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 aplikovanej 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 forma)
**Študijný odbor:** aplikovaná informatika
**Typ záverečnej práce:** diplomová
**Jazyk záverečnej práce:** slovenský
**Sekundárny jazyk:** anglický

**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 Ďurikovič, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

**Poďakovanie:**

Chcela by som sa poďakovať vedúcemu mojej bakalárskej práce doc. RNDr. Milanovi Ftáčnikovi, 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

# Zoznam obrázkov

# Zoznam tabuliek

# 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 navrhnúť 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 dáta. Bude potrebné vyriešiť, ako sa vysporiadať so spracovaním reálnych dát. Kľúčový bude výber dátovej štruktúry na ich spracovanie, ako aj nájdenie vhodného časovo závislého algoritmu.

Sumarizáciou nášho problému je, ako sa vysporiadať s dynamickými dátami, alternatívnymi cestami, viacerými módmi, 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

### 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. Tým pádom 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 predlžuje 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 vzdialeností, je funkcia

$$f(n) = g(n) + h(n), \tag{1.1}$$

kde $g(n)$ predstavuje hodnotu cesty zo začiatočného 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 vzdialenosti 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] tvrdia, 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 prevdepodobnostného rozdelenia času cestého 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 začiatočným 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 najoptimálnejšie trasy, potrebujeme reálne dáta o polohe vozidiel.

Tieto dáta môžeme získať v rôznej forme. Ak by našimi dátami bola pozícii vozidla z GPS, potrebujeme získané pozície vozidiel nejakým spôsobom spracovať. V článku [12] navrhujú, ako spracovať takto získané dáta. 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 dáta 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 zduplikuje 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-dependant model

*Time-dependant* 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 ohodnotenie predsta-

vuje čas potrebný na prestup. Jazda vozidla $t$ je definovaná ako postupnosť zastávok, ktoré vozidlo navštevuje 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. Ťažko 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 začiatočného 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
- $\Delta d$ – priemerná vzdialenosť od všetkých vrcholov ku všetkým susedom

Najskôr vypočítame hodnotu $D = dist(s, t)$, $d$ a $\Delta d$. Prehľadávaný priestor bude mať rozmery $(2d, D)$. Začneme vytvárať cestu tak, že hľadáme vrchol $v_i$, ktorý spĺňa podmienku $dist(v_i, st) \leq d$. Ak nevieme nájsť žiadneho takého kandidáta, navyšujeme $d$ o $\Delta d$, 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

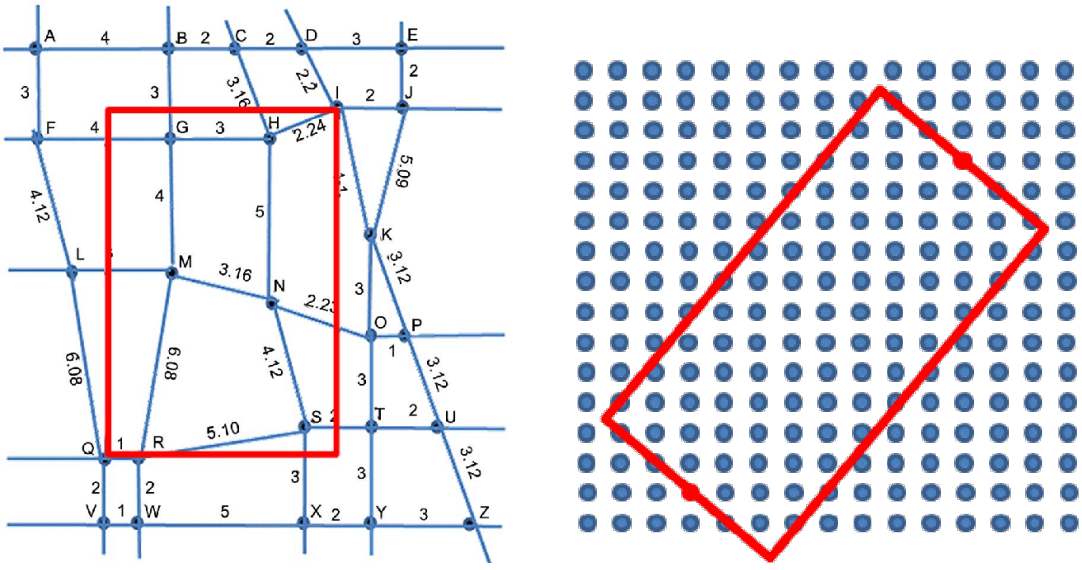
V článku [10] sa autori snažia nájsť spôsob, ako efektívne vypočítať najkratšiu cestu a tak odľahčiť cesty od dopravných zápch. Navrhujú algoritmus na nájdenie najkratšej alternatívnej cesty s minimálnym potrebným výpočtovým časom.

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. Graf je opísaný tabuľkovou maticou a reprezentovaný graficky. Keď poznáme začiatočný a koncový vrchol, vytvoríme si tabuľku všetkých vrcholov, ich súradníc a vzdialeností do začiatočného vrcholu a koncového vrcholu. Podľa súradníc vieme vypočítať vzdialenosti 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.



