

UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

MOBILNÁ APLIKÁCIA NA NÁJDENIE
OPTIMÁLNEJ TRASY V MHD Z REÁLNYCH DÁT
DIPLOMOVÁ PRÁCA

2019
BC. GABRIELA SLANINKOVÁ

UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

MOBILNÁ APLIKÁCIA NA NÁJDENIE
OPTIMÁLNEJ TRASY V MHD Z REÁLNYCH DÁT
DIPLOMOVÁ PRÁCA

Študijný program: Aplikovaná informatika
Študijný odbor: 2511 Aplikovaná informatika
Školiace pracovisko: Katedra aplikovej informatiky
Školiteľ: doc. RNDr. Milan Ftáčnik, CSc.
Konzultant: Mgr. Ľubor Illek

Bratislava, 2019
Bc. Gabriela Slaninková



Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Gabriela Slaninková

Študijný program: aplikovaná informatika (Jednoodborové štúdium,
magisterský II. st., denná formá)

Študijný odbor: aplikovaná informatika

Typ záverečnej práce: diplomová

Jazyk záverečnej práce: slovenský

Sekundárny jazyk: anglicky

Názov: Mobilná aplikácia na nájdenie optimálnej trasy v MHD z reálnych dát

Mobile application for finding the optimal pathway in city public transport from real data

Anotácia: Existuje aplikácia imhd ba, ktorá slúži na plánovanie cesty v MHD v Bratislave na základe statických cestovných poriadkov. Cieľom tejto práce je nájsť spôsob určovania optimálnej cesty a naprogramovať aplikáciu, ktorá to dokáže urobiť s reálnych dát o pohybe vozidiel MHD.

Vedúci: doc. RNDr. Milan Ftáčnik, CSc.

Konzultant: Mgr. Ľubor Illek

Katedra: FMFI.KAI - Katedra aplikovanej informatiky

Vedúci katedry: prof. Ing. Igor Farkaš, Dr.

Dátum zadania: 03.10.2018

Dátum schválenia: 23.10.2018

prof. RNDr. Roman Ďuríkovič, PhD.

garant študijného programu

Roman Ďuríkovič
študent

Milan Ftáčnik
vedúci práce

Pod'akovanie:

Chcela by som sa pod'akovať vedúcemu mojej diplomovej práce doc. RNDr. Milanovi Ftáčníkovi, CSc. za jeho cenné rady, pripomienky, ústretový prístup a usmerňovanie pri tvorbe a písaní práce. Moja vďaka patrí aj Mgr. Ľuborovi Illekovi ze jeho čas a rady.

Abstrakt

Kľúčové slová: RAPTOR algoritmus, hľadanie optimálnej cesty, progresívna webová aplikácia

Abstract

Key words: RAPTOR algorithm, optimal path finding, progressive web application

Obsah

Úvod	1
1 Východiská	2
1.1 Úvod do problematiky	2
1.2 Známe algoritmy hľadania najkratšej cesty	3
1.2.1 Dijkstrov algoritmus	3
1.2.2 A* algoritmus	4
1.3 Časovo závislý algoritmus	4
1.3.1 Multimodálny algoritmus	5
1.4 Spracovanie reálnych dát	5
1.4.1 Time-expanded model	5
1.4.2 Time-dependent model	6
1.5 Optimalizačné metódy	6
1.5.1 Minimalizácia prehľadávaného priestoru v okolí virtuálnej cesty	6
1.5.2 Minimalizácia prehľadávaného priestoru bounding boxom	7
1.6 Zohľadnenie zadania polohy mimo zastávky	8
1.7 Alternatívne cesty	8
1.8 RAPTOR algoritmus	9
1.8.1 Definície pojmov a premenných	9
1.8.2 Základná verzia algoritmu	10
1.8.3 Optimalizácie algoritmu	11
1.8.4 rRAPTOR	12
1.8.5 Dátová štruktúra	12
1.8.6 Vylepšený RAPTOR algoritmus	13
1.9 Podobné existujúce systémy	16
1.9.1 Imhd.sk	16
1.9.2 IDS BK	16
1.9.3 CP	17
1.9.4 CG Transit	17
1.9.5 Ubian	17

2 Návrh	18
2.1 Funkcie aplikácie	18
2.2 Dáta	19
2.2.1 Dáta statických cestovných poriadkov	19
2.2.2 Dáta o meškaní	21
2.2.3 Pešie presuny	22
2.3 Databáza	22
2.4 Parametre pre algoritmus	24
2.5 Algoritmus	25
2.5.1 Použitie a prispôsobenie RAPTOR algoritmu	25
2.6 Dátová štruktúra	28
2.7 Architektúra systému	30
2.7.1 Serverová strana	30
2.7.2 Klientská strana	30
2.7.3 Spracovanie dát	30
3 Implementácia	32
3.1 Technológie	32
3.2 Dáta	32
3.2.1 Parsovanie dát zo súboru	32
3.2.2 Použitie databázy	33
3.2.3 Testovacie dáta	34
3.3 Dátová štruktúra	34
3.3.1 Implementácia dátovej štruktúry	35
3.3.2 Naplnenie dátovej štruktúry	36
3.4 Časový simulátor	36
3.4.1 Vlastná trieda <i>Time</i>	36
3.5 RAPTOR Algoritmus	37
3.5.1 Výsledky RAPTOR algoritmu	37
3.5.2 Hľadanie najbližšej vyhovujúcej jazdy	38
3.5.3 Zapracovanie používateľských preferencií	41
3.6 Klientská strana	42
3.6.1 Implementácia klientskej strany	42
3.6.2 Dátum s čas	43
3.6.3 Zobrazovanie zastávok na mape	43
3.6.4 Zobrazovanie časov ciest	44
3.6.5 Nasledujúce cesty	45
4 Testovanie a evaluácia	47

Zoznam obrázkov

1.1	Bounding box	7
1.2	Hľadanie správnej začiatočnej/konečnej zastávky	8
1.3	RAPTOR - Optimalizácia prechádzania liniek	11
1.4	RAPTOR - Dátová štruktúra	13
1.5	RAPTOR - Dátová štruktúra na zohľadnenie prestupov	13
1.6	RAPTOR - podobné cesty	15
2.1	Entitno-relačný diagram	23
2.2	Vstupné množiny zasáavok	24
2.3	Návrh dátovej štruktúry	29
2.4	Diagram nasadenia	31
3.1	Ukážka exportovaného cestovného poriadku	39
3.2	Hľadanie najbližšej jazdy s prihliadnutím na meškanie	40
3.3	Date picker a Time picker	43
3.4	Zobrazenie zastávok na mape	44
3.5	Možnosti zobrazenia časov v ceste	45
3.6	Finálne zobrazenie časov v ceste	45

Zoznam tabuliek

1.1	Tabuľka premenných pre vylepšený RAPTOR algoritmus	14
2.1	Tabuľka funkciaľít existujúcich aplikácií a navrhovanej aplikácie . . .	19

Úvod

Kapitola 1

Východiská

1.1 Úvod do problematiky

Problém hľadania optimálnej cesty je veľmi rozšíreným problémom nie len v doprave. Doteraz bolo navrhnutých množstvo algoritmov, metód a techník na vyriešenie tohto problému. Pri hľadaní optimálnej trasy vo verejnej doprave je dobré si najskôr ujasniť, čo výraz optimálna cesta znamená.

Najlepšie zhrnutie podproblémov, ktoré môžu nastať sme postrehli v článku [11]. Autori analyzujú ich návrh v najzložitejšom systéme verejnej dopravy - v Hongkongu. Pre verejnú dopravu je praktickejšie navrhnuť viac alternatívnych ciest. Najbežnejšie používateľské preferencie sú:

- minimálny čas
- minimálne poplatky
- minimálny počet prestupov
- minimálna vzdialenosť peších prestupov

Ďalej si treba uvedomiť, že môžu existovať rôzne typy liniek - jednosmerné alebo okružné a taktiež linky závislé na čase - denné, nočné alebo víkendové linky. Vzhľadom na rôzne požiadavky existuje viacero potenciálnych riešení. Neexistuje totiž všeobecne dokonalý spôsob na hľadanie optimálnej cesty, ktorý sedí pre každú verejnú dopravu, najmä kvôli rozloženiu zastávok.

Bratislavská verejná doprava má rôzne typy dopravných prostriedkov (autobus, trolejbus a električka), čo tiež predstavuje určitý problém. Pri prestupoch treba zohľadniť aj pešie presuny. Dopravná sieť, ktorú budeme modelovať v našej práci je teda multimodálna. Multimodálna sieť je definovaná ako kombinácia dvoch a viacerých dopravných prostriedkov na prepravu cestujúcich alebo tovaru z počiatočného miesta na miesto určenia [6].

Ďalším kľúčovým problémom sú podľa zadania reálne dát. Bude potrebné vyriešiť, ako sa vysporiadať so spracovaním reálnych dát o polohe vozidiel. Kľúčový bude

výber dátovej štruktúry na ich spracovanie, ako aj nájdenie vhodného časovo závislého algoritmu.

Sumarizáciou nášho problému je otázka, ako sa vysporiadať s dynamickými dátami, alternatívnymi cestami, viacerými módmi a navigáciou z iného miesta ako zo zastávky. Ďalej chceme vyhovieť používateľským preferenciám, ako je minimálny čas, minimálny počet prestupov a minimálna vzdialenosť peších prestupov. Neposledným problémom je návrh architektúry systému.

V tejto kapitole ďalej opíšeme často skloňované algoritmy pri téme multimodálneho časovo závislého vyhľadávania vo verejnej doprave a priblížime výskumy, ktoré riešili podobné problémy ako my.

1.2 Známe algoritmy hľadania najkratšej cesty

Najčestejším riešením na hľadanie najkratšej alebo optimálnej cesty je použitie niektorého grafového algoritmu alebo jeho modifikácií.

Pod pojmom graf rozumieme dvojicu (V, E) , kde V predstavuje množinu vrcholov a E množinu hrán.

Cesta v grafe $G = (V, E)$ je postupnosť $v_1, e_1, \dots, v_{n-1}, e_{n-1}, v_n$, kde $e_i = (v_i, v_{i+1}) \in E$ pre $i = 1, 2, \dots, n - 1$. Platí, že vrcholy v_i a hrany e_i sa v ceste neopakujú.

1.2.1 Dijkstrov algoritmus

Najznámejším algoritmom hľadania najkratšej cesty v orientovanom, kladne ohodnotenom grafe je Dijkstrov algoritmus. Algoritmus vie nájsť najrýchlejšiu cestu medzi dvomi danými vrcholmi. Známejšou verziou Dijkstrovho algoritmu je hľadanie najkratšej cesty z jedného vrcholu do všetkých ostatných vrcholov v grafe.

Rozhodujúcou hodnotou je vzdialenosť vrcholu $d(n)$, ktorá predstavuje dĺžku cesty od začiatočného vrcholu po vrchol n , pričom cesta vedie cez aktuálny vrchol. Pre začiatočný vrchol sa vzdialenosť $d(n) = 0$ a všetky ostatné vrcholy majú hodnotu $d(n) = \infty$. Na začiatku je začiatočný vrchol označený ako aktuálny. Pri každej iterácii algoritmus preskúma všetky susedné vrcholy aktuálneho vrcholu. Pre každý susedný vrchol prepočíta hodnotu $d(n)$. Ak je táto hodnota menšia ako predtým zaznamenaná, prepíše ju menšou hodnotou. Ďalej algoritmus označí aktuálny vrchol ako navštívený. Potom sa tento vrchol už nebude viac prehľadávať. Aktuálnym vrcholom sa stane zatiaľ nenavštívený vrchol s najmenšou hodnotou $d(n)$.

Ak hľadáme cestu medzi dvomi konkrétnymi vrcholmi, algoritmus končí, keď bol konečný vrchol označený ako navštívený. Inak končí, ak neexistujú žiadne nenavštívené vrcholy alebo majú všetky hodnotu $d(n) = \infty$.

Časová zložitosť Dijkstrovho algoritmu je kvadratická $\mathcal{O}(|V|)$, kde V je množina všetkých vrcholov v grafe. V pôvodnej verzii, kde hľadáme najkratšiu cestu len medzi dvomi vrcholmi, môže algoritmus bežať rýchlejšie.

Nevýhodou algoritmu je veľký prehľadávaný priestor. Existuje viacero algoritmov, ktoré vznikli modifikáciou Dijkstrovho algoritmu. Napríklad Bellman-Fordov algoritmus, ktorý sa vie vysporiadať aj so zápornými hranami alebo A*, ktorý využíva heuristiku na zmenšenie prehľadávaného priestoru.

1.2.2 A* algoritmus

A* algoritmus je grafový optimálny algoritmus na nájdenie najkratšej cesty medzi dvoma bodmi. Vznikol kombináciou heuristických a formálnych prístupov. Udržuje strom ciest začínajúcich v koreni stromu a predĺžuje tieto cesty po jednej hrane. Pri každej iterácii sa rozhodne, ktorú z ciest rozšíri. Algoritmus sa skončí, ak už neexistuje žiadna cesta na rozšírenie alebo pri dosiahnutí koncového vrcholu. Pri prehľadávaní používa stratégiu najskôr najlepšieho. Heuristika, ktorá sa využíva na vyhodnotenie vzdialenosťi, je funkcia

$$f(n) = g(n) + h(n), \quad (1.1)$$

kde $g(n)$ predstavuje hodnotu cesty zo začiatokného vrcholu do vrcholu n a $h(n)$ je heuristická funkcia, ktorá odhaduje najkratšiu cestu z vrcholu n do koncového vrcholu.

Najčastejšie používaná heuristická funkcia na odhadovanie vzdialenosťi je Euklidovská vzdialenosť. Heuristika je prípustná, ak nikdy neprecení skutočné náklady na dosiahnutie cieľa. Potom je zaručené, že algoritmus A* vráti najkratšiu cestu, ak taká v grafe existuje.

Časová zložitosť A* algoritmu závisí od zvolenej heuristiky. V najhoršom prípade je zložitosť algoritmu exponenciálna, v najlepšom prípade môže byť polynomiálna.

1.3 Časovo závislý algoritmus

Stretávame sa s problémom, kedy ohodnotenie hrán v grafe nie je konštantné, ale je závislé od času. Boli navrhnuté rôzne riešenia, ako ohodnotiť hrany, aby sa na model dal aplikovať niektorý z klasických algoritmov hľadania najkratšej cesty.

V článku [5] uvádzajú, že ak dokážeme odhadnúť hodnotu hrany konštantným číslom, potom riešenie klasického problému hľadania najkratšej cesty môže fungovať ako heuristické riešenie problému najkratšej cesty závislej od stavu.

Ďalšia navrhovaná metóda je založena na stochastickom čase jazdy. V článku [14] používajú Bayesovu formulu na získanie pravdepodobnostného rozdelenia času cestného úseku. Na hľadanie optimálnej trasy navrhli a následne použili genetické algoritmy.

1.3.1 Multimodálny algoritmus

Článok [6] navrhuje časovo závislý algoritmus hľadania najkratšej cesty pre multimodálnu dopravnú sieť, pričom využíva A* algoritmus. Navrhnutý algoritmus hľadá len jednu cestu medzi začiatok a koncovým vrcholom.

Autori definujú graf $G = (V, E, M, T)$ ako multimodálny časovo závislý graf, kde V je množina vrcholov, E je množina hrán, M je množina módov a T je množina jázd. Hrana $e_i = (v_i, v_j, m_i)$ je cesta z vrcholu v_i do v_j použitím módu m_i . Vrchol, kde sa menia módy je prestupný vrchol. Ďalej definujú čas prestupu ako súčet času potrebného na prestup medzi dvomi zastávkami, času potrebného na čakanie na prestupný mód a času, ktorý stojí mód na zastávke (týka sa skôr prímestskej dopravy).

Preto je potrebné pridať ďalšie prestupné hrany s tým, že ohodnotenie hrán nebude statické, ale bude to funkcia $cost_{transfer}()$ závislá od času.

Nech hrana $e = (v, v')_m$ má časovo závislú hodnotu $c_e(t)$, jazda pre hranu e je definovaná ako dvojica $travel_e = (t, t')$, kde t je čas odchodu z v a t' je čas príchodu do v' .

1.4 Spracovanie reálnych dát

Verejná doprava má časté meškania spojov z dôvodu nehôd, rôznych porúch spojov alebo servisných prác. Takéto situácie spôsobujú problémy vo verejnej doprave, ale aj pri navrhovaní systému. Aby boli používateľom poskytnuté čo najaktuálnejšie trasy, potrebujeme reálne dátá o polohe vozidiel.

