České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra počítačů

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: **Jakub Chalupa**

Studijní program: Softwarové technologie a management
Obor: Softwarové inženýrství

Název tématu: **Využití grafových databází pro vyhledávání spojů veřejné dopravy**

Pokyny pro vypracování:

V současné době figuruje na českém trhu prakticky pouze jeden centrální vyhledávač spojů veřejné dopravy, který si ale vlastní algoritmus vyhledávání pečlivě střeží. Cílem práce je analyzovat problematiku vyhledávání spojů veřejné dopravy a implementovat co nejefektivnější vyhledávaci algoritmus. Pro tyto účely práce porovná alespoň dva různé přístupy pro ukládání dat a následné vyhledávání spojů z místa A do místa B. Zaměří se zejména na využití grafových databází, které se svou podstatou pro vyhledáváni spojů hodí lépe, než tradiční relační databáze. Výstupem práce bude porovnání obou přístupů z hlediska vhodnosti a výkonnosti a funkční aplikace, která bude vlastní vyhledávání spojů umožňovat.

Seznam odborné literatury:

Sherif Sakr - Eric Pardede: Graph Data Management: Techniques and Applications.
Sherif Sakr - Mohamed Gaber: Large Scale and Big Data: Processing and Management.
Eric Redmond - Jim R. Wilson: Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement.

Vedoucí: doc. Irena Holubová RNDr., Ph.D.

Platnost zadání: do konce letního semestru 2015/2016

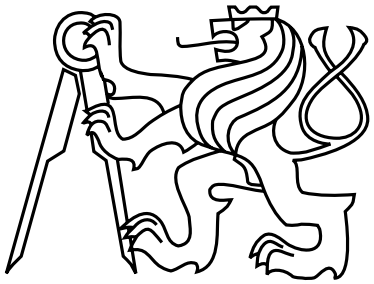
L.S.

doc. Ing. Filip Železný, Ph.D.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 24. 11. 2014

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Bakalářská práce

Využití grafových databází pro vyhledávání spojů veřejné dopravy

Jakub Chalupa

Vedoucí práce: doc. RNDr. Irena Holubová, Ph.D.

Studijní program: Softwarové technologie a management, Bakalářský

Obor: Softwarové inženýrství

19. května 2015

Poděkování

Rád bych tímto poděkoval vedoucí této práce doc. RNDr. Ireně Holubové, Ph.D. za vedení práce a poskytnuté cenné rady. Dále děkuji Jakubu Riedlovi za počáteční impulz pro studium grafových databází, Janu Plhákovi za detailní popis používaného vyhledávacího algoritmu v Biletu, Václavu Henzlovi za detailní popis vyhledávání používaného ve Student Agency a všem mým blízkým, kteří mě po celou dobu studia podporovali.

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.

V Nové Bystřici dne 17. 5. 2015 .....

Abstract

The aim of this work is an analysis of public transport connections and own implementation of a search algorithm. For this purpose the work compares two different ways of data storing and data retrieval. In the first part a detailed analysis of existing implementations of respective search algorithms can be found. Thereafter own search algorithm implementation over relational database PostgreSQL and graph database Neo4j is presented.
The main contribution of this work is the detailed analysis of the most widely used graph database and its possibilities for route planning. Another output of this work is both implementation and testing of own complex web application allowing connections searching.

Abstrakt

Práce se zabývá analýzou problematiky vyhledávání spojů veřejné dopravy a implementuje vlastní řešení vyhledávání. Pro tyto účely porovnává dva způsoby ukládání dat a následné vyhledávání spojů. V první části jsou detailně analyzovány současné implementace vyhledávačů. Poté se práce zaměřuje na implementaci vlastního vyhledávacího algoritmu nad relační databází PostgreSQL a grafovou databází Neo4j.
Hlavním přínosem práce je detailní analýza nejrozšířenější grafové databáze a možnosti jejího využití při plánování cest. Výstupem je i implementace a testování komplexní webové aplikace, která vlastní vyhledávání spojů umožňuje.

Obsah

1 Úvod **1**
- 1.1 Základní pojmy . . . 1
  - 1.1.1 Struktura jízdních řádů . . . 1
  - 1.1.2 Relační databáze . . . 1
  - 1.1.3 Graf . . . 2
    - 1.1.3.1 Traverzování grafem a optimalita cest . . . 2
  - 1.1.4 Grafová databáze . . . 3
- 1.2 Motivace . . . 3

2 Současný stav **5**
- 2.1 Existující vyhledávače spojů v ČR . . . 5
  - 2.1.1 IDOS . . . 5
    - 2.1.1.1 IDOS API . . . 6
    - 2.1.1.2 Další vyhledávače využívající IDOS . . . 6
  - 2.1.2 Jiné aplikace pro vyhledávání spojů s vlastní implementací vyhledávače 6
    - 2.1.2.1 Bileto . . . 7
    - 2.1.2.2 Student Agency / RegioJet . . . 7
    - 2.1.2.3 Smartřády . . . 7
  - 2.1.3 Shrnutí . . . 7

3 Technologické detaily existujících vyhledávačů v České republice **9**
- 3.1 Vyhledávač IDOS . . . 9
- 3.2 Vyhledávač Bileto . . . 9
- 3.3 Vyhledávač dopravce Student Agency / RegioJet . . . 10
- 3.4 Vyhledávač Smartřády . . . 10
- 3.5 Shrnutí . . . 11

4 Analýza vyhledávání spojů **13**
- 4.1 Uživatelské vstupy . . . 13
- 4.2 Parametry kvality vyhledávače . . . 14
  - 4.2.1 Parametry nalezených cest spojů . . . 14
  - 4.2.2 Optimalizace kvality nalezených výsledků . . . 15
  - 4.2.3 Souhrnné nároky . . . 15
- 4.3 Datová struktura . . . 15
  - 4.3.1 Specifika datové struktury . . . 16

4.3.2 Nároky na vyhledávač . . . 17

5 Vyhledávací algoritmus nad relační databází 19
5.1 Přídavný parametr vyhledávacího algoritmu . . . 19
5.2 Relační schéma . . . 20
5.3 Vyhledání spojů bez přestupu . . . 21
5.3.1 Nalezení počátečních zastavení . . . 21
5.3.2 Iterace po následujících zastaveních . . . 22
5.4 Vyhledávání spojů s přestupem . . . 23
5.4.1 Nástin postupu vyhledávání s přestupy . . . 24
5.5 Optimalizace vyhledávacího algoritmu . . . 25
5.5.1 Maximální počet přestupů . . . 25
5.5.2 Vícenásobné přestupy mezi dvěma jízdami . . . 25
5.5.3 Traverzování mezi stanicemi . . . 27
5.5.4 Výběr pareto-optimálních cest . . . 29
5.5.4.1 Identifikace pareto-optimálních cest . . . 29
5.5.4.2 Pořadí zpracování a jeho dopady . . . 31
5.5.4.3 Pareto-optimální cesty nad relační databází . . . 33

6 Vyhledávací algoritmus nad grafovou databází 35
6.1 Struktura grafu . . . 35
6.1.1 Time-dependent model . . . 35
6.1.2 Time-expanded model . . . 36
6.1.3 Srovnání modelů . . . 36
6.1.4 Zvolená struktura grafu . . . 37
6.1.4.1 Kontrola struktury grafu . . . 39
6.1.4.2 Komplexní schéma grafu . . . 42
6.2 Algoritmy pro vyhledávání nejkratších cest grafem . . . 44
6.3 Vyhledávání cest v grafu . . . 45
6.3.1 Nalezení počátečních zastavení . . . 45
6.3.2 Hledání cest od počátečních zastavení . . . 46
6.3.2.1 Kontroly na vrcholech . . . 46
6.3.2.2 Postup traverzování . . . 48
6.4 Optimalizace vyhledávání cest v grafu . . . 50
6.4.1 Maximální počet přestupů . . . 50
6.4.2 Vícenásobné přestupy mezi dvěma jízdami . . . 50
6.4.3 Traverzování mezi shodnými stanicemi . . . 51

7 Implementace 53
7.1 Zvolené technologie . . . 53
7.1.1 Relační databáze PostgreSQL . . . 53
7.1.2 Grafová databáze Neo4j . . . 53
7.1.2.1 Dotazování nad Neo4j . . . 54
7.1.3 Technologie pro webovou aplikaci . . . 55
7.1.3.1 Struktura webové aplikace . . . 55
7.1.3.2 Aplikační server . . . 56

7.2 Implementace vyhledávacího algoritmu nad relační databází 57
7.3 Implementace vyhledávacího algoritmu nad grafovou databází 58
7.3.1 Výběr výchozích vrcholů 59
7.4 Uživatelské rozhraní 60
7.4.1 Vyhledávací formulář 61
7.4.2 Výsledky vyhledávání 61
7.4.3 Administrační část 62

8 Testování **63**
8.1 Testovací data 63
8.1.1 Velikost testovacích dat 63
8.2 Testování rychlosti algoritmů 64
8.2.1 Testovací prostředí 64
8.2.2 Rychlost algoritmu nad relační databází 64
8.2.3 Rychlost algoritmu nad grafovou databází 66
8.2.3.1 Grafické znázornění rychlostí 68
8.2.4 Srhnutí testů rychlosti 70
8.3 Testování správnosti výsledků 71
8.3.1 Testování platnosti všech zastavení 71
8.3.2 Testování kvality výsledků 71

9 Závěr **73**
9.1 Současný stav 73
9.2 Návrh algoritmů pro vyhledávání v jízdních řádech 73
9.3 Implementace 74
9.4 Testování 74
9.5 Budoucí možnosti 74

A Obsah přiloženého CD **81**

B GTFS formát dat **83**

Seznam obrázků

1.1 Orientovaný graf 2

5.1 Schéma GTFS formátu dat 20
5.2 Jízdy a jejich zastávky s výjezdem ze stanice A 21
5.3 Jízdy a jejich zastávky s výjezdem ze stanice A a s přestupní stanicí X 23
5.4 Protisměrné jízdy a jejich přestupy 26
5.5 Stejnosměrné jízdy a jejich přestupy 26
5.6 Traverzování mezi shodnými stanicemi 27
5.7 Traverzování mezi shodnými stanicemi – všechny přestupy 27
5.8 Zastavení ve stanicích a délka cest 29
5.9 Zastavení ve stanicích a shodné délky cest 29
5.10 Jízdy ze stanice A s časem zastavení 32

6.1 Time-expanded (vlevo) a time-dependent model grafu se třemi stanicemi A, B, C. Tři jízdy spojují stanici A se stanicí B (jízdy u, v, w), jedna jízda spojuje stanici B a A (y) a dvě jízdy spojují stanici C a B (x, z). Zdroj: [32] 36
6.2 Základní struktura grafu 37
6.3 Struktura grafu v rámci jedné stanice. Čísla ve vrcholech udávají hodnotu $AbsC_z$. 39
6.4 Obousměrná hrana mezi vrcholy 40
6.5 Zastavení v rámci jedné stanice. Čísla u vrcholů značí C_p do zastavení (vlevo) a C_v ze zastavení. Pro všechny C_v nyní platí $C_v = AbsC_z$. 40
6.6 Zastavení v rámci jedné stanice s obousměrnou hranou mezi vrcholy 41
6.7 Zastavení v rámci jedné stanice s obousměrnými hranami mezi vrcholy 42
6.8 Struktura grafu s přidanými vrcholy jízd (vrcholy J) a intervaly platnosti (vrcholy I). Hrana znázorněná čárkovaně značí příslušnost zastavení k jízdě. Hrana znázorněná tečkovaně značí příslušnost jízdy k intervalu platnosti. . . . 43
6.9 Ilustrační schéma jízdního řádu 46
6.10 Cesta grafem ze zastavení A_1 na jízdu J_1 a k intervalu platnosti I_1 47
6.11 Přestupní hrany v rámci cesty grafem 48

7.1 Standardní architektura Spring aplikace. Zdroj: [11] 56
7.2 Vyhledávací formulář 61
7.3 Tabulka s výsledky vyhledávání 61
7.4 Editační formulář jízdy 62

8.1 Rychlost vyhledávání nad relační databází pro $P = 1$. Osa x reprezentuje čas vyhledávání v sekundách, osa y reprezentuje počet jízd v databázi. 69
8.2 Rychlost vyhledávání nad grafovou databází pro $P = 4$. Osa x reprezentuje čas vyhledávání v sekundách, osa y reprezentuje počet jízd v databázi. 69
8.3 Srovnání rychlostí vyhledávání nad relační (modře) a grafovou (červeně) databází pro $P = 1$. Osa x reprezentuje čas vyhledávání v sekundách, osa y reprezentuje počet jízd v databázích. 70

B.1 Soubor stops.txt z GTFS balíku dat 83
B.2 Soubor agency.txt z GTFS balíku dat 83
B.3 Soubor routes.txt z GTFS balíku dat 84
B.4 Soubor trips.txt z GTFS balíku dat 84
B.5 Soubor stop_times.txt z GTFS balíku dat 84
B.6 Soubor calendar.txt z GTFS balíku dat 84

Seznam tabulek

8.1 Rychlost algoritmu pro 2 252 jízd (relační databáze) 65
8.2 Rychlost algoritmu pro 9 017 jízd (relační databáze) 65
8.3 Rychlost algoritmu pro 22 578 jízd (relační databáze) 66
8.4 Rychlost algoritmu pro 2 252 jízd (grafová databáze) 66
8.5 Rychlost algoritmu pro 9 017 jízd (grafová databáze) 67
8.6 Rychlost algoritmu pro 22 578 jízd (grafová databáze) 67
8.7 Rychlost algoritmu pro 45 146 jízd (grafová databáze) 68

Kapitola 1

Úvod

Plánování cest prostředky hromadné dopravy urazilo během posledních let velký kus cesty a i v České republice tak existuje celá řada vyhledávačů, které plánování cest umožňují. Již dávno proto není nutné studovat papírový jízdní řád od jednotlivých dopravců, ale celou trasu je možné naplánovat prostým zadáním několika údajů. Vyhledávání spojů napříč různými dopravci a jejich jízdními řády ovšem není úkol zcela triviální. Na pozadí vyhledávačů běží složité algoritmy pro vyhledávání cest, které musí v krátkém čase zpracovat velké množství dat. V následujících kapitolách se této problematice budeme podrobně věnovat.

1.1 Základní pojmy

Protože nemusí být čtenář seznámen se všemy pojmy v práci používanými, ty nejdůležitější z nich si detailněji přiblížíme hned na úvod.

1.1.1 Struktura jízdních řádů

Jízdní řád v papírové podobě si jistě každý vybaví. Standardně se jedná o tabulku, která zachycuje příjezdy a odjezdy spojů (vlaků, autobusů...) z různých stanic. Stanice jsou zpravidla seřazené po sobě tak, že tvoří trasu spoje. Po této trase pak každý spoj jede a zastavuje postupně v jednotlivých stanicích v konkrétním čase. Trasa je tedy definována jako seřazená posloupnost stanic a jízda spoje je definována jako seřazená posloupnost zastavení spoje na těchto stanicích v daném čase.

Každý jízdní řád má danou platnost. Tato může být obecně velmi rozličná, často ale dopravci vydávají platné jízdní řády na jeden rok dopředu. Každý ze spojů tedy jede pouze ve dnech platnosti příslušného jízdního řádu. Platnost spoje je dále limitována na konkrétní dny v týdnu, ve který spoj jezdí. Často se proto setkáme se situací, kdy konkrétní spoj jede pouze o nedělích a svátcích. Tato omezení jsou vždy v jízdních řádech označena příslušnými poznámkami.

1.1.2 Relační databáze

Relační databáze je databáze, v níž jsou data strukturována do tabulek, které se skládají z řádků a sloupců. Některé ze sloupců přitom mohou uchovávat informace o relacích

(vztazích) mezi jednotlivými tabulkami.

Pro dotazování nad relační databází slouží jazyk SQL (Structured Query Language) [40]. Pomocí něho jsme schopni efektivně přistupovat k záznamům v databázi uloženým [34].

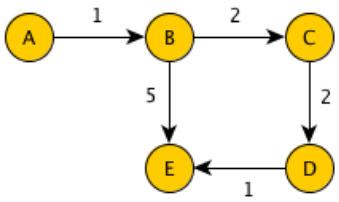
1.1.3 Graf

Graf je struktura definovaná pomocí množiny *vrcholů* a *hran*. Vrcholy mohou obecně reprezentovat libovolné entity a pomocí hran se dají zachytit vazby mezi nimi [17].

Každá hrana v grafu propojuje právě dva vrcholy. Hrana přitom může být určena i svou orientací. Může směřovat pouze z vrcholu A do vrcholu B (v rámci grafu můžeme procházet pouze z vrcholu A do vrcholu B), obráceně nebo i obousměrně. Pomocí orientace jednotlivých hran rozhodneme, jestli je i samotný graf orientovaný, nebo neorientovaný.

1.1.3.1 Traverzování grafem a optimalita cest

Cesta grafem z jednoho vrcholu do dalších pomocí existujících hran se nazývá traverzování. Traverzovat v orientovaném grafu můžeme pochopitelně jen ve směru, který nám jednotlivé hrany umožňují. Traverzování nám umožňuje postupně navštívit všechny existující vrcholy v grafu, způsob traverzování přitom závisí pouze na zvoleném algoritmu [29]. Jednoduchý orientovaný graf si můžeme prohlédnout na obrázku 1.1.



Obrázek 1.1: Orientovaný graf

Na obrázku 1.1 můžeme vidět dvě možné cesty z vrcholu A do vrcholu E. Nyní budeme uvažovat, že délka cesty v grafu z jednoho vrcholu do druhého je definována jako počet vrcholů, které navštívíme po této cestě. První z možných cest je A->B->E, která vede pouze přes jeden vrchol, zatímco cesta A->B->C->D->E vede přes tři vrcholy. Cesta A->B->E je proto *optimální* z hlediska své délky, protože v grafu neexistuje žádná jiná cesta spojující vrcholy A a E, která by byla kratší.

Nyní změníme definici délky cesty grafem. Délka cesty bude vyjádřena jako součet délek jednotlivých hran, které se na této cestě nalézají. Cesta A->B->E má tedy nyní délku 6 a cesta A->B->C->D->E má délku totožnou. Rázem tedy máme k dispozici dvě nejkratší cesty z vrcholu A do vrcholu E. Abychom rozlišili případ, kdy existuje v grafu právě jedna optimální cesta mezi dvěma vrcholy, od případu, kdy těchto cest existuje více, zavedeme pojem *pareto-optimální* cesta. Toto bude každá taková cesta mezi dvěma uzly, která je nejkratší (z hlediska aktuálně používané definice délky cesty), ovšem existence této cesty nevylučuje možnost existence cesty jiné, která má délku totožnou.

1.1.4 Grafová databáze

V posledních letech zažívají velký rozmach tzv. NoSQL úložiště. Jejich základním cílem je efektivněji ukládat a zpracovávat data, která svými vlastnostmi neodpovídají tradičnímu relačnímu modelu. Základní motivací pro vznik těchto úložišť byl zejména prudce rostoucí počet dat, která potřebujeme uchovávat, a jejich vzájemná propojenost [29].

NoSQL úložiště se zpravidla rozdělují do více typů. Jedním ze zástupců těchto úložišť jsou i *grafové databáze*. Tyto databáze pracují s daty přímo ve formátu grafu. Z toho vyplývají i všechny možnosti těchto databází. Každá entita (vrchol grafu), stejně jako každá hrana spojující vrcholy, může obsahovat libovolné množství atributů. Pro vyhledávání v grafu pak můžeme využít traverzování.