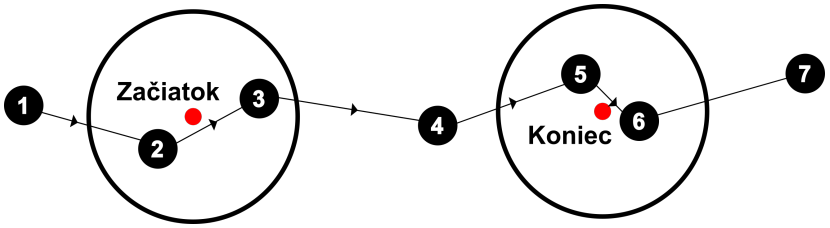

(a) Bounding box (b) Naklonený bounding box

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) nebol optimálny. Rovnako aj výberom zastávky č. 6 ako konečnej zastávky by nebolo súčasťou správneho riešenia hľadania najkratšej cesty.

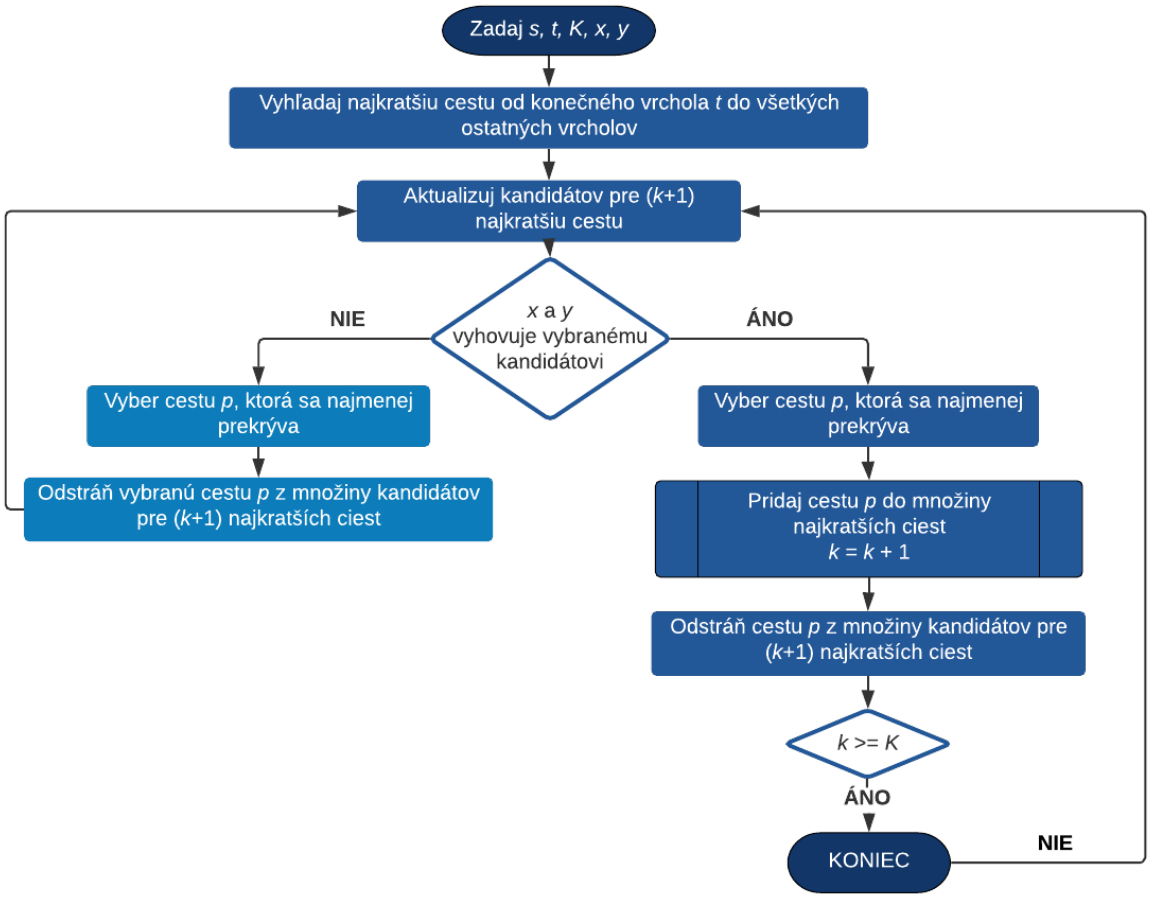



Obr. 1.2: Hľadanie správnej začiatočnej/konečnej zastávky

## 1.7 Alternatívne optimálne cesty

Pri hľadaní optimálnej cesty je potrebné zohľadniť, čo daný používateľ považuje za optimálne. Používateľ pri zadávaní vyhľadávacích parametrov nemusí poznať 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 musí 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. Algoritmus je popísaný vývojovým diagramom na obrázku 1.3.




Obr. 1.3: Algoritmus K najkratších ciest - vývojový diagram

## 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 ťažko 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 paralelizmu. 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 ciest 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. Môže to byť aj platforma. 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 $\tau$. Najzákladnejším kritériom, na ktoré algoritmus prihliada, je *Earliest Arrival Problem*, ktorý hľadá cestu nezačínajucú skôr než $\tau$ v bode $p_s$ a do $p_t$ sa dostane čo najrýchlejšie. Ďalej chceme viacero alternatívnych ciest s tým, že žiadna z ciest nezačína skorej ako $\tau$ 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

$(\tau_0(p), \tau_1(p), ..., \tau_K(p))$, 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 $\infty$. 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 $\infty$. 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 $\mathcal{T}(r) = (t_0, t_1, ..., t_{|\tau(r)|-1})$ 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 $r$ navštevujeme 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 spätne 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 stihnúť 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 niektorej 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 $\tau_k(p_i)$.