Tieto dátá môžeme získať v rôznej forme. Ak by našimi dátami bola pozícia vozidla z GPS, potrebujeme získané pozicie vozidiel nejakým spôsobom spracovať. V článku [12] navrhujú, ako spracovať takto získané dátá. Proces *map-matching* je navrhnutá metóda integrácie údajov z digitálnych máp s údajmi z polohovacieho systému (GPS). Tento proces slúži na identifikáciu správnej linky, po ktorej vozidlo ide a na určenie presnej polohy tohto vozidla v rámci linky.

Ak už dátá máme spracované, potrebujeme model, na ktorom bude časovo-závislý algoritmus bežať. Na vytvorenie modelu boli v článku [13] navrhnuté 2 prístupy: time-expanded model a time-dependent model.

1.4.1 Time-expanded model

Time-expanded model rozdelí čas na konečný počet intervalov a pre každý interval zduplicuje vrchol. Každá udalosť na každej stanici je modelovaná ako vrchol. Model je vhodný pre siete založené na cestovných poriadkoch. Je teda vhodný pre verejnú dopravu. Ľahko sa modeluje, ale vyžaduje veľa pamäte a vykonávanie dotazov je pomalé.

1.4.2 Time-dependent model

Time-dependent model má klasickú topológiu. Počet vrcholov vo výslednom modeli je rádovo menší ako v *time-expanded* modeli. Okrem iného je tento model aj efektívnejší.

Pre každú zastávku p je do modelu vložený *station node*. Navyše pre každú zastávku p v linke r je vytvorený ďalší vrchol *route node* r_p . *Route nodes* sú spojené so *stations nodes* hranami s konštantným ohodnotením. Toto ohodnenie predstavuje čas potrebný na prestup. Jazda vozidla t je definovaná ako postupnosť zastávok, ktoré vozidlo navštievuje podľa daného cestovného poriadku. Jazdy, ktoré pozostávajú z rovnakých vrcholov sú zoskupené do liniek. *Route nodes*, ktoré patria jednej linke sú spojené hranou, ktorej hodnotou je funkcia. Hoci je takto vytvorený model menší, pre algoritmy je komplikovanejší. Tažko sa na ňom vykonávajú dotazy v prípade, že je potrebné brať do úvahy prestupy.

1.5 Optimalizačné metódy

V tejto sekcii spomenieme optimalizačné metódy, ktoré minimalizujú prehľadávanie, aby algoritmy hľadania najkratšej cesty bežali rýchlejšie a efektívnejšie. Jedna z metód, ktorá sa dá použiť na grafe, ktorého vrcholy nemajú reálne súradnice, je algoritmus spomínaný v článku [9]. Autori navrhli heuristiku, ktorá používa 3 indexy na určenie a odrezanie nepotrebných vrcholov pre výpočet najkratšej cesty. Ak poznáme reálne súradnice vrcholov, je efektívnejšie využiť niektoré z techník minimalizácie prehľadávaného priestoru.

1.5.1 Minimalizácia prehľadávaného priestoru v okolí virtuálnej cesty

V článku [6] navrhujú optimalizáciu A* algoritmu založenú na vypočítaní virtuálnej cesty, ktorá predstavuje Euklidovskú vzdialenosť zo začiatocného do koncového vrcholu. Algoritmus hľadá cestu cez vrcholy, ktoré sú blízko virtuálnej cesty. Definujeme parametre:

- virtuálna cesta (st) – Euklidovská vzdialenosť z s do t a zároveň dolné ohraničenie prehľadávaného priestoru
- vzdialenosť d - priemerná vzdialenosť všetkých vrcholov k virtuálnej ceste a predstavuje horné ohraničenie prehľadávaného priestoru
- d_{max} – vzdialenosť najvzdialenejšieho vrcholu od virtuálnej cesty
- Δd – priemerná vzdialenosť od všetkých vrcholov ku všetkým susedom

Najskôr vypočítame hodnotu $D = dist(s, t)$, d a Δd . Prehľadávaný priestor bude mať rozmery $(2d, D)$. Začneme vytvárať cestu tak, že hľadáme vrchol v_i , ktorý splňa

podmienku $dist(v_i, st) \leq d$. Ak nevieme nájsť žiadneho takého kandidáta, navyšujeme d o Δd , až kým nejakého nenájdeme. V každej ďalšej iterácii po výbere vrcholu v_i hľadáme ďalšieho kandidáta. Ak sme prehľadali všetky vrcholy v grafe a nezostavili sme najkratšiu cestu, začneme prehľadávať odznovu s väčším prehľadávaným priestorom.

1.5.2 Minimalizácia prehľadávaného priestoru bounding boxom

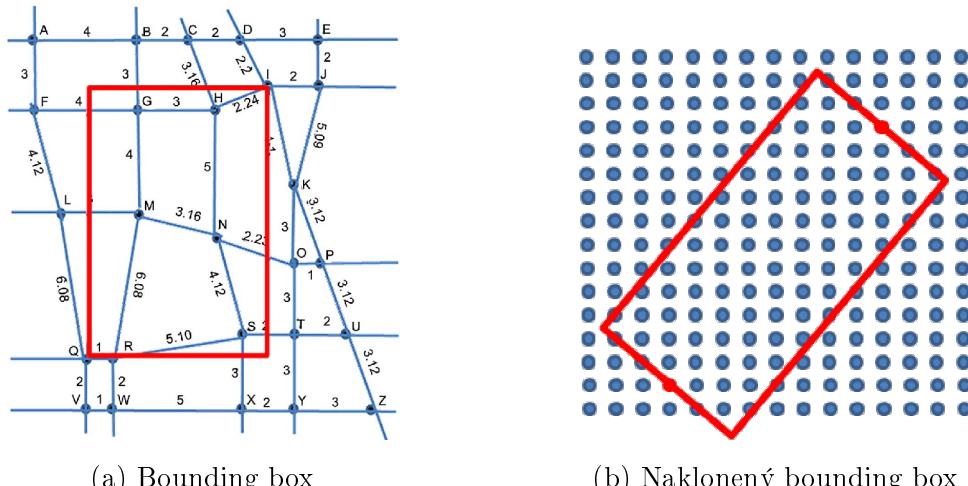
V článku [10] sa autori snažia nájsť spôsob, ako efektívne vypočítať najkratšiu cestu a tak odľahčiť cesty od dopravných zápch. Navrhujú algoritmus na nájdenie najkratšej alternatívnej cesty s minimálnym potrebným výpočtovým časom.

Pre nás zaujímavý je ich prístup k vyhodnoteniu najkratšej cesty a k minimalizácii grafu. Minimalizácia grafu prebieha nepretržite. Vždy, keď zvažujeme ďalší susedný vrchol, definujeme nový *bounding box* a tým odstránime nadbytočné vrcholy, ktoré pri výpočte nie sú relevantné.

Na začiatku potrebujeme vymodelovať graf mesta, kde sú vrcholy popísané zemepisnou šírkou a výškou a hrany sú ohodnotené reálnou vzdialenosťou medzi jednotlivými vrcholmi. Keď poznáme začiatočný a koncový vrchol, vytvoríme si tabuľku všetkých vrcholov, ich súradníc a vzdialostí do začiatočného vrcholu a koncového vrcholu. Podľa súradníc vieme vypočítať vzdialosti medzi jednotlivými vrcholmi. Následne vygenerujeme grafickú reprezentáciu tejto tabuľky.

Bounding box

Bounding box je na začiatku vytvorený medzi začiatočným a koncovým vrcholom, ako je ukázané na obrázku 1.1a. Ďalšiu redukciu vieme dosiahnuť, ak nakloníme bounding box tým spôsobom, že spojnica začiatočného a koncového vrcholu bude osou boxu, ako môžeme vidieť na obrázku 1.1b.

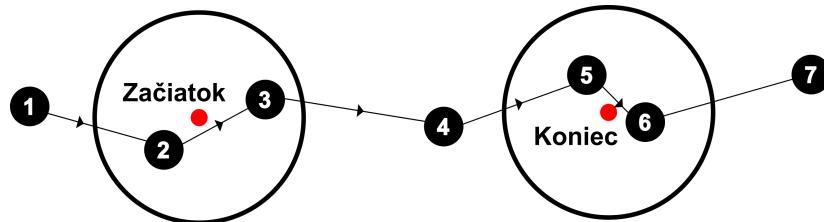


Obr. 1.1: Bounding box [1] [2]

1.6 Zohľadnenie zadania polohy mimo zastávky

V článku [11] autori poukazujú na to, že správny systém na hľadanie optimálnej trasy vo verejnej doprave by mal ponúknuť používateľovi možnosť vyhľadať trasu z miesta (prípadne na miesto), ktoré nie je nutne zastávkou. Ak hľadáme najkratšiu cestu z takto zadaného vrcholu, prvým krokom bude nájdenie prvej zastávky. Autori upozorňujú na to, že vyhľadanie najbližšej zastávky k danej pozícii nemusí byť správnym riešením.

Navrhovaným riešením je nájsť viacero zastávok radiálnym vyhľadávaním, ktorému určíme polomer. Vo vybranej zóne sa môže nachádzať viac zastávok, ktoré prislúchajú tej istej trase, ako môžeme vidieť na obrázku 1.2. V tomto prípade by výber zastávky č. 2 (najbližšej zastávky k zadanej pozícii) neboli optimálny, lebo trvanie pešieho presunu na zastávku č. 2 spolu s časom jazdy na zastávku č. 3 je dlhšie, ako trvanie pešieho presunu na zastávku č. 3. Rovnako aj výberom zastávky č. 6 ako konečnej zastávky by nebolo súčasťou správneho riešenia hľadania najkratšej cesty.



Obr. 1.2: Hľadanie správnej začiatočnej/konečnej zastávky

1.7 Alternatívne cesty

Pri hľadaní optimálnej cesty je potrebné zohľadniť, čo daný používateľ považuje za optimálne. Používateľ pri zadávaní vyhľadávacích parametrov nemusí poznáť svoje preferencie alebo ich môže často meniť. Zadávanie preferencií pri každom vyhľadávaní nie je používateľsky prívetivé. Aby si napriek tomu používateľ mohol zvoliť cestu, ktorá najviac vyhovuje jeho potrebám, systém mu môže ponúknuť viac alternatívnych ciest. Tu sa dostávame k problému K najkratších ciest.

Autori článku [8], poukazujú na to, že väčšina navigačných služieb neposkytuje používateľom viac ciest. A ak áno, využívajú taký algoritmus, ktorý poskytuje viacero alternatívnych ciest, ktoré sú podobné vo veľkej časti úsekov. Správny algoritmus by mal vybrať trasy s najmenšou spoločnou dĺžkou spomedzi tých, ktoré vyhovujú zadaným preferenciám. Autori článku predstavili vývoj heuristického algoritmu, ktorý efektívne vyhľadáva rôzne alternatívne cesty.

1.8 RAPTOR algoritmus

Doteraz spomínané algoritmy riešia problém hľadania optimálnej cesty vo verejnej doprave ako grafový problém. Ako sme uviedli v predchádzajúcich sekciách, najčastejšie sa pre tento problém vytvorí model dopravnej siete pomocou grafu a na ňom sa spustí algoritmus hľadania najkratšej cesty. Týmto riešením však väčšinou dostávame menej optimálne výsledky za vysoké výpočtové časy.

Techniky, ktoré sa opierajú o grafové modely ľahko spracúvajú dynamickosť v systémoch verejnej dopravy, ako časté meškanie liniek, náhle zrušenie liniek, zmeny trás a podobne. Hoci algoritmy hľadania najkratšej cesty sú rýchle, práve vytvorenie modelu spôsobuje spomínané vysoké výpočtové časy.

Autori článku [4] predstavujú RAPTOR – Round-bAsed Public Transit Optimal Router, ktorý dokáže byť oveľa rýchlejší použitím jednoduchých obmedzovacích pravidiel a paraleлизmu. Keďže RAPTOR algoritmus nepotrebuje žiadny grafový model, dokáže fungovať rýchlo a dynamicky. Pre dve zadané zastávky vráti všetky optimálne cesty s minimálnym časom príchodu do konečnej zastávky a minimálnym počtom prestupov. Algoritmus beží v kolách a počíta časy príchodu prechádzaním jednotlivých trás.

Na rozdiel od grafových algoritmov, RAPTOR sa ľahko paralelizuje. Jednoducho rozdelí nezávislé linky medzi rôzne CPU jadrá. Existujú dve rozšírenia algoritmu a to McRAPTOR, ktorý zvláda riešiť viacero kritérií okrem času príchodu a počtu prestupov a rRAPTOR, ktorý vracia množinu cest pre všetky odchody zo zastávky v danom časovom rozsahu. Bez potrebného pre-processingu a post-processingu dokáže spracovať aj kritérium preferovaných prestupných miest.

1.8.1 Definície pojmov a premenných

Definujeme cestovný poriadok ako $(\mathcal{S}, \mathcal{T}, \mathcal{R}, \mathcal{F}, \Pi)$, kde

- $\Pi \subset \mathbb{N}_0$ je čas v sekundách,
- \mathcal{S} je množina zastávok,
- \mathcal{T} je množina jázd,
- \mathcal{R} je množina liniek a
- \mathcal{F} je množina peších presunov.

Zastávka p predstavuje miesto pre nastúpenie a vystúpenie z vozidla. Jazda t reprezentuje postupnosť zastávok konkrétneho vozidla. Každá zastávka p z jazdy t má priradený čas odchodu $\tau_{dep}(t, p) \in \Pi$ a čas príchodu $\tau_{arr}(t, p) \in \Pi$. Každá linka z \mathcal{R} pozostáva z jázd, ktoré majú rovnaké postupnosti zastávok. Pešie prechody z množiny \mathcal{F} reprezentujú prestupy medzi zastávkami. Každý prestup pozostáva z dvoch zastávok p_1 a p_2 a času potrebného na presun medzi nimi $l(p_1, p_2)$.

Výstupom z algoritmu je cesta \mathcal{J} , ktorá je definovaná postupnosťou jázd a peších prestupov. Cesta, ktorá obsahuje k jázd, má presne $k - 1$ prestupov. Majme začiatočnú zastávku p_s a konečnú zastávku p_t a čas odchodu τ . Najzákladnejším kritériom, na ktoré algoritmus prihliada, je *Earliest Arrival Problem*, ktorý hľadá cestu nezačínajúcú skôr než τ v bode p_s a do p_t sa dostane čo najrýchlejšie. Ďalej chceme viacero alternatívnych ciest s tým, že žiadna z ciest nezačína skôr ako τ a cesty môžu obsahovať aj prestupy.

1.8.2 Základná verzia algoritmu

Nech $p_s \in \mathcal{S}$ je začiatočný vrchol a $\tau \in \Pi$ je čas odchodu. Cieľom je vypočítať pre každé k cestu do zastávky p_t s minimálnym časom príchodu, ktorá je zložená z najviac k jázd. Algoritmus beží v k kolách. Kolo k počíta najrýchlejšiu cestu do každej zastávky na $k - 1$ prestupov. Niektoré zastávky nemusia byť dosiahnuteľné. Pri objasňovaní algoritmu bude počet kôl rovný K . Algoritmus pridelí každej zastávke p postupnosť $(\tau_0(p), \tau_1(p), \dots, \tau_K(p))$, kde $\tau_i(p)$ reprezentuje najskorší známy čas príchodu do zastávky p na najviac i jázd.

Na začiatku sú všetky hodnoty vo všetkých postupnostiach pre každú zastávku inicializované na hodnotu ∞ . Nastavíme $\tau_0(p_s) = \tau$. Dodržuje sa invariant: na začiatku kola k je prvých k hodnôt v postupnosti $\tau(p)$ správnych a ostatné hodnoty majú hodnotu ∞ . Cieľom kola k je vypočítať $\tau_k(p)$. Tento výpočet sa vykonáva v troch krokoch.

V prvom kroku kola k (ak $k \neq 0$) sa nastaví pre všetky zastávky p hodnota $\tau_k(p) = \tau_{k-1}(p)$. Týmto nastavíme horné ohraničenie na najskorší príchod do zastávky p na najviac k jázd.