Tohoto typu databáze bývá často využito pro ukládání vzájemně úzce propojených dat, jako jsou například data o uživatelích sociálních sítí.

1.2 Motivace

Analýzou struktury dat jízdních řádů, které se detailně budeme věnovat v příštích kapitolách, zjistíme, že struktura těchto dat je z podstaty grafová. V následujících kapitolách proto využijeme grafové databáze pro uložení dat o jízdních řádech a vyzkoušíme možnosti této databáze pro vyhledávání spojů nad těmito daty. Pro tyto účely jsme vybrali databázi Neo4j [18], která je nejrozšířenější a všeobecně nejlépe hodnocenou existující grafovou databází vůbec [35].

Pro možnost srovnání výhod grafových databází navrhneme i vyhledávací algoritmus nad relační databází. Jako zástupce relačních databází byla zvolena databáze PostgreSQL, která je považována za nejpokročilejší open-source relační databázi[1].

[1] <http://www.postgresql.org/>

Kapitola 2

Současný stav

V České republice vznikl v letech 1998 - 1999 na objednávku Ministerstva dopravy[1] *Celostátní informační systém o jízdních řádech*[2] (CIS JŘ). K provozování systému zákon opravňuje Ministerstvo dopravy, které však od roku 2001 do současnosti pověřuje spravováním systému společnost CHAPS s.r.o[3].

Podle platného zákona o silniční dopravě (zákon číslo 111/1994 Sb. a prováděcí vyhláška 388/2000 Sb.) a zákona o dráhách (zákon číslo 266/1994 Sb.) "*obsahuje CIS JŘ schválené jízdní řády linek veřejné vnitrostátní linkové dopravy (včetně městské autobusové dopravy), schválené jízdní řády linek veřejné mezinárodní linkové dopravy, které mají na území ČR zastávku pro nástup nebo výstup cestujících a schválené jízdní řády veřejné drážní osobní dopravy na dráze celostátní, regionální, tramvajové, trolejbusové, speciální a lanové provozované na území ČR*".

Svým rozsahem je databáze CIS JŘ v Evropě zcela ojedinělým projektem. V ostatních zemích jsou sice pozorovatelné zárodky projektů, které sdružují data o jízdních řádech z více správních celků, nikdy ale ani zdaleka nedosahují celostátní úrovně [39].

2.1 Existující vyhledávače spojů v ČR

Ačkoliv jsou data z CIS JŘ veřejná, veřejnosti jsou poskytována pouze ve formátech PDF a XLS, které prakticky neumožňují další strojové zpracování [4]. Společnost CHAPS s.r.o. má samozřejmě k dispozici i strojově zpracovatelné formáty, tyto ale veřejně neposkytuje. Nad těmito daty ovšem vznikla řada produktů s názvem IDOS (informační dopravní systém), které umožňují vyhledávání v kompletních jízdních řádech z CIS JŘ.

2.1.1 IDOS

IDOS je souhrnné pojmenování pro aplikace poskytované na více platformách, které vyvíjí sama společnost CHAPS. Konkrétně jsou poskytovány aplikace pro následující platformy:

[1] <http://www.mdcr.cz/cs/default.htm>
[2] <http://www.chaps.cz/cs/products/CIS>
[3] <http://www.chaps.cz/>

- IDOS pro lokální počítače a sítě
- IDOS pro mobilní zařízení
- IDOS pro internet
- IDOS pro mobilní telefony

Všechny tyto aplikace mají k dispozici kompletní data o jízdních řádech, která jsou v CIS JŘ. Nejrozšířenější variantou a v České republice bezkonkurenčně nejpoužívanějším vyhledávačem spojení [23] je aplikace IDOS pro internet[4], která v roce 2012 vyhrála v soutěži Evropské komise o nejlepší evropský přeshraniční multimodální[5] plánovač dopravního spojení [7].

2.1.1.1 IDOS API

Společnost CHAPS mimo jiné poskytuje za poplatek přístup k plnému API (application programming interface)[6] webové služby, pomocí kterého je možné získávat informace z jízdních řádů. Díky tomuto faktu vznikla celá řada aplikací (zejména mobilních) sloužících pro vyhledávání spojení, které požadavky uživatelů delegují na zmíněné API. Tvůrci aplikací tak vůbec nemusí řešit vlastní vyhledávání spojení v jízdních řádech. Nevýhoda je ovšem zřejmá – při hledání spojení musí být aplikace připojena k internetu. Mezi tyto aplikace patří např. Pubtran[7], MHDApp[8] apod.

2.1.1.2 Další vyhledávače využívající IDOS

Kromě mobilních aplikací využívajících IDOS API existuje i celá řada webových aplikací, které pro vyhledávání spojů využívají možností IDOSu. Jako nejlepší příklad slouží vyhledávače jednotlivých dopravních podniků velkých měst. Například Dopravní podnik hlavního města Prahy (DPP)[9] má svou vlastní webovou stránku pro vyhledávání spojení i zobrazení výsledků vyhledávání, vlastní vyhledávání ovšem deleguje na IDOS. Dopravní podnik města Brna (DPMB)[10] má naopak vlastní formulář pro zadání parametrů spojení, výsledky vyhledávání se ovšem zobrazují už přímo na doméně IDOS.
Dalším vyhledávačem, který pro vlastní vyhledávání spojení využívá služeb IDOSu je například vyhledávač na webovém portálu Českých drah[11].

2.1.2 Jiné aplikace pro vyhledávání spojů s vlastní implementací vyhledávače

Absence veřejně dostupných strojově zpracovatelných dat z CIS JŘ výrazně omezuje možnost konkurovat aplikacím IDOS. V České republice proto ani neexistuje mnoho vyhledávačů, které by využívaly vlastní data, ale především vlastní vyhledávací algoritmus pro

[4] <http://www.chaps.cz/cs/products/IDOS-internet>
[5] využívající více dopravních prostředků a oborů
[6] rozhraní pro programování aplikací
[7] <https://play.google.com/store/apps/details?id=cz.fhejl.pubtran&hl=cs>
[8] <http://www.mhdapp.cz/>
[9] <http://spojeni.dpp.cz/>
[10] <http://www.dpmb.cz/>
[11] <http://www.cd.cz/>

nalezení spojení. Výjimkou jsou například vyhledávače jednotlivých soukromých dopravců. Tyto vyhledávače ale pochopitelně zpravidla umožňují vyhledávání pouze ve spojích provozovaných tímto dopravcem. Tím pádem jsou používány zejména lidmi, kteří jezdí s tímto dopravcem pravidelně a nebo již znají jeho trasy (jinak řečeno, vědí, že mezi místem A a místem B existuje spojení, které provozuje daný dopravce).

Pro účely této práce bylo vybráno několik vyhledávačů, které při vyhledávání spojení žádným způsobem nekomunikují s IDOSem, a mají tedy vlastní implementaci vyhledávače a data o jízdních řádech lokálně k dispozici. Nyní si jednotlivé vyhledávače přiblížíme, v kapitole 3 se pak podíváme hlouběji na jejich technologické zázemí a implementaci vyhledávacího algoritmu.

2.1.2.1 Bileto

Bileto[12] je poměrně mladý a velmi nadějný projekt z roku 2014. Umožňuje vyhledání spojení napříč velkým počtem dopravců a jednoduché zakoupení jízdenek na všechna spojení současně. Mimo jiné spolupracuje se společností Seznam.cz[13], které data z jízdních řádů poskytuje pro zobrazení na mapách[14] [24].

2.1.2.2 Student Agency / RegioJet

Student Agency[15] je soukromá společnost provozující mezinárodní autobusovou dopravu. Její dceřinou společností je společnost RegioJet, která provozuje mezinárodní železniční dopravu a je největším soukromým železničním dopravcem v České republice [30]. Na svých webových stránkách[16] provozuje vyhledávač, který kombinuje vyhledávání jak v autobusové, tak ve vlakové dopravě.

2.1.2.3 Smartřády

Aplikace Smartřády[17] je mobilní aplikace pro vyhledávání spojů na zařízeních Windows Mobile. Data o jízdních řádek získává od společnosti CHAPS, která za jejich poskytnutí inkasuje licenční poplatek od koncového uživatele. Vlastní vyhledávání nicméně běží oproti lokální kopii těchto dat, a vyhledávací algoritmus je tak implementován přímo v samotné aplikaci.

2.1.3 Shrnutí

Ačkoliv v České republice existuje celá řada vyhledávačů spojení hromadné dopravy, velké množství z nich vůbec neřeší logiku vyhledávání interně, ale deleguje požadavky na API IDOSu. V tomto ohledu tedy můžeme říci, že vyhledávačů, které pracují s velkými daty a řeší vyhledávání interně, zase tak moc na území České republiky není.

[12] <https://www.bileto.cz/>
[13] <http://www.seznam.cz/>
[14] <http://www.mapy.cz/>
[15] <http://www.studentagency.cz>
[16] <https://jizdenky.studentagency.cz/>
[17] <http://www.circlegate.com/cs/smartrady/>

Kapitola 3

Technologické detaily existujících vyhledávačů v České republice

V předchozí kapitole jsme si představili některé existující vyhledávače spojů hromadné dopravy, které mají vlastní implementaci vyhledávacího algoritmu pro nalezení spojů. Pro účely této práce byli ti provozovatelé, kteří nemají technologické zázemí svého vyhledávače již veřejně zpřístupněno, obesláni emailem s žádostí o detailnější technické informace týkající se jejich vyhledávače.

Šlo především o žádost o poskytnutí výčtu technologií, na kterých je jejich vyhledávač postaven. Dále o detailnější popis vyhledávacího algoritmu, který využívají, řádový počet dat, kterými disponují, a v neposlední řadě průměrné časy nutné pro vyhledání spojení. Na základě jednotlivých odpovědí vzniklo následující shrnutí.

3.1 Vyhledávač IDOS

Dne 26.3.2015 jsme na infolinku společnosti CHAPS pro produkty IDOS (idos@chaps.cz) zaslali žádost o poskytnutí výše uvedených údajů. Do dnešního dne bohužel nebyla poskytnuta žádná odpověď.

Algoritmus vyhledávání nebyl veřejně nikdy podrobně zveřejněn [27]. Není tedy možné zjistit, jakým způsobem funguje.

3.2 Vyhledávač Bileto

Bileto je v cloudu [5] běžící software, kam mohou jednotliví dopravci sami nahrávat data o svých spojích [33].

Data o jízdních řádech jsou trvale uložena v relační databázi. Z těchto dat se generují struktury na vyhledávacích serverech. V případě změny v trvalém úložišti jsou tyto servery o změně informovány a ihned si data aktualizují. Celé vyhledávání je napsáno v jazyce C++. Umožňuje vyhledání všech pareto-optimálních výsledků na základě klasifikovatelných kritérií (počet přestupů, průjezdní body, preference určitých dopravců apod.). Při jednom hledání tedy najde nejvýhodnější spoje vzhledem k pareto-optimalitě, v ideálním případě jsou to ale

vždy ty nejrychlejší, respektive ty s nejmenším počtem přestupů. Vyhledávání se zastaví až po prohledání celého prostoru, kdy lze spolehlivě říci, že byla nalezena nejlepší možná spojení pro dané vstupní parametry. Maximální možný počet přestupů je nastaven defaultně na 10.

Databáze Bileta k březnu 2015 obsahovala přibližně 22 000 jízd. Pro tento rozsah dat trvá průměrné vyhledání spoje cca 30 ms. Vyhledávání vyžaduje pouze minimální preprocessing dat (cca 2 s sériového výpočtu).

V současné době tým Bileta pracuje na nové verzi vyhledávače, která by měla umožňovat vyhledávání nad kompletními jízdními řády z celé střední Evropy. Tím jsou myšleny všechny linky na tomto území včetně všech spojů městké hromadné dopravy (MHD), vlaků, autobusů apod. Jen pro ilustraci rozsahu dat – pouze síť autobusové dopravy na území ČR obsahovala již v roce 2009 přes 100 000 jízd [2]. Momentálně je vyhledávací algoritmus ve formě prototypu a na slibné cestě k úspěchu.

3.3 Vyhledávač dopravce Student Agency / RegioJet

Data o jízdních řádech jsou trvale uložena v relační databázi PostgreSQL. Každá jízda spoje je v databázi uložena pro každý den, ve který tento spoj jede. Jedním z atributů jízdy je tedy i datum. V databázi jsou všechny jízdy spojů nagenerovány na několik měsíců dopředu. Zde si povšimněme rozdílného přístupu k ukládání dat od Bileta, kde je každá jízda uložena pouze jednou pro všechny dny platnosti. Rozdílnost těchto postupů při ukládání dat jízdních řádů je detailně popsána v kapitole 4.3.

Celé vyhledávání je napsáno v jazyce Java. Aktuální spoje (na určitý počet dní dopředu) jsou mimo relační databázi uchovávány navíc v rámci operační paměti aplikačního serveru. Při přijetí dotazu na vyhledávání se z dostupných spojů v rámci operační paměti sestaví graf cesty, který se následně ocení a filtruje. Při výběru vhodných cest působí jako jeden z faktorů i částka, kterou klient za cestu zaplatí.

Databáze obsahuje pro jeden den více než 500 spojení. Nad těmito daty trvá průměrné vyhledávání cca 125 ms.

3.4 Vyhledávač Smartřády

Vyhledávací algoritmus byl podrobně popsán a zveřejněn v rámci diplomové práce z roku 2009 [2]. Aplikace disponuje kompletními datovými soubory s informacemi o jízdních řádech z CIS JŘ.

Vyhledávání je implementováno v programovacím jazyce C++. Před vlastním vyhledáváním se data o jízdních řádech načtou do operační paměti. Nad těmito daty se poté vyhledává pomocí tzv. *labelling algoritmu*, který slouží pro vyhledávání všech pareto-optimálních cest v k-kriteriálním[1] grafu.

Datové soubory aplikace obsahovaly v roce 2009 přes 100 000 jízd v rámci linkové autobusové dopravy na území ČR. Na testovacím zařízení trvalo načtení tohoto datového souboru necelých 6 s. Doba vyhledávání pak byla velmi proměnná v závislosti na vstupních parametrech –

[1] Graf, ve kterém můžeme délku cesty mezi dvěma vrcholy klasifikovat na základě konečného počtu různých kritérií.

od jednotek vteřin až po několik desítek vteřin. Podrobný přehled včetně detailní specifikace testovacího zařízení je k dispozici v samotné diplomové práci [2].

3.5 Shrnutí

V předchozích kapitolách jsme si představili vybraná existující řešení vyhledávání spojů na území České republiky. Všechna popsaná řešení mají společné to, že vlastní vyhledávání probíhá výhradně v operační paměti. U vyhledávače IDOS není možné tuto informaci ověřit, nicméně je velmi pravděpodobné, že vlastní vyhledávání probíhá taktéž v rámci operační paměti. Tento přístup je zcela logický – data je v každém případě nejprve třeba dostat do vnitřních struktur programovacího jazyka v požadovaném formátu, aby nad nimi následně mohlo proběhnout vyhledávání. Rychlost přístupu k datům v rámci operační paměti je navíc nesrovnatelně vyšší, než by tomu bylo při nutnosti stálého čtení dat z databáze (která má data persistentně uložena na pevném disku). Vlastní vyhledávací algoritmus pak může pracovat přímo na principu vyhledávání cest v grafu (viz projekt Smartřády).

Tento přístup má však kromě nesporných výhod i řadu nevýhod. V rámci vyhledávačů, které vlastní vyhledávání provádí na koncovém zařízení uživatele (např. projekt Smartřády), je vždy nejprve nutné načíst data o jízdních řádech do operační paměti zařízení, což může trvat delší dobu. Tato data pak mohou potřebovat více operační paměti, než má dané zařízení právě k dispozici. V rámci vyhledávačů, kdy vyhledávání probíhá na vzdáleném serveru (např. Bileto), se problém s nedostatkem operační paměti dá většinou řešit poměrně snadno prostým škálováním operační paměti. Zde ovšem nastává jiný problém, a to problém synchronizace. Data jsou totiž trvale uložena v databázi, která se pro vlastní vyhledávání vůbec nepoužívá. Při každé změně v trvalém úložišti dat se tedy tyto změny musí správně propagovat i na všechny vyhledávací servery, kde je nutné změnit data přímo v operační paměti. Tato synchronizace je přitom obecně problém netriviální a může přinést řadu problémů. Zejména jde o možnost vzniku nekonzistence dat mezi trvalým úložištěm a vyhledávacím serverem, ke které může při nesprávně provedené nebo nedokončené synchronizaci dojít. Vzniklá nekonzistence se poté navíc složitě odhaluje a opravuje.

Tato úvaha nás dovedla zpět k motivaci této práce. Pokud pro ukládání dat využijeme grafovou databázi, která již explicitně udržuje data v grafové struktuře, nebudeme muset vůbec řešit synchronizaci dat mezi trvalým úložištěm a operační pamětí, protože vlastní vyhledávání poběží *v rámci* trvalého úložiště. Současně nám odpadne jakýkoliv preprocessing dat před vlastním vyhledáváním.

Kapitola 4

Analýza vyhledávání spojů

V této kapitole analyzujeme problematiku vyhledávání spojů a definujeme tak základní požadavky, které by měla naše vlastní implementace vyhledávače uspokojovat.

4.1 Uživatelské vstupy

V úvodu práce jsme si představili několik již existujících vyhledávačů spojení v České republice. Většina z nich nabízí velmi podobné či zcela totožné možnosti parametrizace vstupů vyhledávání. Detailnější analýzu možností parametrizace uživatelských vstupů nalezneme například v diplomové práci *Analýza integrace systému vyhledávání spojů se systémem kolizních informací a její využití* [12]. Výsledkem analýzy již existujících vyhledávačů tak vznikl následující výčet vstupů, které bude muset náš vyhledávač akceptovat:

- Výchozí stanice
- Cílová stanice
- Datum a čas
- Výběr, zda chceme hledat podle data a času odjezdu z výchozí stanice, nebo příjezdu do cílové stanice.
- Maximální počet přestupů

Existující vyhledávače často umožňují i další vstupní parametry, například přidání průjezdních bodů, specifikace požadavků na spoje (druh spoje, bezbariérovost...) apod. Cílem této práce ovšem není detailně sledovat parametry jednotlivých spojů, ale vyzkoušet možnosti grafových databází pro účely vyhledávání v jízdních řádech. Výše uvedený výčet vstupů proto budeme považovat pro účely této práce za finální.

4.2 Parametry kvality vyhledávače

Z hlediska uživatelské přívětivosti existují kromě možnosti parametrizace vstupů minimálně dva základní parametry vyhledávače, pomocí kterých koncový uživatel rozeznává kvalitu vyhledávače:

- Rychlost nalezení výsledků
- Kvalita nalezených výsledků

Zatímco *rychlost nalezení výsledků* je parametr velmi snadno měřitelný, parametr *kvalita nalezených výsledků* je z hlediska své měřitelnosti poněkud těžko uchopitelný. Nicméně po definici základních parametrů nalezených cest spojů bude snazší tento parametr nějakým způsobem změřit.

4.2.1 Parametry nalezených cest spojů

Každý nalezený výsledek, který odpovídá uživatelským vstupům pro vyhledávání spojů, se dá popsat pomocí tří základních parametrů:

- Doba jízdy
- Počet přestupů
- Riziko zmeškání přestupného spoje

Doba jízdy je velmi snadno měřitelný parametr. Jedná se o prostý rozdíl času příjezdu do cílové stanice a času odjezdu z výchozí stanice. *Počet přestupů* je taktéž velmi snadno měřitelný. Definujeme jej jako počet přesunů na návazný spoj v rámci přestupní stanice.

