FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	RouteMyWay - plánovač tras veřejné dopravy
Student:	Zdeněk Krupička
Vedoucí:	Ing. Jiří Hunka
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Webové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

Cílem práce je realizovat backendovou aplikaci v jazyce Kotlin vystavující rozhraní pro výpočet cesty mezi dvěma body pomocí veřejné dopravy.
Vstup i výstup z tohoto rozhraní bude ve formátu GTFS. Aplikace bude pracovat s daty o síti veřejné dopravy, která jí jsou poskytnuta ve formátu GTFS.
Aplikace bude schopna v rozumném čase najít spojení mezi dvěma body od zadaného času odjezdu. Tato spojení bude schopna vyhledat minimálně podle nejkratšího času příjezdu, případně, bude-li to možné i dle dalších parametrů (nejméně přestupů, nejmenší cena za jízdné atd..).
Cílem je také vhodně zvolit jednotlivé kroky realizace aplikace (např. analýza, návrh, realizace, testování) a vhodnou metodiku vývoje.

Elektronicky schválil/a Ing. Jaroslav Kuchař, Ph.D. dne 3. února 2022 v Praze.

Bakalářská práce

ROUTEMYWAY - PLÁNOVAČ CEST VEŘEJNÉ DOPRAVY

Zdeněk Krupička

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: Ing. Jiří Hunka
12. května 2022

Obsah

Seznam obrázků

Seznam výpisů kódu

Chtěl bych poděkovat vedoucímu mé bakalářské práce Ing. Jiřímu Hunkovi za jeho trpělivost a ochotu. Také mu děkuji za to, že mne přivedl k takto zajímavému tématu.
Můj obrovský dík dále patří Bc. Filipovi Dolníkovi ze veškerý čas, který mi věnoval při konzultacích k této práci.
Na závěr děkuji mé rodině a přátelům za podporu během studia.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 12. května 2022 .....

Abstrakt

Cílem této práce je implementace backendové aplikace pro vyhledávání tras ve veřejné dopravě. Tato aplikace vystavuje rozhraní pro výpočet cesty pomocí veřejné dopravy mezi dvěma zastávkami. Výsledná cesta je optimalizovaná jak dle nejrychlejšího času odjezdu, tak dle nejmenšího počtu přestupů.

V teoretické části se zabývám rozborem algoritmů, které lze použít k hledání cest ve veřejné dopravě. Popisuji principy jejich fungování a z nich plynoucí výhody a nevýhody. Dále se věnuji datovému formátu GTFS. Zkoumám zde jeho strukturu a možnosti použití.

Výsledkem práce je služba napsaná v jazyce Kotlin. Ta ke svému fungování používá implementaci Profile Connection Scan algoritmu. V praktické části popisuji její architekturu a způsoby použití. Pro účely ukázky a testování byla v rámci této práce vytvořena i webová aplikace komunikující s touto službou.

Klíčová slova veřejná doprava, přestupy ve veřejné dopravě, cestování, výpočet cesty, Kotlin, webová aplikace, GTFS, Connection Scan Algorithm

Abstract

The aim of this thesis is to implement a backend application for finding routes in public transport. This application exposes an interface for calculating a journey using public transport between two stops. The resulting route is optimized according to both the fastest departure time and the smallest number of transfers.

In the theoretical part I deal with the analysis of algorithms that can be used to find routes in public transport. I describe the principles of their operation and the resulting advantages and disadvantages. I also describe the structure of the GTFS data format and its possible uses.

The result of the thesis is an application written in the Kotlin language. It uses the implementation of the Profile Connection Scan algorithm for its operation. In the practical part I describe its architecture and ways of use. For the purposes of demonstration and testing, a web application communicating with this service was created in the scope of this thesis.

Keywords public transport, transfers in public transport, traveling, route calculation, Kotlin, web application, GTFS, Connection Scan Algorithm

Seznam zkratek

API	Application Programming Interface
CRUD	Create Read Update Delete
DDL	Data Definition Language
DI	Dependency injection
DTO	Data Transfer Object
GTFS	General Transit Feed Specification
HTTP	Hyper Text Transport Protocol
ID	Identifier
JDBC	Java Database Connectivity
JPA	Java Persistence API
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
MEAT	Minimal Expected Arrival Time
ORM	Object Relational Mapping
PID	Pražská integrovaná doprava
RAM	Random Access Memory
REST	Representational State Transfer
SMS	Short Message Service
SQL	Structured Query Language
URL	Uniform Resource Locator

Slovník pojmů

Backend	Část aplikace běžící na serveru. Často komunikuje s Frontendem a na jeho popud provádí dané akce. Typicky není uživateli volně přístupná.
Cloud	Termín popisující globální síť serverů poskytující nějaké služby.
Integer	Způsob reprezentace celého kladné číslo pomocí binární soustavy
Dynamické programování	Jsou programovací techniky pro zrychlení výpočtů úloh skládající se z často se opakujících podproblémů.
Framework	Skupina knihoven a nástrojů sloužících k vývoji software. Typicky slouží k určení pravidel a struktury kódu aplikace.
Frontend	Část aplikace, kterou uživatel vidí a interaguje s ní.
Graf	Je uspořádaná dvojice $G = (V, E)$, kde V značí množinu vrcholů (uzlů) a E množinu hran.
HTTP endpoint	Je bod sítě, který slouží jako cíl komunikace.
Jar soubor	Java Archive je souborový formát pro balíčkování výsledné aplikace napsané pro JVM.
Mikroslužba	Aplikace sloužící jako součást jednoho softwarového celku. Typicky má zodpovědnost za jednu funkcionalitu výsledného systému.
Ohodnocený Graf	Je uspořádaná dvojice (G, w), kde $w : E(G) \to (0, \infty)$.
Pareto optimalizace	Je stav, kdy nelze zlepšit jednu vlastnost na úkor zhoršení jiné.
Reverzní proxy	Server fungující jako vstup do sítě aplikací. Slouží pro přeposílání požadavků na požadované endpointy. Může obsahovat další logiku pro autentizaci nebo pravidla pro přeposílání požadavků.
Serverless	Je způsob tvorby aplikací založený na událostech. Tyto aplikace nejsou spuštěny neustále, ale pouze v moment, kdy musí poskytnout nějakou službu.
String	V informatice pojem používaný pro reprezentaci textového řetězce.
Web-scraping	Automatizovaný proces extrakce dat z HTML kódu webových stránek.
ZIP	Všeobecně rozšířený souborový formát pro kompresi a archivaci dat.

Úvod

Cestování pomocí veřejné dopravy je součástí života drtivé většiny obyvatel moderního světa. Ať už tramvaj, autobus, vlak, metro nebo kombinace všech, veřejná doprava je rychlý a levný způsob jak se dostat téměř kamkoliv. V takto složité síti spojů je mnohdy těžké najít tu správnou cestu.

Na trhu již existují řešení tohoto problému. Tvůrci těchto aplikací si ale za své služby nechávají platit nebo poskytují pouze omezené možnosti jejich používání. Má práce je určena pro každého, kdo v rámci svého projektu potřebuje zdarma, jednoduše a spolehlivě vyhledávat trasy mezi dvěma body ve veřejné dopravě.

S tímto tématem mě seznámil Jakub Turcovský, vlastník aplikace Taransit, která poskytuje jízdní řády pro indický trh. Ta ovšem v současné době podporuje pouze hledání přímých spojů. Byla mi tedy nabídnuta možnost účastnit se na tomto projektu a rozšířit funkce této aplikace. Velmi mne zaujal fakt, že případné úspěšné řešení této práce budou využívat v praxi desetitisíce lidí. Také se mi líbilo, že při implementaci a volbě algoritmu mám plně volnou ruku.

Práce je zaměřena na návrh a tvorbu aplikace vystavující API pro plánování tras mezi dvěma body pomocí hromadné dopravy. Potřebná data bude přijímat ve formátu GTFS. Ta budou určovat, nad jakou sítí veřejné dopravy bude aplikace schopna pracovat. Její výstup by navíc měl být jednoduše čitelný pro další zpracování externími aplikacemi.

Cílem teoretické části je prozkoumat existující nástroje pro hledání spojů ve veřejné dopravě. Následně je třeba provést analýzu existujících algoritmů, jejich porovnání a výběr toho správného. Zvolený algoritmus musí podporovat vyhledávání tras minimálně podle nejrychlejšího času příjezdu. Poté je třeba prozkoumat datový formát GTFS, jeho strukturu a využití.

Cílem praktické části je implementace backendové aplikace pomocí programovacího jazyka Kotlin. Tato aplikace bude vystavovat API pro komunikaci s externí prezentační vrstvou. Také musí být schopna přijmout, zpracovat a uložit data ve formátu GTFS, nad kterými bude pracovat. Dalším cílem je návrh metodiky testování výstupu aplikace a vyhodnocení výsledků těchto testů v porovnání s již existujícími službami.

V první kapitole se věnuji rozboru existujících řešení a sběru podkladů pro

praktickou část. Dále se zde zaměřuji na popis algoritmů vhodných pro mé použití a pojmy potřebné k jejich vysvětlení. V neposlední řadě zmiňuji mobilní aplikaci Taransit, na jejíž popud tato práce vznikla. V druhé kapitole tvořím návrh architektury. Obhajuji mou volbu výsledného algoritmu a navrhuji jeho úpravy pro potřeby realizace funkčních požadavků. V následující kapitole objasňuji má implementační rozhodnutí a způsoby řešení funkčních požadavků. Uvádím zde způsob realizace nejen hlavní backendové aplikace, ale i frontendové aplikace, která slouží pro účely ukázky. Nakonec zde popisuji i výzvy spojené s celkovou realizací. V poslední kapitole hodnotím výsledek praktické části, popisuji jeho použitelnost a dotknu se i návrhů na budoucí vylepšení.

Kapitola 1
Analýza

V této kapitole se nejprve zabývám již existujícími nástroji, které poskytují stejné nebo podobné služby. Na to navazuji přehledem algoritmů řešící danou problematiku. Popisuji jejich fungování a zavádím pojmy potřebné k jejich pochopení. Následně definuji problém minimálního očekávaného času příjezdu (tzv. MEAT problém). Dále pokračuji analýzou datového formátu GTFS, jeho specifikacemi a použitím. V neposlední řadě se krátce věnuji mobilní aplikaci Taransit a motivacím vzniku této práce. Na závěr shrnu podklady pro vlastní řešení, definuji funkční a nefunkční požadavky a případy užití.

1.1 Existující řešení

V této sekci jsem se zaměřil na přehled a analýzu aktuálních řešení hledání tras ve veřejné dopravě. Při mém výzkumu jsem narazil na velké množství aplikací, které se soustředí na konkrétní země či města. Jako příklad zde uvedu Transport for London Unified API [1] zaměřenou na Londýn nebo TransitApp [2], která pracuje s několika konkrétními městy. Takové aplikace jsem ze svého přehledu z důvodů relevance vyloučil. Jedinou takovou výjimkou je aplikace IDOS, kterou jsem zahrnul kvůli jejímu zaměření na Českou republiku.

1.1.1 Google directions API

Google Directions API [3], dále jen Directions API, je nejrozšířenější webovou službou, kterou jsem testoval. Je to rozhraní poskytující informace o trasách mezi dvěma či více body. Její výstup je primárně určen pro další zpracování dalšími nástroji. Tato služba je placená v rámci Google Cloud Platform [4], ale poskytuje variantu zdarma s měsíčním omezením na počet požadavků.

Directions API nabízí dva způsoby, jak s ním komunikovat. První je použití poskytnuté knihovny schválené Googlem. Jejich seznam je dostupný v [5]. Tuto možnost lze využít u programovacích jazyků Java, Python, Go a Node.js. Poskyt-

nutý kód obaluje celou funkcionalitu pro komunikaci s Google Maps Web Services [6]. Tedy programátor nemusí nutně využívat pouze Directions API, ale i další služby v rámci tohoto balíčku. Tyto knihovny poskytují i funkce nad rámec jednoduché komunikace. Lze v nich nastavit limity počtu požadavků, chování při chybě nebo asynchronní zpracování vrácených dat. I z těchto důvodů je toto preferovaný způsob komunikace. Pokud ovšem pracujeme v jazyce, pro který není k dispozici knihovna, lze využít druhou metodu. K tomu je potřeba obyčejný HTTP klient podporovaný ve většině moderních jazyků. Programátor tedy musí manuálně odesílat požadavky na danou URL. Zároveň musí dodržet správný formát všech povinných parametrů a je zodpovědný za veškerou další režii, zpracování a práci s výslednou odpovědí.

Jak už jsem zmínil, Directions API slouží pro hledání cest mezi několika zeměpisnými body. To ale neodpovídá mému požadavku pro hledání tras ve veřejné dopravě. Directions API naštěstí podporuje i vyhledávání dle daného typu přepravy. Podporované jsou automobil, chůze, bicykl a veřejná doprava. Při zvolení poslední možnosti lze navíc i vybrat dopravní prostředek nebo zda preferujeme méně přestupů či chůze.

Directions API je mocným nástrojem pro hledání jakékoliv cesty mezi dvěma body. Díky detailním informacím o světové silniční síti a jízdních řádech jsou jeho výsledky velice přesné. K tomu přispívají i samotné dopravní společnosti, které mu v rámci Google Transit API [7] poskytují svá data.

Výstup této služby je ovšem dle mého názoru trochu komplikovaný. Je ve formátu JSON a jeho struktura je určena primárně pro další zpracování v rámci Google Maps Web Services ekosystému. Obsahuje například textové instrukce, velké množství zeměpisných souřadnic nebo zakódované geometrické tvary pro vykreslení v mapách. To se ovšem vzdaluje od mé představy jednoduché posloupnosti spojů a přestupů popisující cestu pomocí veřejné dopravy.

1.1.2 IDOS

IDOS[8] je internetová služba sloužící k vyhledávání tras ve veřejné dopravě. Poskytuje přesné informace o jízdních řádech a dokáže vyhledávat i komplikované cesty s několika přestupy. Svým uživatelům dovoluje zvolit si čas odjezdu, příjezdu nebo použitý dopravní prostředek. Zaměřuje se primárně na jízdní řády české republiky, obsahuje ovšem i informace o většině mezinárodních vlaků a autobusů v Evropě. Svá data navíc dle [9] aktualizují téměř denně.

Její architekturu popsanou v [10] lze rozdělit na dvě části:

- server provádějící veškeré výpočty,
- prezentační vrstva.

Díky tomu je možné ji využívat na celé škále zařízení se stejnými výsledky. Nabízí možnosti interakce pomocí webové a mobilní aplikace, počítačového klienta nebo SMS zpráv.

I když způsob realizace této služby nabízí možnosti napojení externích aplikací, jedná se o uzavřený systém nevystavující veřejné API. Jediné její externí rozhraní je pomocí URL odkazů se zadanými parametry popsaných v [11]. Takový odkaz přímo otevře webovou aplikaci s předvyplněnými informacemi. Tedy nevrací žádná relevantní data vhodná pro strojové zpracování.