Druhé kolo spracuje práve raz každú linku r . Nech $\mathcal{T}(r) = (t_0, t_1, \dots, t_{|\tau(r)|-1})$ je postupnosť jázd pre linku r zoradená od najskôr začínajúcej po poslednú jazdu. Nech $et(r, p_i)$ je najskoršia jazda linky r , na ktorú je možné nastúpiť na zastávke p_i . Táto hodnota nemusí byť vždy definovaná. Pri spracovaní linky r navštěvujeme jej zastávky a hľadáme takú zastávku p , kde je hodnota $et(r, p_i)$ definovaná. Označme prislúchajúcu jazdu ako aktuálnu jazdu pre k . Ďalej prechádzame linku a pre každú zastávku p_j aktualizujeme $\tau_k(p_j)$ použitím tejto jazdy. Aby sme vedeli späť určiť výslednú cestu, nastavíme smerník na zastávku, na ktorej sme nastúpili na jazdu t . Navyše bude potrebné aktualizovať aktuálnu jazdu pre k . Na každej zo zastávok p_i na linke r môžeme stihnuť skoršiu jazdu, pretože sme v predchádzajúcich kolách mohli nájsť rýchlejšiu cestu do p_i . Je potrebné skontrolovať či $\tau_{k-1}(p_i) < \tau_{arr}(t, p_i)$ a aktualizovať t prepočítané $et(r, p_i)$.

V treťom kroku zvažujeme pešie presuny. Zo zastávky p_i sa môžeme dostať do niektorých zastávok peším presunom. Treba overiť, či týmto presunom nedosiahneme skorší čas príchodu do niektornej zo zastávok. Pre každý peší presun $(p_i, p_j) \in \mathcal{F}$ nastaví $\tau_k(p_j) = \min\{\tau_k(p_j), \tau_k(p_i) + l(p_i, p_j)\}$.

Algoritmus sa končí, keď po kole k nebola vylepšená žiadna z hodnôt $\tau_k(p_i)$.

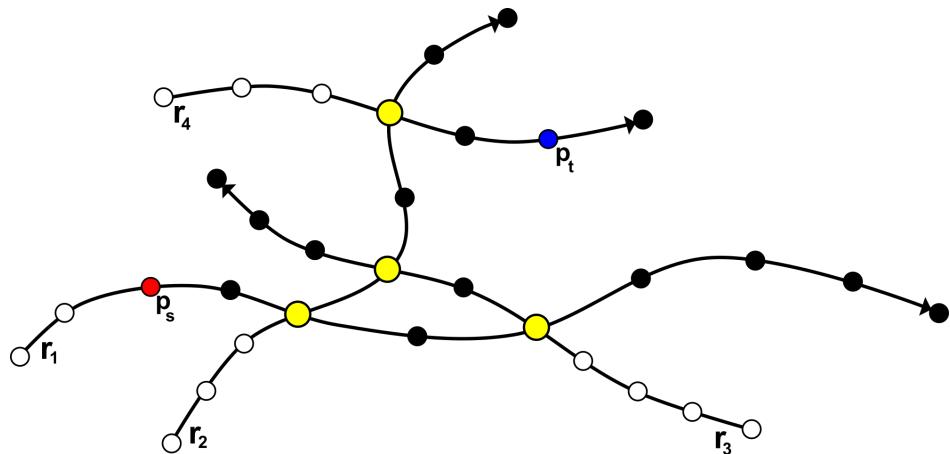
Najhorší odhad časovej zložitosti algoritmu je lineárny, konkrétnie $\mathcal{O}(K(\sum_{r \in \mathcal{R}} |r| + |\mathcal{T}| + |\mathcal{F}|))$, kde K je počet kôl, pričom v každom kole k prechádzame každú linku r najviac jeden krát. Keďže $|r|$ je počet zastávok, spolu prechádzame $\sum_{r \in \mathcal{R}} |r|$ zastávok. Pri počítaní $et(r, p_i)$, prechádzame každú jazdu t linky r najviac jeden krát.

1.8.3 Optimalizácie algoritmu

Pri hľadaní najkratšej cesty je prechádzanie všetkých liniek v každom kole zbytočné. V niektorých prípadoch neexistuje možnosť prestúpiť medzi dvoma linkami, takže niektoré z liniek sú nedosiahnuteľné. Majme linku r a majme zastávku p v r , ktorej posledné vylepšenie času príchodu bolo v kole $k' < k - 1$. Linka r bola opäť navštívená v kole $k' + 1 < k$, ale žiadnej z jej zastávok sa nevylepšil čas príchodu. Nie je dôvod prechádzať linku znova, ak aspoň jedna z jej zastávok nebola vylepšená.

Na implementáciu tejto optimalizácie postačí, ak si v kole $k - 1$ označíme tie zastávky, pre ktoré bol v tomto kole vylepšený čas $\tau_{k-1}(p_i)$. Na začiatku kola k prejdeme cez všetky tieto zastávky, označíme ich a nájdeme všetky linky, ktorým zastávky prislúchajú. Označené zastávky sú potenciálne miesta na prestop medzi linkami v kole k . Stačí nám dokonca prechádzať linku od prvej označenej zastávky v linke. Pridáme linky do pola Q a zapamätáme si jej prvú označenú zastávku.

Príklad môžeme vidieť na obrázku 1.3. Algoritmus hľadá cestu zo zastávky p_s do zastávky p_t . V prvom kole spracuje linku r_1 , v druhom kole linky r_2 a r_3 a v treťom kole linku r_4 . Iterovanie linky začína v prvej označenej zastávke linky (žltá farba). Zastávky označené bielou farbou nebudú po optimalizácii spracované v žiadnom kole.



Obr. 1.3: RAPTOR - Optimalizácia prechádzania liniek

Ďalšou optimalizačnou technikou je *local pruning*. Pre každú zastávku p_i si udržujeme hodnotu $\tau^*(p_i)$, ktorá reprezentuje najskorší známy čas príchodu na zastávku p_i .

Vylepšením je, že zastávku označíme len v prípade, že čas príchodu v kole k je menší ako predchádzajúca hodnota $\tau^*(p_i)$.

RAPTOR algoritmus hľadá cestu zo začiatočnej zastávky do všetkých zastávok, hoci chceme nájsť cestu do konkrétnej konečnej zastávky. *Target pruning* obmedzí hľadanie ciest do všetkých vrcholov na hľadanie jednej cesty. Dosiahneme to, ak v kole k nebudeme označovať zastávky p_i , pre ktoré platí $\tau^*(p_i) > \tau^*(p_t)$.

1.8.4 rRAPTOR

rRAPTOR je rozšírenie RAPTOR algoritmu, ktoré vráti množinu najkratších ciest pre všetky odchody zo zastávky v danom časovom rozsahu.

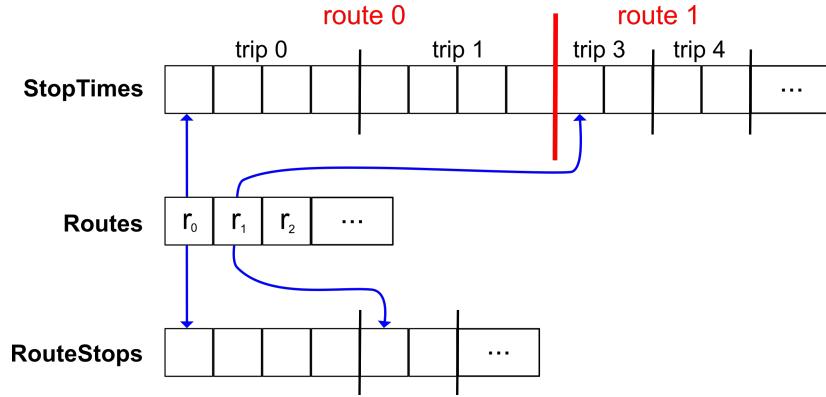
Nech $\Delta \subseteq \Pi$ je časový úsek. Najskôr vložíme do množiny Ψ časy odchodov jázd začínajúcich na danej zastávke, ktoré patria do časového úseku Δ . Pre každý čas odchodu τ z Ψ spustíme RAPTOR algoritmus nezávisle. Znamená to, že hodnota $\tau_k(p)$ bude existovať pre každý čas odchodu τ , zastávku p a kolo k . Nemusí platiť, že všetky nájdené cesty budú optimálne. Na porovnanie ciest využijeme pravidlo dominancie ciest: cesta \mathcal{J}_1 dominuje nad cestou \mathcal{J}_2 , ak platí:

$$\tau_{dep}(\mathcal{J}_1) \geq \tau_{dep}(\mathcal{J}_2) \text{ a } \tau_{arr}(\mathcal{J}_1) \leq \tau_{arr}(\mathcal{J}_2).$$

Aby sme mohli použiť toto pravidlo, zoradíme časy v množine Ψ od najneskoršieho po najskorší čas. Pri iterovaní nebudeme prepisovať hodnotu $\tau_k(p)$ medzi kolami, ale nastavíme vždy na začiatku kola pre všetky zastávky p $\tau_k(p) = \tau_{k-1}(p)$, kde je $\tau_{k-1}(p)$ lepšie ako $\tau_k(p)$. Pri tomto rozšírení algoritmu nemôžeme použiť *local pruning*, keďže nechceme, aby sa hodnota $\tau^*(p)$ porovnávala s ďalším spustením algoritmu pre skoršie časy z množiny Ψ .

1.8.5 Dátová štruktúra

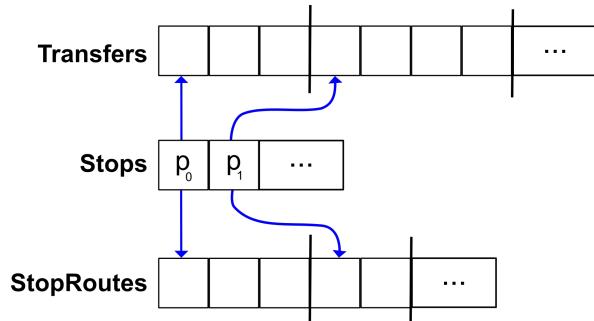
Autori článku navrhli aj štruktúru, ktorá je vhodná pre RAPTOR algoritmus. Linky, jazdy a zastávky indexujeme od 0. Potrebujeme pole liniek *Routes*, ktoré si pre každú linku r_i uchováva informáciu o počte zastávok na linke r_i a smerník na pole *RouteStops*, ktorý označuje začiatok postupnosti zastávok na linke r_i . Podobne aj pre jazdy si linka r_i uchováva smerník, ktorý ukazuje na začiatok bloku v poli *StopTimes*. Jeden blok v poli *StopTimes* obsahuje všetky jazdy prislúchajúce linke r_i zoradené podľa času odchodu z prvej zastávky. Každú jazdu reprezentuje postupnosť časov (čas príchodu a čas odchodu). Štruktúra je zobrazená na obrázku 1.4.



Obr. 1.4: RAPTOR - Dátová štruktúra

V algoritme často potrebujeme iterovať cez zastávky linky r_i . Na to nám slúži pole $RouteStops$. Ak potrebujeme získať najskorší čas odchodu zo zastávky p po čase τ , spôsob akým je pole $StopTimes$ utriedené nám zabezpečí, že čas tejto operácie bude konštantný. Pri kontrolovaní, či sa v predchádzajúcej jazde nezlepšil čas odchodu niektoréj zo zastávok, potrebujeme preskočiť na predchádzajúcu jazdu v poli $StopTimes$. Na urýchlenie tejto operácie máme uloženú informáciu o počte zastávok pre linku r_i .

Takto navrhnutá štruktúra nie je pre algoritmus dostatočná. Na zohľadnenie prestupov potrebujeme ďalšie štruktúry a to pole $Stops$, ktoré obsahuje všetky zastávky. Ďalej pole $StopRoutes$, ktoré má pre každú zastávku priradené linky, ktoré na nej zastavujú. Pole $Transfers$, ktoré obsahuje informácie o peších prechodoch medzi zastávkami. Pre každú zastávku je priradená dvojica: cieľová zastávka a čas potrebný na peší presun na túto zastávku. Náčrt doplnujúcej dátovej štruktúry je na obrázku 1.5.



Obr. 1.5: RAPTOR - Dátová štruktúra na zohľadnenie prestupov

1.8.6 Vylepšený RAPTOR algoritmus

Za výslednú cestu považoval RAPTOR algoritmus tú cestu, ktorá stojí najmenej času nezávisle od toho, koľko prestupov vyžaduje. Keďže cestujúci v skutočnosti nepreferujú cesty s veľkým počtom prestupov, je potrebné zaviesť do algoritmu ich penalizáciu. Ako sme už spomínali v predchádzajúcich sekciách, v prípade verejnej dopravy, ktorá má

viacero módov, je nutné vyhľadať a ponúknuť cestujúcemu viacero alternatívnych ciest. Pôvodný RAPTOR algoritmus vracia len jednu najkratšiu cestu.

V článku [7] bol navrhnutý vylepšený RAPTOR algoritmus, ktorý má v sebe zapracovanú penalizáciu prestupov, korekčný faktor peších prestupov a vracia k alternatívnych ciest. Cesty vypočítané vylepšeným algoritmom sa oveľa viac podobajú cestám, ktoré si v realite cestujúci vyberajú.

V tomto článku nám pribudlo označenie k -tej najkratšej cesty. Kedže písmenom k sme doteraz označovali kolá, budeme hľadať m -tú najkratšiu cestu. Taktiež pribudli nové označenia a premenné, ktoré sú uvedené v tabuľke 1.1.

Premenná	Popis
t_r	jazda t linky r
Tr_c	penalizácia prestupu typu c
$\tau_k(p)$	najskorší známy čas príchodu na zastávku p v k kolách
$\mathcal{J}_k(p)$	množina ciest, ktorými sa vieme dostať na zastávku p v k kolách
$\mathcal{J}_{k,m}(p)$	m -tá najskoršia cesta, ktorou sa vieme dostať na zastávku p v k kolách
$\mathcal{J}_k(p, p_i)$	cesta zo zastávky p do zastávky p_i v k kolách

Tabuľka 1.1: Tabuľka premenných pre vylepšený RAPTOR algoritmus

Penalizácia prestupov

Penalizácia prestupov je pojem zahŕňajúci časové a nečasové prvky prestupu. Medzi časové prvky patrí čakanie na prestupný spoj a čas potrebný na peší presun. Nečasovými prviami môže byť pohodlie a komplikovanosť prestupu, ktoré závisia najmä od toho, či prestupujeme medzi rovnakými módmi alebo sa módy líšia.

Autori článku definujú dva druhy prestupov: horizontálny a vertikálny. Napríklad prestup medzi autobusmi je horizontálny, ale pri prestupe z autobusu na metro je potrebné použiť schody a preto sa jedná o vertikálny prestup. Podľa druhu prestupu c aplikujeme penalizáciu prestupu Tr_c . Pri implementácii si potrebujeme pamätať predchádzajúci mód jazdy, ktorým cestujúci prišiel na zastávku, na ktorej bude prestupovať.

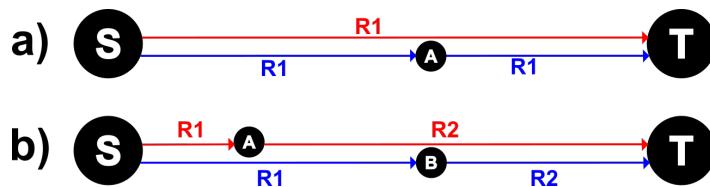
Čo sa týka času potrebného na peší presun medzi jednotlivými zastávkami, priemerná rýchlosť kráčania dospelého človeka bola určená na $1.2m/s$. Väčšinou sa predpokladá, že dĺžka prestupu je Euklidovská vzdialenosť od počiatočnej zastávky po cieľovú. Tento odhad ale nie je výhovujúci, pretože vo väčšine prípadov je trasa prestupu dlhšia ako vzdušná čiara medzi zastávkami. Navrhli preto použiť Manhattanovskú vzdialenosť. Keď je vzdialenosť vzdušnou čiarou rovná 1, Manhattanovská vzdialenosť má hodnotu $\sqrt{2}$.

Hľadanie viacerých ciest

Vylepšený algoritmus hľadá M najkratších ciest. Je potrebné zabrániť tomu, aby v rámci m -nájdených ciest boli podobné cesty. Autori navrhli 2 pravidlá, použitím ktorých zabránia výskytu podobných ciest vo výsledných M cestách.

Prvým pravidlom je, že jazda t linky r by sa už znova nemala vyhľadať zo zastávky, na ktorej cestujúci vystúpi. Príklad je na obrázku 1.6(a). Červená cesta vedie priamo zo začiatocnej zastávky S do konečnej zastávky T a modrá cesta obsahuje prestup na zastávke A . Časy príchodu týchto dvoch ciest sa líšia a preto by sa cesty vyhodnotili ako rôzne. Týmto zabránime tomu, aby červená a modrá cesta boli vybrané súčasne.