Naopak *riziko zmeškání přestupného spoje* je parametr velmi obtížně určitelný. Obecně by měl udávat pravděpodobnost, že některý z přestupných spojů na nalezené cestě odjede z přestupné stanice dříve, než se k ní dostane spoj předchozí. Pravděpodobnost rizika zmeškání spoje můžeme minimalizovat přidáním parametru *počet minut nutných na přestup* (PMP). Tento parametr je ovšem pro každou stanici specifický a žádným způsobem nereflektuje možnost zpoždění předchozího spoje. Umělým zvyšováním PMP přitom zase snižujeme kvalitu nalezených výsledků, protože uživatele připravujeme o spoje, které by za normálních okolností bez problémů stihl. Možnost zpoždění spoje je navíc daná nejen druhem dopravního prostředku, ale také časem a trasou, po které spoj jede. Dále může být spoj pochopitelně zpožděn i jinou, a to i zcela nepředvídatelnou, příčinou (porucha, počasí...).

Vzhledem k těžké měřitelnosti parametru riziko zmeškání spoje proto v našem vyhledávači budeme sledovat pouze dva parametry nalezených výsledků, a to:

- Doba jízdy
- Počet přestupů

V rámci některých existujících vyhledávačů se samozřejmě riziko zmeškání spoje projevuje na výsledcích vyhledávání. K přesnému změření tohoto parametru je ovšem nutné velké množství dat, zejména o dopravní situaci na dané trase. Zpravidla se ale dá říci, že čím menší je časové okno na přestup na návazný spoj, tím větší je riziko zmeškání návazného spoje.

Parametr PMP si ovšem nemůžeme dovolit ignorovat, protože je nutné jej do vyhledávacího algoritmu začlenit. V existujících vyhledávačích může být tento parametr přímo součástí dat z jízdních řádů, případně může být uvažován jako parametr, který může zadat uživatel. V našem případě ovšem pro zjednodušení budeme vždy uvažovat $PMP = 1$ minuta. Detailnější sledování tohoto parametru by nám totiž nepřineslo žádné zajímavé poznatky týkající se hlavního tématu této práce, kterým je návrh vyhledávacího algoritmu.

4.2.2 Optimalizace kvality nalezených výsledků

Na základě parametrů nalezených cest spojů, které jsme si právě definovali, můžeme výsledky vyhledávání mezi sebou porovnávat, určit tak, který výsledek je *kvalitnější*, a zobrazit potom uživateli pouze ty nejrelevantnější výsledky.

Pokud například existují dva výsledky vyhledávání, které do cílové stanice přijedou ve stejný čas, je žádoucí preferovat ten výsledek, který má kratší dobu jízdy, případně menší počet přestupů. V žádném případě ale není nutné ani žádoucí zobrazovat uživateli všechny tyto výsledky.

4.2.3 Souhrnné nároky

Výsledkem předchozí analýzy tak vznikl souhrn nyní již měřitených nároků, které náš vyhledávač bude muset splňovat. Výsledky vyhledávání tedy vždy budeme chtít dostat:

- V co nejrychlejším čase
- V co nejvyšší možné *kvalitě*

4.3 Datová struktura

Celosvětově uznávaný a známý standard pro výměnu dat o veřejné hromadné dopravě je specifikace GTFS (General Transit Feed Specification) [10][41].

GTFS balík dat se zpravidla skládá z několika CSV[1] souborů (s přípnou .txt), kde každý soubor reprezentuje jednu logickou entitu a její parametry. Tyto entity si nyní představíme. Uvedeme si přitom i jejich anglické názvy, abychom se vyhnuli terminologickému chaosu. V dalších částech práce se ovšem budeme držet výhradně českých překladů. Krátké ukázky jednotlivých CSV souborů nalazneme v příloze B.

- **Stops** (stanice) – Obsahuje informace o stanicích (např. stanice Praha, hl. n.).

[1] <https://tools.ietf.org/html/rfc4180>

- **Agency** (dopravci) – Obsahuje informace o dopravcích provozujících spoje (např. Dopravní podnik hlavního města Prahy).
- **Routes** (linky) – Obsahuje informace o jednotlivých linkách, které provozují daní dopravci (např. linka metra B v rámci DPP reprezentuje jeden záznam).
- **Trips** (jízdy) – Jeden záznam zde reprezentuje právě jednu jízdu konkrétní linky.
- **StopTimes** (zastavení jízdy) – Jeden záznam reprezentuje jedno zastavení konkrétní jízdy na konkrétní stanici. Záznam obsahuje informace o času příjezdu a času odjezdu. Pokud tedy vybereme všechna zastavení jízdy pro konkrétní jízdu, získáme tak kompletní informace o celé trase a zastaveních dané jízdy.
- **Calendar** (interval platnosti) – Jeden záznam reprezentuje interval platnosti. Konkrétně záznam obsahuje časový rozsah (např. 24.6.2015 - 23.9.2015) a dále výčet dnů platnosti (např. pouze všední dny). Každý záznam z jízd má referencovaný právě jeden záznam z intervalu platnosti. Díky tomu jsme schopni u každé jízdy zjistit, zda v daný den jede, či nikoliv.

Vzhledem k existenci standardu GTFS není důvod se od něho při implementaci příliš odchylovat. Naopak se celá implementace výrazně zjednoduší, pokud se jej budeme co nejvíce držet. Za prvé velmi elegantním způsobem zachycuje strukturu dat jízdních řádů (všimneme si, že většinu entit z GTFS balíku jsme si již představili na úvod práce při analýze papírových jízdních řádů) a za druhé nám při dodržení tohoto formátu odpadne nutnost případná data získaná ve formátu GTFS zbytečně transformovat do naší databáze.

4.3.1 Specifika datové struktury

Nyní se pokusíme podrobněji rozvést a představit strukturu dat, kterou budeme v aplikaci využívat.

Základem jízdních řádů je stanice. Výchozí a cílová stanice jsou nejdůležitější vstupní parametry vyhledávacího algoritmu. Ve specifikaci GTFS se navíc mohou stanice sdružovat do logických celků, v rámci této práce ovšem tuto možnost nebudeme kvůli jednoduchosti uvažovat.

Jízda reprezentuje jednu jízdu spoje z výchozí do cílové stanice. Jak jsme si už řekli dříve, jízda je tvořena posloupností zastavení. Každé zastavení jízdy obsahuje informace o stanici, na které se právě nacházíme, a dále o času příjezdu a odjezdu z této stanice. Zde je důležité si uvědomit, že jízda ani jednotlivá zastavení nedisponují informacemi o datu, ve kterém tato jízda jede. Platnost jízdy pro konkrétní datum se proto musí zjišťovat až z příslušného záznamu v intervalu platnosti. Pokud jede daná jízda přes půlnoc, pro její platnost se vždy musíme dívat na platnost pro první zastavení této jízdy. Pokud by neexistovala vazba z jízdy do intervalu platnosti, znamenalo by to, že jízda je v provozu každý den. Zde je vhodné si uvědomit, že informace o datu jízdy by mohly být umístěny i na samotných zastaveních jízd. V tom případě bychom mohli zcela vypustit entitu interval platnosti. Jednotlivá zastavení jízdy by pak kromě času příjezdu a odjezdu obsahovala i datum příjezdu a odjezdu. Tím by se sice datový model zjednodušil, velikost dat v databázi by tím ale výrazně narostla. Zatímco při dodržení formátu specifikovaném v GTFS nám stačí mít pouze jeden záznam

o jízdě, která jezdí každý pracovní den od 1.1.2013 do 31.1.2013 s výjezdem v 7:00, tak při použití druhého zmíněného formátu by tato jízda byla v databázi uložena celkem 252x (v roce 2013 bylo přesně 252 pracovních dní), protože by musela být uložena pro každý den zvlášť. Zejména pro větší objemy dat je proto zcela žádoucí striktně se držet specifikace GTFS.

4.3.2 Nároky na vyhledávač

V předchozích sekcích jsme si postupně definovali sadu požadavků, které budeme od vyhledávače očekávat. Stejně tak jsme si definovali datovou strukturu, kterou budeme pro vyhledávání spojů využívat. Jejich spojením nám vzniknou základní předpoklady, na kterých bude muset vyhledávací algoritmus pracovat.

Uvažujme vstupní paremetry vyhledávacího algoritmu zadané uživatelem (pro případ vyhledávání dle času odjezdu, vyhledávání dle času příjezdu by bylo velmi podobné): Výchozí stanice S_v, cílová stanice S_c, datum výjezdu D_v, čas výjezdu C_v a maximální počet přestupů P. Pro souhrnný datum a čas výjezdu zavedeme ještě pomocnou proměnnou DC_v. Vyhledávací algoritmus pak musí najít všechny nejkvalitnější výsledky (sekvence jízd), pro které platí:

- Výchozí stanice je S_v.
- Cílová stanice je S_c.
- Všechny jízdy v rámci nalezené cesty jsou v daný den v platnosti (tedy v datu D_v, případně $D_v + d$ dní, kde d značí počet překlenutí přes půlnoc, pokud jede spoj v rámci více dní).
- Čas výjezdu z S_v je větší nebo roven času C_v (v případě hledání spojů přes půlnoc ovšem toto musí vyhledávací algoritmus správně reflektovat – tedy pokud hledáme spoje s výjezdem v 23:00, samozřejmě je žádoucí najít i spoje s výjezdem v 00:15, které už ovšem jedou příští den).
- Maximální počet přestupů na cestě z S_v do S_c je P.
- Na každé přestupné stanici v rámci cesty máme vždy k dispozici minimálně PMP minut na přestup.

Současně by měl nalézt tyto výsledky v co nejrychlejším čase.

Kapitola 5

Vyhledávací algoritmus nad relační databází

V předchozí kapitole jsme stanovili požadavky, které musí náš vyhledávač splňovat. Nyní musíme navrhnout řešení, kterým se dobereme výsledků. V této kapitole navrhneme řešení, při kterém budeme využívat pouze relační databázi. V kapitole 6 poté navrhneme řešení za použití databáze grafové.

Při návrhu řešení budeme pracovat pouze s možností, že spojení hledáme dle data a času odjezdu. Při hledání výsledků dle data a času příjezdu by situace byla velmi podobná. Současně budeme uvažovat hledání pouze v rámci jednoho dne, vyhneme se tak poměrně složitému ošetřování vyhledávání přes půlnoc. Vlastní implementace vyhledávačů toto nicméně bude muset reflektovat.

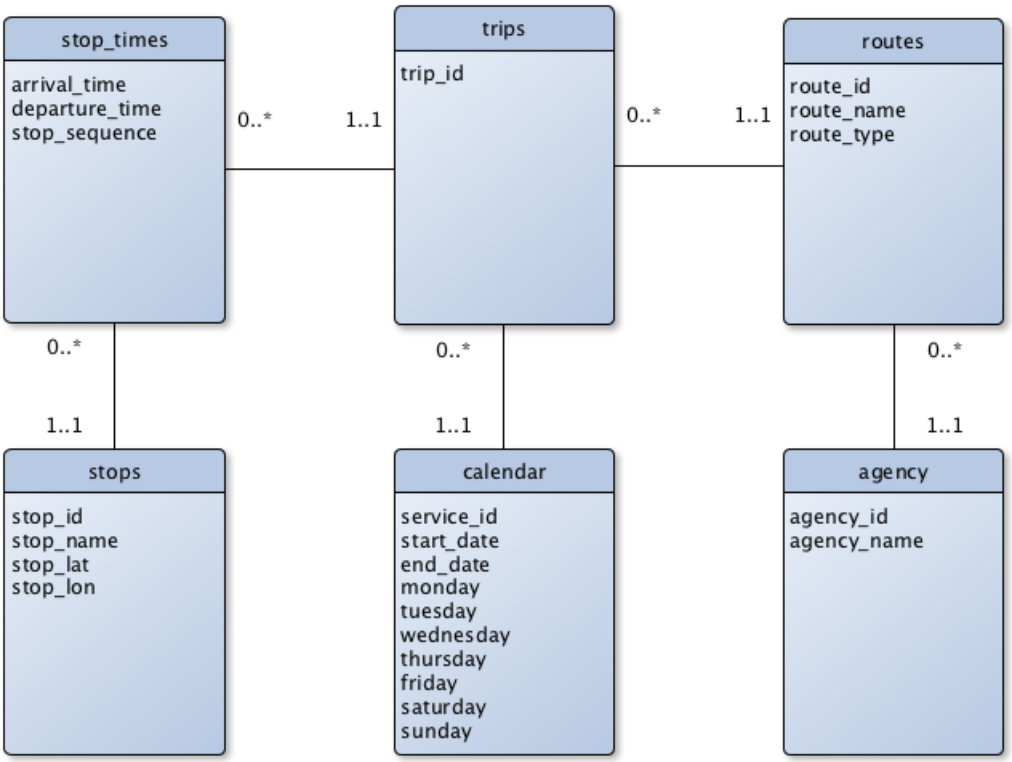
5.1 Přídavný parametr vyhledávacího algoritmu

Pro účely vyhledávání je dobré určit si ještě dodatečný parametr, se kterým bude vyhledávač pracovat. Při analýze jsme určili, že výsledný algoritmus musí najít všechna nejkvalitnější spojení, která mají datum a čas výjezdu z S_v větší nebo roven DC_v (zadané jako vstupní parametry). Pokud bychom tedy chtěli například najít všechny spoje s výjezdem od 20.4.2015 v 01:00, vyhledávací algoritmus by začal hledat všechny odjezdy, které jsou větší nebo rovny tomuto datu a času. Prohledával by tedy celý prostor od zadaného data a času až do konce platnosti jízdního řádu. Ačkoliv by toto chování mohlo být v jistých případech považováno za žádoucí, jeho důsledkem je extrémní nárůst výpočetních požadavků. Ve většině případů nám totiž bude stačit vyhledávat jen spojení, která mají čas příjezdu do S_c menší nebo rovný určitému datu a času. Pro toto datum a čas si zavedeme označení $MaxDC_p$ a budeme jej označovat jako *maximální datum a čas příjezdu*.

Hodnota parametru $MaxDC_p$ může být obecně pro různá hledání velmi odlišná. Záležet přitom bude především na druhu jízdního řádu. Pokud budeme uvažovat hledání pouze v rámci MHD, může být hodnota $MaxDC_p$ - DC_p pouze v řádu jednotek hodin. Pro jízdní řád mezinárodní vlakové dopravy by se ovšem tento rozdíl musel pochopitelně zvětšit, například až na několik desítek hodin.

5.2 Relační schéma

Dříve jsme si představili GTFS formát dat jízdních řádů, který se skládá z více textových souborů ve formátu CSV. Tato data mohou být velmi snadno přetransformována do relační databáze. Jednotlivé soubory budou tabulky, řádky v souborech budou řádky tabulek a údaje v řádcích oddělené čárkou (dle specifikace CSV) budou reprezentovat sloupce tabulky. Pro naše účely si formát GTFS mírně upravíme tak, aby více vyhovoval našim požadavkům. Některé informace, které bychom dále žádným způsobem nevyužili, proto vynecháme. Vznikne nám tak schéma zobrazené na obrázku 5.1.



Obrázek 5.1: Schéma GTFS formátu dat

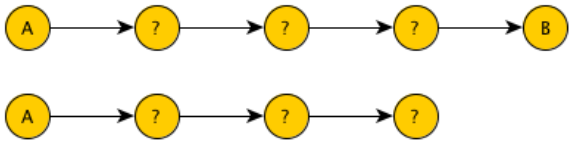
Jednotlivé atributy jsou pojmenovány shodně s pojmenováním v GTFS balíku dat. Při představení tohoto balíku dat jsme si navíc i detailněji popsali, která data v jednotlivých souborech nalezneme. Další informace o jízdních řádech byly řečeny hned na úvod práce. Nicméně i tak by pro čtenáře mohl být některý atribut zejména z tabulky calendar těžko pochopitelný, provedeme tedy krátký sourhn.

Záznam v tabulce calendar bude představovat jeden časový interval platnosti. Tento interval je definován pomocí počátečního a koncového data a výčtu dní v týdnu. Každý den v týdnu pro jeden záznam nabývá hodnoty *true* nebo *false*. Jízdy, které jezdí například jen v neděli proto budou navázány na takový interval platnosti, který má hodnotu *true* pouze u neděle.

U tohoto způsobu definování platností si můžeme povšimnout určité těžkopádnosti pro výjimečné situace, kterými jsou například státní svátky. V tyto dny bývá často dodržován stejný jízdní řád jako o nedělích, nicméně platnost tohoto jízdního řádu je jen na jeden konkrétní den. V tomto případě nám nezbyde nic jiného, než pro každý státní svátek vytvořit nový interval platnosti.

5.3 Vyhledání spojů bez přestupu

Schéma z obrázku 5.1 nyní zachycuje veškeré informace, které potřebujeme pro úspěšný návrh vyhledávacího algoritmu. Představíme si nyní konkrétní situaci: Uvažujme, že se chce cestující dostat ze stanice A (S_v) do stanice B (S_c) v čase T. Ze stanice A v rámci požadovaného času T (interval mezi časem odjezdu a maximálním časem příjezdu) vyjíždí pouze dvě jízdy, z nichž jedna do stanice B opravdu jede.



Obrázek 5.2: Jízdy a jejich zastávky s výjezdem ze stanice A

Očekávaným výstupem vyhledávacího algoritmu by nyní měl být seznam zastavení z obrázku 5.2, které náleží stanicím A a B, pokud jsou tato zastavení propojena jízdou. Zkusíme se nyní k tomuto výsledku dostat pomocí jazyka SQL za předpokladu, že máme v databázi uložená data odpovídající jízdám na obrázku 5.2.

5.3.1 Nalezení počátečních zastavení

Uvažujme následující množinu vstupních parametrů algoritmu (místo názvu stanic již používáme jejich ID):

- S_v ... 5
- S_c ... 9
- D_v ... 20.04.2015
- C_v ... 05:00

K těmto vstupním parametrům si doplníme další důležité parametry, které sice uživatel nezadává, nicméně před samotným spuštěním vyhledávání je nutné tyto parametry určit nebo dopočítat.

- DC_v ... 20.04.2015 05:00

- Den výjezdu Day_v ... pondělí
- $MaxD_p$ ... 20.04.2015
- $MaxC_p$ ... 11:00
- $MaxDC_p$ ... 20.04.2015 11:00

Pro nalezení všech možných cest ze stanice S_v do stanice S_c musíme nejdříve nalézt množinu zastavení, která náleží S_v a současně pro všechna tato zastavení platí, že čas jejich odjezdu náleží intervalu $<DC_v; MaxDC_p)$. Interval si můžeme dovolit nechat na konci otevřený, protože jakýkoliv spoj s výjezdem ve 14:00:00 nemůže do další stanice přijet dříve, než ve 14:00:01. Tato množina zastavení bude reprezentovat, na které ze spojů můžeme na stanici S_v v daném čase nastoupit. Připomeňme, že platnost daných jízd v konkrétní den zjistíme až ze záznamu v intervalu platnosti. Vzhledem k tomu, že platnosti jednotlivých jízd jsou omezeny i konkrétním dnem v týdnu, museli jsme tento den v týdnu pro D_v určit. Záznamy je žádoucí vracet v seřazeném pořadí dle data odjezdu. Tak bude zaručeno, že jako první budeme zpracovávat pro nás nejvíce zajímavá data.