IDOS je velmi užitečná webová služba pro hledání cest ve veřejné dopravě. Její výstup je přehledný a obsahuje pouze nezbytné informace pro popis výsledné sekvence spojů a přestupů. Tato data lze ovšem získat pouze pomocí poskytnutých nástrojů. To prakticky znemožňuje jejich další automatické zpracování nebo vlastní způsob zobrazování. V tomto případě neuvažuji metodu tzv. web-scrapingu, jejíž využití zde považuji minimálně za neetické.

1.1.3 TravelTime

TravelTime [12] je služba poskytující API pro hledání tras mezi zadanými body. Funguje velmi podobně jako Directions API. Její výstup je ve formátu JSON a obsahuje informace pro zobrazení cesty pomocí mapy. Je placená, ale nabízí variantu zdarma s měsíčním omezením.

S jejím rozhraním lze komunikovat pouze jedním způsobem, a to pomocí HTTP klienta. Programátor musí odesílat požadavky pomocí formátu specifikovaném v [13]. Je tedy plně zodpovědný za veškeré následné zpracování získané odpovědi. Existují i oficiálně nepodporované knihovny obalující tuto logiku. Ty jsou ovšem spravované pouze komunitou.

Stejně jako u Directions API, i zde je možné vyhledávat podle způsobu přepravy. Jedním z těchto způsobů je pomocí veřejné dopravy. Mezi další parametry cesty patří například maximální počet přestupů nebo délka čekání na spoj. Nenabízí ovšem podrobnější filtrování dopravních prostředků.

Výstup této služby obsahuje velké množství geografických bodů sloužících pro vykreslování pěších cest v mapách. Z toho důvodu má tendenci být méně přehledný v případech, kdy potřebujeme data pouze o veřejné dopravě. Navíc vyhledávání tras ve veřejné dopravě není její hlavní účel. Dle popisu v [14] se primárně zaměřuje na práci s lokacemi, jejich vzdálenostmi a dosažitelností pro účely cílených reklam.

1.1.4 Shrnutí

Při hledání existujících řešení jsem nenarazil na takové, které by odpovídalo mým požadavkům. Nejvíce se jim přiblížila služba TravelTime, jejíž výstup ovšem nenaplňoval mou představu. Mým cílem bylo nalézt takovou službu, která by se zaměřovala čistě na jednoduché hledání tras ve veřejné dopravě. Tedy jejíž výstup by byl složen ze střídající se posloupnosti spojů a přestupů bez zbytečných dat navíc. Tento požadavek splňovala služba IDOS. Ta ale bohužel neposkytuje veřejné API pro komunikaci s ní.

Po otestování výše zmíněných služeb jsem došel k závěru, že mé vlastní řešení musí komunikovat pomocí HTTP protokolu. Je to standard webových služeb a v kontextu využití mé aplikace dává největší smysl. Výstupní formát musí být také přehledný. Měl by obsahovat pouze data nezbytná pro výpis jednoduché sekvence spojů, kterými musí uživatel jet.

1.2 Existující algoritmy

V této sekci se zabývám algoritmy použitelných k vyřešení dané problematiky. Nejprve definuji pojmy potřebné pro jejich pochopení a jimi využívané datové struktury. Poté krátce popíši jejich princip fungování a nakonec zhodnotím jejich výhody a nevýhody.

Pro účely této práce jsem se rozhodl prozkoumat tři rozdílná pojetí řešení tohoto problému. Jedním z nich je převedení sítě veřejné dopravy na grafovou reprezentaci. Tu lze poté použít jako vstup pro některý ze známých algoritmů pro hledání nejkratší cesty v grafu. Další přístup pracuje se sekvencemi zastávek tvořící jednu cestu dopravního prostředku. Tyto sekvence řadí do tras, které následně prochází a tvoří pareto-optimální výstup dle času příjezdu a počtu přestupů. Nakonec se podívám na algoritmy stavící na tzv. spojeních (connections) a jejich postupným procházení.

Při porovnávání jejich rychlosti používám slovní spojení „velmi pomalé“ nebo „časově drahé“. V těchto případech se jedná o relativní pojmy a často označují časové jednotky v řádech stovek milisekund až jednotek sekund.

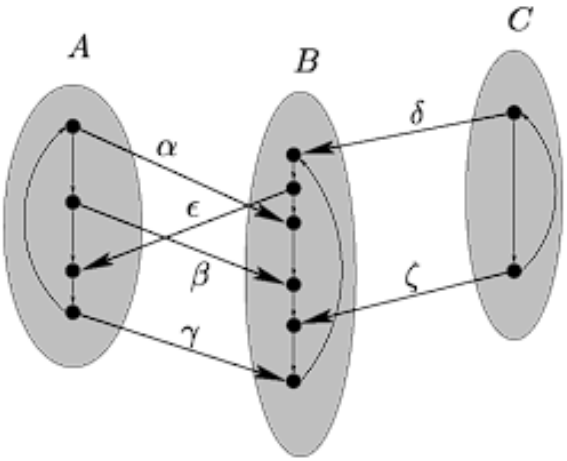
1.2.1 Grafové modely

Jako první jsem zkoumal možnosti reprezentace sítě veřejné dopravy formou grafu. K tomu se dle [15] nejčastěji používají dva způsoby jeho tvorby. První je Časově rozšířený model (Time-extended model) a druhý je Časově závislý model (Time-dependent model). Zde se zaměřuji na první zmíněnou variantu.

Pro účely popisu jejího fungování si nejprve zadefinuji tzv. elementárního spojení. To lze popsat jako uspořádanou pětici $C = (Z, S_1, S_2, T_d, T_a)$. S_1 je počáteční zastávka a S_2 je koncová zastávka. T_d je čas odjezdu z S_1, T_a je čas příjezdu do S_2 a Z je spoj veřejné dopravy. C tedy reprezentuje přímou cestu mezi S_1 a S_2.

Časově závislý model pracuje nad grafem, kde každý vrchol je událost (odjezd/-příjezd) ve stanici, která se udála v daný čas. Hrany odpovídají přímým spojům mezi zastávkami a jsou ohodnoceny jejich délkou.

Ve výsledném grafu se tedy objeví počáteční vrchol (Z, S_1, T_d) a koncový vrchol (Z, S_2, T_a). Platí že $T_a - T_d > 0$. Tyto vrcholy budou propojeny hranou reprezentující Z s váhou $T_a - T_d$. Nechť $N \coloneqq \{n_1, n_2, \cdots, n_k\}$ je množina všech vrcholů stanice S uspořádaná dle jejich času. Množina hran $W \coloneqq \{(n_i, n_{i+1}) \mid 1 \leq i \leq k-1\} \cup \{(n_k, n_1)\}$ tvoří přestupy v rámci jedné stanice. Jejich váhy jsou rozdílem



Obrázek 1.1 Výsledný graf časově rozšířeného modelu [15]

času koncového a počátečního vrcholu. Obrázek (1.1) ukazuje graf Časově rozšířeného modelu s nulovým časem přestupu. Mezi zastávkami A,B,C jsou přímá spojení. Tři spojují A a B (α, β, γ), dva jsou mezi C a B (δ, ζ) a jedno mezi B a A (ϵ).

Pro zohlednění nenulových délek přestupů navíc zavedeme další typ vrcholu. Tyto přestupové vrcholy jsou pouze kopie všech vstupních a výstupních uzlů dané stanice. Existují mezi nimi hrany z množiny W. Z každého vstupního uzlu nyní povede hrana k nejbližšímu přestupovému vrcholu s ohledem na čas potřebný pro přesun. Pokud zde spoj nekončí, vede z něj hrana i k jeho odjezdovému uzlu. Ohodnocení hran je stejné jako ve zjednodušeném modelu. Důkaz o správnosti tohoto postupu je v [15].

Jelikož jsou hodnoty hran nezáporné, lze nad takto sestrojeným grafem některý s algoritmů pro hledání cesty v ohodnoceném grafu. Dle [16] a [15] se ovšem nejčastěji používá klasický Dijkstrův algoritmus definovaný v [17]. Jeho běh můžeme poté zastavit v moment dosažení vrcholu koncové zastávky.

Z popisu tvorby potřebné datové struktury je jasné, že výsledný graf nebude nijak malý. Událostí, které jsou do něj zanesené, mohou být v rámci jedné stanice až tisíce. I to je jeden z důvodů proč je tato varianta méně preferovaná oproti časově závislému modelu. Dále je třeba zmínit, že jakýkoliv grafově založený model špatně reaguje na změny. Pro započtení zpoždění je třeba upravit časy všech navazujících vrcholů, což může být v závislosti na datech časově náročné. I tak je tento model schopen hledání nejrychlejší cesty mezi dvěma body. Navíc s rozšířením použitého algoritmu je schopen filtrování spojů na základě zadaných podmínek a optimalizace počtu přestupů.

1.2.2 RAPTOR

RAPTOR [16], neboli Iterativní trasování veřejné dopravy[1] je algoritmus stavící na fundamentálně odlišném přístupu. Dal by se vzdáleně přirovnat k záplavovým algoritmům na grafových modelech. Místo procházení jednotlivých uzlů ovšem pracuje nad vlastní strukturou reprezentující trasy linek veřejné dopravy. Je navržen tak, aby jeho výstup byl pareto-optimální v dané množině řešení.

Algoritmus funguje nad uspořádanou pětící $R = (\Pi, S, T, R, F)$. Π je množina časových jednotek (lze chápat jako vteřiny dne), S je množina zastávek, T je množina spojů, R je množina tras a F je množina přestupů (pěších cest). Zastávky v S jsou unikátně identifikovaná místo pro nástup nebo výstup z dopravního prostředku. Každý spoj $t \in T$ reprezentuje sekvenci zastávek jednoho dopravního prostředku. Každá zastávka v této sekvenci má přiřazený čas příjezdu $T_a \in \Pi$ a odjezdu $T_d \in \Pi$. Výjimku tvoří první a poslední zastávka, které mají nedefinovaný čas příjezdu respektive odjezdu. Platí, že $T_d \leq T_a$. Cesty $r \in R$ jsou tvořeny množinou spojů sdílející stejnou sekvenci zastávek seřazených dle jejich času odjezdu. Nakonec přestupy $f \in F$ jsou tvořeny uspořádanou dvojicí (s_1, s_2), kde $s_1, s_2 \in S$ s přiřazenou dobou trvání. Tato relace je tranzitivní, tedy $(s_1, s_2) \wedge (s_2, s_3) \Rightarrow (s_1, s_3)$.

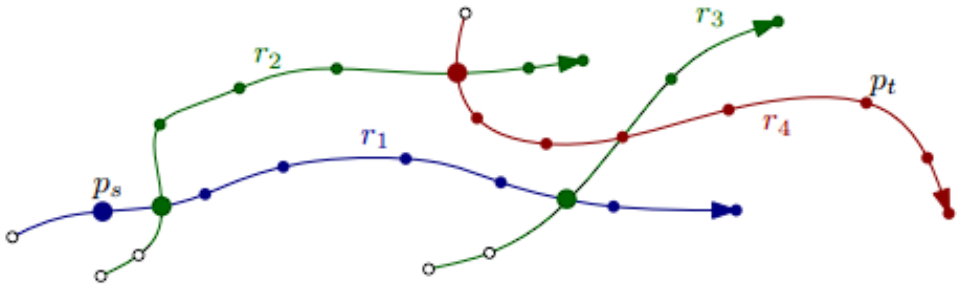
Již jsem zmínil, že algoritmus by se dal vzdáleně přirovnat k záplavovým algoritmům. Je to z toho důvodu, že pracuje v k iteracích. Při každém cyklu zpracuje všechny nově dosažitelné trasy z množiny R. Na konci k-té iterace je vypočtena cesta do každé dosažitelné zastávky pomocí maximálně $k - 1$ přestupů. Obrázek (1.2) zachycuje hledání cesty z p_s do p_t. V první iteraci se prochází cesta r_1. Ve druhé poté cesty r_2 a r_3 a poslední je zpracována cesta r_4. Prázdné zastávky nebyly v tomto průchodu vůbec zváženy.

Nechť s_0 je naše počáteční stanice a π je zvolený čas odjezdu. Před spuštěním výpočtu je každé zastávce nastaven nejkratší čas příjezdu na ∞. Následně označíme s_0 za dosažitelnou a nastavíme jí čas příjezdu na π. Každá iterace se poté skládá ze tří kroků:

1. Nalezení všech spojů, které vyjíždí z nově dosažených zastávek.
2. Výpočet nejkratších časů příjezdu pro všechny zastávky dosažitelné pomocí nalezených spojů.
3. Započtení pěších cest z nové dosažitelných zastávek.

Tento proces se opakuje dokud nedosáhneme našeho cíle, nepřekročíme maximální počet iterací (přestupů) nebo v poslední iteraci nedošlo k vylepšení času příjezdu žádné zastávky. Důkazy a plné definice jsou k nalezení v [16]. Výstup tohoto algoritmu je ve formátu střídající se sekvence přestupů a spojů. Tento model je také schopen efektivně vyhledávat dle více kritérií. Navíc už ze svého principu

[1] Překlad autora



Obrázek 1.2 Hledání cesty RAPTOR [16]

optimalizuje výslednou cestu nejen dle nejrychlejšího času příjezdu, ale i podle nejméně přestupů v pareto smyslu.

1.2.3 CSA

Connection Scan Algorithm [18] (Algoritmus skenující spojení[1]) je další algoritmus vymykající se standardnímu grafovému pojetí. Staví na sekvenčním přístupu procházení jednorozměrného pole. Díky této vlastnosti dokáže velmi rychle vyhledávat trasy i na velkých datech.

Jeho model je založen nad uspořádanou čtveřicí (S, C, R, F). S je množina zastávek, C je množina spojení, R je množina linek veřejné dopravy a F je množina přestupů. Zastávky v S jsou unikátně identifikovatelná místa pro nástup nebo výstup z dopravního prostředku. Linky v R značí plánovanou jízdu dopravního prostředku, například konkrétní autobus nebo vlak. Spojení obsažena v C jsou uspořádané pětice reprezentující cestu mezi zastávkami $(c_{depStop}, c_{arrStop}, c_{depTime}, c_{arrTime}, c_{trip})$. Tyto vlastnosti označujeme jako počáteční zastávka, koncová zastávka, čas odjezdu z počáteční zastávky, čas příjezdu do koncové zastávky a použitá linka v tomto pořadí. Přestupy z F jsou uspořádané trojice $(f_{depStop}, f_{arriStop}, f_{dur})$. Jejich vlastnosti poté nazýváme jako počáteční stanice, koncová stanice a délka přestupu v tomto pořadí. Je vyžadováno, aby $\forall f_i, f_j \in S$ byla relace (f_i, f_j, x) tranzitivní.