Druhé pravidlo považuje cesty za podobné, ak je postupnosť zastávok v oboch cestách rovnaká. Na obrázku 1.6(b) môžeme vidieť, že cesty sa líšia v mieste prestupu a tým pádom aj v čase príchodu do cieľovej zastávky. Preto by tieto cesty boli vyhodnotené ako rozdielne. Po aplikovaní pravidla budú vyhodnotené ako podobné. V $\mathcal{J}_k(p)$ sú uložené cesty, ktorými sme sa dostali na zastávku p v kole k . Ak prídeme na zastávku p v čase $\tau_k(p)$, porovnáme časy príchodov už existujúcich ciest z množiny $\mathcal{J}_k(p)$ a len cesta s minimálnym časom príchodu bude aktualizovaná.



Obr. 1.6: RAPTOR - podobné cesty

Ďalej je popísaný algoritmus 1, ktorý zhodnotí, či je cesta $\mathcal{J}_k(p, p_i)$ podobná ako niektorá z cest zaznamenaných v množine $\mathcal{J}_k(p_i)$.

Algorithm 1 Algoritmus na zistenie podobných ciest

Input: $\mathcal{J}_k(p_i)$, t_r , $\mathcal{J}_k(p, p_i)$

Output: true or false

```

1: for each  $\mathcal{J}_{k,m}(p_i) \in \mathcal{J}_k(p_i)$  do
2:   if last  $t$  of  $\mathcal{J}_{k,m}(p_i) = t_r$  then return false
3:   else if  $\mathcal{J}_k(p, p_i) = \mathcal{J}_{k,m}(p_i)$  then return false
4:   else return true
5:   end if
6: end for

```

1.9 Podobné existujúce systémy

V tejto sekcií popíšeme niektoré existujúce mobilné aplikácie, ktoré umožňujú vyhľadávanie spojov v Bratislavskej mestskej hromadnej doprave. Z používania aplikácie nevieme zistiť, aký algoritmus používajú na vyhľadávanie spojení a v akých dátových štruktúrach udržujú dátá. Vieme však porovnať, aké funkcionality ponúkajú používateľom a zhodnotiť, čo nám chýba alebo prekáža pri používaní týchto aplikácií.

1.9.1 Imhd.sk

Mobilná aplikácia Imhd.sk ponúka vyhľadávanie MHD spojení v Bratislave a v Košiciach. Pri prvom spustení aplikácia stahuje databázu cestovných poriadkov do úložiska zariadenia. Aplikácia funguje aj v offline režime, kedy používa cestovné poriadky, ktoré boli naposledy stiahnuté do zariadenia. Aplikácia ponúka možnosť zobraziť všetky existujúce linky a ich zastávky. Ak používateľ nepozná názov zastávky, z ktorej alebo na ktorú sa chce dostať, môže vybrať zastávku priamo z mapy. Pri vyhľadávaní spojov je možné nastaviť rýchlosť chôdze, minimálny čas potrebný na presun, maximálny počet prestupov, akceptáciu peších presunov, vyfiltrovanie nízkopodlažných spojení alebo spojení na prepravu bicyklov. Používateľ si môže uložiť obľúbené linky, zastávky alebo cesty. Rovnako je možné vyhľadávanie spojení z/do aktuálnej polohy. Aplikácia vyhľadáva spojenia podľa statických cestovných poriadkov. Nezohľadňuje aktuálny stav dopravy a neponúka informácie o prípadnom meškaní spojov. Aplikácia informuje o možnosti kúpy SMS lístkov a v prípade pripojenia na internet zobrazuje aktuálne správy o zmenách, presunoch alebo výlukách v spojoch.

1.9.2 IDS BK

Aplikácia IDS BK je oficiálnou aplikáciou Integrovaného dopravného systému v Bratislavskom kraji. Na rozdiel od predchádzajúcej aplikácie umožňuje nákup lístkov a dobíjanie kreditu na kartu a vyžaduje pripojenie na internet. Rovnako ako v predchádzajúcej aplikácii používateľ vie zvoliť zastávku priamo z mapy. Pri vyhľadávaní spojov vieme nastaviť maximálny počet prestupov a maximálny povolený peší presun. Nevieme ale vyfilterovať nízkopodlažné spoje a nastaviť preferenciu minimálneho potrebného času na prestop medzi spojeniami. Po vyhľadaní spojov aplikácia neponúka zobrazenie postupnosti zastávok konkrétneho spoja. Aplikácia IDS BK rovnako ako Imhd.sk vyhľadáva spojenia zo statických cestovných poriadkov.

1.9.3 CP

Mobilná aplikácia CP je oficiálna aplikácia pre vyhľadávanie v cestovných poriadkoch autobusovej, vlakovej a mestskej hromadnej dopravy celého Slovenska. Pri vyhľadávaní vlakových spojení zobrazí informáciu o prípadných meškaniach a výlukách hľadaných spojov. V aplikácii si používateľ nevie zobraziť linky a postupnosť ich zastávok. Pri vyhľadávaní spojenia vieme nastaviť rôzne preferencie. Na rozdiel od predchádzajúcich aplikácií chýba možnosť nastavenia obmedzenia pešieho presunu. Používateľ si vie zvoliť obľúbené položky a pamätať si historiu hľadania, podľa čoho ponúka používateľovi inteligentné našepkávanie zastávok. Zastávky je možné zadať aj priamo z mapy. Aj v tejto aplikácii je možnosť priamo kúpiť lístky od zmluvných dopravcov.

1.9.4 CG Transit

Aplikácia CG Tranzit ponúka vyhľadávanie v cestovných poriadkoch vlakov, autobusov a MHD v Slovenskej aj v Českej republike. Ponúka aj offline režim, pričom aplikácia sa automaticky aktualizuje po pripojení na internet. Aplikácia ponúka zobrazenie liniek a ich zastávok, zobrazenie zastávok na mape, historiu posledných a obľúbených spojov. Pri vyhľadávaní spojov vieme nastaviť maximálny počet prestupov a prestupové časy pre vlaky, autobusy a MHD zvlášť. Na rozdiel od predchádzajúcich aplikácií nevieme obmedziť peší presun a vyfiltrovať len nízkopodlažné spoje. Aj CG Tranzit aplikácia v online verzii zobrazuje prípadné meškanie vlakov. Pre MHD a autobusové spoje túto funkciu neponúka.

1.9.5 Ubian

Aplikácia ponúka vyhľadávanie v cestovných poriadkoch vlakov, autobusov a MHD na celom Slovensku. Ako jediná aplikácia zobrazuje aktuálne polohy vozidiel na mape a ich meškania aj pre MHD Bratislava. Taktiež ponúkajú možnosť zobraziť najbližšie zastávky v okolí s možnosťou navigácie na zastávku a zobrazenie odchodov z tejto zastávky. Aplikácia UBIAN neponúka možnosť nastavenia preferencií pri vyhľadávaní spojov, ale umožňuje používateľovi vybrať jednu z možností: najskorší odchod, najrýchlejšia cesta, najmenej prestupov, najmenej chôdze alebo najmenej čakania. Aplikácie nefunguje v offline režime a neponúka zobrazenie liniek a ich zastávok.

Hoci aplikácia dokáže zobraziť meškanie spoja, nevyužíva túto informáciu pri vyhľadávaní. Ak používateľ hľadá spojenia, ktoré majú odchod zo zastávky od aktuálneho času, zobrazí mu len tie, ktoré podľa statických cestovných poriadkov majú mať v budúcnosti odchod z danej zastávky. Ak existuje spojenie, ktoré mešká a ešte na zastávku nedorazilo, nezobrazí ho.

Kapitola 2

Návrh

V tejto kapitole sa budeme zaoberať návrhom jednotlivých častí aplikácie. Najskôr spomenieme funkcie, ktoré bude naša aplikácia ponúkať. Ďalej popíšeme, ako sme spomedzi mnohých alternatív vybrali vhodný algoritmus pre potrebu našej aplikácie. Pri navrhovaní aplikácie sme sa venovali analýze získaných statických dát a dát o meškaní. Uvádzame tiež, ako budeme pristupovať k týmto dátam pri implementácii. V neposlednom rade spomenieme, ako bude fungovať naša aplikácia z pohľadu jej architektúry.

2.1 Funkcie aplikácie

Ako sme si mohli všimnúť v 1.9, všetky aplikácie ponúkajú vyhľadávanie z aktuálnej polohy rovnako, ako aj možnosť výberu zastávky priamo z mapy. Tieto funkcie bude používateľovi ponúkať aj naša aplikácia. Väčšina spomenutých aplikácií ponúkala zobrazenie histórie vyhľadávania, ktorá odľahčí používateľa od zadávania parametrov v prípade, že vyhľadáva väčšinou tie isté spoje. Túto funkcionality nájde používateľ aj v našej aplikácii. História vyhľadávania sa bude ukladať do pamäte zariadenia.

Vo väčšine aplikácií si používatelia vedia zobraziť všetky linky v MHD a postupnosti zastávok, ktoré obsluhujú. Túto funkciu bude ponúkať aj naša aplikácia. Čo sa týka nastavenia prídavných preferencií pri vyhľadávaní, aplikácie ponúkajú rôzne preferencie. Najčastejšími z nich sú: maximálny počet prestupov, minimálny čas na prestup, limit pre peší presun a zobrazenie len nízkopodlažých vozidiel. Naša aplikácia bude mať možnosť nastavenia všetkých týchto preferencií.

Jediná aplikácia, ktorá ponúka informácie o reálnom pohybe vozidiel vo forme meškania, je aplikácia *UBIAN*. Predpokladáme však, že aj táto aplikácia vyhľadáva v statických cestovných poriadkoch a pri vyhľadanom spoji len pripíše informáciu vo forme meškania. Uvažujeme tak na základe toho, že po vyhľadaní spojov zo zastávky po zadaní aktuálneho času aplikácia ponúkne také spoje, ktoré podľa statických cestovných poriadkov majú na túto zastávku v blízkej budúcnosti príchod. Ak existuje taký

spoj, ktorý mal odchod z danej zastávky v minulosti, ale má meškanie a na zastávke ešte neboli, aplikácia ho nezobrazí. Naša aplikácia ponúkne aj tie spojenia, ktoré kvôli meškaniu na zastávku ešte nedorazili.

Medzi ďalšie funkcionality patrí zakúpenie lístka priamo cez aplikáciu. Táto funkciu však nesúvisí priamo so zadáním našej práce a navyše je potrebná zmluva s dopravcami. Preto naša aplikácia túto možnosť ponúkať nebude.

Jediná aplikácia, ktorá dokáže vyhľadávať cesty aj v offline režime je *CG Tranzit*. Hoci je užitočné ponúknuť vyhľadávanie bez možnosti prístupu na internet, znamenalo by to, že nás algoritmus by bežal na klientskej strane. Keďže hlavnou úlohou našej aplikácie je vyhľadávanie spojov z reálnych dát, aplikácia bude fungovať online s tým, že zaručí používateľom vždy aktuálne spojenia.

V tabuľke 2.1 je zobrazený prehľad funkcionalít našej navrhovanej aplikácie a iných existujúcich aplikácií na vyhľadávanie spojov v MHD Bratislava.

	Imhd.sk	IDS BK	CP	CG Tranzit	UBIAN	Naša aplikácia
z aktuálnej polohy	✓	✓	✓	✓	✓	✓
výber zástavok z mapy	✓	✓	✓	✓	✓	✓
história vyhľadávania	✓	✗	✓	✓	✓	✓
zobrazenie liniek	✓	✓	✗	✓	✗	✓
max. počet prestupov	✓	✓	✓	✓	✗	✓
min. čas na prestup	✓	✗	✓	✓	✗	✓
limit pre peší presun	✓	✓	✗	✗	✗	✓
len nízkopodlažné vozidlá	✓	✗	✓	✗	✗	✓
meškanie MHD	✗	✗	✗	✗	✓	✓
kúpa lístkov	✗	✓	✓	✓	✓	✗
offline režim	✗	✗	✗	✓	✗	✗

Tabuľka 2.1: Tabuľka funkcionalít existujúcich aplikácií a navrhovanej aplikácie

2.2 Dáta

Pri vývoji aplikácie a pre jej testovanie sú nevyhnutné dáta. Pre účely aplikácie budeme potrebovať dáta zo statických cestovných poriadkov a dáta o meškaní jednotlivých jázd.

2.2.1 Dáta statických cestovných poriadkov

Od Dopravného podniku Bratislava sme získali statické cestovné poriadky, ktoré mali platnosť od 5.2.2018 – 31.12.2018. Dáta sú vo formáte GTFS.

GTFS

General Transit Feed Specification (GTFS) je dohodnutý formát dát, ktorý používajú tisíce poskytovateľov verejnej dopravy a mnohé softvérové aplikácie. Špecifikácia definuje súbory, v ktorých sú reprezentované entity v tabuľke. V stĺpcoch sú popísané vlastnosti entity a v každom riadku je nový záznam. V tejto sekcií analyzujeme len tie súbory, ktoré nám boli poskytnuté a opíšeme, ktoré z poskytnutých údajov využijeme.

Súbor *agency.txt* obsahuje údaje o prepravných spoločnostiach. Tento súbor nebudeme potrebovať, keďže všetky linky spravuje jedna spoločnosť.

Súbor *calendar.txt* obsahuje stĺpce *service_id*, *monday*, *tuesday*, *wednesday*, *thursday*, *friday*, *saturday*, *sunday*, *start_date*, *end_date*. Vlastnosť *service_id* predstavuje unikátny názov typu dňa (pracovný deň, víkend,...). Vlastnosti *monday*, ..., *sunday* môžu nadobúdať hodnoty 0 a 1. Ak je napríklad hodnota *monday* = 1, znamená to, že všetky pondelky medzi dátumom *start_date* a dátumom *end_date* patria do typu dňa *service_id*. Naopak, ak je hodnota rovná 0, tak nejazdí.

V súbore *calendar_dates.txt* sú vlastnosti: *service_id*, *date*, *exception_type*. V prípade, že vlastnosť *exception_type* má hodnotu 1, znamená to, že do *service_id* výnimčne patrí aj deň *date*. Ak je *exception_type* = 2, tak nepatrí.

Zastávky sú definované v súbore *stops.txt*. Zastávka je určená identifikačným číslom *stop_id*, ktoré je unikátné pre každú zastávku. Identifikačné číslo zastávky obsahuje na začiatku viac núl, ktoré môžeme odignorovať. Vlastnosť *stop_name* predstavuje názov zastávky. Táto vlastnosť sa nachádza v súbore viac krát, keďže v rámci zastávky existujú rôzne nástupištia. Unikátnymi vlastnosťami sú aj *stop_lat* a *stop_lng*, teda súradnice zastávky. Zastávka má definovanú aj *zone_id*, teda zónu mesta, do ktorej je zastávka priradená. Jednotlivé stĺpce sú v súbore oddelené čiarkou, avšak čiarka sa môže nachádzať aj v názve zastávky. V takom prípade je názov zastávky uvedený v úvodzovkách.

Súbor *routes.txt* obsahuje zoznam liniek. Pre každú linku definuje identifikačné číslo linky (*route_id*), číslo prepravnej spoločnosti (*agency_id*), skrátený názov linky (*route_short_name*) a mód linky (*route_type*). Podľa štandardu *GTFS* hodnota 0 definuje mód električku, hodnota 3 predstavuje autobus a hodnota 11 trolejbus. Vlastnosť *route_long_name* nie je definovaná a vlastnosť *route_text_color* pre nás nie je potrebná.

Jazdy jednotlivých liniek sú zaznamenané v súbore *trips.txt*, ktorý obsahuje vlastnosti: identifikačné číslo jazdy (*trip_id*), *service_id*, *trip_headsign*, *trip_short_name* a *direction_id*. Podľa GTFS špecifikácie by vlastnosť *service_id* mala predstavovať množinu rôznych *service_id* oddelených pomlčkou. V našich dátach existuje pre jeden záznam jazdy len jeden *service_id*. Vlastnosť *trip_headsign* predstavuje konečnú zastávku jazdy. Túto vlastnosť sice nevyužijeme, ale keďže obsahuje názvy zastávok

rovnako ako v súbore *stops.txt*, môže sa stať, že názov bude obsahovať oddelovač stĺpcov - čiarku. Vlastnosť *trip_short_name* nie je definovaná v žiadnom zázname jazdy. *Direction_id* nadobúda hodnoty 0 a 1. V dátach nám chýba informácia o nízkopodlažných spojoch, ktorú sme chceli využiť pri vyfiltrovaní hľadaných cest.