SQL dotaz pro nalezení množiny výchozích zastavení tedy bude vypadat takto:

```
SELECT s.* FROM stop_times s
INNER JOIN trips t ON s.trip_id = t.trip_id
INNER JOIN calendar c ON t.service_id = c.service_id
WHERE s.station_id = 5 AND c.start_date <= '20.04.2015'
AND c.end_date >= '20.04.2015' AND c.monday = true
AND s.departure_time >= '05:00' and s.departure_time < '11:00'
ORDER BY s.departure_time
```

5.3.2 Iterace po následujících zastaveních

Když máme k dispozici množinu zastavení ve výchozí stanici, můžeme začít *následovat cestu* jednotlivých jízd, kterým tato zastavení přísluší. Pro každé zastavení z počáteční množiny tedy můžeme zjistit všechna následující zastavení této konkrétní jízdy. Pro získání následujících zastavení už si vystačíme pouze s tabulkou stop_times. Vzhledem k tomu, že tato tabulka obsahuje sloupeček stop_sequence, který určuje pořadí zastavení v rámci jízdy, stačí nám vybrat všechna zastavení na aktuální jízdě, která mají hodnotu stop_sequence vyšší, než je hodnota tohoto parametru u výchozího zastavení.

Pro žádné z následujících zastavení již nemusíme kontrolovat, zda je konkrétní jízda daný den v provozu. Toto jsme totiž ověřili již při získání výchozí množiny zastavení. Naopak pro každé následující zastavení musíme kontrolovat hodnotu parametru arrival_time, protože musí vždy platit $arrival\_time <= MaxC_p$. Jednotlivá zastavení na jízdě je vhodné řadit podle pořadí tohoto zastavení v rámci jízdy. Tím bude zaručeno, že zastavení zpracováváme v pořadí, které odpovídá pořadí z jízdního řádu.

Budeme nyní uvažovat, že hledáme všechny následující zastavení jízdy T_1 od zastavení s pořadím 1. SQL dotaz potom bude vypadat takto:

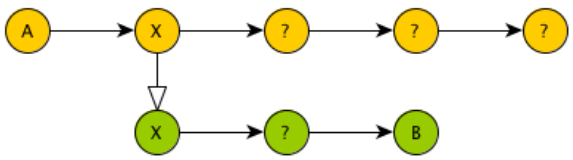
```
SELECT * FROM stop_times
WHERE trip_id = 1 AND stop_sequence > 1
AND arrival_time <= '11:00'
ORDER BY stop_sequence
```

Pro všechna takto nalezená zastavení pak existují dvě možnosti. Buď toto zastavení náleží S_c, nebo jí nenáleží.

Pokud náleží nalezené zastavení S_c, můžeme si výsledek poznamenat a zastavit iteraci po následujících zastaveních aktuální jízdy. Po doběhnutí algoritmu pro všechny jízdy získané prvním SQL dotazem pak budeme mít k dispozici všechny možnosti, jak se dostat z výchozí do cílové stanice v konkrétním čase **bez přestupu**. Vzhledem k tomu, že jsme si záznamy při vybírání z databáze řadili, budou výsledky seřazené vzestupně dle data odjezdu z výchozí stanice.

5.4 Vyhledávání spojů s přestupem

Nyní se pokusíme zdokonalit vyhledávání spojů, které jsme navrhli, tak, aby umožňovalo uvažovat i s možností přestupů. Budeme se chtít dostat ze stanice A (S_v) do stanice B (S_c) v čase T. V konkrétním čase vyjíždí ze stanice A pouze jedna jízda, která bohužel do stanice B nejede. Na druhé zastávce jízdy se dá ovšem přestoupit na jiný spoj, který již do stanice B jede:



Obrázek 5.3: Jízdy a jejich zastávky s výjezdem ze stanice A a s přestupní stanicí X

Na obrázku 5.3 značíme prázdnou šipkou možnost přestupu mezi dvěma jízdami. Tohoto značení se budeme držet i dále v této práci. Očekávaným výstupem vyhledávacího algoritmu by nyní měl být seznam zastavení z obrázku 5.3, která náleží stanicím A, X a B.

V případě, že nalezené zastavení v přechozím případě náleželo S_c, ukončili jsme iteraci po dané jízdě. V opačném případě jsme žádnou akci neprováděli. Pokusíme se nyní ale zamyslet nad tím, co pro nás tento stav vlastně znamenal. Jednoduše řečeno jsme se nacházeli ve stanici X, která nebyla stanicí cílovou (X != S_c). Do této stanice jsme přijeli konkrétní jízdou v konkrétním čase (čas příjezdu do této stanice). Tato situace přesně odpovídá naší výchozí situaci, kdy jsme se nacházeli na stanici S_v v konkrétním datu D_v a čase C_v. Náš aktuální stav se tedy dá velmi jednoduše převést na výchozí stav, a tudíž může být i stejně řešen.

Pokud tedy uvažujeme veškerou logiku pro vyhledání spojů bez přestupu soustředěnou do jedné metody, velmi snadno tak vyřešíme i logiku pro vyhledávání spojů s libovolným počtem přestupů. Pokud při iteraci zastaveními konkrétní jízdy narazíme na stanici, která není

stanicí cílovou, zavoláme jednoduše tutéž metodu, ve které se nacházíme, ovšem s mírně pozměněnými parametry. Výchozí stanice bude stanice, ve které se aktuálně nacházíme v rámci iterace, datum a čas odjezdu poté bude odpovídat datu a času příjezdu do této stanice. Čas příjezdu ovšem musí být navýšený o hodnotu parametru PMP. Ostatní parametry zůstanou totožné.

Narazili jsme tak na situaci, kdy využijeme principu *rekurze*, která v programování představuje opakované vnořené volání stejné funkce [1]. V našem případě nebudeme od vnořeného volání očekávat žádnou návratovou hodnotu, ale metodě pouze předáme strukturu (list, mapu) do které se při prvním i každém dalším vnořeném volání zaznamená, na kterých zastaveních jsme v rámci této větve programu již byli. V okamžiku, kdy narazíme na S_c, se rekurzivní volání zastaví a informace o nalezené cestě se uloží do centrálního úložiště nalezených cest. Pokud na S_c v rámci jedné větve (cesty) rekurze nikdy nenarazíme, zastaví se další rekurzivní volání v okamžiku překročení času $MaxC_p$, nebo v situaci, kdy již nebudou k dispozici žádná další zastavení.

Po doběhnutí algoritmu pak v centrálním úložišti nalezených cest nalezneme všechny možné cesty (sekvence jízd) ze stanice A do stanice B s libovolným počtem přestupů.

Všimneme si, že v právě navrženém řešení vyhledávání jsme již uvažovali, že rekurze bude probíhat v rámci programovacího jazyka a nikoli databáze. V každém kroku rekurze bude nutné volat nový SQL dotaz pro zjištění výchozích zastavení (a jejich jízd) a další SQL dotazy pro zjištění následných zastavení aktuální jízdy. Zde je vhodné uvést, že dnešní databázové systémy umožňují navrhovat a používat tzv. uložené procedury[1], ve kterých by se logika naší metody dala implementovat také. V rámci databáze PostgreSQL se dokonce dají procedury volat i rekurzivně [36]. V případě, že by rekurze běžela přímo v rámci databáze, odpadla by režie nutná pro komunikaci aplikace s databází. Nicméně vlastní logika vyhledávání je natolik komplikovaná, že bylo zvoleno řešení rekurze v rámci aplikace. Čas nutný pro komunikaci s databází je díky přítomnosti *connection poolingu* minimální [38] a kód je v rámci aplikace výrazně čitelnější i lépe testovatelný. Navíc si v rámci rekurzivních volání vždy potřebujeme předávat seznam všech již navštívených stanic a práce s poli je v rámci uložených procedur pomalá [36]. V případě rekurze nad aplikací si navíc jednotlivé výsledky můžeme ihned ukládat pro budoucí použití, zatímco při využití uložené procedury bychom výstupní data museli znovu procházet a zpracovávat, což by stejně vyžadovalo další volání SQL dotazů.

5.4.1 Nástin postupu vyhledávání s přestupy

Představíme si nyní ucelenou ukázku základní verze navrženého rekurzivního algoritmu. Připomeňme si, že v této podobě hledá všechny existující cesty s libovolným počtem přestupů.

function FINDPATHS(S_v, S_c, DC_v, $MaxDC_p$, $visitedStops$, $founded$)
  $D_v \leftarrow datum\ z\ DC_v$
  $C_v \leftarrow cas\ z\ DC_v$
  $Day_v \leftarrow den\ v\ tydnu\ z\ DC_v$
  $MaxD_p \leftarrow datum\ z\ MaxDC_p$

[1] Procedura v rámci databáze. Jedná se o vykonavatelnou část programu, která se vykonává nad daty v databázi uloženými.

```
    MaxC_p ← cas z MaxDC_p

    departureStops ← vsechna vychozi zastaveni pro S_v, D_v, C_v, MaxD_p, MaxC_p, Day_v
    for departureStop in departureStops do
        trip ← departureStop.tripId
        sequence ← departureStop.stopSequence

        currTripStops ← vsechna nasledujici zastaveni pro trip, sequence, MaxC_p
        for currTripStop in currTripStops do
            currStation ← currentTripStop.stationId
            visStopsTmp ← visitedStops.clone()
            visStopsTmp.add(currTripStop)

            if currStation == S_c then
                founded.add(visStopsTmp)
                break
            else
                tmpDC_v ← datum a cas prijezdu z currTripStop
                findPaths(currStation, S_c, tmpDC_v, MaxDC_p, visStopsTmp, founded)
            end if
        end for
    end for
end function
```

5.5 Optimalizace vyhledávacího algoritmu

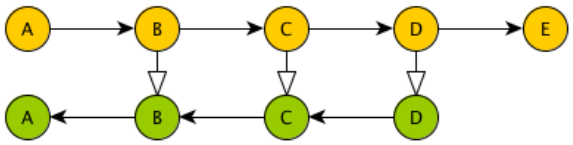
Nyní se pokusíme vyhledávání co nejvíce zoptimalizovat.

5.5.1 Maximální počet přestupů

Za prvé si všimneme, že jsme zatím žádným způsobem do vyhledávacího algoritmu nezakomponovali maximální počet přestupů. Toho dosáhneme jednoduše, můžeme například metodě přidat další vstupní parametr, který bude udávat počet vnořených volání. Tento parametr označíme jako lvl. Pro prvotní volání metody pro vyhledávání bude platit: $lvl = 0$. Při každém vnořeném volání se poté bude metoda volat s parametrem $lvl + 1$. K vnořenému zavolání sebe sama ovšem nesmí dojít, pokud by platilo $lvl >= maxPocetPrestupu$, kde $maxPocetPrestupu$ je maximální počet přestupů, který zadal uživatel jako vstupní parametr vyhledávacího algoritmu. Tímto ošetříme i situaci, kdy budeme chtít hledat zcela bez přestupů, protože rekurzivně metoda sama sebe nikdy nezavolá.

5.5.2 Vícenásobné přestupy mezi dvěma jízdami

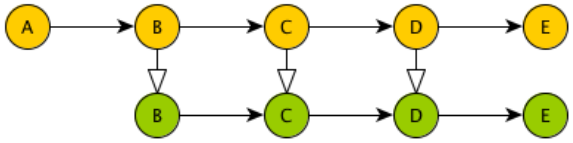
V jízdních řádech můžeme velmi často narazit na situaci znázorněnou na obrázku 5.4. Vidíme zde žlutě jízdu ze stanice A do stanice E a zeleně jízdu ze stanice D do stanice A.



Obrázek 5.4: Protisměrné jízdy a jejich přestupy

Označme si nyní žlutou jízdu z obrázku 5.4 jako $J_{1.1}$ a zelenou jízdu jako $J_{1.2}$. Vzhledem k tomu, že v naší práci neuvažujeme, že by na některých zastaveních jízdy nebylo možné do dopravního postředku nastoupit, nebo z něho vystoupit, nabízí se úvaha, že zcela stačí z $J_{1.1}$ na $J_{1.2}$ přestoupit pouze jedenkrát a to hned při první příležitosti na stanici B. Pokud totiž na $J_{1.2}$ přestoupíme kdykoli později znamená to, že budeme procházet stejné stanice, které jsme již procházeli na $J_{1.1}$.

Další častou a velmi podobnou situací v jízdních řádech je situace znázorněná na obrázku 5.5.



Obrázek 5.5: Stejnosměrné jízdy a jejich přestupy

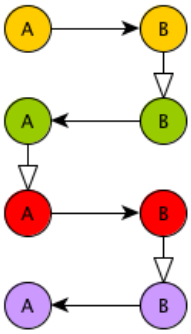
Nyní si žlutou jízdu z obrázku 5.5 označíme jako $J_{2.1}$ a zelenou jízdu jako $J_{2.2}$. I zde můžeme pozorovat, že k přestupu z $J_{2.1}$ na $J_{2.2}$ nám stačilo využít hned první příležitosti na stanici B. Právě při tomto prvním přestupu se nám totiž vytvoří nová větev programu, která následně projde všechna zastavení na $J_{2.2}$ až do stanice E. Pokud ale na $J_{2.2}$ přestoupíme později i na stanicích C a D, budeme znovu zpracovávat zastavení, která už jsme jednou zpracovali.

Předchozí pozorování nás dovádí k závěru, že mezi dvěma konkrétními jízdami stačí vždy přestoupit pouze jedenkrát, a to při první možné příležitosti. O žádná data tímto omezením nepřijdeme, protože v každém případě vždy projdeme všechna pro nás relevantní zastavení. Naopak tímto omezením vyloučíme opakované zpracovávání již zpracovaných (případ z obrázku 5.5), nebo pro nás očividně nedůležitých (případ z obrázku 5.4) zastavení.

V rámci vyhledávacího algoritmus si tedy budeme udržovat i seznam již zpracovaných jízd na cestách. Pokud bychom v první fázi (výběru výchozích zastavení) vybrali i takové zastavení, které náleží jízdě, kterou jsme již dříve zpracovali při dřívějším přestupu mezi oběma jízdami, budeme toto zastavení ignorovat a dále nezpracovávat. Sledováním již navštívených jízd v rámci konkrétních cest navíc obecně můžeme zabránit jakémukoliv budoucímu možnému návratu na jízdu, kterou jsme již v rámci konkrétní cesty zpracovali.

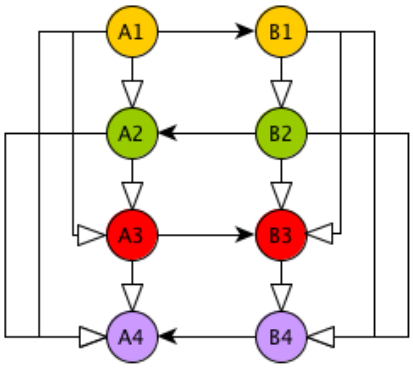
5.5.3 Traverzování mezi stanicemi

Další situací, se kterou se při hledání cest v rámci jízdních řádů setkáme, je neustálé traverzování mezi shodnými stanicemi. Ukázku takového traverzování můžeme vidět na obrázku 5.6, kde je zobrazena shodná část různých jízd.



Obrázek 5.6: Traverzování mezi shodnými stanicemi

Pokud budeme uvažovat, že naší cílovou stanicí není ani A, ani B, začne vyhledávací algoritmus při zpracovávání zastavení na stanici A v rámci žluté jízdy traverzovat mezi těmito stanicemi až do doby, kdy nebude další přestup k dispozici, nebo kdy některý z časů příjezdu převýší $MaxC_p$. Současně si uvědomíme, že v rámci tohoto konkrétního případu je možných přestupů mnohem více, protože např. hned ze žluté jízdy můžeme přímo přestoupit i na červenou či fialovou. Těchto přestupů samozřejmě vyhledávací algoritmus využije také. Všechny možné přestupy jsou znázorněny na obrázku 5.7.



Obrázek 5.7: Traverzování mezi shodnými stanicemi – všechny přestupy

Nabízí se proto úvaha, že aktuální větev vyhledávání je možné vypustit, pokud je první bezprostředně následující stanice po přestupu shodná s poslední bezprostředně předcházející stanicí před přestupem. Příkladem budiž sekvence zastavení A_1–>B_1–>B_2–>A_2. Algoritmus v tomto případě začne zpracovávat zastavení A_2, do kterého se ovšem z A_1 dostal již přímo

(A_1–>A_2). Fakt, že tímto omezením nepřijdeme o žádné výsledky, vychází přímo ze struktury jízdních řádů, nicméně si jej pojďme i dokázat. Zavedeme si nyní funkci p, která bude určovat čas příjezdu do stanice a funci v, která bude určovat čas výjezdu ze stanice. Očividně pro zastavení z obrázku 5.7 platí:

$$p(A_1) \leq v(A_1) < p(B_1) < v(B_2) < p(A_2) \leq v(A_2) \Rightarrow p(A_1) < v(A_2)$$

Nicméně v naší úvaze na toto téma můžeme pokročit ještě dále. Nabízí se otázka, zda neukončit vyhledávání aktuální zpracovávanou cestou, pokud narazíme na stanici, která již v rámci této cesty byla navštívena. Pro cestujícího to totiž znamená, že se navrátil na stanici, na které již jednou byl. Bohužel studiem konkrétních jízdních řádů zjistíme, že plošně není možné toto omezení zavést. Ne výjimečně se totiž setkáme se situací, kdy je i v rámci jednoho spoje (jízdy) navštívena jedna stanice vícekrát. Například příměstská linka autobusu č. 329 v rámci Pražské integrované dopravy (PID) projíždí v části své trasy následujícími stanicemi[2]:

- Koloděje
- Sibřina
- Květnice, rozcestí
- Květnice
- Květnice, rozcestí
- Sibřina
- Sluštice, škola

Všimneme si, že jak stanice Sibřina, tak stanice Květnice, rozcestí jsou navštíveny vícekrát.

Můžeme ovšem zavést méně striktní omezení. V rámci jedné jízdy na nalezené cestě sice musí vyhledávací algoritmus umožnit navštívení jedné stanice vícekrát, navštívení jedné stanice více jízdami současně ovšem můžeme vyloučit. Mezi jízdami, které by stejnou stanici obsahovaly, bychom totiž mohli přestoupit již dříve právě na této společné stanici. To, že přestup by byl v takovém případě možný, jsme dokázali již dříve.

Zavedením tohoto omezení opět zmenšíme prohledávaný prostor, čímž vlastní vyhledávání zrychlíme. V rámci jednotlivých iterací rekurzivního vyhledávacího algoritmu si proto budeme předávat i příslušnost již navštívených stanic k jejich jízdám. Pokud by pak stanice, která přísluší aktuálnímu zpracovávanému zastavení, byla již obsažena v dříve navštívených stanicích na jiných jízdách, můžeme vyhledávání touto větví ukončit. V tomto případě si ovšem uvědomíme, že do této podmínky nesmíme zahrnout právě stanice přestupní. Poslední zastavení na jízdě, ze které přestupujeme, totiž bude náležet stejné stanici jako první zastavení na jízdě, na kterou přestupujeme.

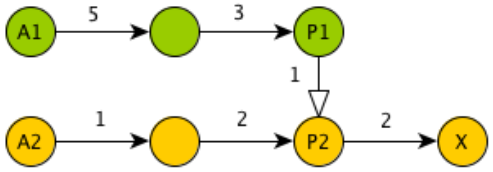
[2] <http://2gis.cz/praha/route/12948454519079111>

5.5.4 Výběr pareto-optimálních cest

U existujících vyhledávačů jsme se často setkali s faktem, že výsledkem vyhledávání jsou všechny pareto-optimální cesty. Pareto-optimalitu jsme si již popsali v úvodu práce, nicméně nyní se na ni podíváme blíže.