Při svém běhu prochází všechna spojení v C vzestupně dle jejich času odjezdu. Při tom přeskakuje všechna spojení, jejichž linka nebo počáteční zastávka nebyly dosud dosaženy. Pokud narazí na přijatelné spojení, pokusí se o zlepšení nejlepšího času příjezdu všech zastávek pěšky dostupných z koncové stanice. (1.3) ukazuje pseudokód tohoto algoritmu včetně podpůrných datových struktur. Při tom si pamatuje sérii spojení, které vedla do dané zastávky pro účely extrakce finální cesty. Tato iterace končí v moment, kdy:

1. se bylo zpracováno poslední dostupné spojení,

[1] Překlad autora

```
1 for all stops x do S[x] ← ∞;
2 for all trips x do reset T[x];
3 for all footpaths f from s do S[f_arr_stop] ← τ + f_dur;
4 for all connections c increasing by c_dep_time do
5     if T[c_trip] is set or S[c_dep_stop] ≤ c_dep_time then
6         raise T[c_trip];
7         for all footpaths f from c_arr_stop do
8             S[f_arr_stop] ← min{S[f_arr_stop], c_arr_time + f_dur};
```

Obrázek 1.3 Pseudokód Connection Scan Algoritmu [19]

2. nejlepší čas příjezdu do koncové zastávky je lepší, než čas odjezdu současného spojení.

Správnost tohoto principu je rozvedena v [18]. I když to je velmi rychlý algoritmus, obsahuje jednu fatální vadu. Nedokáže optimalizovat cestu na nejmenší počet přestupů, pouze nejrychlejší čas příjezdu. Z toho plyne, že jeho výstup může být mnohdy až nesmyslný. Například může donutit uživatele desetkrát přestoupit a být o minutu rychlejší, nežli jet přímým spojem.

Jeho největší výhoda je ve velmi rychlé přípravě dat. Ta spočívá v seřazení množiny spojení dle jejich času odjezdu. To lze volbě správného algoritmu provést v řádu sekund na množině o velikosti několik milionů spojení, viz. [19]. Díky tomu dokáže rychle reagovat na nastalá zpoždění, zrušené spoje nebo výluky.

1.2.4 pCSA

Profile Connection Scan Algorithm (Profilový algoritmus skenující spojení[1]) je rozšíření standardního CSA. Obsahuje v sobě značná vylepšení oproti jeho základní variantě. Umožňuje optimalizaci na základě jak nejrychlejšího času příjezdu, tak dle nejmenšího množství přestupů. Ta může být dle popisu v [19] realizovaná jak v pareto smyslu, tak dle jiné logiky. Pracuje nad stejnou datovou strukturou jako obyčejný CSA. Jeho běh je ovšem velmi odlišný. Skenovaná spojení neprochází vzestupně dle jejich času odjezdu, nýbrž sestupně. Výsledné cesty tedy sestavuje od pozdějších po dřívější. Při tom využívá toho faktu, že dřívější cesta v sobě obsahuje pouze pozdější, již nalezenou cestu. Jedná se tedy o použití techniky dynamického programování.

Při procházení těchto spojení si pro každou zastávku x tvoří tzv. profil $S[x]$. Ten je tvořen množinou uspořádaných dvojic (s, r_c), kde s je čas odjezdu z dané zastávky a r_c značí čas příjezdu do cíle naší cesty. Tato množina je seřazena dle

[1] Překlad autora

prvního parametru vzestupně. Při vkládání nové hodnoty (s_1, t_1) do některého z profilů jsou z něj vyřazeny všechny dvojice (s_2, t_2) splňující $s_1 \geq s_2 \wedge t_1 \leq t_2$. Tím je zajištěna pareto optimalita hodnot této množiny dle času příjezdu.

Ještě je třeba zmínit, odkud jsou brány s a r_c. Parametr r_c spočítáme jako nejmenší hodnotu z následujících případů:

1. čas příjezdu při chůzi do cílové zastávky,
2. čas příjezdu při pokračování současným spojem,
3. čas příjezdu při přestupu na jiný spoj.

Parametr s je poté roven $c_{depTime}$. Nakonec zakomponuj (s, r_c) do profilu každé zastávky, ze které se lze dostat pěšky do $c_{depStop}$ s ohledem na čas přestupu. Díky takto vytvořeným datovým strukturám jsme schopni porovnávat různé cesty do našeho cíle a dělat nad nimi další rozhodnutí. Obrázek (1.4) ukazuje pseudokód těchto operací. K jeho pochopení je ještě potřeba dovysvětlit formát struktury T. Ta reprezentuje seznam obsahující nejrychlejší časy příjezdu do cíle naší cesty sedíme-li ve spoji $x \in R$.

Pro extrakci cesty si je třeba prvky v profilech rozšířit o dva prvky. Pro zastávku z tedy definuji strukturu $(s, t, c_{enter}, c_{exit})$. c_{enter} značí spojení, které cestující použil k opuštění zastávky z a c_{exit} spojení, na jehož konci vystoupil ze spoje. Tyto dvě informace navíc lze poté použít k extrakci optimální cesty mezi zastávkami.

Jedna z optimalizací tohoto algoritmu staví na jeho základní verzi. Jelikož je celkový běh této rozšířené varianty výrazně pomalejší, je dobrý nápad přeskakovat spojení, jejichž linky nejsou dosažitelné. To není možné při procházení tohoto seznamu sestupně. Proto je úkolem základního CSA vygenerovat seznam dostupných linek při jeho normálním běhu. I přes to, že se tato akce může zdát neintuitivní, její využití znatelně zrychluje běh celého algoritmu. Správnost celého postupu a optimalizací je k dočtení v [19].

1.2.5 Shrnutí

Všechny výše zmíněné algoritmy jsou schopné vyřešit problém nejrychlejšího příjezdu. Při jejich prozkoumávání jsem nejprve stál na straně grafového přístupu. Jeho velkou nevýhodou je ovšem náročná příprava dat. Při každé změně je třeba přegenerovat celý graf. To může být ve velké síti veřejné dopravy časově náročné. Takové algoritmy tedy špatně reagují na náhlé události, jako je zpoždění nebo zrušení spoje.

Na druhou stranu mou pozornost upoutaly algoritmy stavící na jiných postupech. RAPTOR je velmi mocný model nabízející optimalizaci počtu přestupů už v jeho základní variantě. Více se ovšem přikláním k CSA. Konkrétně k jeho rozšířené variantě pCSA. Velmi mne zaujala jeho jednoduchost a rychlost. Nepotřebuje pracovat nad nijak složitými datovými strukturami. Celý jeho princip stojí v podstatě na jednom sekvenčním průchodu jednorozměrného pole.

```
1 for all stops x do Initialize stop data structure S[x];
2 for all trips x do Initialize trip data structure T[x];
3 for connections c decreasing by c_dep_time do
      /* 1. Determine arrival time when starting in c
4     τ_1 ← arrival time when walking to the target;
5     τ_2 ← arrival time when remaining seated, uses T[c_trip];
6     τ_3 ← arrival time when transferring, uses S[c_arr_stop];
      /* τ_c = arrival time when starting in c
7     τ_c ← min{τ_1, τ_2, τ_3};
      /* 2. Incorporate τ_c into the data structures
8     Incorporate τ_c into S[x] for all stops x with footpath (x, c_dep_stop);
9     Incorporate τ_c into T[c_trip];
```

Obrázek 1.4 Pseudokód Profile Connection Scan Algoritmu [19]

Pro praktickou část jsem tedy zvolil implementovat pCSA algoritmus. Primární důvod této volby je jeho jednoduchý a přesto dynamický princip fungování.

1.3 MEAT problém

MEAT problém, neboli problém Minimální očekávané doby příjezdu[1], se dle popisu v [19] zabývá poskytnutím alternativních cest při předpokladu zpoždění nějakého spoje. Cílem řešení tohoto problému je pro každou cílovou stanici ve výsledné trase připravit záložní variantu v moment prvotního výpočtu. Ty jsou poté aplikované v případě, že dojde k opoždění či výluce některé navazující linky. Celý tento proces může působit až přehnaně pesimisticky. Místo něj můžeme spustit náš algoritmus znovu s upravenými časy odjezdu a příjezdu. Ovšem cestujeme-li nepředvídatelnou sítí veřejné dopravy, může být poskytnutí alternativy již na začátku naší výpravy žádanou možností.

Tento problém není identický s obyčejným započtením známého zpoždění. Pracuje totiž se všemi možnostmi zpomalení některého spoje na základě definovaných parametrů. Z toho důvodu se velmi liší od klasického přístupu při hledání nejrychlejší cesty. Většina existujících algoritmů dokáže řešit problém započtení zdržení linky na trase. Ovšem nalezení vhodných alternativních spojů vyžaduje další výpočty navíc.

Jedním z algoritmů, který je do jisté míry schopen řešit MEAT problém je pCSA. S drobnou úpravou formátu jeho profilů ukázanou v [19] je schopen v konstantním čase nalézt následující spoj jedoucí do našeho cíle. Tento postup ovšem není dokonalý. Nelze totiž zaručit, že výsledná alternativa bude ta nejrychlejší. Další varianta je sestrojení rozhodovacího grafu místo jediné sekvence zastávek.

[1] Překlad autora

Ta ovšem může být v závislosti na použitém systému velmi časově náročná.

1.4 Formát GTFS

V [20] je formát GTFS definován jako společný formát pro jízdní řády veřejné dopravy a s nimi souvisejících lokací. Jedná se o skupinu textových souborů v jednom zip archivu. Každý soubor poté obsahuje informace o jednom aspektu těchto jízdních řádů. Pro potřeby této práce byly použity následující soubory a jejich parametry:

stops.txt – Tento soubor obsahuje seznam všech lokací figurující v dané síti veřejné dopravy. K jejich konkrétnímu dělení je použit příznak location_type.

- stop_id – Povinný atribut ve formátu textového řetězce. Slouží jako unikátní identifikátor lokace.
- wheelchair_boarding – Příznak, zda lze z lokace nastoupit do dopravního prostředku s invalidním vozíkem. Povolené hodnoty jsou:

  0 nebo prázdné – není známo. Pokud je vyplněn parametr parent_station, použij informaci stanice, pod kterou lokace spadá;

  1 – některé dopravní prostředky podporují nástup s invalidním vozíkem z tohoto místa;

  2 – z místa nelze nastoupit s invalidním vozíkem.
- location_type – Příznak určující typ lokace. Může nabývat hodnot:

  0 nebo prázdné – zastávka nebo nástupiště. Místo pro nástup/výstup do/z dopravního prostředku;

  1 – stanice. Fyzická struktura sdružující několik nástupišť;

  2 – vchod/východ. Místo kudy lze vstoupit/vystoupit do/ze stanice;

  3 – obecný bod. Místo uvnitř stanice neodpovídající žádnému popisu. Může sloužit pro definice pěších cest;

  4 – nástupní prostor. Konkrétní místo na nástupišti.
- stop_name – Název lokace pro účely orientace. Povinný pro zastávky, stanice a vchody/východy.
- stop_lat – Zeměpisná šířka lokace ve tvaru desetinného čísla. Povinná pro zastávky, stanice a vchody/východy.
- stop_lon – Zeměpisná délka lokace. Platí pro ní to samé, jako pro zeměpisnou šířku.
- parent_station – Identifikátor stanice nebo nástupiště, pod které lokace spadá. Je povinný pro vchody/východy, obecné body a nástupní prostory a zakázaný pro stanice.

routes.txt – Obsahuje seznam tras sítě veřejné dopravy. Trasa pod sebou seskupuje linky obsluhující stejnou sekvenci zastávek.

- route_id – Unikátní textový identifikátor trasy.
- route_short_name – Krátký název trasy, který nepopisuje její směr nebo cíl. Jako příklady lze uvést „B“, „R17“ nebo „MHD 1“. Je povinný v případě, kdy není znám její dlouhý název.
- route_long_name – Dlouhý název trasy, který více popisuje její směr nebo cílovou zastávku. Jako příklad uvedu „Praha - Čerčany - Benešov u Prahy“ nebo „Zličín - Černý Most“. Je povinný v případě, kdy není znám její krátký název.
- route_type – Příznak určující typ dopravního prostředku, který jezdí na trase. Povolené hodnoty jsou:
  - **0** – tramvaj nebo jiný nadzemní způsob přepravy v městské oblasti,
  - **1** – metro nebo jiný podzemní způsob přepravy v městské oblasti,
  - **2** – vlaková doprava,
  - **3** – autobus,
  - **4** – trajekt nebo jiný typ vodní přepravy,
  - **5** – železniční vozy na úrovni ulice, které jsou taženy kabelem vedeným pod vozidlem,
  - **6** – lanová doprava, kde jsou kabiny nebo sedačky zavěšeny pomocí jednoho nebo více kabelů,
  - **7** – železniční systém určený pro prudká stoupání,
  - **11** – trolejbus,
  - **12** – jednokolejka.

trips.txt – Obsahuje seznam linek veřejné dopravy. Linka je konkrétní dopravní prostředek jedoucí v daný čas po určené trase.

- route_id – Povinný identifikátor trasy, pod kterou linka spadá. Odkazuje na parametr route_id v souboru routes.txt.
- service_id – Povinný identifikátor služby, která je přiřazena lince. Služby slouží k určení, kdy které linky jezdí. Odkazuje na parametr service_id v souborech calendar.txt a calendar_dates.txt.
- trip_id – Povinný unikátní textový identifikátor linky.
- trip_headsign – Nepovinný text zobrazený na dopravním prostředku linky. Obvykle obsahuje název cílové zastávky.
- trip_short_name – Nepovinný název linky. Slouží k identifikaci dopravního prostředku pro cestující.

- wheelchair_accessible – Nepovinný příznak, zda linka umožňuje bezbariérový přístup. Povolené hodnoty jsou:
  - **0** nebo prázdné – nejsou dostupné informace;
  - **1** – dopravní prostředek linky má místo pro alespoň jeden invalidní vozík;
  - **2** – dopravní prostředek linky nemá místo pro invalidní vozík.
- bikes_allowed – Označuje, zda jsou v dopravním prostředku linky místo pro jízdní kola. Je nepovinný. Povolené hodnoty jsou:
  - **0** nebo prázdné – nejsou dostupné informace;
  - **1** – dopravní prostředek linky má místo pro alespoň jedno jízdní kolo;
  - **2** – dopravní prostředek linky nemá místo pro jízdní kola.

stop_times.txt - Obsahuje časy zastavení linek u jednotlivých zastávek. V podstatě tedy popisuje trasy těchto linek.