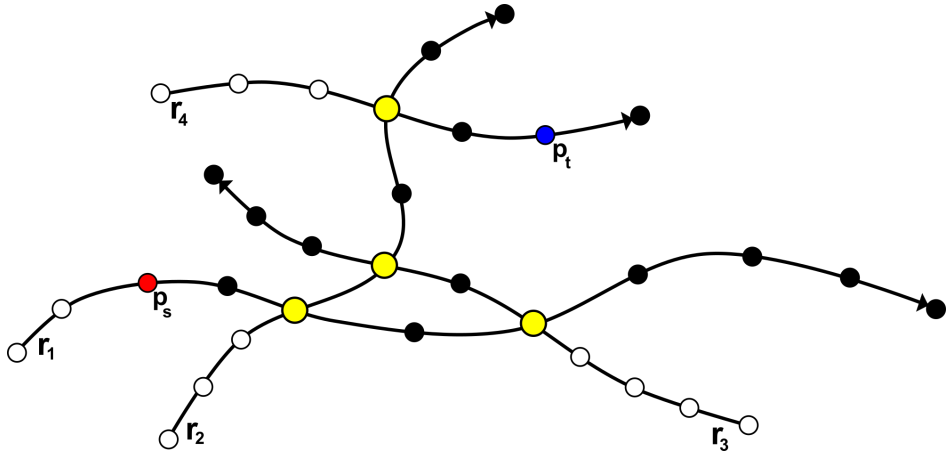
Najhorší odhad časovej zložitosti algoritmu je lineárny, konkrétne $\mathcal{O}(K(\sum_{r \in \mathcal{R}} |r| + |\mathcal{T}| + |\mathcal{F}|)$, kde $K$ je počet kôl, pričom v každom kole $k$ prechádzame každú linku $r$ najviac jeden krát. Keďže $|r|$ je počet zastávok, spolu prechádzame $\sum_{r \in \mathcal{R}} |r|$ 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 znovu, 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 prestup 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 1.4. 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. 1.4: RAPTOR - Optimalizácia prechádzania liniek

Ďalšou optimalizačnou technikou je *local prunning*. 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 prunning* 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 nám vráti množinu najkratších ciest pre všetky odchody zo zastávky v danom časovom rozsahu.

Nech $\Delta \subseteq \Pi$ je časový úsek. Najskôr vložíme do množiny $\Psi$ časy odchodov jázd začínajúcich na danej zastávke, ktoré patria do časového úseku $\Delta$. Pre každý čas odchodu $\tau$ z $\Psi$ spustíme RAPTOR algoritmus nezávisle. Znamená to, že hodnota $\tau_k(p)$ bude

existovať pre každý čas odchodu $\tau$, 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:

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

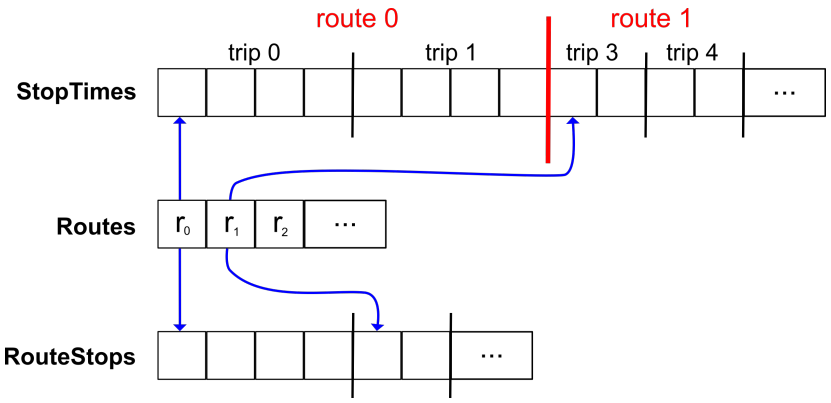
Aby sme mohli použiť toto pravidlo, zoradíme časy v množine $\Psi$ od najneskoršieho po najskorší čas. Pri iterovaní nebudeme prepisovať hodnotu $\tau_k(p)$ medzi kolami, ale nastavíme vždy na začiatku kola pre všetky zastávky $p$ $\tau_k(p) = \tau_{k-1}(p)$, kde je $\tau_{k-1}(p)$ lepšie ako $\tau_k(p)$.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

### 1.8.5 Paralelizácia

Ako sme si mohli všimnúť, linky sú individuálne a nie je potrebné ich iterovať v špecifickom poradí. Ak máme k dispozícii viac CPU jadier, každé z nich môže spracovávať inú podmnožinu liniek. Nákladné môže byť zamedzenie prístupu do spoločného miesta v pamäti ($\tau_k(p)$). Autori navrhli 2 prístupy na riešenie paralelizácie bez využitia zámkov. Jeden prístup počíta s tým, že hardvér zaisťuje atomické zapisovanie do pamäte a druhý sa zaobíde aj bez tejto možnosti.

### 1.8.6 Dátová štruktúra

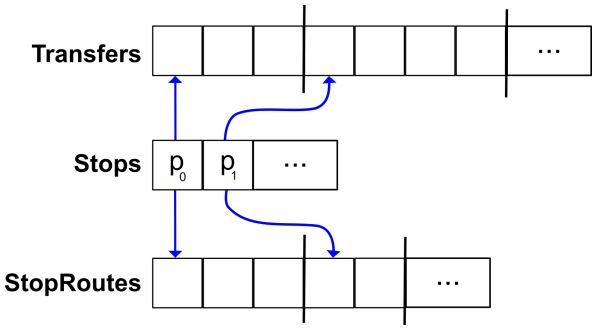
Autori článku navrhli aj štruktúru, ktorá je vhodná pre RAPTOR algoritmus. Linky, jazdy a zastávky indexujeme od 0. Potrebujeme pole liniek *Routes*, ktoré si pre každú linku $r_i$ uchováva informáciu o počte zastávok na linke $r_i$ a smerník na pole *RouteStops*, ktorý označuje začiatok postupnosti 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 ja zobrazená na obrázku 1.5.




