer, boolean onlyLowFloor) {

…

for(int i = this.lastFoundTripsIndices.get(subrouteServiceDay); i >= 0; i--){

SubrouteTrip trip = subrouteTrips.get(i);

Time timePlusMinTransferTime = new Time(time).addMinutes(minTimeForTransfer);

Time tripStopTime = trip.getStopTimeObjects()[stopIndex].getTime();

if**(!onlyLowFloor || trip.isLowFloor()**) {

if (**tripStopTime.isBefore(timePlusMinTransferTime)**) { break; }

else { lastFound = trip; }

lastFoundTripsIndices.put(subrouteServiceDay, i);

}

}

return lastFound;

}

#### 3.4.6.4 Vyfiltrovanie len nízkopodlažných vozidiel

Informáciu o nízkopodlažných vozidlách sme nedostali v dátach statických cestovných poriadkoch ale náhodne sme si ich vygenerovali pri importovaní dát a teda každá jazda linky má priradenú informáciu o type vozidla.

V predchádzajúcom kóde môžeme vidieť podmienku, ktorá zabezpečí, aby v prípade nastavenia vyfiltrovania len nízkopodlažných spojov, algoritmus hľadal len jazdy s nízkopodlažnými vozidlami.

### 3.4.7 Naledujúce cesty

- ako s oneskoreným kliknutím na nasledujuce

- na frontende vypočíta nový čas, zvyšné parametre okopíruje

## 3.5 Frontend

V tejto sekcii popíšeme ako je aplikácia implementovaná na klientskej strane a objasníme ako sme riešili problém so zobrazením zastávok v mape.

### 3.5.1 Implementácia frontendovej časti

Pri navrhovaní funkcionalít aplikácie sme si načrtli obrazovky, podľa ktorých sme postupovali pri implementácii.

Aplikáciu z pohľadu klienta tvoria 4 podstránky: hlavné menu, zoznam liniek, vyhľadávanie spojov a výsledky vyhľadávania. Hoci existujú rôzne knižnice znovu použiteľných komponentov, nenašli sme také, ktoré by vyhovovali našim požiadavkám. Vytvorili sme si preto vlastné komponenty. Niektoré z nich sú špecifické pre jednu podstránku, ako napríklad komponent Preferences.vue. Tento komponent sa správa ako vysúvacia lišta, ktorá zobrazí možnosť navolenia vyhľadávacích parametrov. Naopak komponenty Card.vue alebo Loader.vue sú použité takmer na každej podstránke. Údaje, ktoré sú zdieľané medzi podstránkami a komponentami sú uložené vo vuex store.

História vyhľadávania sa ukladá do lokálneho úložiska prehliadača, aby aplikácia mohla ponúknuť používateľovi posledné parametre vyhľadávania.

### 3.5.2 Zobrazovanie zastávok na mape

Aplikácia ponúka možnosť výberu zastávky priamo z mapy v prípade, že používateľ nepozná názvy zastávok. Na prácu s mapou používame knižnicu vue-google-maps. Po zvolení možnosti zadania zastávky z mapy, vycentruje pohľad mapy na aktuálnu polohu používateľa.

Najskôr sme zobraziť všetky existujúce zastávky naraz. Bežné mobilné zariadenie však nestíhalo prekresľovať obrazovku pri posúvaní pohľadu z dôvodu hustého rozsadenia zastávok. Bolo potrebné optimalizovať zobrazenie zastávok.