- trip_id – Povinný unikátní identifikátor linky. Odkazuje do souboru trips.txt.
- arrival_time – Čas příjezdu do zastávky. Je ve formátu HH:MM:SS. Může nabývat hodnoty větší než 24:00:00 v případě, kdy linka vyjíždí před půlnocí a končí po ní. Je povinný pro první a poslední zastávku pro daný spoj.
- departure_time – Čas odjezdu ze zastávky. Je ve stejném formátu jako arrival_time. Na rozdíl od něj ale není povinný.
- stop_id – Identifikátor zastávky. Odkazuje do souboru stops.txt.
- stop_sequence – Pořadí, ve kterém linka navštívila danou zastávku.

calendar.txt - Některé linky veřejné dopravy jezdí pouze několik dní v týdnu. Tento soubor obsahuje rozpisy služeb. Ty sdružují tyto linky a určují, kdy jsou dostupné na týdenní bázi.

- service_id – Unikátní identifikátor služby. Je ve formátu textového řetězce.
- monday, tuesday, wednesday, thursday, friday, saturday, sunday – Sedm po sobě jdoucích parametrů, které reprezentují dny v týdnu od pondělí do neděle v tomto pořadí. Tyto příznaky značí, zda jsou linky služby dostupné v daný den. Mohou nabývat hodnoty 0 a 1. Pokud tedy příznak monday obsahuje hodnotu 1, linky služby jsou dostupné ve všechna pondělí ze zadaného rozsahu.
- start_date – Počáteční datum, od kdy rozpis služeb platí.
- end_date – Koncový datum, do kdy rozpis služeb platí.

calendar_dates.txt – V tomto souboru lze konkrétně definovat dostupnosti služeb v konkrétní dny. Lze ho použít dvěma způsoby. První je v kombinaci se souborem calendar.txt. V rozpisech jízdních řádů se často objevují výjimky.

Například některé linky nejsou dostupné ve státní svátky nebo při mimořádných akcích. Tento soubor poté může obsahovat tyto nepravidelnosti. Toto je preferovaná varianta použití. Druhá je samostatně bez souboru calendar.txt. V tom případě zde musí být popsány dostupnosti všech služeb v konkrétní dny.

- service_id – Id služby.
- date – Konkrétní den, kterého se týká úprava dostupnosti služby.
- exception_type – Typ změny dostupnosti. Může obsahovat následující hodnoty:
  - **1** – služba byla zařazena do provozu,
  - **2** – služba byla vyřazena z provozu.

pathways.txt – V některých případech nejsou přestupy mezi zastávkami úplně přímočaré. V tomto souboru lze definovat pěší cesty mezi dvěma body, které jsou definované v souboru stops.txt. GTFS reference ovšem povoluje definovat tyto cesty pouze v rámci jedné stanice.

- pathway_id – Unikátní identifikátor této pěší cesty. Je ve tvaru textového řetězce.
- from_stop_id – Id počáteční zastávky. Odkazuje na id v souboru stops.txt.
- to_stop_id –Id koncové zastávky. Odkazuje na id v souboru stops.txt.
- is_bidirectional – Příznak, zda je cesty obousměrná. Povoleny jsou tyto hodnoty:
  - **0** – cesta je jednosměrná a vede z počáteční zastávky do koncové,
  - **1** – cestu lze použít oběma směry.
- traversal_time – Čas v sekundách udávající průměrnou délku cesty.

Seznam všech povinných a nepovinných souborů a jejich obsah je popsán v [21]. Jejich struktura je podobná formátu CSV. Díky tomu jsou velmi dobře strojově čitelné.

Při bližším prozkoumání tohoto formátu jsem si uvědomil nesplnitelnost části mého zadání. Konkrétně části „Vstup i výstup z tohoto rozhraní bude ve formátu GTFS“. GTFS slouží k ukládání a přenosu statických informací o síti veřejné dopravy, nikoliv pro popis cesty v ní. Jedním z možných řešení tohoto problému je výstup mé aplikace „ohnout“ do jeho tvaru. V tu chvíli by se ovšem nejednalo o formát GTFS dle jeho definice. Výsledek by mu byl pouze podobný strukturou, ale obsah by byl nevalidní pro jakýkoliv další nástroj. V reakci na toto zjištění jsem se rozhodl využít jiný datový formát pro výstup mé služby, konkrétně JSON. Jeho přesnou strukturu poté definuji v (2.3)

1.5 Taransit

Taransit [22] je mobilní aplikace, která poskytuje vlakové jízdní řády pro indický trh. Tato data dokáže poskytovat jak v online, tak offline režimu. To je vzhledem k relativně nízkému pokrytí mobilních sítí v Indii dle [23] žádaná vlastnost. Na Google play má již přes 50 tisíc instalací [24].

Tato aplikace je neustále ve vývoji a její ambice nekončí u poskytování statických jízdních řádů. Jejím primárním cílem je sběr informací o zpožděních vlaků na základě dat od jejích uživatelů. Ty chce následně využít pro plánování komplexních tras v této vlakové síti. Indičtí dopravci je totiž málokdy poskytují online a častokrát nejsou známy až do příjezdu vlaku do další stanice. Dle informací z [25] se zpoždění některých vlaků může pohybovat v řádech i desítek hodin.

Již jsem nastínil, že proces získávání informací o zpožděních vlaků má fungovat formou tzv. live dat. Idea je taková, že člověk jedoucí v daném spoji povolí sdílení své polohy. Sběr těchto dat bude tedy plně automatický s minimálním vstupem od uživatele. Navíc je v plánu vedení této činnosti formou hry. Za sdílení své polohy by uživatelé získávali nějakou herní měnu či jiné odměny. Ty mají sloužit jako motivace tato data poskytovat.

Další zajímavostí o této aplikaci je, že neobsahuje reklamy. To je na indickém trhu spíše rarita.

1.6 Podklady pro návrh vlastního řešení

V závěru této kapitoly se podívám na případy užití a požadavky kladené na mé řešení (dále jen služba). Ty byly sestaveny na základě konzultací s Jakubem Turcovským pro potřeby aplikace Taransit. Slouží jako podklady pro praktickou část a způsoby jejího vývoje. Jsou v nich také zohledněné informace získané z předchozích částí.

Tato služba je cílena na firmy (dále zákazníky) provozující vlastní webovou nebo mobilní aplikaci. Tu poté nazývám klientskou aplikací. Její uživatelé jsou z mého pohledu i mí uživatelé.

1.6.1 Funkční a nefunkční požadavky

Následující výčet zahrnuje všechny požadavky na vyvíjenou službu.

Tato služba je určena k internímu použití. Uživatelé s ní tedy nekomunikují přímo, ale přes nějakého prostředníka. Tím může být například reverzní proxy. Z toho důvodu zde nejsou požadavky na její zabezpečení.

Funkční požadavky

F1: *Hledání cesty* – Uživatel může vyhledat cestu mezi dvěma zastávkami.

F2: *Optimální cesta* – Výsledná cesta bude optimalizovaná nejen podle nejrychlejšího času příjezdu, ale i dle počtu přestupů.

F3: *Čas odjezdu* – Uživatel bude moci zadat čas odjezdu z jeho vybrané počáteční stanice.

F4: *Zpoždění* – Služba bude schopna zapracovat vzniklé zpoždění některého spoje do svého výsledku.

F5: *Filtrování* – Uživatel bude schopen zadat filtry při hledání cesty. Mezi ně patří daný typ prostředku, bezbariérový přístup a místo pro jízdní kolo.

F6: *Aktualizace dat* – Služba bude schopna přijmout a zpracovat data ve formátu GTFS. Tato data si poté uloží a bude nad nimi pracovat.

Nefunkční požadavky

N1: *Databáze* – Služba musí být schopna pracovat nad různými relačními databázemi.

N2: *Paměťová náročnost* – Algoritmus je v současné době určen pro běh na serveru. V budoucnosti by ale mohl běžet v režimu bez přístupu k internetu na mobilních zařízeních. V takových případech je velikost dostupné RAM velmi omezená, jak je popsáno v [26].

N3: *Rozhraní* – Služba bude komunikovat s uživatelskou aplikací pomocí HTTP protokolu.

N4: *Výstup* – Výstup služby by měl být ve formátu JSON, jednoduše čitelný a ve formě střídající se sekvence spojů a přestupů.

1.6.2 Případy užití

V případech užití této služby figurují dva účastníci. První je její uživatel, který ji využívá pro hledání cesty ve veřejné dopravě. Druhý je její provozovatel, který ji spravuje.

U1: *Vyhledání cesty* – Uživatel se dotazuje na cestu ze zastávky *A* do zastávky *B*.

U2: *Čas odjezdu* – Uživatel svou cestu může začít nejdříve v zadaný čas.

U3: *Bezbariérový přístup* – Uživatel pro svou cestu potřebuje využít spoje a zastávky, které poskytují bezbariérový přístup.

U4: *Cestování na kole* – Uživatel cestuje na bicyklu a potřebuje využít spoje veřejné dopravy, které mají místo pro jízdní kolo.

U5: *Výběr dopravního prostředku* – Uživatel chce pro svou cestu využít pouze vybrané typy dopravních prostředků veřejné dopravy. Může například chtít jet pouze vlakem, protože autobusy mu jsou nepohodlné.

U6: *Aktualizace dat* – Provozovatel této služby aktualizuje data o síti veřejné dopravy.

U7: *Zadání zpoždění* – Provozovatel této služby může zadat zpoždění některého spoje.

Kapitola 2
Návrh architektury

V této kapitole se věnuji návrhu architektury pro implementaci praktické části této práce.
Navrhuji zde způsoby realizace a fungování jejích klíčových funkcí. Nejprve odůvodňuji mou volbu použitého algoritmu. Následně popisuji jeho úpravy pro účely splnění funkčních požadavků z (1.6.1) a dokazuji jejich správnost. Na to navazuji popisem práce s perzistentními daty. Tvořím zde návrh databázového modelu a ukazuji nezbytné transformace vstupních dat do reprezentace potřebné pro správný běh zvoleného algoritmu. V neposlední řadě definuji konkrétní formu výstupu mé služby a popisuji mnou zvolenou metodiku vývoje praktické části.

2.1 Volba použitého algoritmu

Na samotném začátku jsem potřeboval zvolit vhodný algoritmus pro výpočet tras. Ten musel mít všechny vlastnosti pro splnění požadavků z (1.6.1). Při tomto výběru jsem vycházel z analýzy provedené v (1.2). Troufám si říci, že se jedná o jedno z nejdůležitějších rozhodnutí v rámci implementace praktické části této práce. Odvíjí se od něj jak nezbytné transformace vstupních dat, tak databázový model pro jejich efektivní uložení.

Hned na začátku jsem ze svého výběru vyloučil základní CSA. Ten nesplňoval požadavek na optimalizaci výsledné trasy dle počtu přestupů. Dále jsem po důkladném zvážení rozhodl odpustil od grafových algoritmů. I když jsou při využití Časově rozšířeného modelu schopny splnit všechny mé kladené požadavky, nevýhody popsané v (1.2.1) mě od nich odradily. Největší vliv na toto rozhodnutí měla velikost výsledného grafu a z toho vyplývající komplikovaná reakce na vzniklé zpoždění některého spoje.

Má výsledná volba se tedy odehrála mezi pCSA a RAPTOR algoritmem. Oba jsou schopny splnění všech funkčních požadavků a každý z nich staví na fundamentálně rozdílném principu. RAPTOR nabízí již vestavěný způsob hledání optimální cesty dle více parametrů, zatímco pCSA vyniká svou jednoduchou přípravou dat.

Obě tyto varianty mají tedy své silné stránky. Nakonec jsem se ovšem rozhodl pro využití pCSA. Zapůsobily na mě primárně jeho jednoduchost a rychlost vycházející z experimentů v rámci [19].

2.1.1 Specifikace parametrů cesty

V této sekci se zabývám způsoby parametrizace výsledné trasy. Jako příklad zde uvedu možnost vyhledání trasy pouze za použití vybraných typů dopravních prostředků. Tyto parametry vychází z případů užití vyjmenovaných v (1.6.2).

Jde tedy o omezení použitelných spojů a zastávek dle zadaných parametrů. To je konceptuálně velmi jednoduché. Stačí z této sítě dané zastávky a spoje odebrat. Po tomto kroku lze triviálně prohlásit, že výsledkem je stále validní síť veřejné dopravy.

Tuto operaci lze aplikovat na všechny způsoby filtrování specifikované v případech užití. Zbývá tedy popsat efektivní proces, jak tuto síť veřejné dopravy omezit při běhu zvoleného algoritmu.

Jak je popsáno v (1.2.4), pCSA pracuje nad datovou strukturou tzv. spojení. Ty následně vkládá do seznamu a sekvenčně prochází. Toto spojení v sobě mimo jiné obsahuje počáteční a koncovou zastávku a linku, která je spojuje. Lze tedy z tohoto seznamu vynechat všechna spojení, jejichž zastávky/spojení nesplňují zadané parametry. Ve výsledku se tedy nejedná o úpravu algoritmu, nýbrž o omezení dat nad kterými pracuje.

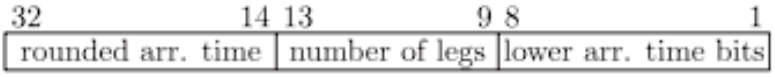
2.1.2 Optimalizace počtu přestupů

[19] ukazuje dvě možnosti optimalizace cesty dle počtu přestupů pro pCSA. Při popisu potřebných změn vycházím z pojmů definovaných v (1.2.4).

První z nich je optimalizace v pareto smyslu. I když tato varianta dává nejpřesnější výsledky, její komplexita znatelně zpomaluje výsledný běh algoritmu. Tato optimalizace spočívá v nahrazení všech původních profilů uspořádanou dvojicí $(s_{depp}, (A_1, A_2, \cdots, A_{max}))$. Index $max \in \mathbb{N}$ označuje maximální povolený počet přestupů. Element A_l reprezentuje nejrychlejší příjezd do naší cílové stanice pomocí maximálně l přestupů. Tedy každý profil nyní obsahuje nejrychlejší cestu z dané zastávky pro každý uvažitelný počet přestupů. Tuto strukturu upravujeme při výpočtu délky cesty při přestupu, podobně jako v klasickém pCSA.

Druhá možnost ze může zdát na první pohled více komplexní, její princip není ovšem nijak komplikovaný. Spočívá totiž v zakódování počtu přestupů do samotného času příjezdu do cílové zastávky.

Uvažujme reprezentaci času jakožto časové značku s rozlišením na sekundy. Jelikož v jednom roce jich je zhruba 31 536 000, k tomu určitě bude stačit bezznaménkový 32-bitový integer. Nyní do tohoto čísla zakóduji počet přestupů. Rozdělím jeho 32 bitů na 2 části, levou a pravou. Levá část nyní obsahuje časovou značku na bitech 32-6. V pravé části o velikosti 5 bitů je poté uložen počet přestupů. Vzhle-



Obrázek 2.1 Struktura časové značky pro pCSA

dem k bitové povaze porovnávání čísel v procesoru se nejprve porovná hodnota časové značky v levé části. Počet přestupů je brán v potaz pouze v případě, kdy jsou levé části stejné. Tato metoda tedy optimalizuje počet přestupů v případě, kdy jsou časy příjezdů identické. Stále ovšem mohou nastat situace, kdy je o vteřinu rychlejší cesta s více přestupy preferovaná před přímým spojem.