Obr. 1.5: 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 $\tau$, spôsob akým je pole *StopTimes* utriedené nám zabezpečí, že čas tejto operácie bude konštantný. Pri kontrolovaní, či sa v predchádzajúcej jazde nezlepšil čas odchodu niektorej 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 doplňujúcej dátovej štruktúry je na obrázku 1.6.




Obr. 1.6: RAPTOR - Dátová štruktúra na zohľadnenie prestupov

### 1.8.7 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 $c$ |
| $l(p, p_i)$ | čas pešieho prestupu zo zastávky $p$ do zastávky $p_i$ |
| $\tau_k(p)$ | najskorší známy čas príchodu na zastávku $p$ v $k$ kolách |
| $\tau_k(t, p)$ | najskorší známy čas príchodu na zastácku $p$ použitím jazdy $t$ v $k$ kolách |
| $\mathcal{J}_k(p)$ | množina ciest, ktorými sa viam dostať na zastávku $p$ v $k$ kolách |
| $\mathcal{J}_{k,m}(p)$ | $m$-tá najskoršia cesta, ktorou sa viem 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 prvkami 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.

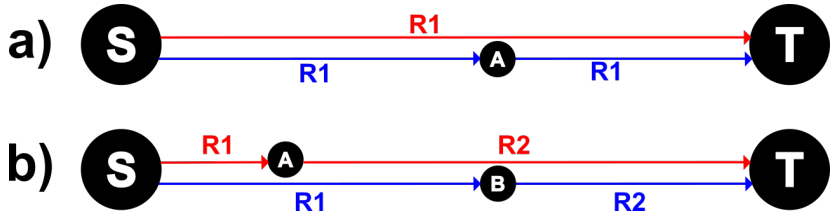
Čo sa týka času potrebného na peší presun medzi jednotlivými zastávkami, priemerná rýchlosť kráčania dospelého človeka bola určená na $1.2m/s$. Väčšinou sa predpokladá, že dĺžka prestupu je Euklidovská vzdialenosť od počiatočnej zastávky po cieľovú. Tento odhad ale nie je vyhovujúci, pretože vo väčšine 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 1.7(a). Červená cesta vedie priamo zo začiatočnej zastávky $S$ do konečnej zastávky $T$ a modrá cesta obsahuje navyše 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 1.7(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. 1.7: RAPTOR - podobné cesty

Ďalej je popísaný algoritmus 1, ktorý zhodnotí, či je cesta $\mathcal{J}_k(p, p_i)$ podobná ako niektorá z ciest 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**

**Zhrnutie vylepšeného RAPTOR algoritmu**

Algoritmus nám vráti $M$ najskorších ciest, ktoré vedú na zastávku $p_t$. Vstupom pre algoritmus je čas odchodu $\tau$, začiatočná zastávka $p_s$ a konečná zastávka $p_t$. Začíname v kole $m = 0$. Pre každú zastávku $p$ nastavíme hodnotu $\tau_k(p) = \infty$, okrem začiatočnej zastávky $p_s$. Tá bude mať hodnotu $\tau_k(p_s) = \tau$ a bude označená.

**KROK 1:** Ak nie sme v kole 0, pre každú zastávku $p$ nastavíme $\tau_k(p) = \tau_{k-1}(p)$ a $\mathcal{J}_k(p) = \mathcal{J}_{k-1}(p)$.
**KROK 2:** Pre každú zastávku $p$ označenú v kole $k-1$ hľadáme všetky zastávky $p_i$, na ktoré sa vieme dostať peším presunom zo zastávky $p$. Ak $\tau_k(t, p)$ cesty $\mathcal{J}_k(p, p_i)$ je skôr ako $\tau_k(p_i)$, tak do $\tau_k(p_i)$ vložíme hodnotu $\tau_k(t, p)$ a označíme zastávku $p_i$. Ak počet ciest v množine $\mathcal{J}_k(p_i) < K$, pridaj cestu $\mathcal{J}_k(p, p_i)$ do tejto množiny. Ak je veľkosť množiny rovnaká ako $K$, nahraď cestu $\mathcal{J}_{k,m}(p_i)$ cestou $\mathcal{J}_k(p, p_i)$ a označ zastávku $p_i$. Ak nastane nejaká zmena v množine $\mathcal{J}_k(p_i)$, zotrieď ju podľa času príchodu do zastávky $p_i$.
**KROK 3:** Pre každú označenú zastávku $p$, vložíme všetky dvojice $(r, p)$ do poľa $Q$, pričom platí, že linka $r$ obsluhuje zastávku $p$.
**KROK 4:** Pre každú dvojicu $(r, p)$ hľadáme jazdu $t_r$, ktorá príde na zastávku $p$ po čase $\tau_k(p) + Tr_c$.
**KROK 5:** Ak je posledná linka v množine $\mathcal{J}_{k,m}(p)$ identická s linkou $r$, preskoč KROK 6 a KROK 7.
**KROK 6:** Hľadaj jazdu $t_r$ linky $r$ a pre každú zastávku $p_i$, ktorá patrí tejto linke over, či existuje podobná cesta ceste $\mathcal{J}_k(p, p_i)$. Ak už podobná cesta existuje, pokračuj KROKOM 7. Inak, ak počet ciest v $\mathcal{J}_k(p_i) < K$, pridaj cestu $\mathcal{J}_k(p, p_i)$ do tejto množiny. Ak je veľkosť množiny rovnaká ako $K$, porovnaj čas $\tau_k(p_i)$ cesty $\mathcal{J}_{k,m}(p_i)$ s časom $\tau_k(t, p)$ cesty $\mathcal{J}_k(p, p_i)$. Ak je čas $\tau_k(t, p)$ menší, nahraď $\mathcal{J}_{k,m}(p_i)$ cestou $\mathcal{J}_k(p, p_i)$ a označ zastávku $p_i$. Ak nastane nejaká zmena v množine $\mathcal{J}_k(p_i)$, zotrieď ju podľa času príchodu do zastávky $p_i$.
**KROK 7:** Nájdi cestu, ktorá má rovnakú postupnosť liniek akú má $\mathcal{J}_k(p, p_i)$ a ak platí $\tau_k(t, p)$ cesty $\mathcal{J}_k(p, p_i) < \tau_k(p_i)$ cesty $\mathcal{J}_{k,m}(p_i)$, tak aktualizuj $\tau_k(p)$, označ zastávku $p_i$ a zotrieď množinu $\mathcal{J}_k(p_i)$
**KROK 8:** Ak existuje zastávka, ktorá má hodnotu $\tau_k(p)$ aktualizovanú v kole $k$, pokračuj v algoritme znovu od KROKU 1. Inak ukonči proces.