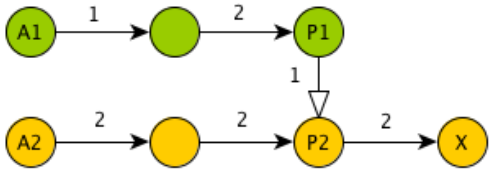
5.5.4.1 Identifikace pareto-optimálních cest

Především je třeba zdůraznit, že tento pojem jsme definovali v případě nacházení cest grafem. Momentálně ovšem graf neprocházíme a nemáme k němu ani blízko, což je dáno strukturou dat v relační databázi. Ovšem i tak můžeme nyní pareto-optimální cestu určit. Uvažujme nyní strukturu jízdního řádu z obrázku 5.8.



Obrázek 5.8: Zastavení ve stanicích a délka cest

Pokud budeme uvažovat, že jsou obě jízdy v platnosti v rámci časového intervalu pro vyhledávání, máme k dispozici dvě cesty ze stanice A do stanice X. Vzhledem k tomu, že momentálně máme jízdní řád zobrazen ve formě grafu, můžeme snadno určit délku obou cest ze stanice A do stanice X na základě délky jednotlivých hran po cestě. Cesta A_1–>P_1–>P_2–>X má tedy délku 11 a cesta A_2–>P_2–>X má délku 5. Zcela očividně tak platí, že druhá z cest je výhodnější, protože $5 < 11$. Kromě toho první z cest vyžaduje přestup na stanici P, z tohoto pohledu se její výhodnost ještě snižuje. I kdyby totiž obě cesty byly shodně dlouhé, vždy je pro cestujícího příjemnější jet tou cestou, na které nemusí přestupovat. Již dříve jsme si zavedli počet přestupů jako jedno z kritérií kvality nalezených výsledků vyhledávání. Nyní je vhodná chvíle zavést pojem *penalizace za přestup*, která počet přestupů promítne i do samotné doby jízdy, což pro nás bude při vyhledávání výhodné. Penalizaci za přestup budeme dále značit jako PP. Uvažujme nyní strukturu jízdního řádu z obrázku 5.9.



Obrázek 5.9: Zastavení ve stanicích a shodné délky cest

Na obrázku 5.9 je zobrazena stejná struktura jízdního řádu jako na obrázku 5.8, změnily se ovšem délky obou cest mezi stanicemi A a X. Obě z cest mají nyní shodnou délku 6. Cesta

A_1–>P_1–>P_2–>X ovšem vyžaduje přestup, zatímco cesta A_2–>P_2–>X jej nevyžaduje. Očividně je tedy pro cestujícího pohodlnější jet druhou z cest. Abychom mohli tuto výhodnost určit v průběhu vyhledávání, využijeme právě zavedeného pojmu penalizace za přestup. Před vlastním vyhledáváním určíme, jaká bude hodnota PP. V našem případě například mějme $PP = 2$. Pokud nyní během vyhledávání využijeme přestupu, přičteme za každý hodnotu PP k délce cesty. Tím pádem nyní můžeme znovu určit délky obou cest z obrázku 5.9. Zatímco cesta A_1–>P_1–>P_2–>X má stále stejnou délku 6, cesta A_2–>P_2–>X má rázem délku 8. Je důležité si ovšem uvědomit, že tohoto měření délek cest využíváme pouze pro účely vyhledávání. Pokud by délky jednotlivých hran, stejně jako hodnota PP byly udávány v minutách, v žádném případě to neznamená, že první z cest zabere 8 minut, zatímco druhá jich zabere pouze 6. Právě naopak – obě cesty zaberou pouze 6 minut.

Vyjasněním těchto faktů a díky definici penalizace za přestup se můžeme vrátit k výběru pareto-optimálních cest. Pokud bychom nyní spustili vyhledávací algoritmus nad daty z obrázku 5.9 (pro hledání ze stanice A do stanice X) víme, že zastavení P_2 by bylo zpracováváno v rámci obou možných cest. Současně už víme, že ačkoliv jsou obě cesty správnými výsledky vyhledávacího algoritmu, druhá z cest (A_2–>P_2–>X) je výhodnější a uživateli stačí zobrazit právě tuto cestu. Obecně uživateli vždy chceme zobrazit každé zastavení pouze na jedné cestě a to té nejvýhodnější. To nás vede k úvaze, že první z cest jsme vůbec nemuseli zpracovávat. Nyní si ještě musíme uvědomit, jak tuto cestu v rámci vyhledávání detekujeme, abychom její zpracování mohli přeskočit.

Prvním společným zastavením, které náleží oběma cestám, je zastavení P_2. V rámci tohoto zastavení jsme samozřejmě schopni určit délku cesty, kterou jsme do tohoto zastavení absolvovali od zastavení výchozího. Tuto délku již budeme uvažovat penalizovanou, tedy s připočtenými penalizacemi za přestup. Délka cesty A_1–>P_1–>P_2 je tak 6, zatímco délka cesty A_2–>P_2 je jen 4. Obě cesty nás ale dovedly na společné zastavení P_2, kterým samozřejmě dále můžeme pokračovat zcela shodně. Již jsme si řekli, že každé zastavení chceme uživateli zobrazit pouze v rámci nejlepší možné cesty. V rámci vyhledávání si proto budeme udržovat globální seznam již navštívených zastavení. U každého zastavení pak budeme sledovat i délku cesty, kterou jsme se ke každému zastavení od výchozího zastavení dostali. Tyto délky budeme udržovat s připočtenými penalizacemi za přestup.

Při zpracování každého zastavení se poté nejdříve podíváme do seznamu již navštívených zastavení. Pokud by v tomto seznamu aktuální zastavení opravdu bylo, porovnáme aktuální délku cesty s délkou cesty již uloženou. Pokud bude aktuální délka cesty větší, než již uložená hodnota, ukončíme další vyhledávání v této větvi. V opačném případě uložíme aktuální délku cesty se zpracovávaným zastavením do globálního seznamu a budeme pokračovat ve vyhledávání v aktuální větvi. Nyní si zavedeme nová označení:

- $actStopId$ ... ID aktuálního zastavení
- $actPathLength$ ... délka cesty od výchozího do aktuálního zastavení s připočtenými penalizacemi za přestup
- $globalMap$ ... glogální mapa již navštívených zastavení typu $stopId$ –> $pathLength$

V rámci každého zpracovávaného zastavení pak vykonáme následující logiku:

```
function findPaths(...)
    ...
    if globalMap contains actStopId then
        prevPathLength ← globalMap.get(actStopId)
        if prevPathLength < actPathLength then
            return
        else
            globalMap.put(actStopId, actPathLength)
        end if
    else
        globalMap.put(actStopId, actPathLength)
    end if
    ...
end function
```

Tímto postupem z vyhledávání vyloučíme cesty, které bychom stejně po nalezení všech výsledků vyfiltrovali pryč. V ideálním případě tak výsledkem vyhledávání budou všechny pareto-optimální cesty mezi výchozí a cílovou stanicí.

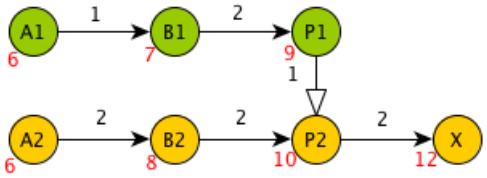
5.5.4.2 Pořadí zpracování a jeho dopady

Bohužel postup pro nalezení pareto-optimálních cest z kapitoly 5.5.4.1 platí opravdu jen hypoteticky. Snadno nahlédneme, že nutnou podmínkou pro nalezení **pouze** všech pareto-optimálních cest je procházet jednotlivá zastavení v pevně daném pořadí. Pokud se vrátíme k obrázku 5.9, dříve nebo později se dostaneme do okamžiku, kdy mezi sebou budeme porovnávat délku první a druhé cesty na zastavení P_2. Již víme, že výhodnější je cesta A_2–>P_2–>X. Aby tedy vyhledávací algoritmus ukončil vyhledávání na méně výhodné cestě A_1–>P_1–>P_2–>X, musí být uzel P_2 v rámci výhodnější cesty zpracován dříve, než tentýž uzel v rámci cesty méně výhodné.

Zavedeme si nyní novou proměnnou – *čas zastavení*, která bude nabývat času příjezdu do stanice v rámci zastavení, pokud tento čas existuje. V opačném případě bude odpovídat času výjezdu ze stanice v rámci zastavení. Budeme ji značit C_z.

Nově definovanou proměnnou ihned využijeme. Pokud bychom totiž jednotlivá zastavení zpracovávali v pořadí, které odpovídá hodnotě C_z, snadno nahlédneme, že bychom pak vždy zpracovávali jedno zastavení v rámci všech možných cest neprodleně po sobě. Nyní nám tento výrok platí například pro zastavení P_2 z obrázku 5.9. Toto zastavení bude pochopitelně zpracováváno v rámci obou možných cest, nyní ale již v přímém sledu za sebou. Bohužel jsme ještě zcela nevyřešili náš požadavek, aby se toto zastavení zpracovávalo nejdříve v rámci výhodnější cesty. Na určení pořadí zpracovávání jednotlivých zastavení tedy musí mit kromě C_z podíl i délka cesty z výchozího zastavení do aktuálního. Délka cesty se přitom bude již brát s přičtenou hodnotou PP.

Zaměříme se nyní na schéma jízdního řádu z obrázku 5.10. Červeně jsou zde značeny časy jednotlivých zastavení. Vypíšeme si nyní všechna zastavení na obou možných cestách ze stanice A do stanice X včetně času každého zastavení s přičtenou hodnotou $PP = 2$ za každý přestup.



Obrázek 5.10: Jízdy ze stanice A s časem zastavení

1. A_1 (6), B_1 (7), P_1 (9), P_2 (12), X (14)
2. A_2 (6), B_2 (8), P_2 (10), X (12)

Shrneme si, dle jakých kritérií chceme jednotlivá zastavení řadit:

1. Dle C_z braným vzestupně
2. Dle délky cesty k zastavení od výchozího zastavení včetně PP branou vzestupně

Pokud dojde k situaci, kdy budou obě hodnoty, podle kterých řadíme, v rámci dvou zastavení shodná, můžeme se pak uchýlit k náhodnému výběru. Seřazením všech zastavení na obou cestách ze stanice A do stanice X z obrázku 5.10 tak získáme následující seznam zastavení (v závorce u každého zastavení nyní bude uvedeno číslo cesty, na které zastavení uvažujeme):

$$A_1\ (1),\ A_2\ (2),\ B_1\ (1),\ B_2\ (2),\ P_1\ (1),\ P_2\ (2),\ P_2\ (1),\ X\ (2),\ X\ (1)$$

Skutečně (a nepřekvapivě) se zastavení P_2 zpracovalo dříve na výhodnější cestě. Pokud se tedy nyní v rámci vyhledávacího algoritmu při zpracování každého zastavení (na libovolné cestě) nejdříve podíváme do globální mapy již zpracovaných zastavení, jsme schopni vyhledávání nevýhodnou první cestu na zastavení P_2 ukončit (jak jsme si ukázali dříve). Tento fakt platí pochopitelně obecně, což není potřeba žádným způsobem dokazovat. Vyplývá to totiž přímo z časových závislostí v rámci jízdních řádů.

V této podobě by tedy výstupem z vyhledávacího algoritmu byly opravdu **pouze** všechny pareto-optimální cesty. Každé zastavení je totiž přednostně zpracováno na nejvýhodnější cestě.

Zpracovávání prvků v právě definovaném pořadí má další velmi výhodnou vlastnost. Protože všechna zastavení zpracováváme dle hodnot C_z vzestupně, znamená to, že po nalezení prvního výsledku (první cesty z výchozí do cílové stanice) máme jistotu, že tato cesta je nejlepší možná ze všech existujících cest. Mezi aktuálně zpracovávaným zastavením (Z_a) a jakýmkoliv dalším zastavením, zpracovaným v následujících krocích (sourhnně označíme Z_o) totiž zcela jistě platí nerovnost $C_z\ (Z_o) >= C_z\ (Z_a)$. Pokud bychom tedy chtěli uživateli vrátit pouze n nejlepších výsledků (což je velmi častý případ), po nalezení právě n prvních výsledků můžeme celé vyhledávání ukončit. Tato skutečnost vede k výraznému zrychlení vyhledávání zejména v situacích, kdy v rámci časového intervalu pro vyhledávání existuje větší počet možných cest. Prohledávaný prostor se tak totiž výrazným způsobem zmenší.

5.5.4.3 Pareto-optimální cesty nad relační databází

V našem návrhu řešení pomocí rekurzivního algoritmu očividně není právě požadované pořadí prvků definované v kapitole 5.5.4.2 dodrženo. V současné chvíli se po vybrání prvního výchozího zastavení a dalších zastavení na vybrané jízdě ihned začneme větvit ve vyhledávání v rámci rekurze. Toto pořadí vybírání zastavení ke zpracovávání je sice možné zcela přesně předpovídat (s ohledem na princip rekurze), nicméně samo vybrání zastavení nám neříká nic o jeho čase. Z tohoto pohledu tedy momentálně algoritmus vybírá zastavení prakticky náhodně. Abychom dodrželi požadované pořadí, nesměli bychom ihned začít s rekurzivním voláním, ale právě naopak bychom si již nalezená zastavení museli ukládat stranou a až následně je vybírat v požadovaném pořadí.

K tomuto účelu bychom mohli využít například prioritní frontu [15]. V našem případě by priorita prvků byla udána dvojicí již definovaných parametrů. Celý algoritmus bychom pak museli výrazně pozměnit. Po vybrání výchozích zastavení bychom všechna uložili do prioritní fronty a začali se zpracováním prvního zastavení na řadě. Zde nastává otázka, co dál. V ideálním případě bychom vybrali první následující zastavení po aktuálním vybraném zastavení na příslušné jízdě. Toto zastavení bychom opět vložili do prioritní fronty a vybrali další prvek v pořadí. To by ovšem znamenalo volání jednoho SQL dotazu pro každé zastavení v rámci jízdního řádu, což by výrazným způsobem navýšilo počet volání databáze. Druhým možným řešením by proto bylo, stejně jako to děláme nyní, vybrat všechna následující zastavení na jízdě po aktuálním vybraném zastavení. Všechna tato zastavení bychom postupně vložili do prioritní fronty a nakonec vybrali další prvek ke zpracování. Tímto způsobem by se nám ovšem do prioritní fronty v každém kroku dostávalo vícenásobně více prvků, než by z ní bylo vybíráno. Snadno by pak došlo k situaci, kdy bychom měli významnou část všech zastavení z databáze vloženou v prioritní frontě, která data uchovává v rámci operační paměti. Dostáváme se tak do situce, kdy by pro nás bylo výhodnější mít data o jízdních řádech v operační paměti permanentně. Cílem návrhu tohoto vyhledávacího algoritmu je ale pravý opak – nemít data uložena v operační paměti, ale vyhledávat přímo nad relační databází. Samozřejmě už v současné době jsme si zavedli optimalizace, které vyžadují uchovávání mezivýsledků v operační paměti. Tyto mezivýsledky se ovšem svým rozsahem v žádném případě nerovnají ukládání význačné části všech zastavení z databáze do prioritní fronty, kdy každé zastavení navíc může být uvažováno v rámci velkého počtu cest, a tudíž uloženo mnohonásobně.

Vzhledem k těmto skutečnostem nebude v rámci tohoto algoritmu prioritní fronta použita a prvky nadále budou vybírány v pořadí udaném rekurzivním voláním. Za cenu, že tak najdeme i neoptimální výsledky (které vyfiltrujeme až posléze), se zbavíme nutnosti uchovávání velkého množství dat v operační paměti. V nejhorším případě tak ovšem nalezeneme vždy nejdříve všechny neoptimální výsledky, než ty pareto-optimální. Zda k této nejhorší možné variatně někdy dojde, je pak dáno vlastní podobou uložených jízdních řádů. Statisticky vzato bychom ale museli mít extrémní smůlu, aby nastala.

Kapitola 6

Vyhledávací algoritmus nad grafovou databází

V kapitole 5 jsme navrhli vyhledávací algoritmus nad relační databází. Nyní navrhneme algoritmus nad databází grafovou. Nejdříve navrhneme strukturu grafu, která bude obsahovat všechny pro nás důležité informace z jízdních řádů a nad kterou bude možné vyhledávat výsledky v takové podobě, aby odpovídaly nárokům na vyhledávač definovaným v kapitole 4.3.2. Poté se seznámíme s existujícími algoritmy pro vyhledávání nejkratších cest grafem a navrhneme vlastní verzi vyhledávání, která bude reflektovat všechny nároky na vyhledávač kladené.

6.1 Struktura grafu

Připomeňme si, že jediná data, která máme k dispozici, jsou data ve formátu GTFS. Při výběru správné struktury grafu tedy neopomeneme fakt, že čím bude struktura grafu (jednotlivých vrcholů a hran, které je spojují) více odpovídat struktuře dat v GTFS formátu, tím snazší pro nás bude data do grafu ztransformovat.

Na téma výběru správné struktury grafu pro účely uložení a následné vyhledávání v datech z jízdních řádů vznikla v minulosti již řada studií. Výsledkem těchto studií je návrh dvou základních modelů grafů, které jsou pro tyto účely vhodné.

6.1.1 Time-dependent model

Time-dependent model poprvé publikovali autoři Ariel Orda a Raphael Rom v [19] a [20]. Další studie tohoto modelu nalezneme například v [31] nebo [2].

V tomto modelu vrcholy grafu reprezentují stanice. Mezi vrcholy (tedy stanicemi) existuje hrana, pokud mezi těmito stanicemi v rámci jízdního řádu existuje přímé spojení. Pokud mezi dvěma vrcholy existuje více přímých spojení, hrana zůstává vždy jen jedna. Z toho vyplývá, že hrana nemůže mít explicitně přiřazenou délku. Tato délka se musí dopočítávat až při traverzování grafem, kdy se určí na základě času výjezdu (C_v) z výchozího vrcholu a času příjezdu (C_p) do cílového vrcholu. Délka hrany pak nabyde hodnoty C_p - C_v.

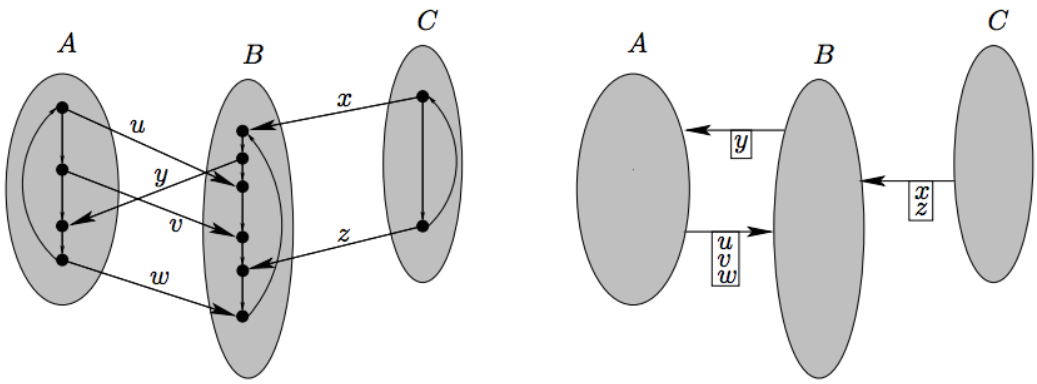
6.1.2 Time-expanded model

Time-expanded model byl publikován mimo jiné v [21] nebo [32]. Další studie tohoto modelu nalezneme i v již zmíněných publikacích [31] nebo [2].