Rozšíříme tedy tuto optimalizaci následovně. Rozdělme si jeden den do segmentů o velikosti 2^8 sekund. Nyní všechny nejrychlejší časy spadající do jednoho segmentu můžeme prohlásit za stejně rychlé a jsme schopni vybrat ten s nejméně přestupy. V podstatě „zaokrouhlujeme“ tyto časy na násobky 256. Nové kódování naší časové značky bude vypadat jako na obrázku (2.1). Bity 32-14 obsahují horní, „zaokrouhlenou“, část našeho času. Bity 13-9 reprezentují počet přestupů a bity 8-1 obsahují spodní část původní časové značky. Opět je zde využita vlastnost binárního porovnávání celých čísel. Díky ní jsme tedy schopni optimalizovat dle následující kritérií:

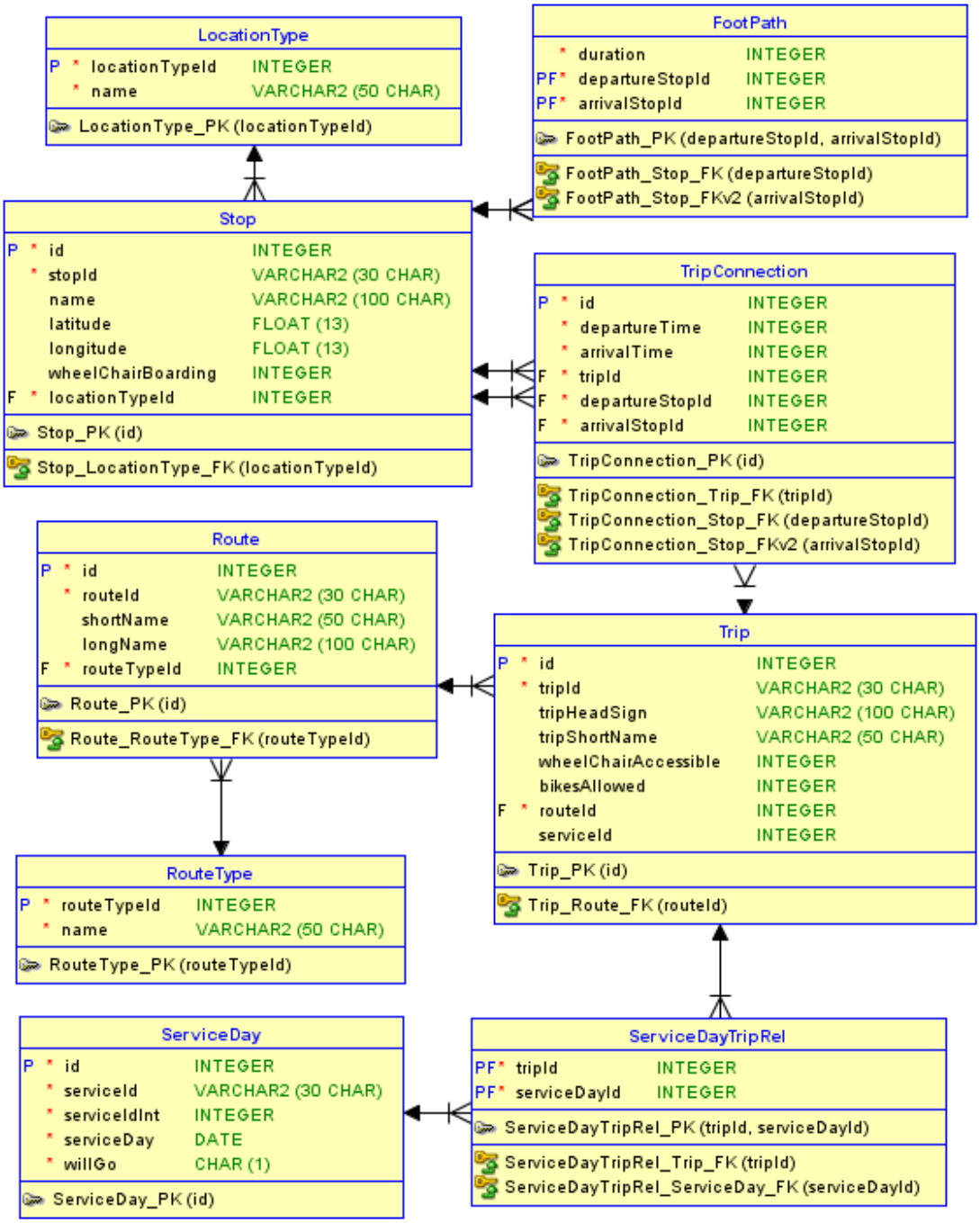
1. „zaokrouhleného“ času příjezdu,
2. nejmenšího počtu přestupů,
3. skutečného času příjezdu.

Tato metoda kompletně neřeší případy, kdy optimální cesta obsahuje větší množství přestupů. Dle [19] ovšem snižuje množství jejich výskytů na přijatelně nízké množství.

Po důkladném zvážení všech kladů a záporů jsem se rozhodl pro implementaci druhé varianty. I když neřeší tento problém dokonale, její výrazně jednodušší implementace a rychlejší běh považuji za důležité vlastnosti. Zároveň mě fascinoval způsob jejího fungování za použití základních bitových operací.

2.2 Práce s daty

Pro účely správného fungování výsledné služby je třeba zpracovat a uložit poskytnuté informace o síti veřejné dopravy. V této části se tedy nejprve věnuji zvolenému databázovému modelu s ohledem na zvolený algoritmus v (2.1). Následně popisuji proces transformace poskytnutých GTFS dat do tohoto modelu.



Obrázek 2.2 Relační model databáze

2.2.1 Návrh databáze

Z důvodu splnění požadavku (1.6.1) jsem měl při realizaci perzistentního úložiště na výběr pouze relační databázové systémy. Pro jednoduchost jsem se tedy rozhodl při testování využít H2 [27] databázi. Ta není tak úplně databázovým systémem. Jedná se o knihovnu napsanou v jazyce Java [28] poskytující JDBC rozhraní. Dokáže ukládat data jak dočasně v operační paměti, tak trvale do souboru na disku. V (3) poté popisuji možnosti a způsoby použití jiných relačních databází.

Jak již bylo řečeno výše, pro správné fungování výsledné služby je třeba zvolit formu ukládaných dat. Obrázek (2.2) ukazuje můj výsledný relační databázový model. Jeho struktura je poměrně jednoduchá, jelikož data potřebná k běhu algoritmu a sestavení výstupu nejsou nijak komplikovaná. Základní kostra tohoto modelu vznikla ještě před začátkem implementace. V průběhu byl ovšem několikrát upravován pro řešení konkrétních situací.

Ve zbytku této sekce stručně popisuji jednotlivé tabulky a jejich sloupce. Explicitně zde neuvádím většinu primárních klíčů (sloupce id). Ty slouží pouze pro účely tvorby vazeb mezi tabulkami a nejsou v aplikaci jinak využívány. Primární a cizí klíče jsou na diagramu zdůrazněny písmeny **P** a **F** v tomto pořadí před jejich názvem. Jeden sloupec může zastávat obě role, v tom případě je označen oběma písmeny.

- **LocationType** – Tabulka sloužící jako číselník typů zastávek. Formát GTFS ve své specifikaci klasifikuje zastávky dle jejich typu.
  - *locationTypeId* – id lokace přesně odpovídající referenci GTFS (1.4)
  - *name* – název této lokace
- **Stop** – Reprezentuje zastávky v síti veřejné dopravy.
  - *stopId* – id zastávky extrahované z GTFS dat
  - *name* – název zastávky
  - *latitude* – zeměpisná šířka umístění této zastávky
  - *longitude* – zeměpisná délky umístění této zastávky
  - *wheelChairBoarding* – příznak, zda zastávka má rampu pro vozíčkáře; Může nabývat hodnoty 0, 1 nebo 2. Ty znamenají neznámo, ano a ne v tomto pořadí.
  - *locationTypeId* – cizí klíč typu zastávky
- **RouteType** – Slouží jako číselník typů tras. V GTFS je každé trase přiřazen její typ specifikující dopravní prostředek, jaký po ní jezdí.
  - *routeTypeId* – id typu trasy přesně odpovídající referenci GTFS (1.4)
  - *name* – název této lokace

- **Route** – Obsahuje trasy v síti veřejné dopravy. Trasa pod sebou sdružuje konkrétní linky. Jako příklad bych uvedl trasu metro C (Letňany - Ládví - Háje).
  * *routeId* – id trasy extrahované z GTFS dat
  * *shortName* – krátký název této trasy; Pokud vycházím z příkladu výše, zde bude obsažena hodnota „C“.
  * *longName* – dlouhý název této trasy; Z příkladu výše zde bude hodnota „Letňany - Ládví - Háje“
  * *routeTypeId* – cizí klíč typu této trasy; Tedy dopravního prostředku fungujícího na této trase.
- **ServiceDay** – V této tabulce jsou uloženy provozní dny. Tedy konkrétní dny, kdy které linky jezdí. Ty jsou sdruženy pod tzv. službami.
  * *serviceId* – id služby extrahované z GTFS souboru.
  * *serviceIdInt* – číselný ekvivalent k *serviceId*; Tedy každá hodnota z *serviceId* má přiřazené právě jedno celé číslo. Tento sloupec existuje z důvodu optimalizace některých operací při běhu samotného algoritmu.
  * *serviceDay* – den ke kterému se vztahuje daný záznam
  * *willGo* – příznak, zda linka v tento den jede nebo ne
- **Trip** – Obsahuje informace o konkrétních linkách veřejné dopravy. Jako příklad bych zde uvedl vlak R705 směr České Budějovice.
  * *tripId* – id linky extrahované z GTFS dat.
  * *tripHeadSign* – text objevující se na značce na dopravním prostředku této linky; Slouží k identifikaci cíle cesty tohoto spoje. Z příkladu výše by zde byla hodnota „České Budějovice“.
  * *tripShortName* – krátký identifikátor této linky; Z příkladu výše by zde byla hodnota „R705“.
  * *wheelChairAccessible* – příznak, zda má tato linka místo pro invalidní vozík
  * *bikesAllowed* – příznak, zda má tato linka místo pro jízdní kolo
  * *routeId* – cizí klíč trasy
  * *serviceId* – číselné id služby, pod kterou tato linka spadá; Neslouží jako cizí klíč, tedy se neodkazuje do tabulky **ServiceDay**. Tento sloupec slouží k optimalizaci při výběru linek z databáze.
- **ServiceDayTripRel** – Spojující tabulka pro propojení linek s jejich provozními dny.
  * *tripId* – cizí klíč linky

  * *serviceDayId* – cizí klíč provozního dne

- **FootPath** – Reprezentuje pěší cesty mezi zastávkami. Ty jsou důležité pro započítání přestupů.
  * *duration* – délka chůze mezi zastávkami v minutách
  * *departureStopId* – cizí klíč počáteční zastávky této cesty
  * *arrivalStopId* – cizí klíč cílové zastávky této cesty
- **TripConnection** – Obsahuje přímá spojení mezi zastávkami. Struktura této tabulky přesně odpovídá spojení zadefinovaném v (1.2.3).
  * *departureTime* – čas odjezdu z počáteční zastávky
  * *arrivalTime* – čas příjezdu do cílové zastávky
  * *tripId* – cizí klíč linky tohoto spojení
  * *departureStopId* – cizí klíč počáteční zastávky tohoto spojení
  * *arrivalStopId* – cizí klíč cílové zastávky tohoto spojení

2.2.2 Transformace dat

Z popisu databázového modelu výše vyplývá, že nezbytné transformace dat nebudou nijak komplikované. Většina tabulek se totiž svou strukturou velmi podobá konkrétním souborům z GTFS formátu. Jedinou výjimkou je tabulka TripConnection, jejíž formát je odvozen od datové struktury zvoleného algoritmu.

Pro její plnění využiji soubor stop_times.txt, který je povinný v GTFS datech. Podrobněji ho popisuji v [21]. Ten obsahuje časy zastavení konkrétních linek na jejich zastávkách. Pro účely ukázky zde uvedu některé jeho parametry potřebné pro danou transformaci:

- *trip_id* – id linky zastavující na dané zastávce
- *arrival_time* – čas příjezdu do zastávky
- *departure_time* – čas odjezdu ze zastávky
- *stop_id* – id zastávky
- *stop_sequence* – pořadí, ve kterém je daná zastávka navštívena

Z jednoduchého pozorování vyplývá, že pro získání jednoho přímého spojení mezi dvěma zastávkami potřebuji dva po sobě jdoucí záznamy. Data ze stop_times.txt si tedy seřadím nejprve dle trip_id a poté dle stop_sequence vzestupně. Výsledná posloupnost poté obsahuje bloky záznamů seskupené dle jejich linek a seřazené od prvního zastavení po poslední.

Mějme tedy záznamy D a A pro které platí $D_{trip\_id} = A_{trip\_id} \wedge D_{stop\_sequence} < A_{stop\_sequence}$. Z toho dle popisu v [21] plyne i $D_{departure\_time} < A_{arrival\_time}$. Výsledné přímé spojení c lze poté popsat jako uspořádanou pětici $(D_{stop\_id}, A_{stop\_id}, D_{departure\_time}, A_{arrival\_time}, A_{trip\_id})$.

2.3 Výstupní formát

Jak jsem již avizoval v (1.4), jako výstupní formát jsem se rozhodl využít JSON. Jeho strukturu jsem navrhl tak, aby reprezentovala jednoduchou posloupnost spojů a přestupů. Nejprve si zadefinuji jeho dílčí části a ty poté spojím do výsledné formy. K tomu používám programovací jazyk TypeScript [29], který rozšiřuje jazyk JavaScript o statické datové typy. Z toho důvodu ho považuji za dobrý způsob, jak tento výstup popsat.

Ukázka kódu (1) vyobrazuje výstupní formát pro reprezentaci zastávky ve veřejné dopravě. Jeho obsah je strohý, protože je potřeba pouze pro identifikaci místa pro nástup/výstup do/z dopravního prostředku.

```
interface Stop {
  name: string;
  stopId: string;
}
```

Výpis kódu 1 DTO výstupu zastávky ve veřejné dopravě

Ukázka (2) popisuje výstupní formát pro trasu veřejné dopravy. Obsahuje jak její textovou identifikaci pro cestující, tak její typ. Ten slouží k určení druhu dopravního prostředku, pro který je tato trasa určena.

```
interface Route {
  routeId: string;
  shortName: string;
  longName: string;
  routeType: {
      routeTypeId: number;
      name: string;
    };
}
```

Výpis kódu 2 DTO výstupu trasy ve veřejné dopravě

Ukázka (3) následně obsahuje výstupní formát pro spoj veřejné dopravy. Slouží

k určení konkrétní linky, kterou musí cestující jet. Má v sobě jak její textovou identifikaci, tak i objekt reprezentující trasu, pod kterou spadá.

```
interface Trip {
  tripId: string;
  route: Route;
  tripHeadSign: string;
  tripShortName: string;
}
```

Výpis kódu 3 DTO výstupu linky ve veřejné dopravě

V neposlední řadě ukázky (4) a (5) popisují výstupní struktury, které tvoří dílčí části výsledné cesty. Obě obsahují parametr *name*, který slouží pro jejich rozlišení.