## 1.9 Podobné existujúce systémy

V tejto sekcii 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 dáta. Vieme však porovnať, aké funkcionality ponúkajú používateľom a zhodnotiť, čo nám chýba alebo prekáža pri používaní týchto aplikácii.

### 1.9.1 Imhd.sk

Mobilná aplikácia Imhd.sk ponúka vyhľadávanie MHD spojení v Bratislave a v Košiciach. Pri prvom spustení aplikácia sťahuje 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 vyfiltrovať nízkopodlažné spoje a nastaviť preferenciu minimálneho potrebného času na prestup 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. Aplikácia pracuje online a používa vždy aktuálne cestovné poriadky. 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 aplikácii chýba možnosť nastavenia obmedzenia pešieho presunu.

Používateľ si vie zvoliť obľúbené položky a pamätá si históriu 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.

Ani aplikácia CP neponúka vyhľadávanie spojov z MHD so zohľadnením reálnej polohy vozidiel MHD.

### 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, históriu 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 funkcionalitu 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. Pre Bratislavu majú od Dopravného podniku Bratislava online polohu vozidiel, rovnako aj od Železničnej spoločnosti Slovensko a mnohých autobusových medzimestských prepravcov.

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 meška 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. Spomíname tiež, ako budeme pristupovať k týmto dátam pri implementácii, teda akým spôsobom ich uložíme do databázy a následne do dátovej štruktúry. 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 funkcionalitu nájde používateľ aj v našej aplikácii. História vyhľadávania sa bude ukladať do pamäte zariadenia.

Okrem aplikácie *UBIAN* a *CP* 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 ces-

tovných 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 nebol, aplikácia ho nezobrazí. Naša aplikácia ponúkne aj tie spojenia, ktoré kvôli meškaniu na zastávku ešte nedorazili. Pre vyhľadaný spoj zobrazí čas odchodu zo statických dát cestovného poriadku a pripíše k nemu informáciu o meškaní.

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

Len aplikácia *CG Tranzit* funguje aj v offline režime. Hoci je užitočné ponúknuť vyhľadávanie bez možnosti prístupu na internet, znamenalo by to, že náš 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 aplikácii 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 Algoritmus

Pri hľadaní algoritmu na nájdenie optimálnej cesty sme najskôr 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, môžeme pri vyhľadávaní optimalizovať prehľadávaný priestor. 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 dáta. 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 najoptimálnejšiu cestu, prihliadal na prestupy medzi rôznymi módmi a popri tom počítal s ďalšími pridanými kritériami. V 1.3 boli spomenuté návrhy časovo závislých algoritmov. Algoritmus 1.3.1 sa dokáže vysporiadať aj viacerými módmi. Vráti však len jednu cestu. V 1.7 je popísaný algoritmus, ktorý efektívne vyhľadáva rôzne alternatívne cesty.

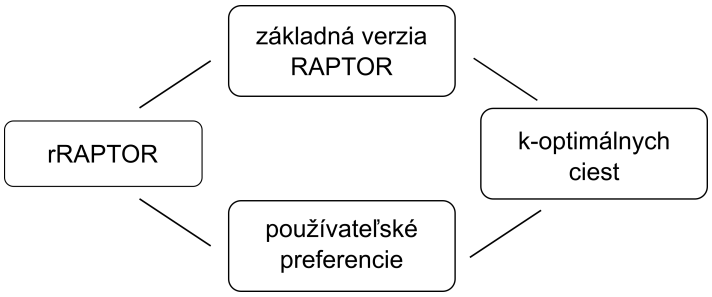
Kvôli dynamickej povahe verejnej dopravy grafový prístup v kombinácii s vyhľadávacím grafovým algoritmom vyžaduje veľa pre-processingu a to sa odráža na výpočtových časoch. Výhodou vo verejnej doprave je, že vozidlá sa pohybujú po vyznačených linkách, ktorých trasy poznáme. Schéma verejnej dopravy sa 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 riešiť multimodalitu hromadnej dopravy. Ľahšie zvláda dynamickosť dát ako meškanie linky, zrušenie linky alebo zmenu trasy.