Súbor *stop_times.txt* definuje identifikačné číslo jazdy (*trip_id*), identifikačné číslo zastávky (*stop_id*), čas príchodu a odchodu jazdy na zastávku (*arrival_time* a *departure_time*), *stop_sequence*, *stop_headsign*, *pickup_type* a *drop_off_type*. Vlastnosti *arrival_time* a *departure_time* sú vždy rovnaké, preto jednu z nich budeme ignorovať. Vlastnosť *stop_sequence* predstavuje poradie zastávky v rámci linky. Poradnie nie je iterované od 1, ale začína rôznymi kladnými číslami. Položka *stop_headsign* nie je v žiadnom zázname definovaná. Vlastnosti *pickup_type* a *drop_off_type* nadobúdajú hodnoty 0 a 3. V *GTFS* špecifikácii znamená číslo 0, že jazda na tejto zastávke stojí vždy a 3 určuje, že jazda na zastávke stojí len na znamenie. Hodnoty sú pre obe vlastnosti vždy rovnaké, takže jednu z nich môžeme opäť ignorovať.

2.2.2 Dáta o meškaní

Dopravný podnik Bratislava nám poskytol aj informácie o meškaní za rok 2018, pre mesiac február, marec a apríl. Pre každý deň v mesiaci existuje súbor vo formáte .csv. Každý záznam obsahuje údaje: identifikačné číslo záznamu, dátum a čas, kedy bol záznam o meškaní zaevidovaný, identifikačné číslo vozidla, názov linky, poradie, číslo zastávky, názov zastávky a meškanie.

Z pozorovania dát vyplýva, že vlastnosť hodnota meškania nadobúda hodnotu 0, kladné a záporné celé čísla alebo hodnotu *n/a*. Zaujímavú nás len záporné čísla. Na prelome dní sa v dátach vyskytuje hodnota -1229 , ktorú budeme ignorovať.

Aby sme boli schopní správne zohľadniť meškanie v našom algoritme, potrebujeme hodnotu meškania zapísať do dátovej štruktúry. Na to potrebujeme vedieť identifikačné číslo zástavky a konkrétnu jazdu linky, na ktorej meškanie vzniklo.

Z poskytnutých dát vieme zistiť číslo zastávky, ktoré sa však nezhoduje s identifikačnými číslami zastávok získaných z dát statických cestovných poriadkov. Po menšej analýze sme zistili, že poskytnuté čísla zastávok sa podobajú na tie identifikačné. Ak sa vyskytuje v číslu zastávky nula, tak k nule, ktorá je najviac vpravo pridáme ďalšie tri nuly. Týmto získame identifikačné číslo zastávky.

Ďalej potrebujeme vedieť priradiť poskytnutý názov linky ku konkrétnnej linke zo statických dát. Názov linky je v statických dátach unikátny, takže linku môžeme vyhľadať aj podľa názvu. V dátach o meškaní existujú také názvy liniek, ktoré sa nenachádzajú v tých statických. Sú to napríklad trojciferné čísla začínajúce cifrou 4. Ak cifru 4 zmeníme na písmeno *N*, názvy liniek sa zhodujú s názvami nočných spojov. Existujú aj ďalšie názvy liniek, ktoré nenájdeme v zozname liniek statických cestovných poriadkov.

Tieto už budeme ignorovať.

Pôvodne sme si mysleli, že údaj poradie určuje poradie jazdy v rámci linky. Nie je to však tak, keďže hodnoty sú príliš malé. Máme však už všetky údaje potrebné na špecifikovanie správnej jazdy. Poznáme identifikačné číslo linky r , poznáme hodnotu meškania d jazdy t na zastávku p a poznáme aj čas a dátum τ , kedy boli tieto údaje zaznemenané. Po odpočítaní d minút z času τ , získame približný čas, kedy stojí hľadaná jazda t linky r na zastávke p . Neexistujú dve jazdy jednej linky, ktoré stoja na zastávke v rovnakom čase v rovnaký typ dňa. Týmto spôsobom získame jazdu, ktorej meškanie prisľúcha.

Získané dáta umožňujú našej aplikácii ponúknuť správne cesty s prihliadnutím na meškanie spojov od 5.2.2018 – 30.4.2018. Statické vyhľadávanie bez meškania bude správne fungovať od 5.2.2018 do konca roka 2018.

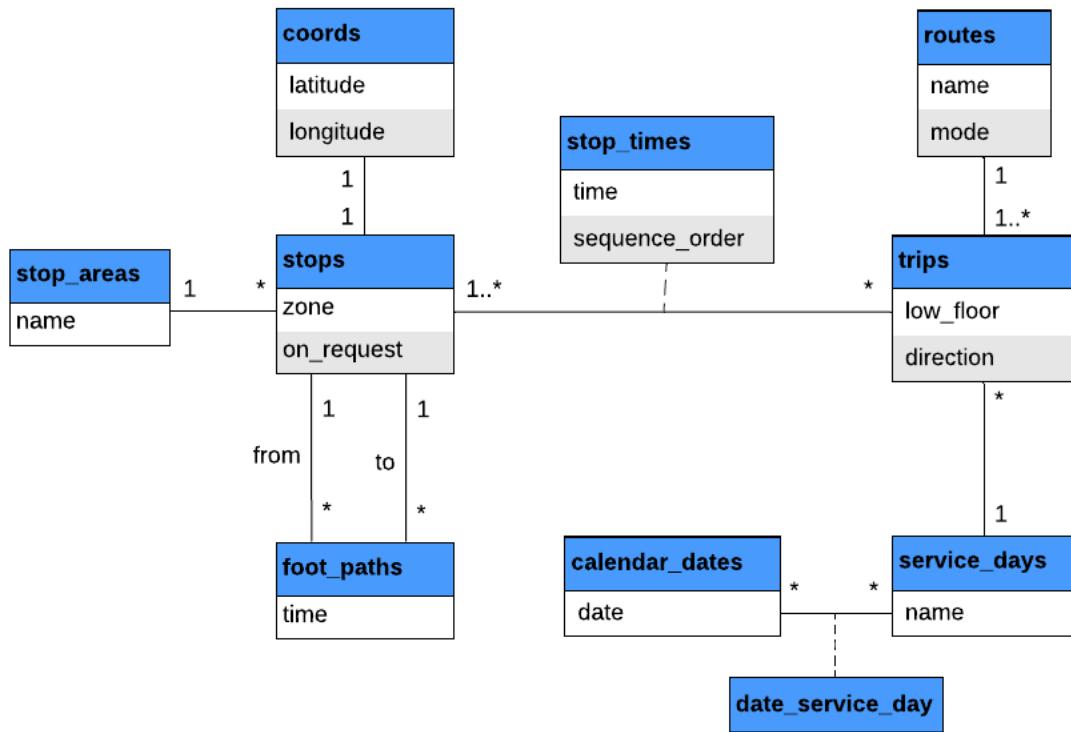
2.2.3 Pešie presuny

V dátach sa nenachádzajú informácie o peších presunoch. V sekcii 1.8.6 bolo navrhnuté, ako vypočítať peší presun s použitím Manhattanovej vzdialenosť, prípadne penalizovať prestupy podľa ich typu. napokon sme sa rozhodli využiť *Google API*, ktoré okrem iného ponúka možnosť vyhľadania peších vzdialenosťí medzi 2 bodmi prostredníctvom *Distance Matrix API*. Počet bezplatných dopytov na *Distance Matrix API* je však obmedzený. Z toho dôvodu nebudeme vyhľadávať pešie presuny dopytovaním na *Distance Matrix API* pri hľadaní cesty. Vytvoríme jednorazovo súbor *foot_paths.txt*, ktorý bude obsahovať údaje o peších presunoch, ktoré sú pre nás zaujímavé. Nebudeme vyhľadávať pešie vzdialenosť medzi každou dvojicou zastávok, nakoľko je to nepotrebné.

Budeme postupovať nasledovne: pre každú zastávku p nájdeme zastávky v okolí 800 metrov radiálnym vyhľadávaním. Pomocou *Distance Matrix API* zistíme vzdialenosť nájdených zastávok od zastávky p . Každú dvojicu zastávok uložíme do súboru spolu so zistenými vzdialosťami určenými v minútach.

2.3 Databáza

Na serveri budú uložené dáta aplikácie v PostgreSQL databáze. Databáza sa naplní pri prvotnom spustení aplikácie na serveri. Schéma databázy je popísaná entitno-relačným diagramom na obrázku 2.1.



Obr. 2.1: Entitno-relačný diagram

Entita *stop_areas* obsahuje zoskupenie zastávok, ktoré majú rovnaké názvy (*name*).

V entite *stops* evidujeme zónu mesta (*zone*), do ktorej zastávka patrí. Hodnotu *on_request*, určuje, či sa na zastávke nastupuje a vystupuje na znamenie. Každá zastávka má priradené súradnice, ktoré sa udržujú v entite *coords*.

V entite *coords* sú súradnice určené atribútmi zemepisná šírka (*latitude*) a zemepisná výška (*longitude*).

Entita *foot_paths* obsahuje atribút *time*, ktorý určuje čas v minútach potrebný na peší presun zo zastávky *from* na zastávku *to*.

V entite *routes* sa udržuje zoznam liniek jazdiacich aktuálne v bratislavskej MHD. Linka je určená názvom (*name*) a módom (*mode*). V Bratislave jazdia 3 rôzne módy: električka, trolejbus a autobus. Každá linka má počas dňa viaceré jazdy (*trips*).

Entita (*trips*) uchováva informáciu o tom, či je vozidlo, ktoré bolo pridelené konkrétnej jazde nízkopodlažné (*low_floor*), ktorým smerom ide (*direction*) a zároveň počas akých typov dní (*service_day*) jazda premáva. Každá jazda linky je tvorená postupnosťou zastávok, ktoré linka obsluhuje.

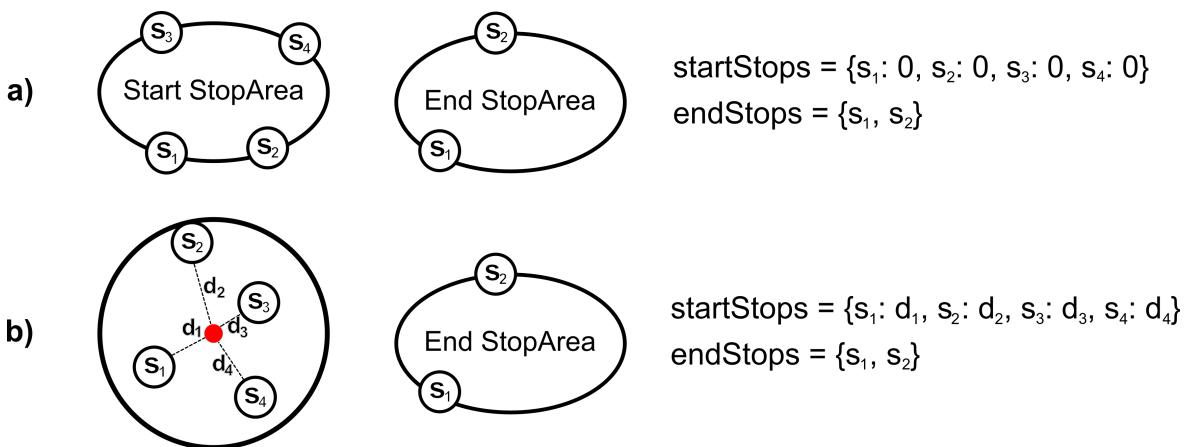
V entite *stop_times* je zachytený čas (*time*), kedy jazda stojí na zastávke a v akom poradí sú zastávky v rámci jazdy (*sequence_order*).

V entite *service_days* sú názvy rôznych typov dní, v ktorých premávajú jazdy liniek.

Entita *calendar_dates* obsahuje zoznam všetkých dátumov (*date*), v rozsahu platnosti aktuálneho cestovného poriadku. Ku každému dátumu je priradený typ dňa (*service_day*). Jeden dátum môže prislúchať k viacerým typom dňa a naopak jeden typ dňa môže prislúchať viacerým dátumom.

2.4 Parametre pre algoritmus

Používateľ môže pre vyhľadanie ciest zadať začiatočnú a konečnú zastávku tak, že vyberie zastávky z ponúkaného zoznamu zastávok. V zozname prestavuje zastávka *stop area*, ktorá zoskupuje zastávky s rovnakým názvom a rôznymi identifikačnými číslami. Preto vyhľadávanie ciest medzi začiatočnou a konečnou zastávkou bude vyhľadávanie ciest medzi dvomi množinami zastávok, ako je ilustrované na obrázku 2.2 a).



Obr. 2.2: Vstupné množiny zastávok

Vyhľadávanie z aktuálnej polohy

Začiatočný bod vie používateľ vybrať aj ako aktuálnu polohu, na ktorej sa nachádza. Radiálnym vyhľadávaním nájdeme okolité zastávky, ako bolo spomenuté v 1.6. Najbližšia vyhľadaná zastávka k aktuálnemu bodu nemusí znamenať optimálne riešenie a preto budeme považovať za začiatočnú zastávku každú z nich. Pre každú z týchto zastávok zistíme vzdialenosť od aktuálnej lokality v minútach použitím *Distance Matrix API*.

Tieto zastávky vytvoria množinu začiatočných zastávok s informáciou o iniciálnych vzdialostiach z aktuálnej polohy do začiatočnej zastávky. Tento prípad je zobrazený na obrázku 2.2 b). Do finálneho riešenia zakomponujeme peší presun od aktuálnej polohy po začiatočnú zastávku vybraných optimálnych ciest. Pri vstupných parametroch bude informácia o tom, či používateľ vyhľadával z aktuálnej lokality.

Vstupom pre algoritmus bude teda množina začiatočných zastávok aj s iniciálnymi vzdialenosťami. V prípade, že nevyhľadávame z aktuálnej lokality budú mať vzdialenosť hodnotu 0. Vstup algoritmu tvorí aj množina konečných zastávok, dátum a čas, 4 parametre predstavujúce používateľské preferencie a informácia o vyhľadávaní z aktuálnej lokality.

2.5 Algoritmus

Pri hľadaní algoritmu na nájdenie optimálnej cesty sme na jskôr siahli po najznámejšom vyhľadávacom grafovom algoritme. Dijkstrov algoritmus 1.2.1 sa zdal vhodný, avšak jeho vylepšená verzia A* algoritmus 1.2.2 je pri správne zvolenej heuristike efektívnejšia. Keďže zastávky, ktoré predstavujú vrcholy v grafe majú dané súradnice, uvažovali sme aj o optimalizovaní prehľadávaného priestoru. Pri štúdiu článkov sme narazili na rôzne optimalizácie prehľadávaného priestoru. Minimalizácia v okolí virtuálnej cesty a minimalizácia *bounding boxom* sú spomenuté v 1.5.

V prípade cestovných poriadkov je náročné správne namodelovať graf, ktorý dokáže efektívne spracovať časovo závislé dátá. V 1.4 boli spomenuté dva overené prístupy *Time dependent* a *Time expanded model*, ktoré tento problém riešia.

Ďalšou výzvou je prispôsobiť grafový vyhľadávací algoritmus, aby dokázal vypočítať optimálnu cestu, prihliadal na prestupy medzi rôznymi módmi a popri tom počítal s ďalšími pridanými kritériami. V 1.3 boli spomenuté návrhy časovo závislých algoritmov, ktoré niektoré z týchto problémov riešia. Algoritmus 1.3.1 sa dokáže vysporiadať aj viacerými módmi. Vráti však len jednu cestu. Na hľadanie alternatívnych ciest, by sme mohli použiť algoritmus spomenutý v 1.7.

Kvôli dynamickej povahé verejnej dopravy grafový prístup v kombinácii s vyhľadávacím grafovým algoritmom vyžaduje veľa pre-processingu a to sa odráža na výpočtových časoch. Výhodou vo verejnej doprave je, že vozidlá sa pohybujú po vyznačených linkách, ktorých trasy poznáme. Schéma verejnej dopravy sa preto dá zachytiť do pomerne jednoduchých dátových štruktúr. Tento fakt si všimli aj autori algoritmu RAPTOR, ktorý sme opísali v 1.8.

V našej aplikácii sme sa rozhodli použiť tento negrafový algoritmus. Jeho výhodou je, že nie je potrebné vytvárať model a nie je potrebné osobitne riešiť multimodalitu hromadnej dopravy. Ľahšie zvláda dynamickosť dát ako meškanie linky, zrušenie linky alebo zmenu trasy.