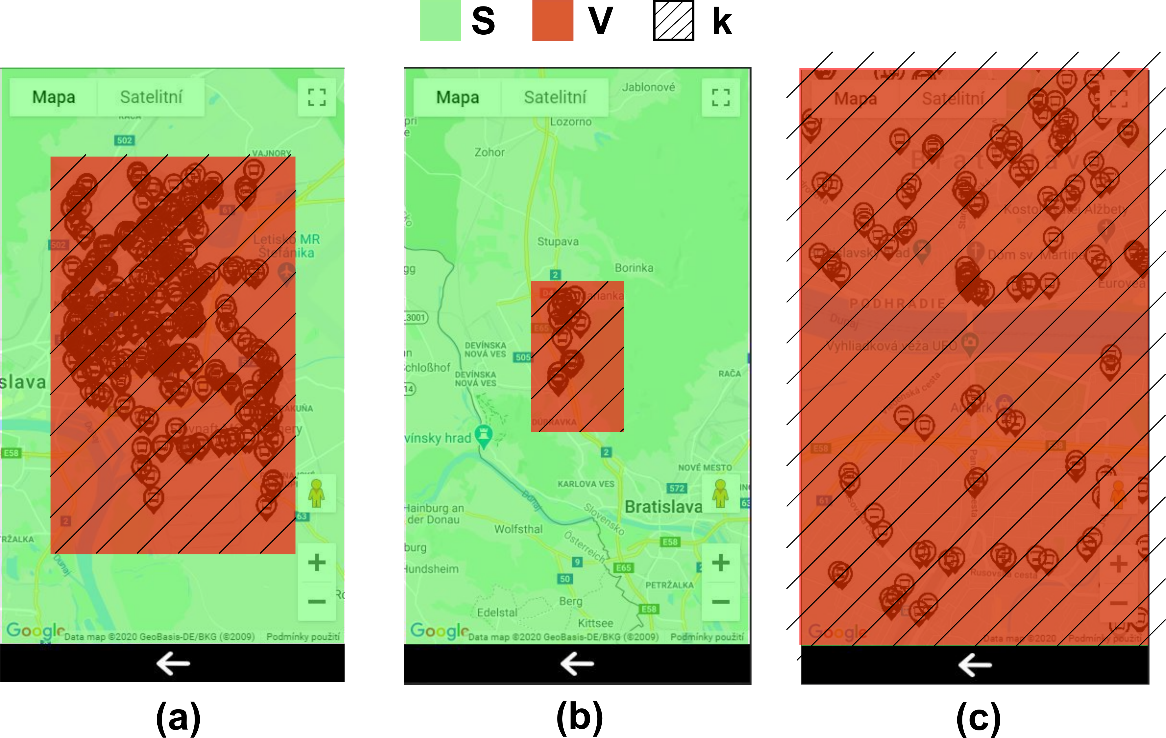
Ďalej sme zvažovali sme zobrazovanie zastávok rovnakým spôsobom ako ich zobrazujú Google Maps, a to tak, že používateľ vidí zastávky len ak je dostatočne priblížený. V prípade, že používateľ vidí celú Bratislavu, nevidí žiadne zastávky.

Nakoniec sme zobrazovanie zastávok vyriešili tak, že zastávky sa budú zobrazovať len v určitom výreze V, ktorý je centrovaný v strede obrazovky. Tento výrez V bude obdĺžnikového tvaru a jeho veľkosť sa bude z pohľadu používateľa meniť podľa hodnoty priblíženia pohľadu.

Pri každej zmene hodnoty priblíženia pohľadu, získame súradnice hraničných bodov zobrazenej plochy. Pomocou získaných súradníc, vypočítame reálnu rozlohu ohraničenej plochy S. Zvolili sme si konštantu k, ktorá určuje maximálnu rozlohu výrezu V.

Ak je S > k, zastávky zobrazíme zastávky vo výreze V s rozlohou k, ako môžeme vidieť na obrázku (a) a (b). Z pohľadu používateľa je rozloha výrezu V na obrázkoch rôzna, avšak v skutočnosti je rozloha rovnaká, pretože sa jedná o reálnu rozlohu.

Ak je S < k, tak zastávky zobrazím na celej obrazovke ako môžeme vidieť na obrázku (c). V tomto prípade je dokonca k > V.



### 3.5.3 Práca s časom

Time simulator, vlastný timepicker,

### 3.5.4 Zobrazovanie časov ciest

dilema či zobraziť len reálny alebo aj ten druhy + obrazovky

--------------------unofficial--------------------

Ak chcem vytvoriť a naplniť databázu nastavíme spring.jpa.hibernate.ddl-auto=create-drop. A zavoláme request api/import. Ak je databáza už vytvorená nastavíme túto vlastnosť na validate.

V application.properties máme podľa profilu nastavenú cestu k GTFS dátam.