V tomto modelu vrcholy grafu reprezentují *události* na stanicích. Událostí v tomto případě myslíme příjezd do stanice nebo výjezd z ní. Dříve v této práci jsme si zavedli pojem zastavení. Uvědomíme si, že zastavení je vlastně nadřízená jednotka události, protože v sobě zahrnuje jak příjezd, tak výjezd (pokud v rámci daného zastavení obě události existují). Jednotlivé události (tedy nyní vrcholy) jsou propojeny dvěma typy hran. Jedná se o hrany *přestupní*, které reprezentují možnost přestupu mezi jízdami v rámci jedné stanice a hrany *přejezdové*, které reprezentují možnost přepravy dopravním prostředkem z jedné stanice do druhé. Každý vrchol má přiřazen čas, a to buď C_v nebo C_p v závislosti na druhu události. Tento čas označíme jako *absolutní čas vrcholu* a budeme jej značit $AbsC_v$. V rámci stanice pak nejsou propojeny přestupními hranami všechny události, které přestup umožňují, ale propojeny jsou pouze dvě bezprostředně po sobě následující události v rámci dne. Poslední událost v rámci jednoho dne je pak přestupní hranou propojena s první událostí dne následujícího. Protože všechny hrany propojují vrcholy, které mají vždy pevně stanovený čas, mohou mít i samotné hrany explicitně přiřazenou délku. Pokud je $AbsC_{v1}$ absolutní čas výchozího vrcholu a $AbsC_{v2}$ absolutní čas cílového vrcholu, délku hrany určíme jako $|AbsC_{v1}$ - $AbsC_{v2}|$.

6.1.3 Srovnání modelů

Na obrázku 6.1 můžeme vidět znázorněné oba právě představené modely grafu.



Obrázek 6.1: Time-expanded (vlevo) a time-dependent model grafu se třemi stanicemi A, B, C. Tři jízdy spojují stanici A se stanicí B (jízdy u, v, w), jedna jízda spojuje stanici B a A (y) a dvě jízdy spojují stanici C a B (x, z). Zdroj: [32]

Z popisů obou modelů i grafického znázornění ihned můžeme usoudit, že time-dependent model bude obsahovat výrazně méně vrcholů i hran, než tomu bude u time-expanded modelu. Typicky se totiž v rámci jízdního řádu během dne na jedné stanici uděje výrazně více událostí než jedna. Na druhou stranu je to právě time-expanded model, který přirozeně zachycuje strukturu jízdního řádu mnohem elegantněji.

Autoři práce [3] dokázali, že vyhledávání cest v time-dependent modelu pomocí *Dijkstrova algoritmu* (popsán v kapitole 6.2) je pro specifické situace rychlejší, než vyhledávání v time-expanded modelu. V případě, že ale chceme do vyhledávání zavést více parametrů, podle kterých cesty poměřujeme a vyhledáváme (počet přestupů, minimální čas na přestup...), dokázali autoři studie [25], že v těchto případech se rozdíly v rychlosti smazávají a naopak je výhodnější využívat time-expanded modelu. Tento závěr je i detailněji rozveden v [2].

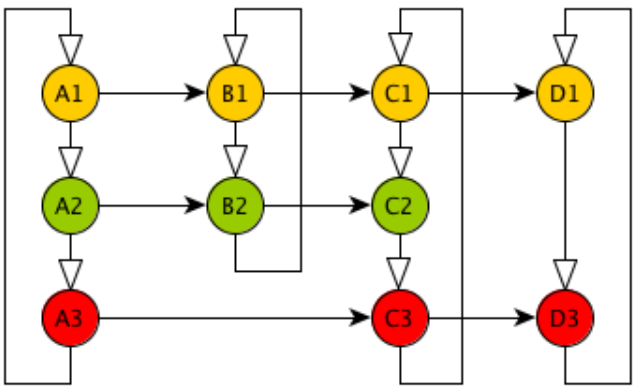
Již nyní si také můžeme všimnout, že time-expanded model je ve své podstatě strukturálně velmi podobný našim datům z GTFS balíku. Pokud bychom uvažovali za vrcholy grafu jednotlivá zastavení namísto událostí (v tomto případě je sjednocení snadné), vrcholy grafu nyní můžeme vytvořit přímo z jednotlivých zastavení. Přejezdové hrany pak můžeme doplnit velmi snadno v rámci každé jízdy a vzhledem k tomu, že jsme schopni mezi sebou časově porovnat zastavení v rámci jedné stanice, není problém ani doplnit hrany přestupní.

Vzhledem k těmto okolnostem byl pro účely této práce zvolen time-expanded model, který je pro nás v mnoha ohledech vhodnější. Navíc se volbou tohoto modelu neubíráme cestou, která by byla v některé ze studií označena za neefektivní. Právě naopak volíme řešení, které se v rámci mnoha studií citovaných dříve jeví jako nejlepší možné.

6.1.4 Zvolená struktura grafu

Vzhledem k tomu, že již máme vybranou strukturu grafu, kterou budeme používat, můžeme se pustit do jejího návrhu s námi dostupnými daty, tedy daty z GTFS balíku.

Abychom se co nejvíce přiblížili GTFS struktuře dat a také graf zjednodušili, zavedeme opatření, které jsme navrhli již dříve. Jednotlivé vrcholy grafu pro nás nebudou události ve stanicích, ale budou jimi přímo jednotlivá zastavení ve stanicích. Za prvé tímto zjednodušením snížíme počet vrcholů v grafu téměř na polovinu a za druhé se tím graf i opticky zpřehlední. Tento mírně upravený expanded-model tedy bude mít podobu zobrazenou na obrázku 6.2.



Obrázek 6.2: Základní struktura grafu

Všimneme si, že již od začátku uvažujeme, že zastavení uložená v grafu budou mít pouze informaci o čase a nikoliv o datu. Tím pádem jsme byli schopni definovat přestupní hranu

z posledního zastavení v rámci dne do prvního zastavení v rámci dne. V grafu tak jsou uložena zastavení, která přímo odpovídají zastavením z GTFS balíku. Pochopitelně bychom mohli graf i rozšířit a ke každému zastavení přidat i datum. Tím pádem by pak bylo každé zastavení uloženo tolikrát, v kolika dnech platnosti jízdního řádu daný spoj jede. Toto řešení je intuitivnější, protože by nám umožnilo velmi elegantně propojovat jednotlivá zastavení v jízdním řádu. Tento model byl dokonce popsán jako speciální případ time-expanded modelu v [31], kde je označován názvem *fully time-expanded graph*. Jak jsme ale konstatovali již v kapitole 4.3.1, je pro nás z hlediska velikosti dat výhodnější používat model dat pouze pro jeden den. Nicméně to pro nás znamená, že nějakým způsobem budeme muset v grafu udržovat informaci o tom, ve kterých dnech který spoj jede. Později se tedy nevyhneme přidání dalších vrcholů.

Vzhledem k tomu, že jsme tedy jako vrcholy grafu určili jednotlivá zastavení, můžeme už i určit atributy jednotlivých vrcholů typu zastavení. Tyto atributy budou prakticky odpovídat atributům, o kterých již víme, že je jednotlivá zastavení mají k dispozici. Konkrétně jimi budou (již v českých překladech):

- ID zastavení
- ID stanice
- ID jízdy
- Čas příjezdu C_p
- Čas výjezdu C_v

Ne všechna zastavení mají definovanou jak hodnotu C_v, tak hodnotu C_p. V tomto případě samozřejmě bude přítomna pouze ta existující. Parametry C_v a C_p navíc můžeme definovat i v rámci celých jízd. V tomto kontextu je C_v čas výjezdu z prvního zastavení na jízdě a C_p je čas příjezdu do posledního zastavení na jízdě.

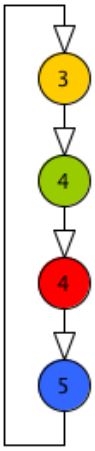
Nyní si detailněji rozebereme strukturu grafu z obrázku 6.2. Již při návrhu algoritmu pro relační databázi jsme shodné značení používali, protože je pro zaznamenávání informací z jízdních řádů velmi vhodné. Barevně shodné vrcholy jsou zastavení v rámci jedné jízdy. Zastavení, která jsou pod sebou (např. všechna zastavení začínající A), jsou zastavení v rámci jedné stanice (zde tedy právě stanice A). Uložení jízdy do grafu pro nás nepředstavuje žádný problém. Zastavení totiž má informaci o příslušnosti ke stanici, jízdě a navíc i pořadí v rámci jízdy. Stačí tedy jednotlivá zastavení uložit jako vrcholy a bezprostředně po sobě následující zastavení propojit přejezdovou hranou. Menší problém nastane při propojování vrcholů v rámci jedné stanice přestupními hranami. Zde je nutné propojit jednotlivá zastavení ve správném pořadí. Dříve jsme si definovali absolutní čas vrcholu. To ovšem bylo ještě v době, kdy jsme jako vrcholy uvažovali události a nikoli zastavení. Nyní tedy zavedeme nový pojem *absolutní čas zastavení* $AbsC_z$, který bude definován jako C_v, pokud tento čas existuje. V opačném případě bude definován jako C_p. Pokud nyní všechna zastavení v rámci jedné stanice seřadíme podle parametru $AbsC_z$ a každé dva po sobě bezprostředně následující vrcholy propojíme přestupní hranou a navíc propojíme poslední vrchol s prvním, dostaneme se do situace z obrázku 6.2.

6.1.4.1 Kontrola struktury grafu

Nyní musíme zkontrolovat, zda propojením vrcholů definovaným v kapitole 6.1.4 získáme správný model grafu. U přejezdových hran není co řešit, přejezd je v každém případě možný, protože jde o informaci přímo z jízdního řádu. Konkrétně tedy např. zastavení A_1 má pořadí v rámci jízdy nižší než zastavení A_2, což jsme zjistili přímo z GTFS balíku dat. U přestupních hran ovšem sami musíme zaručit, že je možné mezi danými vrcholy přestoupit.

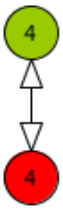
Pro C_v a C_p v rámci jednoho zastavení vždy očividně platí $C_p <= C_v$. Pro hodnotu $AbsC_z$ každého zastavení tedy tedy platí $AbsC_z >= C_v >= C_p$. Pokud není C_v nebo C_p v rámci zastavení definováno (k čemuž může dojít pouze na výchozím nebo cílovém zastavení), pochopitelně pak bude tato hodnota z naší nerovnosti vynechána. Dále si uvědomíme, že k úspěšnému přestupu mezi dvěma jízdami (označme J_1 a J_2) je nutné, aby C_p (J_1) $<= C_v$ (J_2). Pokud je tato nerovnost dodržena, je mezi jízdami umožněn přestup. Z vlastnosti $C_p <= C_v$ pak ale i přímo vyplývá, že pokud pro pro J_1 a J_2 platí C_v (J_1) $<= C_v$ (J_2), je přestup mezi J_1 a J_2 také zcela jistě možný. Proto pokud jednotlivá zastavení v rámci stanice seřadíme dle času $AbsC_z$ a následně je postupně propojíme, bude mezi všemi takto propojenými zastaveními vždy přestup možný.

Bohužel předchozí způsob propojení nás obral o některé možné přestupy, které v rámci jízdního řádu mohou existovat. Za prvé velmi často dojde k situaci, kdy se budou hodnoty $AbsC_z$ více zastavení v rámci stanice rovnat. V tom případě nám po seřazení zastavení dle hodnoty $AbsC_z$ v rámci jedné stanice může vzniknout situace zobrazená na obrázku 6.3.



Obrázek 6.3: Struktura grafu v rámci jedné stanice. Čísla ve vrcholech udávají hodnotu $AbsC_z$.

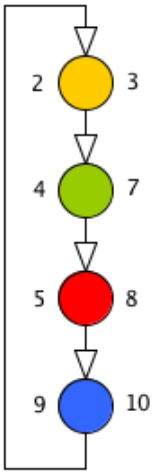
Na obrázku 6.3 vidíme, že máme správně propojený zelený vrchol s červeným přestupní hranou. Bohužel v tomto případě by měl být propojen i červený vrchol se zeleným, protože mají shodnou hodnotu $AbsCz$. Nyní máme více možností, jak tento problém vyřešit. Jedno z možných řešení je, že můžeme hranu mezi zeleným a červeným vrcholem navrhnout jako obousměrnou, jak je zobrazeno na obrázku 6.4.



Obrázek 6.4: Obousměrná hrana mezi vrcholy

Pokud nyní tuto hranu zakomponujeme do grafu na obrázku 6.3, vyřešíme tak elegantně i možnost přestupu z červeného vrcholu na zelený. Nicméně si uvědomíme, že jsme při návrhu struktury grafu zatím žádným způsobem neuvažovali parametr PMP (počet minut nutných na přestup), který v rámci vyhledávacího algoritmu budeme chtít využívat, jak jsme si řekli již v kapitole 4.2.1. Tento parametr sice žádným způsobem ve statické části grafu nemůže být zakomponován, v kapitole 4.2.1 jsme si ale již určili jeho hodnotu ($PMP = 1$ minuta). Protože je tento parametr nenulový, musíme již nyní počítat s tím, že možnost přestupu bude v těchto případech stejně muset kontrolovat samotný vyhledávací algoritmus.

Aby se nám situace zkomplikovala, v rámci jízdních řádů bohužel existují i poměrně vzácné, ale ne nemožné situace, na které je náš současný model nedostačující. Tuto situaci můžeme pozorovat na obrázku 6.5.

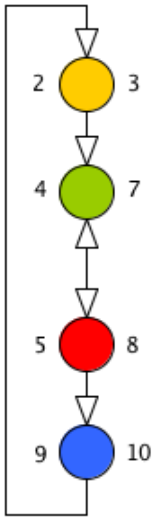


Obrázek 6.5: Zastavení v rámci jedné stanice. Čísla u vrcholů značí C_p do zastavení (vlevo) a C_v ze zastavení. Pro všechny C_v nyní platí $C_v = AbsC_z$.

Obrázek znázorňuje zastavení v rámci jedné stanice. Zastavení byla správně seřeazena dle hodnoty $AbsC_z$ a následně propojena přestupními hranami. Bližším studiem tohoto grafu ale zjistíme, že obsahuje vážnou chybu. C_p do červeného vrcholu je 5 a C_v ze zeleného vrcholu je 7. Očividně je tedy umožněn přestup mezi jízdami, na kterých tato zastavení leží. Bohužel

v našem grafu není momentálně přestup mezi červenou a zelenou jízdou umožněn. Toto je samozřejmě závažná chyba, kterou musíme odstranit.

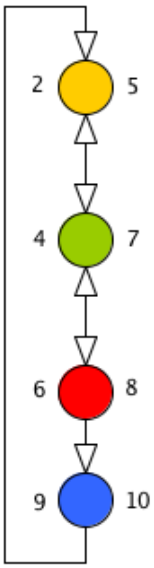
K řešení využijeme již dříve navrhnuté možnosti – obousměrných hran. V tomto případě nám ani nic jiného nezbývá, protože přestup je opravdu fyzicky možný jak ze zelené na červenou, tak z červené na zelenou jízdu. Pokud tedy při tvorbě grafu a propojování zastavení v rámci jedné stanice přestupními hranami narazíme na situaci, kdy po seřazení zastavení dle $AbsC_z$ bude mezi některou dvojicí vrcholů V_1, V_2 (kdy po seřazení V_1 předchází V_2) platit $C_p\ (V_2) < C_v\ (V_1)$, musíme hranu mezi V_1 a V_2 vytvořit obousměrnou. Pokud tento postup aplikujeme při vytváření grafu z obrázku 6.5, vznikne nám tak model zobrazený na obrázku 6.6.



Obrázek 6.6: Zastavení v rámci jedné stanice s obousměrnou hranou mezi vrcholy

Dalším studiem návazností v jízdních řádech se ale ukazuje, že ani toto řešení ještě správným způsobem neošetřuje všechny možnosti. Pokud budeme uvažovat schéma jízdního řádu z obrázku 6.7, zjistíme, že je sice správně vytvořené s ohledem na právě definovaný postup, obsahuje ale logický nesmysl.

Umožňuje totiž přestup z červeného na žlutý vrchol, který ale časově není možný. Toto pozorování nás vede k závěru, že o možnosti či nemožnosti přestupu bude muset rozhodovat až samotný vyhledávací algoritmus. Všechny vrcholy v rámci jedné stanice proto seřadíme dle hodnoty $AbsC_z$ a jednotlivé dvojice vrcholů propojíme vždy obousměrnými hranami. Po každém využití přestupní hrany v rámci vyhledávacího algoritmu pak musíme ověřit, zda je přestup možný (zjistíme z časů C_p a C_v jednotlivých vrcholů). Pokud ne, můžeme traverzování po přestupní hraně v rámci této stanice a tohoto směru ukončit. Nutnost tohoto postupu je daň za zjednodušení struktury grafu, kdy jako jednotlivé vrcholy nepoužíváme události na stanicích, ale přímo jednotlivá zastavení. Nicméně i tak jsme situaci stále zjednodušili, protože díky seřazení vrcholů dle $AbsC_z$ od nás tento postup nyní vyžaduje vždy jen jedno otestování v rámci traverzování navíc. Pokud bychom tedy uvažovali, že vrcholy



Obrázek 6.7: Zastavení v rámci jedné stanice s obousměrnými hranami mezi vrcholy

na obrázku 6.7 jsou všechny propojeny obousměrnými hranami a výchozí vrchol je pro nás červený, traverzování směrem nahoru se v každém případě zastaví po otestování C_v na žlutém vrcholu a dále tímto směrem nebude pokračovat. Protože k vlastnímu traverzování směrem nahoru v rámci zastavení náležících jedné stanici bude docházet velmi zřídka, budeme pro zjednodušení v obrázcích grafů nadále používat pouze jednosměrné přestupní hrany.