2.5.1 Použitie a prispôsobenie RAPTOR algoritmu

RAPTOR algoritmus počíta s tým, že každá linka pozostáva z jázd, ktoré majú rovnaké postupnosti zastávok ako bolo spomenuté v 1.8.1. Pri skúmaní dát sme zistili, že v

prípade našich dát táto vlastnosť nie je splnená. Bude potrebné prispôsobiť algoritmus aj dátovú štruktúru, aby zohľadňovali rôzne postupnosti zastávok v rámci linky.

Optimalizácia

Na základnú verziu RAPTOR algoritmu popísanú v 1.8.2 použijeme aj jeho optimalizáciu opísanú v 1.8.3, kedy označujeme zastávky, aby sme nemuseli prechádzať tie linky, ktorým sa nevylepšil čas $\tau_{k-1}(p)$. Označené zastávky predstavujú potenciálne prestupné zastávky. Rovnako využijeme aj optimalizácie *local-prunning* a *target-prunning*, ktoré nám zredukujú počet označených zastávok. Pri optimalizácii *target-prunning* však nebude využívať hodnotu $\tau^*(p_t)$, ktorá predstavuje najlepší čas do konečnej zastávky p_t . Nás zaujíma najlepší čas zo všetkých konečných zastávok, ktoré sú zoskupené v *stop area* $\tau^*(P_t)$.

Zapracovanie dát o meškaní

Kedže cieľom našej práce je vyhľadávanie ciest s prihliadnutím na prípadné meškania spojov, potrebujeme dosiahnuť, aby algoritmus pri hľadaní optimálnych ciest prihliadal na zaznamenané meškania. Spoliehame sa na to, že dátová štruktúra si bude držať pri každej zastávke v každej jazde hodnotu meškania. Prihliadať na meškania budeme len v prípade, že používateľ vyhľadáva v aktuálny deň. Inak bude údaje o meškaniach ignorovať a vyhľadávať cesty zo statických cestovných poriadkov.

Preferencie

Okrem základných povinných vstupných parametrov ponúkame používateľovi možnosť zvoliť si prídavné parametre vyhľadávania. Okrem začiatočnej a konečnej zastávky, času a dátumu vyhľadávania si bude môcť používateľ zvoliť aj maximálny počet prestupov, maximálnu dĺžku pešieho presunu v minútach, ako aj minimálny čas na prestup medzi dvoma linkami. Používateelia s hendičkom, kočíkom alebo bicyklom si budú môcť vyhľadávať len nízkopodlažné spoje. Ďalším vylepšením algoritmu je, že algoritmus bude zohľadňovať všetky tieto používateľské kritériá.

Alternatívne cesty

Po dobehnutí RAPTOR algoritmu, získame tabuľku najlepších časov pre každé kolo a každú zastávku. Pod najlepším časom v kole k pre zastávku p rozumieme najskorší možný čas, ktorým sa vieme dostať na zastávku p na $k-1$ prestupov. Keďže používame optimalizáciu *local-prunning*, máme pre každú zastávku evidovaný čas $\tau^*(p)$, ktorý predstavuje celkovo najlepší čas pre zastávku p bez ohľadu na počet prestupov. Našim cieľom však nie je získať jednu najkratšiu cestu, ale alternatívne optimálne cesty.

V sekcii 1.8.6 sme spomenuli vylepšený RAPTOR algoritmus, ktorý pre každú zastávku nehľadá najlepší čas, ale K optimálnych cest. Prihliada na to, aby cesty neboli podobné na veľkej časti úsekov. Po dobehnutí algoritmu máme pre každú zastávku najviac K optimálnych cest.

Pôvodne sme použili spomínané navrhnuté vylepšenie RAPTOR algoritmu, prichádzali sme však na problém, ako správne zvoliť premennú K , aby sme neodfiltrovali tie cesty, ktoré môžu byť správne. Navyše po každom vylepšení času $\tau_k^*(p)$ bolo potrebné prejsť všetky cesty v množine $\mathcal{J}_k(p)$ a odfiltrovať najmenej optimálne.

Rozhodli sme sa napokon využiť základnú verziu algoritmu, kedy získavame tabuľku najlepších časov. Budeme si pamätať ako sme dosiahli každý z vylepšených časov. Budeme evidovať z ktorej zastávky sme sa na označenú zastávku dostali a akým spôsobom, či už spojom alebo peším presunom. Cesty vytvoríme len také, ktoré končia v niektornej z konečných zastávok a začínajú v niektornej zo začiatočných zastávok. Tieto cesty budeme filtrovať podľa potrebných filtrov, aby sme získali čo najoptimálnejšie cesty.

Filtre

Všetky cesty, ktoré budú výsledkom dopytu musia vyhovovať nasledujúcim filtrom. RAPTOR algoritmus operuje v kolách a v každom kole hľadá zastávky, ktorým vieme vylepsiť čas. Tieto zastávky dosahuje akciami. Každá akcia začína v niektornej z označených zastávok p_m v kole $k - 1$ a končí v inej dosiahnuteľnej zastávke p_i v kole k . Ako sme už spomínali zastávka p_m je prestupné miesto. Pod akciou teda rozumieme buď peší presun zo zastávky p_m do zastávky p_i alebo úsek jazdy pričom platí, že p_m je zastávka na nastúpenie a zastávka p_i je zastávka na vystúpenie.

Cesta je vlastne postupnosťou zastávok (p_0, p_1, \dots, p_n) a akcií $(a_0, a_1, \dots, a_{n-1})$. Nechceme povoliť, aby v rámci cesty boli obe akcie p_i, p_{i+1} akciami pešieho presunu. V súčte by časy trvania po sebe nasledujúcich peších presunov mohli prekročiť maximálne zadané trvanie pešieho presunu.

Ďalej nechceme povoliť, aby zastávka p_0 a p_1 patrili do rovnakej *stop area*. Toto obmedzenie platí aj pre zastávky p_{n-1} a p_n .

Ďalej filtrujeme cesty porovnávaním medzi sebou. Majme cesty, ktoré už prešli cez všetky vyššie spomínané kritériá. Nech cesta J_1 začína v čase dt_1 , končí v čase at_1 a obsahuje n prestupov. Nech cesta J_2 má začiatočný čas dt_2 , konečný čas at_2 a obsahuje n prestupov. V sekcii 1.8.6 boli okrem iného spomenuté aj podobné cesty. Problém podobných ciest sme ilustrovali na obrázku 1.6. V prípade že platí $dt_1 = dt_2, at_1 = at_2$ a $m < n$, tak cestu J_2 chceme odfiltrovať. V inom prípade nech platí, že $dt_1 = dt_2, at_1 = at_2$ a $m = n$ a zároveň platí, že jednotlivé úseky jázd patria rovnakej linke, tak odfiltrujeme jednu z nich. Tieto cesty sa líšia len prestupným miestom.

Chceme filtrovať nevýhodné cesty. Nech platí $dt_1 \leq dt_2$ a $ar_1 > ar_2$, tak cestu J_1

odfiltrujeme. Rovnako aj keď platí $dt_1 < dt_2$ a $ar_1 \geq ar_2$ chceme cestu J_1 odfiltrovať.

Dostávame sa ešte k takým cestám, pre ktoré platí $dt_1 \leq dt_2$, $at_1 < at_2$ a $m > n$. Cestujúcich môže zaujímať aj cesta J_2 , ktorou sa síce nedostanem do konečnej zastávky najrýchlejšie, ale obsahuje menší počet prestupov ako cesta J_1 . Otázkou je s akým omeškaním času príchodu je cesta J_2 ešte priateľná. Rozhodli sme sa prestup penalizovať x minútami. Nech platí že cesta J_1 je cesta, ktorou sa dostanem najskôr do konečnej zastávky a nech $(at_2 - at_1) > (m - n) * x$, tak cestu J_2 odfiltrujeme.

Nasledujúce cesty

Takto vylepšený algoritmus nám nájde niekoľko optimálnych ciest najskôr od hľadaného času a dátumu. My však chceme používateľovi zobraziť určitý počet ciest na jednu podstránku a teda potrebujeme vyhľadané cesty doplniť nasledujúcimi cestami. Riešením tohto problému by mohol byť rRAPTOR algoritmus spomínaný v 1.8.4. Takto vylepšený algoritmus však nepočítá s tým, že cesta môže začínať peším presunom. Budeme preto používať iný mechanizmus hľadania nasledujúcich ciest. Zvolíme si konštantu x , ktorá predstavuje minimálny počet ciest na podstránke. Keď vyhľadáme cesty po čase τ , algoritmus nájde k optimálnych ciest, pričom začiatok čas poslednej cest je t . Ďalšie vyhľadávanie spustíme s časom $t + 1$ minúta. Vyhľadané cesty pridáme k predchádzajúcim a na tieto cesty aplikujeme ešte niektoré z filtrov. Takto budeme pokračovať, kým nemáme vzhľadaných aspoň x ciest.

Zhrnutie

Výstupom z nášho algoritmu bude množina aspoň x ciest začínajúcich na niekorej zastávke p_s zo *stop area* P_s , po čase τ a končiacich v niekorej zastávke p_t zo *stop area* P_t . Jednotlivé cesty sú optimálne, nie sú si podobné na väčšine úsekov a vychovávajú prípadným používateľským preferenciám.

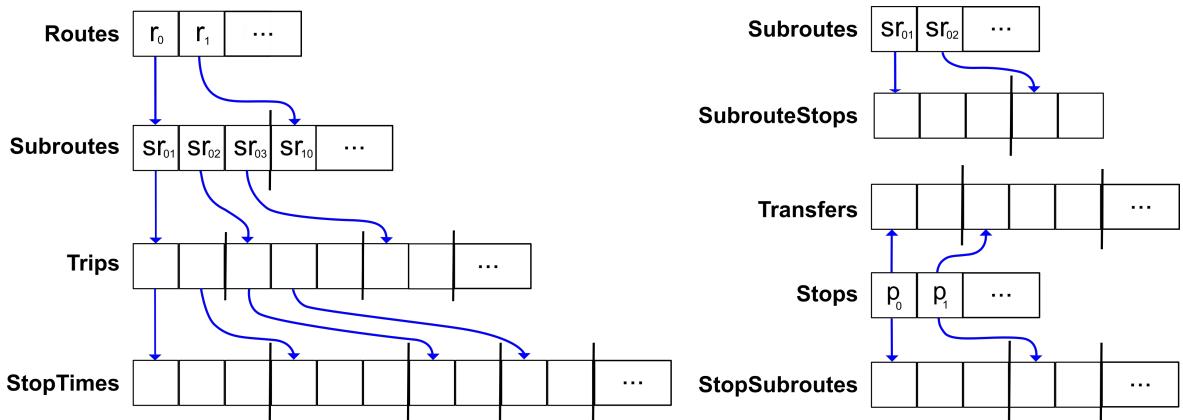
2.6 Dátová štruktúra

Aby algoritmus dokázal rýchlo nájsť optimálne cesty, potrebuje efektívne navrhnutú dátovú štruktúru. Pri jej návrhu sme sa inšpirovali dátovou štruktúrou z 1.8.5, ktorá bola navrhnutá pre základnú verziu RAPTOR algoritmu. Dátová štruktúra uvedená v článku počíta so skutočnosťou, že jednotlivé jazdy v rámci linky majú rovnakú postupnosť zastávok. Avšak v našich dátach to tak nie je.

Linky v našich dátach obsahujú jazdy, ktoré idú jedným aj druhým smerom. Zástavky, cez ktoré prechádza linka majú súčasne rovnaký názov, ale majú iné identifikačné čísla, ktorými sú definované a najmä ich postupnosť je iná. Okrem rôznych smerov obsahuje linka aj také jazdy, ktorých postupnosť zastávok je iná ako pri väčšine. Najmä v

ranných a večerných hodinách prechádzajú niektoré jazdy len cez určitú podpostupnosť zastávok.

RAPTOR algoritmus potrebuje, aby všetky jazdy, ktoré patria konkrétnej linke mali rovnakú postupnosť zastávok. Rozhodli sme sa preto zoskupiť jazdy linky s rovnakou postupnosťou zastávok do úsekov linky (*subroutes*). Teraz platí, že 1 linka (*route*) má viacero úsekov a jednému úseku linky prislúcha viac jazd linky (*trips*). Upravená štruktúra je zachytaná na obrázku 2.3.



Obr. 2.3: Návrh dátovej štruktúry

V algoritme sa potrebujeme často dopytovať na všetky linky, ktoré stoja na konkrétnej zastávke. Úseky linky pre zastávku vieme jednoducho získať z poľa *StopSubroutes* a hľadanú linku z poľa *Routes*. Pole *SubrouteStops* priraďuje postupnosť zastávok konkrétnemu úseku linky. Zastávky sú zoradené podľa časového príchodu linky na jednotlivé zastávky.

Algoritmus potrebuje často k svojim výpočtom nájsť jazdu linky t , ktorá stojí na zastávke p najskôr od zadanej času τ v zadanom type dňa (pracovný deň, víkend, ...). Aby sme pri hľadaní jazdy linky nemuseli prechádzať všetky prvky poľa *Trips*, zoskupili sme navyše jednotlivé jazdy v rámci úseku podľa typu dňa.

V poľi *StopTimes* nebude prvok obsahovať len informáciu o čase, kedy podľa statického poriadku má stáť jazda linky na zastávke, ale aj informáciu o predpokladanom časovom meškaní danej jazdy na zastávku. K tomuto údaju bude algoritmus pristupovať len vtedy, ak používateľ vyhľadáva v aktuálny deň.

Poľo *Transfers* obsahuje pre každú zastávku p pole zastávok, ktoré sa nachádzajú v blízkosti zastávky p spolu s informáciu o časovej vzdialenosť v minútach. Maximálna vzdialenosť peších presunov je 8 minút.

2.7 Architektúra systému

Klient sa bude dopytovať na server pre vyhľadanie spojenia. Server spustí výpočet nad dátovou štruktúrou, ktorá má aktuálne cestovné poriadky s informáciou o prípadnom meškaní spojov a vráti odpoveď klientovi.

Algoritmus bude pracovať nad dátovou štruktúrou, ktorá bude obsahovať stále aktuálne dátá. Dátová štruktúra bude rovnako ako algoritmus uložená na serverovej strane.

2.7.1 Serverová strana

Na serverovej strane bude bežať aplikačný server *Tomcat*. Na uchovanie dát použijeme relačnú databázu. Na komunikáciu s klientom budeme používať *REST API*.

2.7.2 Klientská strana

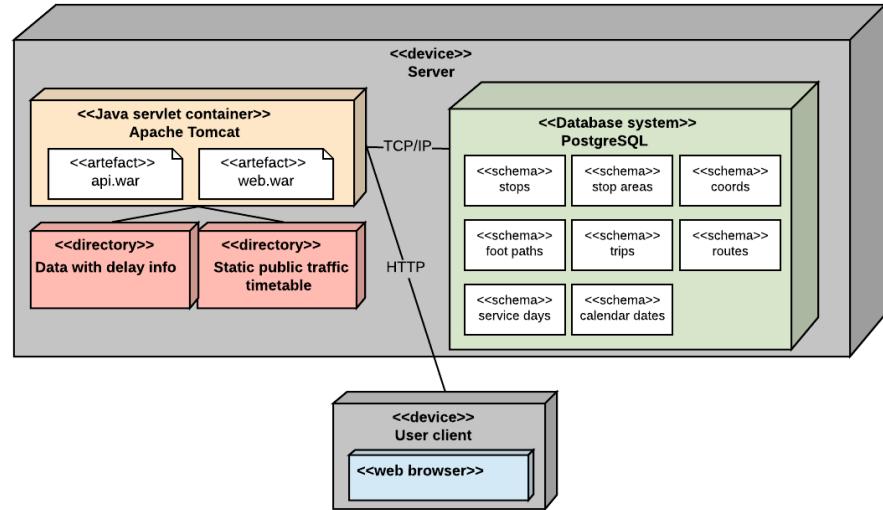
Na klientskej strane sme sa rozhodli pre progresívnu webovú aplikáciu. Je to webová aplikácia, ktorá sa dokáže správať ako mobilná aplikácia, neustále sa aktualizuje, pričom nie je potrebná jej inštalácia. Po návšteve webovej stránky na mobilnom zariadení používateľ dostane upozornenie od stránky, či si ju chce uložiť do zariadenia ako mobilnú aplikáciu. Progresívna webová aplikácia zaberá minimum miesta v pamäti a má svoj vlastný úložný priestor, kde sa budú ukladať preferencie a história vyhľadávania.