TripConnection reprezentuje cestu v dopravním prostředku mezi dvěma zastávkami. Říká „nastup v zastávce *departureStop* v čas *departureStop* do spoje *trip* a vystup v *arrivalStop* v čase *arrivalTime*“. Tyto zastávky nemusí být hned za sebou.

FootConnection popisuje pěší cestu, neboli přestup mezi zastávkami. Lze ho interpretovat jako „cesta ze zastávky *departureStop* do zastávky *arrivalStop* trvá *durationInMinutes* minut“.

```
interface TripConnection {
  trip: Trip;
  departureTime: string;
  arrivalTime: string;
  name: string;
  departureStop: Stop;
  arrivalStop: Stop;
}
```

Výpis kódu 4 DTO výstupu cesty dopravním prostředkem ve veřejné dopravě

```
interface FootConnection {
  durationInMinutes: number;
  name: string;
  departureStop: Stop;
  arrivalStop: Stop;
}
```

Výpis kódu 5 DTO výstupu pro přestup ve veřejné dopravě

Nakonec v ukázce (6) spojuji předchozí části do výsledné cesty. Mým primárním cílem byla jednoduchost a čitelnost výstupu. Jeho struktura je tedy jednoduchý seznam, který obsahuje střídající se *TripConnection* a *FootConnection*.

```
interface Path{
  connections: Array<TripConnection | FootConnection>
}
```

Výpis kódu 6 DTO výstupu pro výslednou cestu

Kapitola 3
Realizace

V této kapitole popisuji, jak jsem postupoval při tvorbě této práce. Dále zde zdůvodňuji volbu mnou použitých technologií. Nakonec popíši některé problémy, na které jsem narazil při implementaci a mé způsoby jejich řešení.

Vývoj této práce probíhal klasickým vodopádovým modele. V rámci realizace jsem ale se často vracel a upravoval části z návrhu architektury. Příkladem může být databázový model (2.2.1) nebo výstupní formát (2.3). Ke těm jsem se vracel při řešení konkrétních problémů, při implementaci.

Pro účely ukázky jsem vytvořil také webovou aplikaci v jazyce JavaScript. Ta slouží k interakci s uživatelem a umožňuje mu hledat trasy ve veřejné dopravě. Nabízí možnosti filtrování těchto tras dle případů užití v (1.6.2). Byla také využita při vývoji výsledné backendové služby. Používal jsem ji pro aktualizaci dat o veřejné dopravě a pro rychlé manuální testování změn. V následující sekcích referuji na tuto webovou aplikaci jako na „frontend“

3.1 Backend

Začal jsem tedy s implementací backendové webové služby. V tento moment jsem ještě neuvažoval o tvorbě frontendu. K jejímu testování jsem nejprve používal aplikaci Postman [30].

3.1.1 Volba technologií

Na výběr jsem měl několik technologií, pomocí kterých lze tvořit webové služby. Díky tomu, že Kotlin je interoperabilní s Javou, jsem měl možnosti využití i Java technologií. Rozhodoval jsem se tedy mezi frameworky Ktor [31], Micronaut [32] a Spring [33].

Ktor je framework pro jazyk Kotlin, který je založen na asynchronním přístupu zpracování požadavků. Slouží k tvorbě webových služeb a klientských aplikací. Programuje se v něm explicitním způsobem, na rozdíl od zbylých dvou variant.

Micronaut je framework určený pro programovací jazyky běžící v JVM. Jeho hlavní cíl je tvorba modulárních mikroslužeb a serverless aplikací. Stejně jako Ktor je postaven na asynchronním přístupu. K tomu využívá Netty webový server [34]. K deklaraci svých komponent používá anotace. Nevyužívá ovšem reflexi, jako tomu je u jiných JVM frameworků. Tyto anotace analyzuje a zpracovává při kompilaci kódu. Je tzv. full-stack, obsahuje v sobě tedy knihovny pro práci s databází, konfiguraci zabezpečení nebo DI.

Spring je ekosystém projektů, které dohromady slouží k vytvoření komplexních aplikací. Jeho hlavním zaměřením je tvorba webových a cloudových služeb. Při jejich tvorbě lze využít jakoukoliv kombinaci těchto modulů. Pokud chce komunikovat s databází, může do své aplikace přidat modul Spring Data. Pokud potřebuje zabezpečit svou služby, existuje projekt Spring Security. Takto modulární design dává vývojáři možnost začlenit novou funkcionalitu do své aplikace v moment, kdy ji potřebuje. Pro deklarace svých komponent Spring využívá anotace. Na rozdíl od Micronautu ale silně staví na reflexi.

Pro implementaci této práce jsem se nakonec rozhodl použít Spring Boot. Jedná se o projekt v rámci Springu, který již obsahuje základní konfiguraci pro svůj běh. Mám s ním největší zkušenosti a jeho moduly mi ve výsledku velmi ulehčili výslednou implementaci. Jako příklad uvedu práci s databází, kde jsem využil modul Spring Data JPA.

3.1.2 Architektura kódu

Rozhodl jsem se použít klasickou třívrstvou architekturu, tedy datovou, aplikační a prezentační vrstvu. Jedná se o standardní architekturu pro servery.

Spring Boot staví na tomto dělení do vrstev, které mezi sebou komunikují. K jejich deklaraci lze použít poskytnuté anotace. Jako příklad zde uvedu anotace `@Service` a `@RestController`.

3.1.3 Datová vrstva

Jak jsem zmiňoval výše, pro práci s databází jsem se rozhodl využít projekt Spring Data, konkrétně jeho modul Spring Data JPA [35]. Ten funguje jako abstrakce nad konkrétními implementacemi JPA. Pro tuto práci jsem zvolil framework Hibernate [36]. Dohromady poskytují možnosti ORM, tedy mapování databázových tabulek na entity v kódu.

V ukázce kódu (7) je poté vidět způsob tohoto mapování. Pro reprezentaci entit používám obyčejné datové třídy. Ty jsou popsány pomocí anotací tak, aby odpovídali konkrétním tabulkám. Typicky ORM pracují tak, že databázové vazby modelují jako reference na jejich entity. Ty poté načítá z databáze v moment, kdy jsou potřeba. Z toho důvodu musí všechny operace nad danou entitou probíhat v rámci jedná transakce. Tomuto omezení jsem se chtěl vyhnout, a tak jsem tyto

reference nahradil jejich ID. Vzhledem k tomu, že v databázi to jsou obyčejné sloupce si s tímto mapováním Hibernate dokáže poradit.

```
@Entity
@Table(name = "tripConnection")
data class TripConnection(
    val departureStopId: Int,
    val arrivalStopId: Int,
    @Column(name = "departureTime") val departureTimeDb: Int,
    @Column(name = "arrivalTime") val arrivalTimeDb: Int,
    val tripId: Int,
    @Id @Column(name = "id") val tripConnectionId: Int = 0
)
```

Výpis kódu 7 Databázová entita

Ještě je třeba popsat způsob, jakým jsem vytvořil tabulky v databázi. Jelikož jsem nevyužil ORM typickým způsobem, aplikace není schopna tyto tabulky vytvořit sama. Z toho důvodu jsem použil klasický sql soubor obsahující všechny potřebné DDL příkazy. O jeho spuštění se stará nástroj Flyway [37]. Ten slouží k automatickému spouštění databázových migrací. Dokáže inkrementálně upravovat schéma cílové databáze a díky tomu verzovat její změny. Tuto funkci jsem ovšem nevyužil a pro mé potřeby jsem vytvořil pouze finální verzi.

Pro komunikaci s databází jsem použil JpaRepository, které lze vidět v ukázce kódu (8). Jedná se o rozhraní, které poskytuje definice CRUD operací, jako jsou `save()`, `delete()` nebo `find()`. Jako parametr přijímá entitu, nad kterou pracuje a typ jejího ID. Spring poté sám vygeneruje konkrétní implementaci tohoto rozhraní a všech jeho metod. Ta je následně použita při běhu aplikace v místech, kde je její rozhraní použito ve spojení s DI.

Spring Data JPA dokáže generovat i těla metod, které nejsou v základním rozhraní
`JpaRepository`. Ukázka níže obsahuje metodu pro výběr zastávky podle sloupce, který není jejím ID. Název této metody samozřejmě musí odpovídat daným pravidlům, která jsou k dočtení v [38].

```
interface StopJpaRepository : JpaRepository<Stop, Int> {
    fun findByName(name: String): Stop?;
}
```

Výpis kódu 8 JPA Repository

Tuto vrstvu jsem rozdělil v podstatě na dvě. První z nich jsou již zmíněná rozhraní

`JpaRepository<>`. Ty ale v aplikační vrstvě nepoužívám přímo. K tomuto účelu jsem vytvořil vlastní repozitáře, tzv. repository, které jejich funkce obalují. Ukázka kódu (9) obsahuje příklad takové třídy. Každá entita má tedy přiřazený vlastní JpaRepository a jeho obalující repository. Díky tomuto přístupu lze kdykoliv změnit způsob komunikace s databází s minimálním zásahem do zbytku aplikace.

```
@Repository
class TripConnectionRepository() : GeneralRepository<TripConnection,
↪  Int, TripConnectionJpaRepository>() {
}
```

Výpis kódu 9 Repository obalující JpaRepository

Vzniklé třídy ovšem obsahují velké množství identického kódu pro CRUD operace. Z toho důvodu jsem pro jejich implementaci využil generického předka. Jeho část je vidět v ukázce kódu (10). Ten přijímá jako parametr entitu se kterou pracuje, typ jejího ID a JpaRepository. Jeho metody lze navíc překrýt implementací z jeho potomků v případě, kdy je to potřeba.

```
abstract class GeneralRepository<Entity : Any, Id : Any, Repository :
↪  JpaRepository<Entity, Id>>() {

    @Autowired
    protected lateinit var jpaRepository: Repository

    open fun save(entity: Entity): Entity {
        return jpaRepository.save(entity)
    }

    open fun findById(id: Id): Entity? {
        return jpaRepository.findByIdOrNull(id)
    }
}
```

Výpis kódu 10 Část generického repository

3.1.4 Aplikační vrstva

Aplikační vrstva obsahuje funkcionalitu pro hledání trasy a pro zpracování vstupních dat. K deklaraci komponent z této vrstvy Spring používá anotaci `@Service`. Tato vrstva poskytuje funkce prezentační vrstvě a využívá datovou vrstvu pro ukládání a načítání dat.

Existují zde třídy sloužící pouze jako prostředníci mezi prezentační a datovou vrstvou, tzv. service. Poskytují tedy obyčejnou CRUD funkcionalitu. Ukázka kódu (11) zobrazuje takovou třídu. Opět zde pro každou entitu existuje samostatná service.

```
@Service
class TripConnectionsService() : GeneralService<TripConnection, Int,
↪  TripConnectionRepository>() {
```

Výpis kódu 11 Service pro přístup k datové vrstvě

Stejně jako v datové vrstvě by i zde tedy existovalo mnoho duplikátního kódu. Z toho důvodu jsem aplikoval stejný postup, jako pro repozitáře výše. Vytvořil jsem generického předka, který přijímá jako parametry entitu, typ jejího ID a repository. Jeho část lze vidět v ukázce kódu (12).

```
abstract class GeneralService<Entity : Any, Id : Any, Repository :
↪  GeneralRepository<Entity, Id, out JpaRepository<Entity, Id>>>() {

    @Autowired
    protected lateinit var repository: Repository;

    @Transactional
    open fun save(connection: Entity): Entity {
        return repository.save(connection)
    }

    @Transactional
    open fun findAll(): List<Entity> {
        return repository.findAll()
    }
}
```

Výpis kódu 12 Část generické service

Toto dělení se může zdát neefektivní, poskytuje ale velkou kontrolu nad prací s daty. Pokud se změní nějaká logika pro vkládání dat, tak ji lze jednoduše upravit v dané service. Takový zásah poté bude mít minimální vliv na zbytek aplikace.

3.1.4.1 Algoritmus a filtrování

Implementace zvoleného algoritmu se skládá ze dvou kroků. První z nich je výpočet profilů, které obsahují cestu ze dané zastávky do cíle cesty. V tomto kroku se nejprve zavolá základní varianta CSA. Jeho výstup je seznam spojů, pomocí

kterých se lze dopravit do cílové zastávky. Tato operace slouží k optimalizaci běhu pCSA.

Algoritmus na svém vstupu očekává seznam spojení seřazený dle času odjezdu sestupně. K tomuto účelu jsem se rozhodl využít Kotlin sekvence [39]. Ty fungují stejně jako jiné kolekce, přes které lze iterovat. Nabízí ovšem možnost aplikovat filtry na jejich prvky v moment, kdy jsou procházeny. Díky tomu lze tedy jednoduše omezit procházený seznam spojení bez vytváření nové kolekce. Tento přístup jsem využil pro filtrování spojení, jak je popsáno v (2.1.1). Příklad použití je vidět v ukázce kódu (13).

```
sequence.filter { route ->
    usesAllowedVehicle(route)
}
```

Výpis kódu 13 Příklad filtrování sekvence

Druhý krok je z těchto profilů extrahovat výslednou nejrychlejší cestu. Následující pseudokód popisuje tento proces. K jeho pochopení je třeba popis datové struktury *profilu* z (1.2.4).

```
s = počáteční zastávka
r = čas odjezdu
výstup = prázdný seznam

opakuj, dokud není dosažena koncová zastávka:
    p = první profil zastávky s, jehož odjezd >= r
    pokud takový profil neexistuje, pak
        skonči a hledaná cesta neexistuje

    d = pěší cesta ze zastávky s do cílové zastávky

    pokud je délka d kratší než délka cesty pomocí p, pak
        přidej d do výstupu
        skonči

    přidej cestu obsaženou v p do výstupu
    s = koncová zastávka p
    r = čas výstupu v koncové zastávce p

vrať výstup
```

3.1.5 Prezentační vrstva

Prezentační vrstva se skládá z jediného řadiče (controlleru), který vystavuje rozhraní pro komunikaci s touto službou. K jeho deklaraci jsem použil Spring anotaci `@RestController`. Ta na rozdíl od anotace `@Controller` pouze zjednodušuje výslednou implementaci. Konkrétní HTTP endpointy jsou poté reprezentovány pomocí metod. Jak je ve Springu zvykem, tyto metody musí mít anotace popisující jejich vlastnosti. V ukázce kódu (14) lze vidět příklad vytvoření controlleru a konkrétního endpointu. Zde jsem použil speciální anotaci `@GetMapping`, která přímo specifikuje HTTP metodu GET.

```
@RestController
class Controller() {

    @GetMapping("/path/json")
    fun path(): ResponseEntity<Path> {
        return ResponseEntity.ok(findPath())
    }
}
```

Výpis kódu 14 Příklad controlleru v prezentační vrstvě

3.1.5.1 Formát rozhraní

Ke komunikaci s touto službou jsem se rozhodl použít HTTP protokol. Je třeba tedy ještě popsat konkrétní formát požadavků, které zpracovává.