Využijeme základnú verziu RAPTOR algoritmu popísanú v 1.8.2 a súčasne využijeme aj jeho optimalizáciu opísanú v 1.8.3, kedy označujeme zastávky, aby sme nemuseli prechádzať tie linky, ktorým sa nevylepšil čas $\tau_{k-1}(p)$. Optimalizáciu *local-prunning* nevyužijeme, keďže chceme použiť rozšírenú verziu RAPTOR algoritmu a to je rRAPTOR algoritmus popísaný v sekcii 1.8.4. Algoritmus rRAPTOR nám nevráti len jednu najkratšiu cestu, ktorá začína najskôr od zadaného času, ale vráti nám množinu najkratších ciest, začínajúcich v zadanom časovom úseku.

Cesta, ktorú nám vráti základná verzia RAPTOR algoritmu je najkratšia a nezáleží na tom, koľko prestupov bude obsahovať. Našou úlohou je používateľovi poskytnúť optimálnu cestu a keďže optimálna cesta môže byť pre každého používateľa iná, ponúkneme mu viacero alternatív. Túto funkciu ponúka vylepšený RAPTOR algoritmus popísaný v 1.8.7. Algoritmus nám vráti viacero optimálnych ciest, pričom prihliada na to, aby jednotlivé cesty neboli rovnaké na veľkej časti úseku.

Našim cieľom je zlúčiť rRAPTOR algoritmus s vylepšeným RAPTOR algoritmom na hľadanie viacerých ciest a zakomponovať mechanizmus schopný zohľadniť zadané používateľské preferencie, ktoré bude algoritmus prijímať ako vstupné parametre. Preferencie, ktoré bude vedieť algoritmus zohľadniť: minimálny čas na prestup, maximálny počet prestupov, maximálna dĺžka pešieho prestupu a vyhľadanie len nízkopodlažných spojov. Na obrázku 2.1 sú zachytené jednotlivé podalgoritmy, ktoré budú tvoriť náš výsledný algoritmus.




Obr. 2.1: Návrh algoritmu

Výstupom z nášho algoritmu bude množina ciest začínajúcich na zastávke $p_s$, po čase $\tau$ a končiacich v zastávke $p_t$. Jednotlivé cesty sú optimálne, vyhovujú prípadným používateľským preferenciám a ich časy odchodov sú z časového intervalu $<\tau, \tau + \Delta\tau>$.

### 2.2.1 Vyhľadávanie z aktuálnej polohy

Ak používateľ zadá ako začiatočnú zastávku aktuálnu polohu, nájdeme zastávky v okolí radiálnym vyhľadávaním, 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 zastávku zvlášť spustíme algoritmus, ktorý nám pre každú zo zastávok vráti množinu optimálnych ciest a na záver z nich vyberieme $n$ najoptimálnejších ciest. Do finálneho riešenia zakomponujeme peší presun od aktuálnej polohy po začiatočnú zastávku vybraných optimálnych ciest.

## 2.3 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.

### 2.3.1 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 tekto sekcii analyzujeme len tie súbory, ktoré nám boli poskytnuté a opíšeme, ktoré z poskytnutých údajov využijeme.

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

Súbor *celandar.txt* obsahuje stĺpce *service_ id*, *monday*, *tuesday*, *wednesday*, *thursday*, *friday*, *saturday*, *sunday*, *start_ date*, *end_ date*. Vlasntosť *service_ id* predstavuje unikátny názov typu dňa (pracovný deň, víkend,...). Vlasnosti *monday, ..., sunday* môžu nadobúdať hodnoty 0 a 1. Ak je napríklad hodnota $monday = 0$, znamená to, že všetky pondelky medzi dátumom *start_ date* a dátumom *end_ date* patria do typu dňa *service_ id*. Naopak, ak je hodnota rovná 1, 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* výnimočne 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 unikátne pre každú zastávku. Vlastnosť *stop_ name* predstavuje názov zastávky. Táto vlastnosť sa nachádza v súbore viac krát, kedže 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 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_ day* oddelených pomlčkou. V našich dátach existuje pre jeden záznam jazdy len jeden *service_ day*. V prípade, že jazda patrí viacerým *service_ day*, je vytvorený nový záznam s rovnakými vlastnosťami jazdy pre každý *service_ day*. Vlastnosť *trip_ headsign* predstavuje konečnú zastávku jazdy. Túto vlastnosť síce nevyužijeme, ale kedže 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 hľadanýhch ciest.

Súbor *stop_ times.txt* definuje indentifikačné 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 na zastávke stojí len na znamenie. Hodnoty sú pre obe vlastnosti vždy rovnaké, takže jednu z nich môžeme opäť ignorovať.

### 2.3.2 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:

- Id – identifikačné číslo záznamu
- Datum a cas – dátum a čas, kedy bol záznam o meškaní zaevidovaný
- Vozidlo – identifikačné číslo vozidla
- Linka – identifikačné číslo linky
- Poradie – poradie jazdy v linke
- Cislo zastavky – identifikačné číslo zastávky
- Nazov zastavky – názov zastávky
- Meskanie – hodnota meškania v minútach.