2.7.3 Spracovanie dát

Pri spustení aplikácie alebo po aktualizácii cestovných poriadkov sa spustí služba, ktorá z úložiska, kde sú aktuálne cestovné poriadky namapuje dátá do našej databázy a do dátovej štruktúry.

Ďalšia služba bude vytvorená na spracovanie údajov o meškaní. Hoci máme v súbore pre konkrétny deň údaje o meškaní jázd na celý deň, chceme sa čo najviac priblížiť reálnemu nasadeniu. Budeme teda počítať s tým, že nové údaje o meškaní pribúdajú po minúte. Služba bude spúšťaná každú minútu. Bude čítať súbor pre aktuálny deň, získa záznamy, ktoré pribudli v poslednej minúte a aktualizuje meškanie pre konkrétné jazdy. Údaje o meškaní sú evidované pre zastávku s , na ktorej meškanie vzniklo. Aktualizácia meškania jazdy bude prebiehať tak, že pre všetky zastávky jazdy od zastávky s až po konečnú zastávku jazdy zapíše do dátovej štruktúry hodnotu získaného meškania.

Spôsob akým bude aplikácia nasadená je znázornená na obrázku 2.4.



Obr. 2.4: Diagram nasadenia

Kapitola 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.

3.1 Technológie

Serverovú časť aplikácie sme implementovali v jazyku *Java*, konkrétnie vo verzii *Java 11* 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*, konkrétnie verziu *Vue 2*. Zároveň sme použili knižnicu *Vuex* na spravovanie stavu aplikácie.

3.2 Dáta

3.2.1 Parsovanie dát zo súboru

Dáta statických cestovných poriadkov

Náš projekt obsahuje dáta vo formáte GTFS (2.2.1), ktoré reprezentujú cestovné poriadky v bratislavskej MHD platné pre obdobie v minulosti. Dáta zo súboru *stops.txt* sme zachytili v troch 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.

Kedže v dátach nemáme informáciu o peších presunoch medzi jednotlivými zastávkami, implementovali sme *FootPathsFileGenerator*. Táto služba nám využitím *Google*

Distance Matrix API zistí vzdialenosť medzi jednotlivými zastávkami a vytvorí súbor *foot_paths.txt*. Súbor obsahuje len také dvojice zastávok, ktoré sme definovali v sekcií 2.2.3. 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 zistuje, ktoré typy dní prislúchajú konkrétnemu dátumu. Objekty *CalendarDate* budú uchovávať dátumy z rozmedzia dátumov platnosti cestovného poriadku. V objektoch *Service-Day* 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*.

Dáta o meškaní vozidiel

V projekte máme aj dáta o meškaniach vozidiel. Ich formát sme spomínali v 2.2.2. Potrebujeme aby sa každú minútu prečítali tie dáta ktoré patria k aktuálnemu dňu a vznikli v predchádzajúcej minúte. Použili sme springovskú anotáciu *@Scheduled* s parametrom *fixedRate=60000*, ktorá zaručí, že každú minútu sa bude vykonávať funkcia, ktorá s súbore hľadá záznamy o meškaní z predchádzajúcej minúty. Vyfiltrujú sa tie záznamy, ktoré majú zápornú hodnotu meškania a snažíme sa nájsť jazdu, ktorej tieto údaje prislúchajú. Zaujímavé údaje sú zastávka *p*, linka *r*, čas záznamu *t* a meškanie *d*. Ak nájdeme v linke *r* jazdu, ktorá stojí na zastávke v čase *t-d*, nastavíme jej hodnotu meškania *d* v dátovej štruktúre, ako aj zastávke *p* a všetkým nasledujúcim za ňou.

3.2.2 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 ľaháť 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.

3.2.3 Testovacie dátá

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 počítał čo najoptimálnejšie cesty. Pri veľkej dátovej sade sa ľahko 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 jazd a 185 záznamov *stopTimes*. Pre porovnanie reálnej sady obsoahuje 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 profile z frameworku *Spring*. Každý profil má definované premenné prostredia, ktoré pre nás 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.

3.3 Dátová štruktúra

Dátová štruktúra sa vytvára pri spustení serverovej aplikácie. Najskôr sme začali implementovať vytvorenie dátovej štruktúry podľa návrhu, teda ľaháním údajov z databázy. Z dôvodu pomalého ľahania údajov z databázy sme sa rozhodli obísť databázu a dátovú štruktúru vytvárame hneď pri parsovaní súboru. Tento proces trvá približne 5 - 7 minút v závislosti od zariadenia. Toto trvanie je prijateľné pri produkcií keďže aplikácia sa bude spúštať len pri aktualizácii cestovných poriadkov a v prípade nejakých výpadkov. V procese implementácie je však 7 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.

Premýšľali sme aj o umiestnení dátovej štruktúry a algoritmu na klientskej strane. Toto riešenie by umožňovalo používateľovi vyhľadávať aj offline, kedy by aplikácia nevyhľadávala s prihlásením na meškanie vozidla. 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.

3.3.1 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 implementačnej fáze. `@Component` je vlastne `@Bean` a ten sa v *Springu* správa ako *Singleton*. Trieda *DataStructureModel* definuje model dátovej štruktúry pre RAPTOR algoritmus, ktorá je tvorená šiestimi dátovými štruktúrami.

```
public class DataStructureModel implements Serializable {
    private Map<Route, List<Subroute>> routeSubroutes;
    private Map<Long, List<Transfer>> stopTransfers;
    private Map<Stop, List<String>> stopSubroutes;
    private Map<String, Subroute> subroutesByIndex;
    private Map<String, Map<Long, Integer>> stopIndexInSubroute;
    private List<CalendarDate> calendarDates;
}
```

Štruktúra *routeSubroutes* je definovaná slovníkom, 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 .

Slovník *stopSubroutes* ma zastávku p ako kľúč a hodnotou je pole indexov úsekov linky, ktoré stoja na zastávke p . Index úseku linky je textový reťazec, pretože má formát {identifikačné číslo linky} _ {číslo úseku v rámci linky}. Tento index sa vytvára pri vytváraní dátovej štruktúry, ako aj celý objekt *Subroute*.

Slovník *subroutesByIndex* obsahuje zoznam všetkých úsekov liniek, s tým že je indexovaný podľa unikátneho reťazca.

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.

3.3.2 Naplnenie dátovej štruktúry

Pri plnení dátovej štruktúry sa zavolá *LoadService*, ktorý parsuje dáta zo súborov GTFS do polí objektov *StopArea*, *Stop*, *Coords*, *Trip*, *Route*, *CalendarDate*, *ServiceDay*, *FootPath*, *StopTime*. Tieto objekty sa navzájom na seba odkazujú. Ako sme spomínali v sekcií 3.2.1, tieto polia nie sú len kópiou GTFS dát, ale obsahujú nejaké zmeny. Napríklad zoskupenie zastávok s rovnakým menom do objektu *StopArea* alebo súradnice do objektov *Coords* a pod.. Pri vytváraní dátovej štruktúry z polí objektov ešte navyše zoskupujeme jazdy s rovnakou postupnosťou zastávok do úsekov linky *Subroute*.

Ako sme už spomínali na začiatku, dátovú štruktúru po naplnení dátami serializujeme, aby sme neboli pri implementácii obmedzovaný časom spustenia aplikácie. Objekty v dátovej štruktúre sa však na seba odkazujú cyklicky a preto je potrebné cyklické referencie pri plnení dátovej štruktúry odstrániť.

3.4 Časový simulátor

Keďže máme k dispozícii historické dáta o meškaní vozidiel a platnosť cestovných poriadkov je tiež v minulosti, potrebujeme v aplikácii simulať čas v minulosti. V konfigurácii nastavujeme čas, ktorý, aplikácia začína spustením. Na prelome dňa sa čas vynuluje a navýší 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 aktualizujeme 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. Získané hodnoty meškania v minútach priradzujeme k získanej zastávke a ku všetkým zastávkam nasledujúcim za ňou.

3.4.1 Vlastná trieda *Time*

Na serverovej strane sme si vytvorili vlastnú triedu *Time*, keďže s časom potrebujeme často pracovať nezávisle od dátumu. Jej parametrami 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 $t=23:59:59$, 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.

3.5 RAPTOR Algoritmus

3.5.1 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:

```
private Map<Integer, List<StopArrivalTime>> roundArrivals;
```

Kľúčom je kolo a jeho hodnotou je pole objektov *StopArrivalTime*. Tento objekt zdeleniuje 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:

```
private 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. Definovali sme slovník *roundActions*.

```
private 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. Môžu totiž existovať také 2 akcie, že zastávku sme dosiahli čas 6:00 v kole 2 a zároveň sme tento čas dosiahli v kole 3. Druhá akcia je nevýhodnejší

takže je 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 začína v aktuálnej lokalite, pripojíme k ceste na začiatok ešte prvotný peší presun z aktuálnej lokality do začiatočnej zastávky.

Target pruning

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

```
private 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 niektoréj zo zastávok v spoločnej *stopArea* dostali skorej, čas nevylepšíme a zastávku neoznačíme ako prestupné miesto na ďalšie prehľadávanie.

3.5.2 Hľadanie najbližšej vyhovujúcej jazdy

RAPTOR algoritmus funguje tak, že hľadáme najskôršiu vyhovujúcu jazdu linky r po čase τ , ktorá stojí na zastávke p a jazdí v niektorý zo zadaných typov dní (*ServiceDay*). V algoritme nepracujeme s linkami, ale s objektami *Subroute*. Tento objekt obsahuje slovník, ktorý je indexovaný objektom *ServiceDay* a hodnotou je pole jázd (*SubrouteTrip*). Tým pádom jazdu nehľadáme vo všetkých jazdách, ktoré patria linke, ale len v takých, ktoré stoja na zastávke p v niektorý požadovaných typov dní.

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 príchodu jazdy na zastávku p menší ako τ , 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 τ , máme v premennej *lastTrip* vyhovujúcu jazdu. Index tejto jazdy si uložíme do poľa *foundTripsIndices* v prípade, že by sme pre túto *subroute* potrebovali nájsť ešte nejakú skôršiu jazdu v ďalšom hľadaní. Túto optimalizáciu môžme spraviť, keďže jazdy, ktoré prídu na zastávku neskôr ako označená jazda, nás už nezaujímajú.

Môže nastať prípad, že všetky jazdy stoja na zastávke p skôr ako v čase τ , preto sme museli pridať podmienku, ktorá overí, či príchod poslednej jazdy na zastávku p je neskôr ako čas τ . Ak nie je, nevrátime žiadnu cestu.

Časť implementácie vyhľadávania najbližšej vhodnej jazdy môžeme vidieť v ukážke kódu:

```

if (arrivalTimeOfLastTripEarlierThanSearchedTime(subrouteTrips, searchedTime,
    stopIndex)) { return null; }
for (int i = this.foundTripsIndices.get(subrouteServiceDay); i >= 0; i--) {
    SubrouteTrip trip = subrouteTrips.get(i);
    Time tripStopTime = trip.getStopTimeObjects()[stopIndex].getTime();
    if (!onlyLowFloor || trip.isLowFloor()) {
        if (tripStopTime.isBefore(searchedTime)) { break; }
        else { lastFound = trip; }
        foundTripsIndices.put(subrouteServiceDay, i);
    }
}

```

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.

Stále sme však prichádzali na problém, že linky nie sú zoradené správne. Naimplementovali sme sližbu, ktorá nám vyvori csv súbor s čitateľným cestovním poriadkom. Po hlbšom skúmaní sme zistili, že atribút *sequence order* nezoraduje 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 čas 23:59:59. Dokonca existujú také jazdy, ktoré prichádzajú na všetky zastávky linky až na druhý deň ako môžeme vidieť na obrázku 3.1.

42903	Soboty_1					Nedäle+Sv_2				
	24:30:00	25:30:00	26:30:00	3:30	23:30	24:30:00	25:30:00	26:30:00	3:30	23:30
9300007	24:30:00	25:30:00	26:30:00	3:30	23:30	24:30:00	25:30:00	26:30:00	3:30	23:30
35100004	24:30:00	25:30:00	26:30:00	3:30	23:30	24:30:00	25:30:00	26:30:00	3:30	23:30
36800002	24:31:00	25:31:00	26:31:00	3:31	23:31	24:31:00	25:31:00	26:31:00	3:31	23:31
38800002	24:32:00	25:32:00	26:32:00	3:32	23:32	24:32:00	25:32:00	26:32:00	3:32	23:32
1100002	24:33:00	25:33:00	26:33:00	3:33	23:33	24:33:00	25:33:00	26:33:00	3:33	23:33
20600002	24:34:00	25:34:00	26:34:00	3:34	23:34	24:34:00	25:34:00	26:34:00	3:34	23:34
16400002	24:34:00	25:34:00	26:34:00	3:34	23:34	24:34:00	25:34:00	26:34:00	3:34	23:34
31500002	24:36:00	25:36:00	26:36:00	3:36	23:36	24:36:00	25:36:00	26:36:00	3:36	23:36
50100002	24:37:00	25:37:00	26:37:00	3:37	23:37	24:37:00	25:37:00	26:37:00	3:37	23:37
35800001	24:39:00	25:39:00	26:39:00	3:39	23:39	24:39:00	25:39:00	26:39:00	3:39	23:39
2100004	24:40:00	25:40:00	26:40:00	3:40	23:40	24:40:00	25:40:00	26:40:00	3:40	23:40
33900002	24:42:00	25:42:00	26:42:00	3:42	23:42	24:42:00	25:42:00	26:42:00	3:42	23:42
13500002	24:43:00	25:43:00	26:43:00	3:43	23:43	24:43:00	25:43:00	26:43:00	3:43	23:43

Obr. 3.1: Ukážka exportovaného cestovného poriadku

Tieto výnimky v dátach nám po menšej analýze nedávali zmysel a keďže sme boli už v pokročilej fáze vývoja algoritmu, rozhodli sme sa takéto dátu 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 indexov *subroutes*, ktoré na zastávke p operujú. *Subroute* je zoskupenie jázd s rovnakou postupnosťou zastávok v rámci linky. Subroute si udržuje pole jázd indexovaných identifikačné číslami *serviceDay*, v ktorý táto jazda operuje. Toto pole jázd je zoradené podľa času príchodu jazdy na zastávku.

Prihliadanie na meškanie vozidiel

Dostávame sa k jadru našej práce a prispôsobeniu algoritmu, aby prihliadal na dátu 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. Konkrétnie dátu vkladáme do štruktúry *stopSubroutes*, obsahujúcej aj objekty *StopTimeObject*, 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*).

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 zmená, 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 t_1 ma príchod na zastávku p_1 v čase 15:05 ale má meškanie $d_1 = 10$ minút. Predpokladaný príchod jazdy t_1 na zastávku p_1 je teda 15:15. Za ňou nasledujúca jazda t_2 má na zastávku p_1 prísť 15:10 a nemá zistené žiadne meškanie. V prípade, že by používateľ zadal vyhľadávanie v aktuálny deň napríklad o $\tau = 14:59$ zo zastávky p_1 , algoritmus by vybral jazdu t_2 ako najskoršiu jazdu, ktorú vie používateľ chytiť na zastávke p_1 . Správnym riešením by však bolo vybrať jazdu t_0 . Ilustráciu tejto situácie môžeme vidieť na obrázku 3.2.

	t_0/d_0	t_1/d_1	t_2/d_2	t_3/d_3	...
p₁	15:00/0	15:05/10	15:10/0	15:15/0	...
...					

Obr. 3.2: Hľadanie najbližšej jazdy s prihliadnutím na meškanie

Pri hľadaní najbližšej jazdy teda zvažujeme dve rôzne situácie. V prípade, že vyhľadávame v iný deň ako v aktuálny deň, hľadáme jazdu tak ako sme spomínali na začiatku

sekcie. Ak hľadáme v aktuálny deň, prechádzame cez všetky jazdy linky, kým nenájdeme takú jazdu t , pre ktorú platí: $stopTime(t, p).time + stopTime(t, p).delay \geq \tau$ a zároveň je tento čas najmenší možný.

3.5.3 Zapracovanie používateľských preferencií

Maximálny počet prestupov

RAPTOR algoritmus končí vtedy, ak už neexistujú žiadne ďalšie označené zastávky, teda ak sa v predchádzajúcim kole nevylepšil žiadny z časov. RAPTOR algoritmus beží v kolách. Prvé kolo znamená 0 prestupov, druhé kolo 1 prestup, atď. Aby bol algoritmus obmedzený počtom prestupov, stačí pridať podmienku a obmedziť algoritmus navyše maximálnym počtom kôl. V kóde je táto podmienka umiestnená na začiatku vyhľadávania:

```
public RaptorResults search(SearchParams searchParams) {
    Set<Stop> markedStops = initializeMarkedStops(searchParams);
    RaptorResults raptorResults = initializeRaptorResults(searchParams);
    while(markedStops.size() > 0 &&
        raptorResults.getRound() <= searchParams.getMaxNumberOfTransfers()){
        ...
    }
    return raptorResults;
}
```

Maximálny čas pešieho presunu

Pri prechádzaní zastávok, ktoré boli v predchádzajúcim kole vylepšené hľadáme pešie presuny do okolitých zastávok. Okolité zastávky boli pre každú zastávku radiálne vyhľadané v okolí x metrov pri parsovaní dát. 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. Umiestnenie v kóde:

```
private void traverseTransfers(Set<Stop> markedStops, int maxTimeOfWalking){
    for(Transfer transfer: transfers){
        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 pred tvorením zoznamu výsledných ciest.