/load Je endpoint, který slouží k přijetí nových GTFS dat. Požaduje následující parametry:

- file - Zip soubor, který obsahuje nová data.
- password - Heslo potřebné pro spuštění aktualizace. Lze ho konfigurovat na straně služby.

/path/json Je endpoint pro získání výsledné cesty. Přijímá následující parametry:

- departureStopId - GTFS ID počáteční zastávky hledané cesty.
- arrivalStopId - GTFS ID cílové zastávky hledané cestu.

- departureTime - Nepovinný parametr, který určuje den a čas odjezdu. Pokud není přítomný je použit čas odeslání požadavku. Musí být v ISO 8601 formátu [40].
- bikesAllowed - Nepovinný parametr udávající, zda výsledná cesta musí započítat místo pro kolo. Povolené hodnoty jsou *true* a *false*.
- wheelChairsAllowed - Nepovinný parametr udávající, zda výsledná cesta mít bezbariérový přístup. Povolené hodnoty jsou *true* a *false*.
- vehiclesAllowed - Seznam identifikátorů povolených dopravních prostředků. Tyto hodnoty musí být odděleny čárkou.

/delay Je endpoint pro zadání zpoždění spoje. Jeho parametry jsou:

- tripId - GTFS ID spoje, u kterého vzniklo zpoždění.
- durationMinutes - Velikost tohoto zpoždění v minutách.
- dateString - Datum vzniklého zpoždění ve formátu ISO.

3.2 Frontend

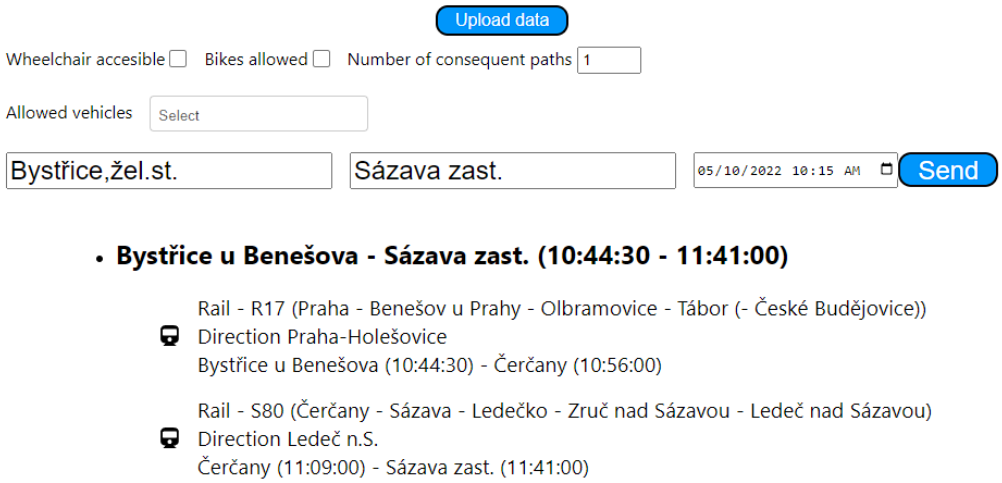
Zhruba v polovině vývoje backendu jsem se rozhodl vytvořit doprovodnou webovou aplikaci. Jedná se o malý projekt napsaný v jazyce JavaScript, který poskytuje možnosti interakce s výslednou službou. Lze s její pomocí otestovat všechny případy užití z pohledu uživatele a zároveň aktualizovat data na backendu. Při její tvorbě jsem si kladl za cíl, aby její spuštění a použití bylo co nejjednodušší.

Z toho důvodu jsem ji zakomponoval přímo do výsledné služby. Framework Spring totiž dokáže automaticky poskytovat statické soubory webových stránek. Tato aplikace tedy bude dostupná v moment, kdy je spuštěn backend. Uživatel poté musí pouze přejít na správnou URL pomocí prohlížeče a otevře se mu tato webová aplikace. Obrázek 3.1 ukazuje její výslednou formu.

3.2.1 Volba technologií

Pro implementaci jsem nechtěl použít čistý JavaScript. Při tvorbě komplexních webových aplikací může totiž být dle mého názoru velmi nepřehledný. Rozhodl jsem se tedy sáhnout po některém z mnoha dostupných frameworků pro tvorbu dynamických webových stránek. Chtěl jsem, aby běh této aplikace byl omezen čistě na prohlížeč uživatele. Tedy aby její spuštění nevyžadovalo samostatný server. Nakonec se má výsledná volba odehrála mezi frameworkem Vue.js [41] a knihovnou React.js [42].

Obě varianty nabízí možnosti tvorby dynamických webových aplikací. Vue.js staví na HTML šablonách k deklaraci komponent a jejich propojení s vnitřní logikou. React.js je na druhou stranu výrazně minimalističtější. Také využívá komponenty, ovšem k jejich reprezentaci používá třídy nebo obyčejné funkce.



Obrázek 3.1 Výsledná webová aplikace

Po jejich porovnání jsem se rozhodl použít React.js. Tato webová aplikace není nijak komplexní a z toho důvodu mi přišel přístup této knihovny jako správná volba.

K odesílání HTTP požadavků na backend jsem se rozhodl použít knihovnu Axios [43]. V tomto případě obaluje funkce zabudovaného HTTP klienta v jazyce JavaScript [44]. Z mého pohledu ale nabízí srozumitelnější rozhraní a způsob jeho používání.

3.2.2 Architektura kódu

Vzhledem k povaze této aplikace jsem se rozhodl použít dvouvrstvou architekturu. Tedy pouze datovou a prezentační vrstvu. Její primární účel je „překládat“ požadavky uživatele pro backend a zobrazovat výsledné cesty.

3.2.3 Datová vrstva

Datová vrstva se stará pro komunikaci s backendem. K tomu využívá protokol HTTP. Ukázka kódu (15) obsahuje příklad odeslání požadavku na server. Zde konkrétně se jedná o získání seznamu zastávek, jejichž název je podobný se zadaným textem.

3.2.4 Prezentační vrstva

Prezentační vrstva interaguje s uživatelem a umožňuje mu vyhledávat trasy ve veřejné dopravě. V tomto případě je tvořena z React.js komponentami. Každá taková komponenta reprezentuje jinou část webové aplikace. Dále si v sobě ukládají data

```
function loadStopsByName(name: string): Promise<Array<Stop>> {
    return axios.get('/stops?name=' + name, {}).then(response =>
    ↪  response.data)
}
```

Výpis kódu 15 Odeslání požadavku na server

potřebná pro jejich fungování. Ukázka kódu 16 obsahuje komponentu definovanou pomocí jednoduché funkce. Ta v tomto případě slouží k zobrazení jednotlivých částí výsledné cesty veřejnou dopravou.

```
function Path(props: PathViewProps) {
  return (
    {props.path.connections map((connection: Connection, index:
    ↪  number) => showConnection(connection, index))}
  );
}
```

Výpis kódu 16 React.js komponenta

3.3 Problémy při implementaci

Při tvorbě této práce jsem se setkal primárně se dvěma problémy. První z nich byla implementace samotného algoritmu. Po každé úpravě jsem jeho výstup porovnával s výstupem z Google Direction API a IDOSu. Po čase se ovšem ukázalo, že se mé výsledky více a více liší od výsledků těchto služeb. Tento fakt mě donutil několikrát upravit mou implementaci zvoleného algoritmu. Nakonec se ovšem ukázalo, že chyba se nacházela na straně dat, která jsou používal k testování. Konkrétně se jednalo o nepřesná data o pěších cestách mezi zastávkami. Tento problém jsem vyřešil úpravou způsobu mého testování. Více se tomuto problému věnuji v (4.1).

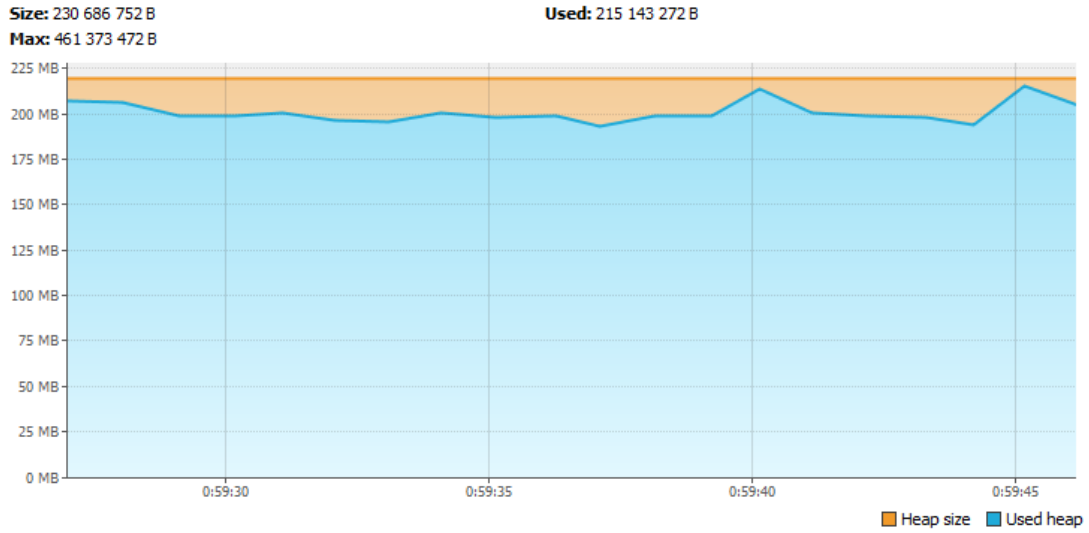
Druhý se týkal výpočetních zdrojů, které výsledná služba potřebovala ke svému běhu. Přesněji šlo o velikost operační paměti, kterou algoritmus potřeboval ke svému běhu. Tento problém je úzce spojený se splněním požadavku (1.6.1). Při mém testování jsem s použitím dat o Pražské integrované dopravě zaznamenal využití RAM okolo 500 MB pouze pro data. To při běhu na serveru nemusí být velký problém. Pokud ovšem uvažuji o použití tohoto algoritmu na mobilních zařízení jedná se o velkou překážku. K jeho běhu nejsou ale potřeba všechny informace o síti veřejné, které jsou uloženy v databázi. Například nepotřebuje data o názvech zastávek nebo linek. Tyto informace jsou nezbytné až pro sestavení výstupu z této služby.

Z tohoto důvodu jsem v rámci datové vrstvy vytvořil entity, které se mapují na již existující tabulky, ale obsahují pouze minimum údajů. Příklad je vidět v ukázce kódu 17. I když se tato entita mapuje na tabulku *Stop*, tak neobsahuje například její název. Takto vytvořené „minimální“ entity používám při běhu algoritmu. Data nezbytná k sestavení kompletního výstupu jsou poté načtena z databáze v moment, kdy jsou potřeba.

```
@Entity
@Table(name = "stop")
data class StopBase(
    @Id @Column(name = "id") val id: Int,
    val wheelChairBoarding: Int?
)
```

Výpis kódu 17 Příklad minimalizované entity pro běh algoritmu

Po této úpravě jsem zaznamenal výrazné snížení použité paměti RAM pro data. K běhu samotného algoritmu stačilo zhruba 225 MB, jak ukazuje obrázek (3.2). Při tomto testování jsem opět použil data o Pražské integrované dopravě.



Obrázek 3.2 Využití RAM při běhu algoritmu

Kapitola 4
Vyhodnocení

V této kapitole se věnuji vyhodnocení funkčnosti výsledné služby. Popisuji zde zvolený způsob jejího testování a interpretuji jeho výstup. Následně zhodnotím splnění cílů a požadavků na výsledek této práce. Nakonec navrhnu budoucí vylepšení pro tuto práci. Ta nejsou nezbytná pro její nasazení, vylepšují nebo rozšiřují ale její fungování.

4.1 Testování

Způsob testování výstupu této služby se ukázal více komplikovaný, než jsem předpokládal. Mým původním plánem bylo porovnávat mé výsledky s výsledky ze služby Google Directions API (1.1.1). K tomu jsem chtěl využít automatické integrační testy. Při jejich implementaci jsem ovšem narazil na několik problémů, které tento proces velmi omezují. Nejprve zde uvedu zdroj dat, která jsem použil při vývoji. Na to naváži popisem problémů, které se od nich odvíjí.

Pro účely testování jsem zvolil GTFS data o Pražské integrované dopravě [45]. Ta jsou dostupná zde [46] a jsou aktualizována každý den. Neobsahují ovšem délky přestupů mezi všemi zastávkami. Tyto údaje jsou nezbytné pro správné fungování mé služby. Neexistuje ale možnost jejich obsažení v datech o sítích veřejné dopravy.

Z tohoto důvodu jsem se rozhodl pro výpočet odhadů vzdáleností mezi všemi dostupnými zastávkami. K tomu jsem použil jejich zeměpisné polohy definované ve formátu GTFS (1.4). Tyto odhady nejsou ani zdaleka přesné. Umožňují ovšem vyhledání správných tras v případech, kdy jejich délka nehraje kritickou roli. Jako příklad uvedu přestupy v rámci jedné zastávky nebo přestupy mezi spoji s dostatečným časovým odstupem.

Provozovatel této služby může upravovat parametry pro výpočet těchto odhadů, nebo je úplně vyloučit. V tom případě musí tato data poskytnout v rámci souboru *pathways.txt* definovaný ve formátu GTFS. Platí pro ně ovšem omezení popsaná v (1.4).

Hledání tras ve veřejné dopravě je velmi citlivé na přesnost dat, která má daný

systém k dispozici. Rozdíl jedné minuty v délce přestupu může často úplně změnit výslednou cestu. To platí i při hledání pareto-optimální cesty dle nejrychlejšího času a počtu přestupů. Z těchto důvodů nelze předpokládat, že dvě služby pracující nad lehce rozdílnými daty najdou vždy stejnou cestu.

K výslednému otestování výstupů této služby jsem tedy použil kombinaci dvou způsobů. První z nich jsou automatické integrační testy. Jejich tvorba nebyla ovšem jednoduchá. Musel jsem pracovat s takovými trasami, kde se nepřesnosti mých dat projeví pouze minimálně. Z toho důvodu jsem se omezil primárně na hledání cest ve vlakové a autobusovou dopravě. Jejich jízdní řády jsou obvykle sestaveny s většími časovými odstupy mezi jednotlivými spoji. Navíc jsem zde nemohl porovnávat názvy zastávek přímo. Výstup z Directions API totiž neobsahuje interpunkci. Uchýlil jsem se tedy k porovnávání jejich zeměpisných poloh s přesností na několik desítek metrů.