Z pozorovania dát vyplýva, že hodnoty meškania majú hodnotu 0, záporné celé čísla alebo hodnotu $n/a$. V prípade, že narazíme na hodnotu 0 alebo $n/a$, tento záznam neberieme do úvahy. Môže ale platiť, že hodnota 0 znamená, že meškanie je z intervalu $< 0min, 1min)$ minúta, hodnota 1 z intervalu $< 1min, 2min)$. Na prelome dní sa tam vyskytuje hodnota $-1229$.

Takže naša aplikácia bude môcť 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.

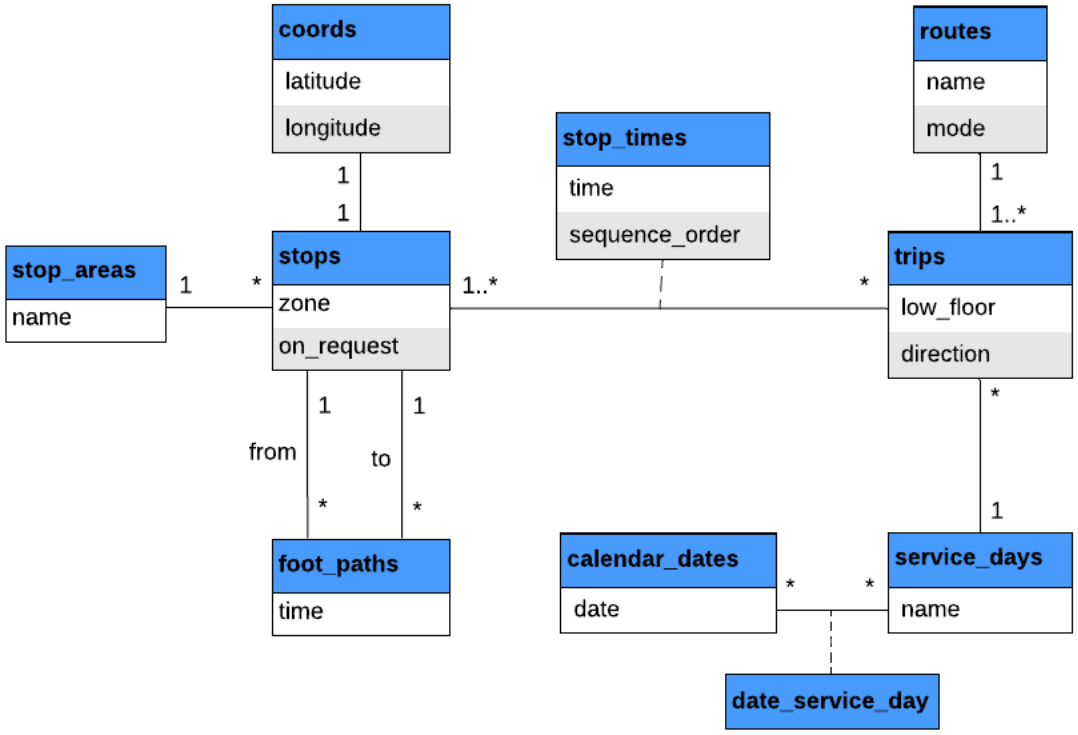
### 2.3.3 Pešie presuny

V dátach sa nenachádzajú informácie o peších presunoch. *Google API* ponúka možnosť vyhľadania peších vzdialeností medzi 2 bodmi. Počet bezplatných dopytov na *Distance Matrix API* je obmedzený. Nebudeme vyhľadávať pešie vzdialenosti 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* zistím vzdialenosti nájdených zastávok od zastávky $p$. Každú dvojicu zastávok uložíme do súboru *foot_paths.txt* spolu so zistenými vzdialenosťami určených v minútach.

## 2.4 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 2.2.




Obr. 2.2: 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.

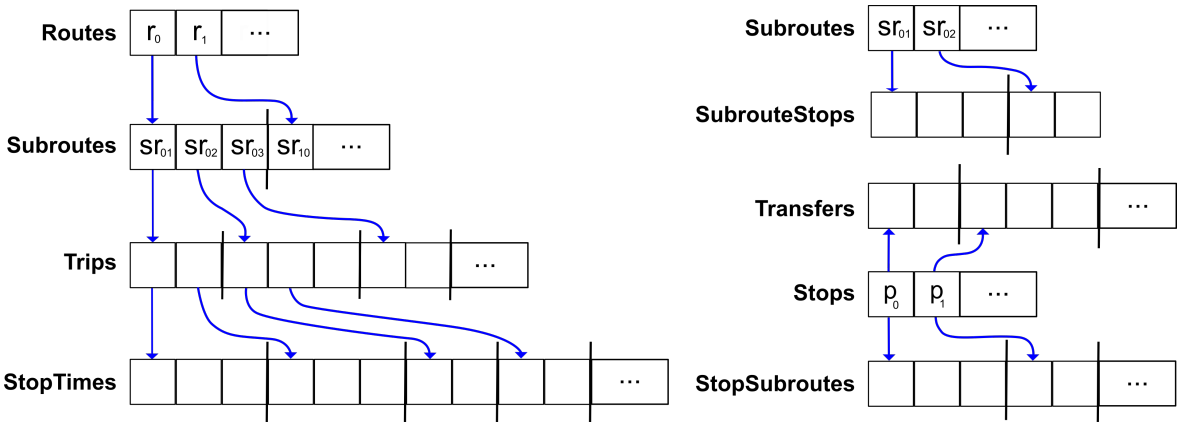
V entite *stop_times* je zachytený čas (*time*), kedy jazda stojá na zastávke a v akom poradí sú zastávky v rámci jazdy (*sequence_order*).