Minimálny čas na prestup

Nech nastane prípad, že na zastávke A vystúpime v čase 6:00 a hodnota minimálneho času na prestup sú 2 minúty. Nech jazda t_1 stojí na zastávke A v čase 6:00 a jazda t_2 v čase 6:05. Hľadáme najskoršiu jazdu linky r , ktorú vieme stihnuť na zastávke A . Správnym riešením je jazdu t_2 , nie jazdu t_1 .

V prípade, že sa jedná o prvú akciu v ceste, nechceme pripočítavať minimálny čas na prestup. Pri prvom vyberaní jazdy ešte neprestupujeme. Začiatocný čas, teda nastavíme podľa kola, v ktorom sa algoritmus práve nachádza. V ukážke je ukízené ako získavame čas τ , ktorý následne posielame do triedy *TripFinder*, ktorá nám vráti najskorsšiu využívajúcu jazdu po čase τ .

```
private Time getStartingTime(RaptorResults raptorResults, int minTimeForTransfer,
    Time arrivalTime) {
    return raptorResults.getRound() == 1
        ? arrivalTime
        : new Time(arrivalTime).addMinutes(minTimeForTransfer);
}
```

Vyfiltrovanie len nízkopodlažných vozidiel

Informáciu o nízkopodlažných vozidlách sme nedostali v dátach statických cestovných poriadkoch. Vygenerovali sme si ich náhodne pri importovaní dát a teda každá jazda linky má priradenú informáciu o type vozidla. V kóde v sekcii môžeme vidieť podmienku, ktorá zabezpečí, aby v prípade nastavenia vyfiltrovania len nízkopodlažných spojov, algoritmus hľadal len také jazdy, ktoré obsluhujú nízkopodlažné vozidlá.

3.6 Klientská strana

3.6.1 Implementácia klientskej strany

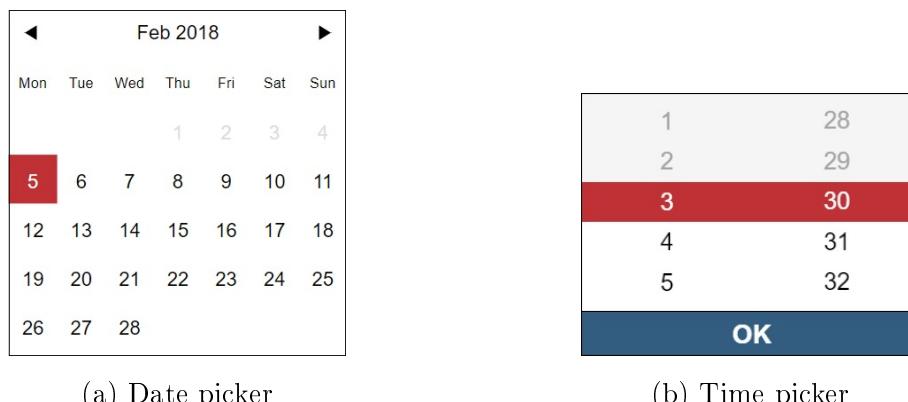
Pri navrhovaní funkcionálít aplikácie sme si načrtli obrazovky, podľa ktorých sme postupovali pri implementácii. Aplikáciu z pohľadu klienta tvoria 4 podstránky: hlavné menu, zoznam liniek, vyhľadávanie spojov a výsledky vyhľadávania. Hoci existujú rôzne knižnice znova použiteľných komponentov, nenašli sme také, ktoré by vyslovili našim požiadavkám. Vytvorili sme si preto vlastné komponenty napríklad komponent *Card.vue*, *Loader.vue* alebo *Preferences.vue*. Tento komponent sa správa ako vysúvacia lišta, ktorá zobrazí možnosť navolenia vyhľadávacích parametrov. Údaje, ktoré sú zdieľané medzi podstránkami a komponentami sú uložené vo *vuexstore*. História vyhľadávania sa ukladá do lokálneho úložiska prehliadača, aby aplikácia mohla ponúknut používateľovi posledné parametre vyhľadávania.

3.6.2 Dátum s čas

Na klientskej strane sa potrebujeme tiež vysporiadať s časom v minulosti. Keďže chceme, aby aplikácia bola používateľsky prívetivá, predvolíme pri vyhľadávaní aktuálny dátum a čas. Na serverovej strane sa simuluje aktuálny dátum a čas a po spustení aplikácie na klientskej strane sa dotiahne zo servera aktuálny simulovaný dátum a čas. Klientská strana pokračuje v simulovaní času.

Na výber dátumu sme použili komponent *datepicker*. Tento komponent umožňuje nastaviť predvolený dátum a znemožniť výber dátumov v simulovanej minulosti. Ukážka komponentu je na obrázku 3.3a.

Na zvolenie času sa si vytvorili vlastný komponent *timepicker*, pretože žiadny z dostupných komponentov nespĺňal naše požiadavky. Náš komponent na výber času predvolí aktuálny simulovaný čas. V prípade, že vyhľadávame v aktuálnej simulované deň, komponent zneaktívni časy v minulosti. Na obrázku 3.3b je ukážka komponentu.



(a) Date picker

(b) Time picker

Obr. 3.3

3.6.3 Zobrazovanie zastávok na mape

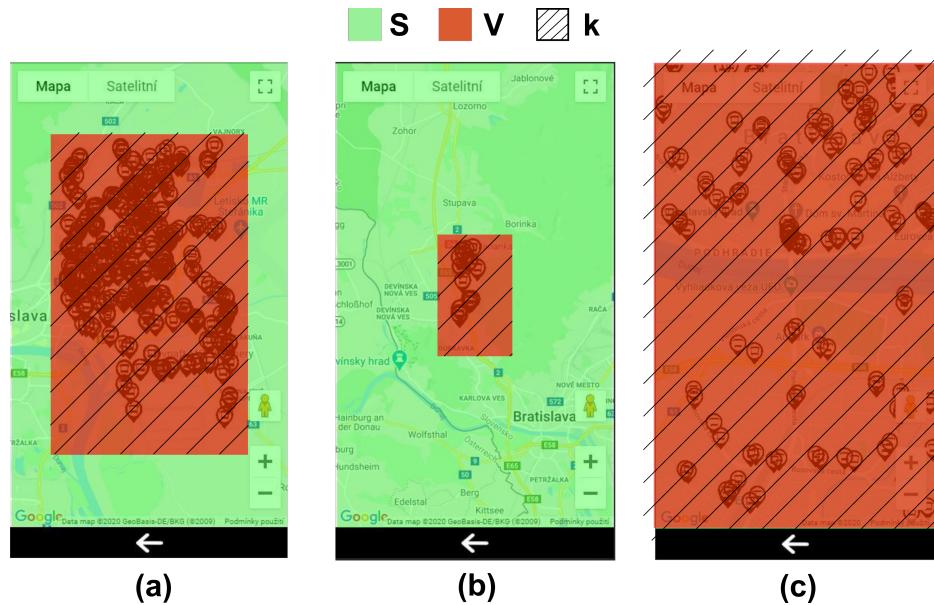
Aplikácia ponúka možnosť výberu zastávky priamo z mapy v prípade, že používateľ nepozná názvy zastávok. Na prácu s mapou používame knižnicu *vue-google-maps*. Po zvolení možnosti zadania zastávky z mapy, vycentruje pohľad mapy na aktuálnu polohu používateľa.

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

Ďalej sme zvažovali, že zobrazenie zastávok rovnakým spôsobom ako ich zobrazuje aplikácia *GoogleMaps*, a to tak, že používateľ vidí zastávky len ak je dostatočne priblížený. V prípade, že používateľ vidí celú Bratislavu, nevidí žiadne zastávky.

Nakoniec sme zobrazenie zastávok vyriešili tak, že zastávky sa budú zobrazovať len v určitom výreze V , ktorý je centrovaný v strede obrazovky. Tento výrez V bude ob-

dĺžnikového 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 3.4 (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 zobrazíme na celej obrazovke ako môžeme vidieť na obrázku 3.4(c). V tomto prípade je dokonca $k > V$.



Obr. 3.4: Zobrazenie zastávok na mape

3.6.4 Zobrazovanie časov ciest

Pri vypisovaní jednotlivých cest na klientskej strane sme sa museli viackrát zamyslieť, ako zobraziť meškanie vozidla. Najskôr sme sa inšpirovali aplikáciami, ktoré tiež zobrazujú meškania spojov. Vypísali sme pri zastávke na nastúpenie a pri zastávke na vystúpenie časy príchodov podľa statických cestovných poriadkov. K tomu sme pripísali aj informáciu o meškaní prípadne nemeškaní spoja. Používateľ by si však musel sám prepočítať, kedy má spoj predpokladaný príchod v prípade meškania. MHD spoje často meškajú a používateľ by často musel prepočítavať. To nie je veľmi používateľsky prívetivé. Ukážka zobrazenia je na obrázku 3.5a.

Ďalšou možnosťou je zobraziť oba časy – čas podľa statických cestovných poriadkov a aj ten s priponaným omeškaním. Ako môžeme vidieť na obrázku 3.5b toto riešenie pôsobí chaoticky, pretože na jednej stránke je zobrazených príliš veľa časov.

5.2.2018	25 min	
39 Trávna-DREVONA Račianske mýto	06:50 včas 07:02	
59 Račianske mýto Kollárovo nám. meškanie 6 min	06:58 07:03	
6 min → Mariánska	07:15	

5.2.2018	25 min	
39 Trávna-DREVONA Račianske mýto	06:50 včas 07:02	
59 Račianske mýto Kollárovo nám. meškanie 6 min	06:58 07:04 07:03 07:09	
6 min → Mariánska	07:15	

(a)

(b)

Obr. 3.5: Možnosti zobrazenia časov v ceste

Rozhodli sme sa tento problém vyriešiť tak, že budeme zobrazovať len pripočítaný čas, ktorý udáva, kedy má spoj príchod na zastávku podľa statických cestovných poriadkov s pripočítaným predpokladaným meškaním. Časy spoja budú označené farebne. V prípade, že vozidlo už vyrazilo a nemá žiadne meškanie, časy budú zafarbené zelenou farbou. Meškajúce spoje budú mať červené časy a spoje, ktoré ešte nevyrazili budú mať časy čierne. Po kliknutí na cestu sa zobrazí detail cesty, kde je pri každej zastávke zobrazený čas zo statických cestovných poriadkov a pri spoji je zobrazený stav meškania. Zobrazenie cesty aj detail cesty je zobrazený na obrázku 3.6.

Trávna-DREVONA Mariánska	05.02.2018	25 min
59 včas		
Trávna-DREVONA 101 06:50 Pekná cesta 101 06:53 ŽST Vinohrady 100 06:56 Pionierska 100 06:53 Račianske mýto 100 07:02 → Antolská		
39 meškanie 6 min		
Račianske mýto 100 06:58 Povaznícka 100 06:59 STU 100 07:01 Kollárovo nám. 100 07:03 → Cintorín Slávičie údolie		
presun 6 min → Mariánska		

5.2.2018	25 min	
39 Trávna-DREVONA Račianske mýto	06:50 včas 07:02	
59 Račianske mýto Kollárovo nám.	07:04 07:09	
6 min → Mariánska	07:15	

Obr. 3.6: Finálne zobrazenie časov v ceste

3.6.5 Nasledujúce cesty

Pre vyhľadanie optimálnych ciest používateľ zadáva povinné vstupné parametre: začiatočnú a konečnú zastávku, dátum a čas τ . Ostatné parametre sú dobrovoľné a sú predvolené. Klient si vždy uloží posledné vyhľadávané parametre do stavu a pošle do-

pyt na server. Ako odpoveď dostane aspoň x ciest, ktoré nezačínajú skôr ako čas τ . Server sa teda stará o to, aby bol dostatočný počet ciest na jednu stránku. V prípade, že používateľ chce nasledujúce cesty, pod poslednou vyhľadanou cestou sa nachádza tlačidlo "Nasledujúce". Kliknutím na toto tlačidlo klient pripočíta k času odchodu poslednej cesty jednu minútu. Následne pošle dopyt so zapamätanými vyhľadávacími parametrami na server s tým, že zmení čas prípadne aj dátum.

Kapitola 4

Testovanie a evaluácia

Záver

Literatúra

- [1] Shortest alternate path discovery through recursive bounding box pruning - scientific figure on researchgate - figure 13. bounding box consideration of approx 89 nods. https://www.researchgate.net/figure/Moving-Q-to-I-bounding-box-Q-I_fig3_316188932. Navštívené: 18. august 2019.
- [2] Shortest alternate path discovery through recursive bounding box pruning - scientific figure on researchgate - figure 5. moving q to i; bounding box (q, i). https://www.researchgate.net/figure/Bounding-box-consideration-of-approx-89-nods_fig7_316188932. Navštívené: 18. august 2019.
- [3] Kristóf Bérczi, Alpár Jüttner, Marco Laumanns, and Jácint Szabó. Arrival time dependent routing policies in public transport. *Discrete Applied Mathematics*, 251:93 – 102, 2018.
- [4] Daniel Delling, Thomas Pajor, and Renato F. Werneck. Round-based public transit routing. *Transportation Science*, 49, 01 2012.
- [5] Lunce Fu and Maged Dessouky. Algorithms for a special class of state-dependent shortest path problems with an application to the train routing problem. *Journal of Scheduling*, 21(3):367–386, Jun 2018.
- [6] Abdelfattah Idri, Mariyem Oukarfi, Azedine Boulmakoul, Karine Zeitouni, and Ali Masri. A new time-dependent shortest path algorithm for multimodal transportation network. *Procedia Computer Science*, 109:692 – 697, 2017. 8th International Conference on Ambient Systems, Networks and Technologies, ANT-2017 and the 7th International Conference on Sustainable Energy Information Technology, SEIT 2017, 16-19 May 2017, Madeira, Portugal.
- [7] I Jeon, H Nam, and C Jun. Improved public transit routing algorithm for finding the shortest k-path. *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XLII-4/W9:255–264, 10 2018.

- [8] Yeon-Jeong Jeong, Tschangho John Kim, Chang-Ho Park, and Dong-Kyu Kim. A dissimilar alternative paths-search algorithm for navigation services: A heuristic approach. *KSCE Journal of Civil Engineering*, 14(1):41–49, Jan 2010.
- [9] Y. Li, H. Zhang, H. Zhu, J. Li, W. Yan, and Y. Wu. Ibas: Index based a-star. *IEEE Access*, 6:11707–11715, 2018.
- [10] R. Parmar and B. Trivedi. Shortest alternate path discovery through recursive bounding box pruning. *Journal of Transportation Technologies*, 7:167 – 180, 2017.
- [11] Lilian S.C. Pun-Cheng and Albert W.F. Chan. Optimal route computation for circular public transport routes with differential fare structure. *Travel Behaviour and Society*, 3:71 – 77, 2016.
- [12] Mohammed Quddus and Simon Washington. Shortest path and vehicle trajectory aided map-matching for low frequency gps data. *Transportation Research Part C: Emerging Technologies*, 55:328 – 339, 2015. Engineering and Applied Sciences Optimization (OPT-i) - Professor Matthew G. Karlaftis Memorial Issue.
- [13] Frank Schulz. *Timetable Information and Shortest Paths*. PhD thesis, Universität Karlsruhe (TH), Fakultät für Informatik, 2005.
- [14] Y. Zhao Z. Sun, W. Gu and C. Wang. Optimal path finding method study based on stochastic travel time. *Journal of Transportation Technologies*, 3(4):260 – 265, 2013.