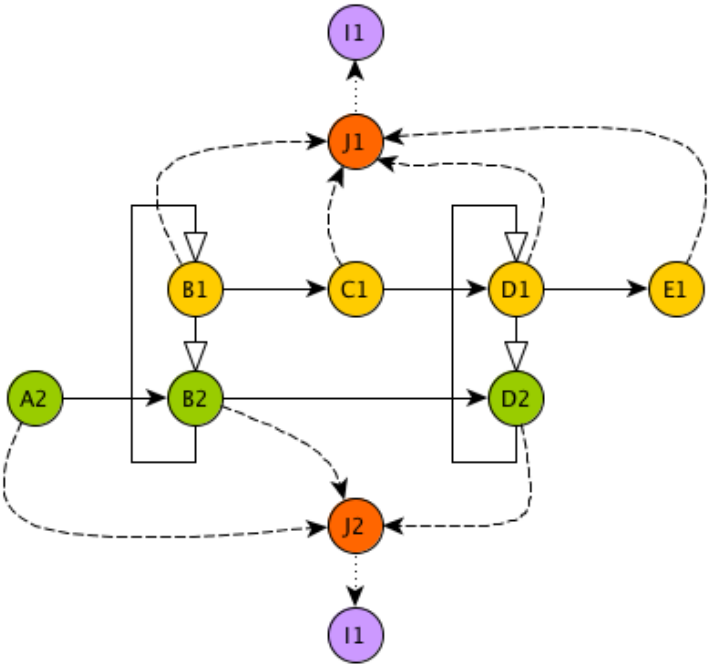
6.1.4.2 Komplexní schéma grafu

V grafu tedy již nyní máme správně uloženou kompletní strukturu jízd. Vzhledem k tomu, že využíváme model dat pouze pro jeden den, řekli jsme si už dříve, že do grafu bude nutné přidat další vrcholy, které budou obsahovat informace o platnosti jednotlivých jízd (a tedy i konkrétních zastavení těchto jízd). Potřebujeme tedy do grafu dostat vrcholy, které budou reprezentovat intervaly platnosti definované v GTFS balíku dat. Tyto vrcholy pak musí obsahovat všechny atributy, které jsme si pro interval platnosti určili již při definici relačního schématu dat. Atributy si pro připomenutí zopakujeme (s použitím českých názvů):

- ID intervalu platnosti
- Datum platnosti od
- Datum platnosti do
- Pondělí
- Úterý
- Středa

- Čtvrtek
- Pátek
- Sobota
- Neděle

Jednotlivé dny v týdnu nabývají pouze hodnoty *true* nebo *false*. Vrchol intervalu platnosti by měl být logicky referencován z objektu jízdy. Vrcholy jízd ale v grafu momentálně také nemáme, protože jsme si zatím vystačili s jednotlivými zastaveními. Nyní máme dvě možnosti. Do grafu přidáme i vrcholy, které budou reprezentovat jednotlivé jízdy a tyto vrcholy poté propojíme s příslušným vrcholem intervalu platnosti, nebo můžeme jednotlivá zastavení ihned propojit s intervalem platnosti. Zvolíme první řešení, protože po změně intervalu platnosti pro některou z jízd nám bude stačit nahradit v grafu pouze jednu hranu. Navíc je tato struktura mnohem přehlednější. Aktuální návrh struktury grafu můžeme vidět na obrázku 6.8.



Obrázek 6.8: Struktura grafu s přidanými vrcholy jízd (vrcholy J) a intervaly platnosti (vrcholy I). Hrana znázorněná čárkovaně značí příslušnost zastavení k jízdě. Hrana znázorněná tečkovaně značí příslušnost jízdy k intervalu platnosti.

Schéma grafu z obrázku 6.8 budeme považovat pro naše účely za finální. Máme v něm totiž uloženy všechny informace, které budeme při vyhledávání cest potřebovat. Ještě si připomeneme, že stejně jako v relačním schématu budou i zde vrcholy, které reprezentují jízdy, obsahovat pouze jediný parametr, kterým je ID jízdy. Žádná další informace na těchto

vrcholech totiž není potřeba. Současně si povšimneme chybějících vrcholů pro stanice. Tohoto zjednodušení se nyní můžeme dopustit, protože data o stanicích, která do vyhledávacího algoritmu žádným způsobem nezasahují (pouze se v nich vyhledá ID počáteční a cílové stanice), již máme v relační databázi. Pokud bychom chtěli mít všechna data v grafové databázi, přidali bychom vrcholy zastavení stejným způsobem, kterým jsme přidali vrcholy jízd. Vrcholy stanic by pak obsahovaly shodné atributy, které jsme u stanic definovali pro relační databázi.

6.2 Algoritmy pro vyhledávání nejkratších cest grafem

Pravděpodobně nejznámějším a nejpoužívanějším algoritmem pro hledání nejkratších cest mezi vrcholy grafu je *Dijkstrův algoritmus* [6]. Tento algoritmus můžeme využít pro hledání nejkratší cesty vždy, když neobsahuje záporně ohodnocené hrany. V rámci jízdních řádů zápornou hranu nikdy neuvažujeme, tuto podmínku tedy můžeme považovat za splněnou. Originální verze algoritmu byla určena pro hledání nejkratší cesty mezi dvěma vrcholy, dá se ale snadno modifikavat takovým způsobem, že nalezne nejkratší cesty od výchozího vrcholu ke všem ostatním vrcholům v grafu. Krátce si nyní funkčnost algoritmu představíme:

1. Přiřaď každému vrcholu v grafu vzdálenost od výchozího vrcholu na hodnotu ∞. Výjimkou je sám výchozí vrchol, kde vzdálenost nastav na 0.
2. Všechny uzly označ jako nenavštívené. Výchozí uzel označ jako aktuální uzel a vytvoř množinu zatím nenavštívených uzlů, která bude obsahovat všechny uzly kromě aktuálního.
3. Projdi všechny sousedy aktuálního uzlu a spočti jejich vzdálenost od výchozího uzlu. Vzdálenost je udána součtem délek všech hran od výchozího uzlu až k uzlu právě zpracovávanému. Pokud je tato vzdálenost kratší než již dříve zaznamenaná vzdálenost k tomuto uzlu, ulož si tuto vzdálenost k danému uzlu.
   Po zpracování všech sousedů aktuálního uzlu vyjmi aktuální uzel z množiny nenavštívených uzlů (jinak řečeno označ aktuální uzel jako navštívený).
4. Pokud jsme již označili cílový uzel jako navštívený, nebo nejmenší aktuální vzdálenost mezi uzly v množině nenavštívených uzlů je ∞, ukonči algoritmus.
5. Vyber z množiny nenavštívených uzlů ten s nejkratší vzdáleností od výchozího uzlu, označ jej za aktuální a pokračuj bodem 3.

Po doběhnutí algoritmu je jeho výstupem délka nejkratší cesty z výchozího uzlu do cílového uzlu (nebo všech ostatních v závislosti na zvolené variantě). Popis algoritmu byl volně přepsán z [26].

Časová složitost Dijkstrova algoritmu je přímo závislá na výběru struktury, kterou používáme pro ukládání nenavštívených uzlů. Z této struktury totiž vybíráme uzly dle jejich vzdálenosti od uzlu výchozího. Pro tyto účely se výborně hodí prioritní fronta. S využitím prioritní fronty implementované jako Fibonacciho halda [9] je časová složit algoritmu rovna $O(|E| + |V|log|V|)$, kde $|E|$ je počet hran a $|V|$ je počet vrcholů v grafu.

Pro vyhledávání v našem grafu nicméně není možné přímou implementaci Dijkstrova algoritmu využít. Všimneme si totiž, že tento algoritmus pro správnou funkčnost požaduje přímou porovnatelnost dvou uzlů na základě jejich vzdálenosti od výchozího uzlu.

V kapitole 5.5.4 jsme si ale ukázali, že v rámci jízdních řádů potřebujeme porovnávat jednotlivé cesty na základě více kritérií. Mezi tato kritéria patří nejen vzdálenost uzlu od výchozího, ale také čas příjezdu do tohoto uzlu či počet přestupů v rámci cesty. Dále jsme si v téže kapitole ukázali, že naším cílem je v rámci dat najít všechny pareto-optimální cesty a nikoliv jen nejkratší cestu. Po bližším zkoumání kapitoly 5.5.4 tak zjistíme, že jsme algoritmus pro vyhledávání všech pareto-optimálních cest v grafu již navrhli a můžeme jej tedy využít. Současně si povšimneme výrazné podobnosti námi navrženého algoritmu s právě představeným Dijkstrovým algoritmem.

6.3 Vyhledávání cest v grafu

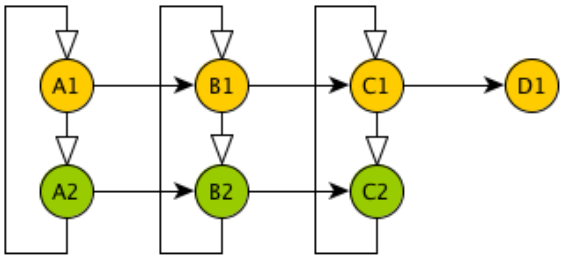
Požadavky pro vyhledávání spojů nad grafovou databází se pochopitelně neliší od požadavků pro vyhledávání nad databází relační. V ideálním případě by nám naopak oba způsoby vyhledávání nad shodnými daty měly pro stejné vstupní parametry vrátit zcela shodné výsledky. Protože jsme již postup vyhledávání jednou popsali v kapitole 5, nemá nyní smysl tento postup znovu celý popisovat. V této kapitole proto naopak budeme vycházet z již nabytých poznatků pro vyhledávání z kapitoly 5. Může tak dojít k situaci, kdy některé zřejmé skutečnosti již zmíněné v kapitole 5 zde nebudou znovu rozváděny.

6.3.1 Nalezení počátečních zastavení

Nalezení výchozích zastavení bude muset splňovat všechna pravidla definovaná v kapitole 5.3.1. Jediný rozdíl oproti postupu ve zmíněné kapitole je, že nyní hledáme vrcholy grafu. Konkrétní způsob, kterým tyto vrcholy z grafové databáze vybereme, si představíme až v implementační části v kapitole 7. Způsob dotazování nad námi zvolenou databází Neo4j je totiž specifický právě jen pro tuto databázi. V případě jazyka SQL toto neplatilo, protože se jedná o standardizovaný dotazovací jazyk nad relačními databázemi obecně a nikoli jen nad jednou konkrétní implementací.

Na tomto místě si tak nově pouze určíme způsob, kterým počáteční zastavení vybereme. Máme totiž dvě celkem rovnocenné možnosti. Uvažujme nyní schéma jízdního řádu z obrázku 6.9.

Pokud se v rámci jízdního řádu zobrazeného na obrázku 6.9 budeme chtít dostat ze stanice A do stanice C, je možné vybrat počáteční zastavení dvojím způsobem. V kapitole 5.3.1 jsme si definovali, že vybereme všechna počáteční zastavení, ze kterých je možné v rámci jízdního řádu dále vyhledávat. Tímto způsobem můžeme provést výběr počátečních zastavení pochopitelně i v rámci grafu. Do množiny počátečních zastavení bychom tak vybrali zastavení A_1 i zastavení A_2. V tomto případě si uvědomíme, že traverzování v grafu musí od každého počátečního zastavení pokračovat *pouze* přejezdovou hranou. Pokud bychom totiž v rámci počátečního zastavení k traverzování využili i hrany přestupní, dostali bychom se tak ze zastavení A_1 na zastavení A_2, které ale již bylo taktéž vybráno do množiny počátečních zastavení. Toto je pochopitelně stav, který nechceme připustit.



Obrázek 6.9: Ilustrační schéma jízdního řádu

Druhou možností by bylo vybrání pouze prvního výchozího zastavení ze všech možných. První by pro nás v tomto případě znamenalo to s nejnižším časem výjezdu (C_v). V rámci grafu bychom pak od tohoto zastavení ihned začali traverzovat i po přestupních hranách, a dostali bychom se tak nakonec i do všech zbývajících počátečních zastavení, která jsou v rámci vyhledávání možná.

Z představených řešení bylo zvoleno to první, které je pro naše účely vhodnější. V rámci každé cesty při vyhledávání bude totiž jednoznačně definováno počáteční zastavení, což u druhého způsobu neplatí. Výchozí uzel by pro nás totiž nebyl uzlem, který reprezentuje zastavení, na kterém jsme nastoupili do dopravního prostředku, ale právě naopak by byl pro všechny cesty shodný a museli bychom podle toho složitě upravovat vyhledávací algoritmus.

6.3.2 Hledání cest od počátečních zastavení

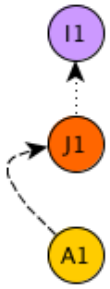
Při návrhu struktury grafu jsme zavedli několik typů hran: přejezdové, přestupní a dále hrany spojující zastavení s konkrétní jízdou a hrany spojující jízdy s intervaly platnosti. Nejdůležitější jsou pro nás nyní hrany přejezdové a přestupní, protože po nich budeme traverzovat mezi zastaveními. Při traverzování samozřejmě musíme dodržovat orientace jednotlivých hran.

6.3.2.1 Kontroly na vrcholech

Pro každý vrchol, na který se v rámci traverzace dostaneme, musíme zkontrolovat, zda pro jeho čas (C_z) neplatí $C_z > MaxC_p$. Pokud by totiž tato nerovnost byla splněna, musíme traverzování v aktuální větvi ukončit (dostali jsme se mimo platnost intervalu pro vyhledávání spojení). Tímto způsobem nepřijdeme o žádné výsledky, protože struktura našeho grafu je tvořena takovým způsobem, že pro každé dva vrcholy V_1, V_2, mezi kterými existuje přejezdová hrana platí, že pokud V_1 –> V_2, pak také nutně C_z (V_1) <= C_z (V_2). Zadefinujeme si nyní novou proměnnou – *inverzní čas zastavení* C_{zi}, která bude definována jako C_v ze zastavení, pokud tato hodnota existuje, jinak bude definována jako C_p do zastavení. Pro každé dva vrcholy V_1, V_2 propojené přestupnými hranami platí, že pokud V_1 –> V_2, pak také nutně C_{zi} (V_1) <= C_{zi} (V_2). Tímto jsme dokázali, že ukončením traverzování po překročení hodnoty $MaxC_p$ nepřijdeme o žádné výsledky.

Dále určíme, ve kterých případech budeme kontrolovat, zda je aktuálně zpracovávaný vrchol platný v rámci aktuálního data vyhledávání. Stejně jako u vyhledávacího algoritmu nad relační databází totiž tuto informaci nemusíme kontrolovat vždy. Pokud jsme se totiž do vrcholu dostali přejezdovou hranou, znamená to, že jsme již na aktuální jízdu nastoupili někdy dříve, a tudíž jsme již měli zkontrolovat její platnost pro datum vyhledávání. Kontrolu platnosti jízdy (zastavení) tedy budeme provádět pouze tehdy, pokud byla poslední hrana předcházející aktuálnímu vrcholu v rámci cesty hranou přestupní.

Pro zjištění, zda je aktuální jízda v provozu pro datum vyhledávání, musíme nejprve v rámci grafu po hraně spojující zastavení s jízdou dotraverzovat na vrchol jízdy, odkud pak musíme dotraverzovat na vrchol intervalu platnosti. Z tohoto vrcholu již vyčteme potřebné parametry pro určení, zda je daná jízda pro den vyhledávání v provozu. Schéma grafu, kterým musíme ze zastavení až k intervalu platnosti traverzovat, je zobrazeno na obrázku 6.10.



Obrázek 6.10: Cesta grafem ze zastavení A_1 na jízdu J_1 a k intervalu platnosti I_1

Pokud ovšem z intervalu platnosti zjistíme, že aktuální jízda (zastavení) není pro den vyhledávání v platnosti, nemůžeme traverzování touto cestou zcela ukončit. Tato skutečnost pro nás totiž pouze znamená, že k traverzování nesmíme dále využít přejezdovou hranu (která vede na další zastavení v rámci aktuální jízdy, které je samozřejmě také k aktuálnímu datu vyhledávání neplatné). Naopak přestupní hranu můžeme využít k dalšímu traverzování. Dostaneme se tak na další zastavení, na kterém opět musíme zkontrolovat platnost dané jízdy.

Připomeňme si, že pro zjednodušení uvažujeme celé vyhledávání pouze v rámci jednoho dne. Vlastní implementace vyhledávače ale samozřejmě bude muset umožňovat vyhledávání přes více dní. Pro tyto účely je také nutné si uvědomit, že platnost jízdy pro konkrétní den musíme zjišťovat vždy z prvního zastavení každé jízdy, jak jsme si řekli již v kapitole 4.3.1. Může se totiž stát, že jízda vyjíždí v jiný den, než ve kterém se aktuálně v rámci vyhledávání pohybujeme. V našem případě to při implementaci bude znamenat, že pro zjištění platnosti jízdy vždy musíme dotraverzovat v protisměru přejezdových hran na první zastavení aktuální jízdy, pro které teprve platnost budeme zjišťovat. Při vlastní implementaci však můžeme i sáhnout ke *cachování*[1] těchto dat, abychom se vyhnuli výraznému navýšení počtu vrcholů ke zpracování v rámci vyhledávání.

[1] Ukládání dat do mezipaměti.

6.3.2.2 Postup traverzování

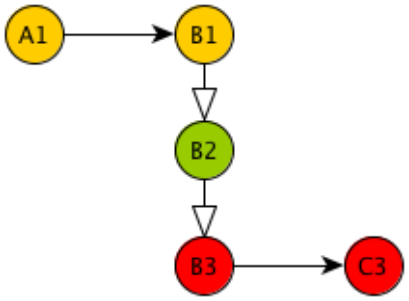
Díky definici základních parametrů, které musíme pro všechny vrcholy v rámci traverzování kontrolovat, se nyní můžeme pustit do konkrétního návrhu vyhledávání cest. Dostali jsme se do situace, kdy využijeme hledání všech pareto-optimálních cest v grafu. Postup pro hledání těchto cest byl detailně popsán v kapitole 5.5.4. Nyní si jej tak pouze shrneme a zpřesníme pro využití na vyhledávání cest v grafu.

Ve zmíněné kapitole jsme navrhli, aby se jednotlivá zastavení zpracovávala v pevně definovaném pořadí. Toto pořadí bylo určeno pomocí času zastavení C_z a délky cesty k zastavení od zastavení výchozího. Délku cesty jsme přitom počítali již s připočtenými penalizacemi za přestup (PP).

Nyní definujeme novou proměnnou Z_v, která bude určovat výchozí zastavení v rámci cesty (výchozí vrchol) a proměnnou Z_a, která bude určovat aktuálně zpracovávané zastavení (vrchol) v rámci traverzování grafem. Pro určení parametru C_z vrcholu Z_a nám stačí ze Z_a přečíst hodnoty C_v a C_p, které se pro určení C_z používají.

Abychom mohli pokračovat dále, musíme si definovat, jakým způsobem budeme počítat délku cesty a počet přestupů na cestě. Uvědomíme si, že jsme hranám v grafu nepřiřadili žádné délky. Nyní si ukážeme, že to není opomenutí, ale záměr. Žádné délky hran v grafu totiž vůbec nepotřebujeme.

Výpočet délky cesty mezi Z_v a Z_a bez PP je pro nás triviální, neboť se jedná o hodnotu C_v (Z_v) - C_z (Z_a). To pro nás mimochodem znamená, že v rámci traverzování grafem musíme mít v každém kroku k dispozici informaci o výchozím vrcholu této cesty. Toto pro nás nicméně není žádný problém, protože již dříve v kapitole 6.3.1 jsme vhodně zvolili způsob výběru výchozích vrchoů. Pro určení paramteru PP očividně musíme mít ale přehled i o tom, kolikrát jsme na aktuální cestě přestoupili z jednoho spoje na druhý. Definice přestupu nicméně není v rámci našeho grafu zcela triviální, proto si ji nyní zavedeme. Jeden přestup v rámci cesty bude reprezentován jednou dvojicí bezprostředně po sobě následujících hran typu přestupní a přejezdová. Nejedná se tedy pouze o počet přestupních hran v rámci cesty. Tento fakt můžeme vidět na obrázku 6.11.



Obrázek 6.11: Přestupní hrany v rámci cesty grafem

Na tomto obrázku je znázorněna cesta ze stanice A do stanice C přes přestupní stanici B. Vidíme, že po cestě jsme využili dvě přestupní hrany, nicméně pochopitelně jsme přestoupili

pouze jednou, a to ze žluté jízdy na červenou. Tento přestup jsme nyní již schopni detekovat, protože jsme na cestě bezprostředně po sobě použili přestupní a přejezdovou hranu.