Druhý způsob zahrnoval manuální testování pomocí několika nástrojů. Použil jsem webovou aplikaci z (3.2) k hledání cest pomocí všech dopravních prostředků. Tyto výsledky jsem následně porovnával s výstupem ze služby IDOS (1.1.2). V případech, kdy se výsledné cesty lišily jsem použil jízdní řády dostupné v [47] ke zjištění důvodu jejich rozdílu.

V průběhu tohoto testování jsem zjistil, že IDOS a Directions API často nemají aktuální informace o sítích veřejné dopravy. Při každém spuštění mých testů jsem použil nejnovější GTFS data ze stránek PID. Ta ovšem často obsahovala nové spoje, o kterých zmíněné služby „nevěděli“. Na druhou stranu ale data PID často neobsahují například výluky nebo zrušené spoje.

Z výše popsaných důvodů nelze jednoznačně říci, že výstup z této služby je správný. Pro přesné testování by bylo třeba použít identická data, jaká používají ostatní služby.

V rámci testování jsem se také zaměřil na rychlost výpočtů hledaný cest. K tomu jsem vytvořil experiment, při kterém jsem měřil čas potřebný k nalezení 100 cest mezi náhodnými zastávkami. V mém testovacím prostředí celý tento proces zabral zhruba 25 sekund. Tedy algoritmus potřeboval k nalezení jedné trasy průměrně 250 milisekund.

4.2 Splnění cílů

Z pozorování provedených výše jsem došel k závěru, že správné fungování mé práce plně závisí na datech, která jí jsou poskytnuta. Její výsledky byly stejné s porovnávanými službami v případech, kdy měli dostatečné podobná data. Ve zbylých případech byl pak její výstup ovlivněn primárně nepřesnými délkami přestupů.

Služba je schopna vyhledávat trasy mezi dvěma zastávkami veřejné dopravy. Tyto trasy jsou optimalizované jak podle nejrychlejšího času příjezdu, tak dle počtu přestupů. Umožňuje uživatelům zadávat parametry výsledné cesty, jako je použitý dopravní prostředek, bezbariérový přístup a místo pro jízdní kolo. Dále

z testování vyplývá, že výsledné cesty dokáže hledat velice rychle.

Dokáže přijmout data o veřejné dopravě ve formátu GTFS, zpracovat je a uložit do nastavené databáze Dokáže také změnit druh relační databáze, nad kterou pracuje bez zásahu do jejího kódu.

Služba dokáže do svého výstupu zakomponovat zpoždění některého spoje. Princip fungování této vlastnosti ovšem není finální. Požadavek na její existenci vznikl na popud aplikace Taransit (1.5). Ta ale zatím nedokáže zpracovávat data o zpožděních vlaků. Z toho plyne, že způsob práce se zpožděním v rámci této služby se bude muset přizpůsobit výslednému fungování aplikace Taransit.

I přes to je možné již nyní zadat zdržení některého spoje do použitých dat. To lze provést dvěma způsoby.

První z nich je započtení zpoždění spoje přímo do GTFS souborů poskytnutých službě. To ale není ideální, jelikož pro každou vzniklou změnu je třeba provést kompletní aktualizaci dat.

Druhý způsob jeho zadání je pomocí definovaného HTTP endpointu. Ten přijímá ID spoje, den kdy nastalo zpoždění a jeho velikost v minutách. Tato varianta je o poznání elegantnější. Stále ovšem slouží pouze k demonstraci možností výsledné služby.

4.3 Budoucí vylepšení

V této sekci navrhnu několik možných vylepšení a rozšíření mé práce. Výslednou službu lze plně používat v jejím současném stavu. Pro účely jejího komerčního nasazení jsem ale navrhl několik úprav, které by mohlo být vhodné přidat.

Optimalizace přestupů Služba již nyní optimalizuje výslednou cestu dle počtu přestupů. Můj přístup k tomuto problému popsaný v (2.1.2) ovšem trpí několika nedostatky. Pro budoucí použití by mohlo být užitečné upravit implementaci algoritmu tak, aby hledala pareto-optimální cestu dle počtu přestupů. V tom případě ale bude třeba vzít v potaz případné zpomalení jeho běhu.

Výstupní formát Při tvorbě výstupního formátu z mé služby jsem nebral v potaz jejich velikost při posílání po síti. V tomto případě se jedná hlavně o dvojitý výskyt některých zastávek. Ty figurují jak v přestupech, tak ve spojích veřejné dopravy. Pro účely komprese by mohlo být vhodné odkazovat na tyto zastávky pomocí ID do nějakého slovníku, který bude součástí výstupu.

Rozšíření filtrování Současné možnosti parametrizace hledané cesty staví na omezení vstupních dat. Uživatelé této služby mohou ale například požadovat nastavení maximální ceny jízdného. Tento typ úprav ovšem vyžaduje zásah do implementace použitého algoritmu.

Kapitola 5
Závěr

Cílem této práce bylo vytvořit backendovou službu, která bude poskytovat rozhraní pro výpočet tras pomocí veřejné dopravy. Hlavní požadavek byl kladen na její výstup. Nalezená cesta musela být nejrychlejší možná od zadaného času odjezdu.

Pro účely analýzy jsem otestoval tři existující služby, které poskytují stejné nebo podobné funkce. Při jejich používání jsem došel k závěru, že výstupy těchto řešení neodpovídají mé představě jednoduchého formátu popisující trasy ve veřejné dopravě. Tento fakt poté dále přispěl k formulaci výstupního formátu této práce.

V rámci analýzy jsem dále porovnal několik algoritmů, které lze použít k řešení tohoto problému. Po jejich důkladném prozkoumání jsem si pro účely mé implementace zvolil algoritmus pCSA. K této volbě přispěli primárně jeho rychlost a možnosti přizpůsobení.

Následně jsem při rozboru formátu GTFS identifikoval data nezbytná pro správný běh zvoleného algoritmu a sestavení výstupu mé služby. V tento moment jsem také odhalil fakt, že nemohu splnit celé zadání této práce. Konkrétně nelze splnit část, která určuje výstup z této služby a to z toho důvodu, že formát GTFS neslouží k popisu cest ve veřejné dopravě. Pro popis výsledných tras jsem tedy zvolil formát JSON s vlastní strukturou.

Pro komunikaci s touto službou jsem zvolil protokol HTTP. Pomocí něj je schopna přijímat i odesílat všechna potřebná data.

K otestování jsem zvolil kombinaci integračních a manuálních testů. Tuto volbu jsem učinil primárně kvůli citlivosti hledání tras ve veřejné dopravě na přesnost poskytnutých dat. Na základě výsledků těchto testů jsem poté zhodnotil, zda výsledná služba funguje dle požadavků. Ukázalo se však, že přesnost dat je větší problém, než jsem předpokládal. V případě, kdy obě porovnávané služby nemají identická data, tak není možné aby poskytovali stejné výsledky. Výstup mé služby jsem tedy testoval na několika speciálních trasách, kde tyto rozdíly byly zanedbatelné.

Výsledek této práce splňuje její zadání nejlépe, jak je to možné s ohledem na informace zjištěné v rámci analýzy této práce.

Bibliografie

1. VELKÉ BRITÁNIE A SEVERNÍHO IRSKA, Spojené království. *Transport for London Unified API* [online]. [B.r.] [cit. 2022-04-20]. Dostupné z: `https://api.tfl.gov.uk/`.
2. TRANSIT. *TransitApp* [online]. [B.r.] [cit. 2022-04-20]. Dostupné z: `https://transitapp.com/`.
3. GOOGLE. *Google Directions API*. 2022. Dostupné také z: `https://developers.google.com/maps/documentation/directions`.
4. GOOGLE. *Google Cloud platform*. 2022. Dostupné také z: `https://cloud.google.com/gcp`.
5. GOOGLE. *Client Libraries for Google Maps Web Services*. 2022. Dostupné také z: `https://developers.google.com/maps/documentation/directions/client-library`.
6. GOOGLE. *Google Maps Web Services*. 2022. Dostupné také z: `https://developers.google.com/maps/documentation`.
7. GOOGLE. *Google Transit APIs*. 2022. Dostupné také z: `https://developers.google.com/transit`.
8. CHAPS. *IDOS*. 2022. Dostupné také z: `https://idos.idnes.cz/pid/spojeni/`.
9. CHAPS. *Champs - Aktualizace* [online]. [B.r.] [cit. 2022-04-20]. Dostupné z: `https://www.chaps.cz/cs/updates`.
10. CHAPS. *IDOS - Řešení pro internet* [online]. [B.r.] [cit. 2022-04-20]. Dostupné z: `https://www.chaps.cz/cs/products/IDOS-internet`.
11. CHAPS. *IDOS Veřejné API* [online]. [B.r.] [cit. 2022-04-20]. Dostupné z: `https://www.chaps.cz/files/idos/IDOS-API.pdf`.
12. TRAVELTIME. *TravelTime*. 2022. Dostupné také z: `https://traveltime.com/`.

13. TRAVELTIME. *TravelTime Routes API Reference*. 2022. Dostupné také z: https://docs.traveltime.com/api/reference/routes.

14. TRAVELTIME. *TravelTime Analysis*. 2022. Dostupné také z: https://traveltime.com/analytics.

15. PYRGA, Evangelia; SCHULZ, Frank; WAGNER, Dorothea; ZAROLIAGIS, Christos. Efficient models for timetable information in public transportation systems. *ACM Journal of Experimental Algorithmics, v.12, 2.4.1-2.4.39 (2007)*. 2007, roč. 12. Dostupné z DOI: 10.1145/1227161.1227166.

16. DELLING, Daniel; PAJOR, Thomas; WERNECK, Renato. Round-Based Public Transit Routing. *Transportation Science*. 2012, roč. 49. Dostupné z DOI: 10.1287/trsc.2014.0534.

17. DIJKSTRA, E. A note on two problems in connexion with graphs Numerische Mathematik. *Journal of Computer and System Sciences - JCSS*. 1959, roč. v. 177.

18. STRASSER, Ben; DIBBELT, Julian; PAJOR, Thomas; WAGNER, Dorothea. Intriguingly Simple and Fast Transit Routing. In: 2013, sv. 7933. ISBN 978-3-642-38526-1. Dostupné z DOI: 10.1007/978-3-642-38527-8_6.

19. DIBBELT, Julian; PAJOR, Thomas; STRASSER, Ben; WAGNER, Dorothea. Connection Scan Algorithm. *Journal of Experimental Algorithmics*. 2017, roč. 23. Dostupné z DOI: 10.1145/3274661.

20. GOOGLE. *GTFS Static Overview* [online]. [B.r.] [cit. 2022-04-20]. Dostupné z: https://developers.google.com/transit/gtfs.

21. GOOGLE. *GTFS Reference* [online]. [B.r.] [cit. 2022-04-20]. Dostupné z: https://developers.google.com/transit/gtfs/reference.

22. TARANSIT. *Taransit* [online]. [B.r.] [cit. 2022-04-20]. Dostupné z: https://taransit.com/.

23. NPERF. *3G / 4G / 5G coverage map, India* [online]. [B.r.] [cit. 2022-04-20]. Dostupné z: https://www.nperf.com/en/map/IN/-/-/signal/.

24. TARANSIT. *Kolkata Sub & local train time* [online]. [B.r.] [cit. 2022-04-20]. Dostupné z: https://play.google.com/store/apps/details?id=com.taransit.transport.

25. RUNNINGSTATUS. *Running Status* [online]. [B.r.] [cit. 2022-04-20]. Dostupné z: https://runningstatus.in/history.

26. GOOGLE. *Overview of memory management* [online]. [B.r.] [cit. 2022-04-20]. Dostupné z: https://developer.android.com/topic/performance/memory.

27. THOMAS MUELLER. *H2 Database Engine*. 2022. Dostupné také z: https://www.h2database.com/.

28. ORACLE. *Java*. 2022. Dostupné také z: https://www.java.com/en/.
29. MICROSOFT. *TypeScript*. 2022. Dostupné také z: https://www.typescriptlang.org/.
30. POSTMAN, INC. *Postman*. 2022. Dostupné také z: https://www.postman.com/.
31. JETBRAINS. *Ktor Framework*. 2022. Dostupné také z: https://ktor.io/.
32. THE MICRONAUT FOUNDATION. *Micronaut Framework*. 2022. Dostupné také z: https://micronaut.io/.
33. VMWARE. *Spring Framework*. 2022. Dostupné také z: https://spring.io/.
34. THE NETTY PROJECT. *Netty*. 2022. Dostupné také z: https://netty.io/.
35. VMWARE. *Spring Data Jpa*. 2022. Dostupné také z: https://spring.io/projects/spring-data-jpa.
36. HIBERNATE. *Hibernate*. 2022. Dostupné také z: https://hibernate.org/.
37. RED GATE SOFTWARE LTD. *Flyway*. 2022. Dostupné také z: https://flywaydb.org/.
38. VMWARE. *Spring Data JPA - Reference Documentation*. 2022. Dostupné také z: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories.query-methods.query-creation.
39. JETBRAINS. *Kotlin Sequence*. 2022. Dostupné také z: https://kotlinlang.org/docs/sequences.html.
40. ISO. *ISO 8601 DATE AND TIME FORMAT* [online]. [B.r.] [cit. 2022-04-20]. Dostupné z: https://www.iso.org/iso-8601-date-and-time-format.html.
41. EVAN YOU. *Vue.js*. 2022. Dostupné také z: https://vuejs.org/.
42. META PLATFORMS, INC. *React.js*. 2022. Dostupné také z: https://reactjs.org/.
43. AXIOS. *Axios*. 2022. Dostupné také z: https://axios-http.com/docs/intro.
44. MOZILLA FOUNDATION. *Fetch API*. 2022. Dostupné také z: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API.
45. ROPID. *Pražská integrovaná doprava* [online]. [B.r.] [cit. 2022-04-20]. Dostupné z: https://pid.cz/.
46. ROPID. *Pražská integrovaná doprava GTFS* [online]. [B.r.] [cit. 2022-04-20]. Dostupné z: https://pid.cz/o-systemu/opendata/.
47. ROPID. *Jízdní řády linek PID* [online]. [B.r.] [cit. 2022-04-20]. Dostupné z: https://pid.cz/jizdni-rady-podle-linek/.

Obsah přiloženého média

- `readme.txt` ..... stručný popis obsahu média
- `exe` ..... adresář se spustitelnou formou implementace
- `src`
  - `RouteMyWay` ..... zdrojové kódy implementace
    - `Backend` ..... zdrojové kódy backendu
    - `Frontend` ..... zdrojové kódy frontendu
    - `README.md` ..... popis konfigurace a spuštění služby
  - `prace` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `prace.pdf` ..... text práce ve formátu PDF