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.

## 2.5 Dátová štruktúra

Aby algoritmus vedel rýchlo a efektívne pracovať potrebuje dobre navrhnutú dátovú štruktúru. Pri jej návrhu sa budeme inšpirovať dátovou štruktúrou z 1.8.6, 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. Linka obsahuje jazdy, ktoré idú jedným aj druhým smerom. Zástavky cez ktoré prechádza linka majú síce rovnaký názov, ale majú iné identifikačné čísla, ktorými sú definované a 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 s rovnakou postupnosťou zastávok do úsekov (*subroutes*). Teraz platí, že 1 linka (*route*) má viacero úsekov a jednému úseku linky prislúcha viac jázd linky (*trips*). Upravená štruktúra je zachytaná na obrázku 2.3.




Obr. 2.3: Návrh dátovej štruktúry

V algoritme sa potrebujeme často dopytovať na všetky úseky linky, ktoré stoja na konkrétnej zastávke. Toto vieme okamžite nájsť v poli *StopSubroutes*. Pole *SubrouteStops* priraďuje postupnosť zastávok konkrétnemu úseku linky.

V poli *StopTimes* nebude 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.

Aby sme pri hľadaní jazdy linky nemuseli prechádzať všetky prvky poľa *Trips*, budeme si pri jej prvkoch držať aj informáciu o tom, počas akých typov dní (pracovné dni, víkendy, ...) jazda linky jazdí.

## 2.6 Architektúra aplikácie

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

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

### 2.6.1 Serverová strana

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

### 2.6.2 Klientská strana

Na klientskej strane sme sa rozhodli pre progresívnu webovú aplikáciu *PWA*. 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

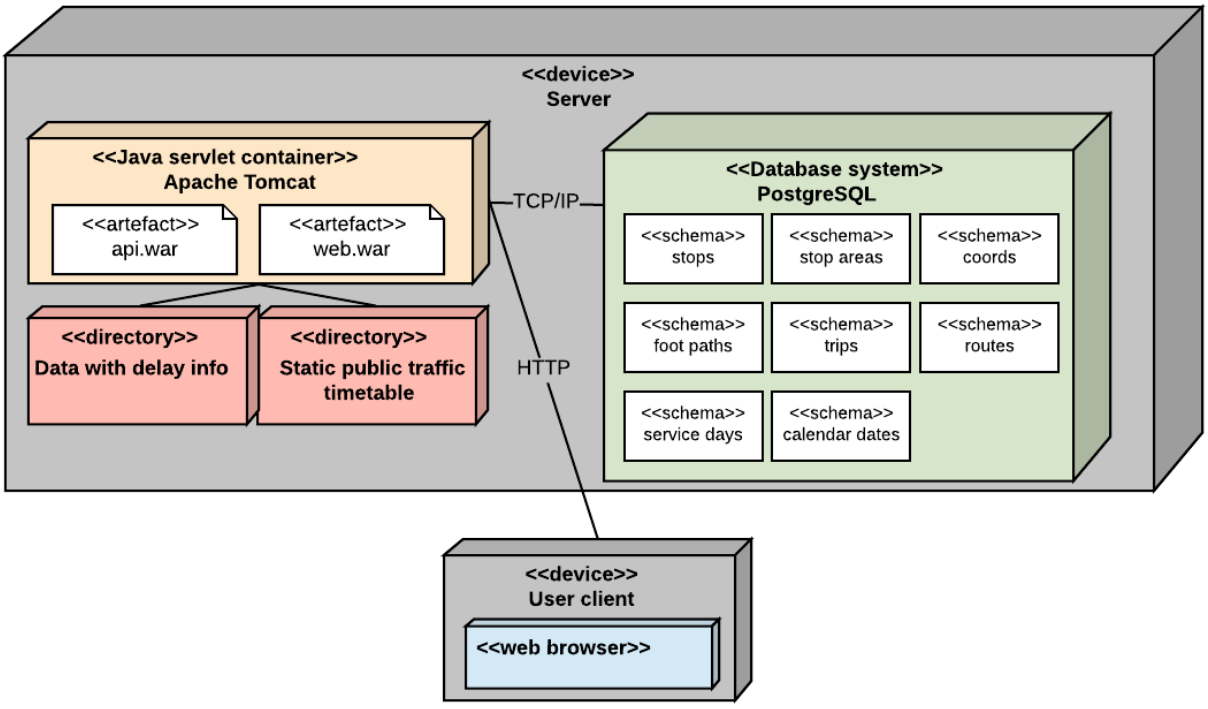
zariadení používateľ dostane upozornenie od stránky, či si ju chce uložiť do zariadenia ako mobilnú aplikáciu. Progresívna webová aplikácia zaberá minimum miesta v pamäti a má svoj vlastný úložný priestor, kde sa budú ukladať preferencie a história vyhľadávania.

### 2.6.3 Spracovanie dát

Pri spustení aplikácie alebo po aktualizácii cestovných poriadkov sa spustí služba, ktorá nám z úložiska, kde sú aktuálne cestovné poriadky namapuje dáta do našej databázy. Po tom, ako budú dáta uložené v databáze sa spustí ďalšia služba, ktorá obnoví dátovú štruktúru podľa nových cestovných poriadkov.

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 rátať 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 konkrétnu jazdu. 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

# 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.