Abychom mohli v průběhu traverzování pro Z_a určit PP, musíme tedy mít vždy informaci o všech předcházejících hranách na aktuální cestě. Tyto hrany navíc musí být v pořadí, ve kterém byly navštíveny. Počet přestupů po cestě je pak dán počtem bezprostředně po sobě následujících hran typu přestupní a přejezdová. Jsme tedy nyní schopni určit délku cesty včetně penalizací, kterou jsme potřebovali pro správné řazení vrcholů pro zpracování. Nutnost sledování všech hran, ale i vrcholů v rámci cesty pro nás není na přítěž, protože bychom je tak jako tak museli sledovat. Pokud bychom si totiž informace o každé z cest někde neukládali, neměli bychom jak poté uživateli nalezené cesty zobrazit. Nestačí nám tedy pouze zjišťovat délky jednotlivých cest, ale naopak potřebujeme znát kompletní strukturu všech nalezených cest.

Označme si nyní délku cesty včetně penalizací za přestupy proměnnou CPP. Vyhledávání všech pareto-optimálních cest je už nyní díky předchozí detailní analýze velmi přímočaré:

1. Vlož všechna počáteční zastavení do prioritní fronty.
2. Vyber z prioritní fronty vrchol ke zpracování Z_a (priorita je určena C_z a CPP). Pokud ve frontě již není žádný prvek, ukonči celý algoritmus.
3. Pokud již byl vrchol Z_a zpracováván dříve v rámci této cesty, ukonči další traverzování touto cestou a pokračuj krokem 2.
4. Pokud již byl vrchol Z_a zpracováván algoritmem dříve v rámci jiné cesty a současně platí CPP (dříve zpracovaného uzlu) $< CPP$ (Z_a), ukonči další traverzování touto cestou a pokračuj krokem 2.
5. Pokud je Z_a zastavením v cílové stanici, ukonči další traverzování touto cestou a ulož si výsledek. Pokračuj dále krokem 2.
6. Prozkoumej všechny sousedy Z_a dostupné pomocí orientovaných hran. Typy hran, které mohou být využity pro hledání sousedů, budou specifikovány první z možností, která pro aktuální Z_a platí:
   (a) Pokud je Z_a prvním vrcholem v rámci cesty, sousedy uvažuj pouze po přejezdové hraně.
   (b) Pokud byla poslední hrana po cestě k Z_a hranou přestupní a současně není Z_a v platnosti pro aktuální datum vyhledávání, sousedy uvažuj pouze po přestupní hraně.
   (c) Pokud neplatí (a) ani (b), uvažuj sousedy po přejezdové i přestupních hranách.

   Pro každého z vybraných sousedů zkontroluj jeho C_z. Pokud by pro některý vrchol platilo $C_z > MaxC_p$ nebo $C_z < (C_p\ (Z_a) + PMP)$, vyluč tento vrchol z množiny sousedních vrcholů vrcholu Z_a.
7. Všechny zbylé sousedy v množině sousedů z kroku 6 vlož do prioritní fronty a pokračuj krokem 2.

Výsledkem právě představeného algoritmu budou všechny pareto-optimální cesty z výchozího do cílového vrcholu v rámci časového intervalu pro vyhledávání s libovolným počtem přestupů. Na každý přestup je zaručen minimální čas na přestup PMP.

6.4 Optimalizace vyhledávání cest v grafu

I v rámci vyhledávání cest grafem můžeme zavést optimalizace, které jsme si představili již v kapitole 5.5, kde nalezneme i detailnější popis jednotlivých optimalizací. Jejich zakomponování do vyhledávacího algoritmu ale pochopitelně bude odlišné, proto si jej nyní představíme.

6.4.1 Maximální počet přestupů

Zakomponování parametru $maxPocetPrestupu$ do vyhledávacího agloritmu nad grafem je možné více způsoby. Pro každý uzel Z_a například můžeme kontrolovat, zda po cestě k tomuto uzlu z Z_v nebyl již překročen $maxPocetPrestupu$. Tento postup nicméně není příliš vhodný, protože abychom tento stav identifikovali, musíme vyhledávacímu algoritmu v rámci jedné cesty umožnit $maxPocetPrestupu + 1$ přestup. Mnohem výhodnější pro nás bude další přestup v rámci cesty vůbec neumožnit, pokud jsme již po cestě využili maximálního počtu přestupů.

Pro každý zpracovávaný uzel Z_a tak budeme kontrolovat, kolikrát jsme již po cestě ze Z_v využili přestup. Pokud by platilo *počet přestupů* $= maxPocetPrestupu$, k dalšímu traverzování v rámci této cesty budeme moci využívat již pouze přejezdových hran. Tímto způsobem zamezíme překročení hodnoty $maxPocetPrestupu$ v rámci jedné cesty.

Právě představený způsob ošetření maximálního počtu přestupů se nicméně dá ještě vylepšit. Všimneme si, že kontrolu na překročení parametru $maxPocetPrestupu$ stačí provádět pouze v takových vrcholech Z_a, ke kterým jsme se při traverzaci dostali přejezdovou hranou. Při detekci maximálního počtu přestupů tak totiž už nikdy v rámci dané cesty neumožníme využít přestupní hrany. Kontrolovat překročení maximálního počtu přestupů tedy není nutné, pokud jsme se k aktuálnímu Z_a při traverzování dostali přestupní hranou.

6.4.2 Vícenásobné přestupy mezi dvěma jízdami

Problém vícenásobných přestupů mezi shodnými jízdami byl popsán v kapitole 5.5.2. Nyní si uvědomíme, že náš návrh pro vyhledávání pareto-optimálních cest již tento problém pro vyhledávání v grafu vyřešil. Díky procházení vrcholů v pevně daném pořadí totiž vždy budeme kontrolovat všechny cesty vedoucí přes shodný vrchol v přímém sledu po sobě. Algoritmus pak bude preferovat vždy jen ty nejvýhodnější možné cesty vedoucí přes daný vrchol (a tedy i jízdu). Traverzování po nevýhodných cestách přes již navštívené vrcholy a jízdy tak bude vždy ihned ukončeno.

Tuto skutečnost si můžeme blíže představit nejdříve na situaci z obrázku 5.4 v kapitole 5.5.2. Pokud budeme uvažovat za výchozí vrchol žlutý vrchol A, máme velmi mnoho možných cest, kterými můžeme dotraverzovat do zeleného vrcholu A. Očividně nejvýhodnější je pro nás nicméně cesta $A{->}B{->}B{->}A$. Tato cesta jako jediná také bude výstupem vyhledávacího

algoritmu. Všechny ostatní možné cesty totiž s tou nejlepší možnou budeme porovnávat nejpozději v zeleném vrcholu B, kde existence zmíněné nejlepší cesty zamezí všem ostatním cestám v dalším traverzování grafem od tohoto vrcholu.

Pokud budeme uvažovat situaci z obrázku 5.5 v kapitole 5.5.2 a budeme se chtít dostat ze žlutého vrcholu A do zeleného vrcholu E, uvědomíme si, že výstupem algoritmu bude opět pouze jedna nejvýhodnější cesta. Nicméně v tomto případě je určení nejvýhodnější cesty problematické, protože všechny existující cesty jsou stejně dlouhé (výhodné). Algoritmus nicméně všechny existující cesty bude porovnávat nejpozději v zeleném vrcholu D, kde také umožní další traverzování pouze nejvýhodnější cestě, kterou bude cesta $A{->}B{->}C{->}D{->}D$. Při traverzování k tomuto vrcholu totiž na této cestě (jako jediné) zatím nebylo využito přestupu, a tudíž se její délka bude počítat bez penalizace za přestup. Zde si můžeme povšimnout zajímavé vlastnosti. Zatímco algoritmus pro relační databázi jsme navrhli tak, že vždy zvolí pouze první možný přestup, nynější algoritmus zvolí nad daty z obrázku 5.5 poslední možný přestup. Dříve jsme si řekli, že ideálně by měly oba algoritmy vracet zcela totožné výsledky. Toto je ale situace, kdy nám rozdílnost výsledků vadit nebude. Nalezené cesty totiž v obou případech budou stejně dlouhé i výhodné. Úprava jednoho z algoritmů tak, aby se choval stejně jako algoritmus druhý, by navíc nebyla zcela triviální. Protože by nám tato úprava nepřinesla žádné výrazné zlepšení a naopak bychom museli jeden z algoritmů ohýbat, nebudeme se jí dále zabývat.

6.4.3 Traverzování mezi shodnými stanicemi

Problém traverzování mezi shodnými stanicemi byl podrobně popsán již v kapitole 5.5.3. Nyní si tedy pouze navrhneme způsob, kterým této nežádoucí situaci zamezíme při traverzování grafem.

Výsledkem úvahy ve zmíněné kapitole bylo konstatování, že pokud aktuálně zpracováváme zastavení ve stanici, která byla již dříve v rámci této cesty jinou jízdou navštívena, můžeme další vyhledávání touto cestou ukončit. Při návrhu vyhledávacího algoritmu nad grafem jsme již konstatovali, že pro každý zpracovávaný vrchol musíme mít k dispozici informaci o všech předchozích zastaveních. Díky tomu jsme schopni traverzování mezi shodnými stanicemi zabránit, protože každé zastavení má informaci o stanici i jízdě. Pokud tedy při zpracovávání zastavení v rámci vyhledávání zjistíme, že na stanici, které se aktuální zastavení týká, jsme již dříve na jiné jízdě byli, můžeme traverzování touto větví ukončit.

Kapitola 7

Implementace

Vlastní implementace obou vyhledávačů bude díky velmi detailní analýze a návrhu řešení poměrně přímočará. Nejprve se seznámíme s technologiemi, které byly pro implementaci vybrány. Poté si představíme implementaci vyhledávání nad relační i grafovou databází.

7.1 Zvolené technologie

Nyní si představíme zvolené technologie, ve kterých budeme oba navržené algoritmy a vlastní aplikaci umožňující vyhledávání implementovat.

7.1.1 Relační databáze PostgreSQL

Již v úvodu této práce jsme za zástupce relačních databází, které budou použity pro vyhledávání spojů, zvolili databázi PostgreSQL[1]. Jedná se o open-source [22] databázový systém, za jehož vývojem stojí především vývojářská komunita a který se vyvíjí již od 80. let minulého století. Patří mezi nejrozšířenější open-source databázové systémy na světě a právě mezi těmito systémy je považován za jeden z nejpokročilejších [14]. Z tohoto důvodu byla pro účely implementace vyhledávacího algoritmu nad relační databází vybrána právě databáze PostgreSQL.

7.1.2 Grafová databáze Neo4j

Zejména v poslední době se začíná objevovat stále více implementací grafových databází. Mimo jiné například AllegroGraph[2], FlockDB[3] nebo Sones GraphDB[4]. Pro účely této práce jsme nicméně již v úvodu zvolili databázi Neo4j[5], která je jednou z nejrozšířenějších a nejpokročilejších grafových databází [8].

[1] <http://www.postgresql.org/>
[2] <http://franz.com/agraph/allegrograph/>
[3] <https://github.com/twitter/flockdb>
[4] <https://github.com/sones/sones>
[5] <http://neo4j.com/>

Databáze Neo4j pochází od společnosti Neo Technology, Inc.[6]. Je nabízena ve verzi Community, která je zcela zdarma, a Enterprise, která je již zpoplatněna a obsahuje v sobě navíc například i oficiální podporu. První verze byla vydána již v roce 2007 [28].

Celý databázový systém je napsaný v Javě[7] [37]. Pro integraci s aplikací se dá využít buď tzv. *embedded módu*, který je vlastně Java archiv (JAR)[8], případně může databáze běžet ve *standalone módu*, kdy k databázi můžeme přistupovat plnohodnotně pomocí REST (Representational State Transfer) API[9] rozhraní.

Jednotlivé vrcholy i hrany v databázi mohou obsahovat libovolné množství atributů typu *key –> value*. Tato vlastnost se výborně hodí pro ukládání informací k vrcholům, které jsme definovali v kapitole 6.1.4. Jednotlivé atributy navíc mohou být indexovány, čímž se umožní velmi rychlé vyhledávání výchozích vrcholů dle hodnot atributů.

7.1.2.1 Dotazování nad Neo4j

Jednou z možností pro dotazování nad databází Neo4j je Cypher query language (Cypher) [16], který byl vytvořen speciálně pro Neo4j. Cypher je deklarativní jazyk inspirovaný jazykem SQL, díky čemuž také obsahuje velké množství klíčových slov z jazyka SQL převzatých.

Detailní analýza jazyka Cypher není předmětem této práce, nicméně si pro názornost představíme několik krátkých ukázek.

- Výběr jednoho vrcholu (node) z databáze

```
MATCH (node) RETURN node
```

- Výběr dvou vrcholů (node1, node2) spojených orientovanou hranou (rel)

```
MATCH (node1)-[rel]->(node2) RETURN node1, rel, node2
```

Ukázky dalších jazykových konstruktů jsou uvedeny například v [16]. V rámci Cypheru je možné i pokročilé traverzování grafem s možností parametrizace cesty.

Neo4j nicméně také poskytuje velmi rozsáhlé Traversal Java API[10], které umožňuje traverzování grafem řídit velmi podrobně v každém jednotlivém kroku traverzace. V tomto ohledu umožňuje řídit vyhledávání v grafu mnohem podrobněji, než je v technických možnostech jazyka Cypher. Příkladem, kdy již jazyk Cypher není pro traverzování vhodný, může být například nutnost traverzování po více různých hranách, kdy chceme určit i maximální počet využití daných hran v rámci cesty. Cypher sice umožňuje určit maximální počet hran použitých po cestě, avšak tento počet je dán vždy pro hrany všech typů. Pokud tedy hledáme cestu, kde můžeme po hraně h traverzovat do nekonečna, nicméně po hraně i v rámci cesty nejvýše 3x, Cypher selhává. Je sice možné i tuto podmínku zakomponovat, řešení je ovšem složité a vyžaduje nejdříve nalezení všech cest (i těch, kdy byla hrana i použita více než 3x) a teprve až poté je nutné požadované cesty ze všech nalezených vyfiltrovat.

[6] <http://neo4j.com/company/>
[7] <http://www.oracle.com/technetwork/java/index.html>
[8] <http://www.oracle.com/technetwork/java/archive-139210.html>
[9] <http://www.zdrojak.cz/clanky/rest-architektura-pro-webove-api/>
[10] <http://neo4j.com/docs/stable/tutorial-traversal-java-api.html>

Protože vyhledávání, které jsme definovali v kapitole 6, je poměrně složité, v žádném případě nám pro jeho implementaci nebude stačit jazyk Cypher. Využijeme tedy právě Traversal Java API.

7.1.3 Technologie pro webovou aplikaci

V dnešní době existuje nespočet možných technologií, ve kterých by se aplikace pro účely vyhledávání spojů mohla implementovat. Protože jsme nicméně zvolili grafovou databázi Neo4j a právě jsme si řekli, že pro vyhledávání v grafu budeme chtít využít Traversal Java API, implementuje webovou aplikaci také v Javě. Zde je vhodné zmínit, že je možné databázi Neo4j integrovat i v rámci velkého množství jiných programovacích jazyků (Ruby[11], Python[12], .NET[13]...), žádná z možností ale nenabízí tak kvalitní API pro traverzování grafem, jako je právě to psané v Javě.

Tuto volbu nebudeme připisovat pouze na vrub zvolené databázi. Java totiž není jen objektově orientovaný programovací jazyk, ale i platforma, která umožňuje tvorbu robustních webových aplikací. Pro tvorbu webových aplikací slouží Java EE (enterprise edition)[14], která v sobě zahrnuje například i aplikační server, ve kterém aplikace běží, řeší správu databázových spojení, řídí práva a uživatelské role aplikace a mnoho dalšího [13].

Pro vývoj enterprise aplikací v Javě vznikl v roce 2003 nyní již velmi populární *framework*[15] Spring[16]. Tento framework v mnoha ohledech usnadňuje vývoj enterprise aplikací v Javě, díky celé řadě již hotových komponent pro okamžité využití. Pro nás je nicméně Spring zajímavý ještě z jiného pohledu. Jeho volnou součástí je totiž projekt Spring Data[17] a jeho součást Spring Data Neo4j[18], která v sobě zahrnuje i možnost využít mapování jednotlivých vrcholů a hran databáze na Java třídy. Tímto způsobem jsme schopni navrhnout strukturu grafové databáze z aplikace velmi elegantním způsobem.

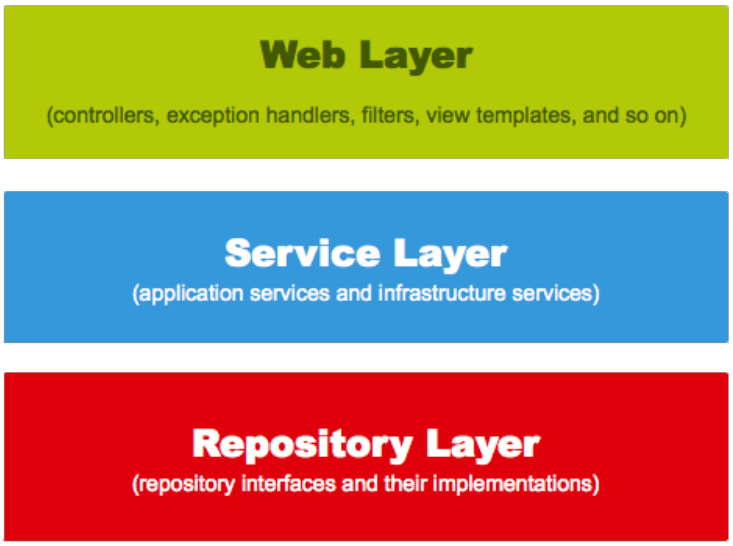
7.1.3.1 Struktura webové aplikace

Struktura vlastní aplikace se řídí standardní doporučovanou architekturou pro aplikace psané ve Springu [11]. Architektura aplikace se tedy dělí do tří základních vrstev:

- Webová vrstva (Web layer) – Zpracovává uživatelské dotazy a vrací na ně odpovědi.
- Servisní vrstva (Service layer) – Obsahuje kompletní logiku aplikace, řídí transakce, zabezpečení apod.
- Datová vrstva (Repository layer) – Stará se o komunikaci s datovými úložišti (databázemi).

Tyto vrstvy také můžeme vidět graficky znázorněné na obrázku 7.1.

[11] <https://www.ruby-lang.org/en/>
[12] <https://www.python.org/>
[13] <http://www.microsoft.com/net>
[14] <http://www.oracle.com/technetwork/java/javaee/overview/index.html>
[15] Souhrn struktur (knihoven) sloužících pro podporu vývoje aplikace
[16] <https://spring.io/>
[17] <http://projects.spring.io/spring-data/>
[18] <http://projects.spring.io/spring-data-neo4j/>



Obrázek 7.1: Standardní architektura Spring aplikace. Zdroj: [11]

Logika vyhledávání spojů bude v rámci architektury obsažena pochopitelně v datové vrstvě. Pro tyto účely můžeme definovat rozhraní vyhledávání, které potom budou obě implementace využívat. Metoda pro vyhledávání (*findRidesByDepartureDate()*) tedy bude mít předpis definovaný níže.

```
/**
* najde cesty (paths) z vychozi do cilove stanice dle data odjezdu
* @param stationFromId stanice z
* @param stationToId stanice do
* @param departure odjezd
* @param maxArrival max prijezd
* @param maxTransfers max pocet prestupu
* @return list vysledku
*/
List<SearchResultWrapper> findRidesByDepartureDate(
    long stationFromId, long stationToId, Date departure,
    Date maxArrival, int maxTransfers);
```

7.1.3.2 Aplikační server

Jako aplikační server, na kterém bude aplikace nasazena, byl zvolen WildFly[19]. Nicméně vzhledem k faktu, že aplikace je psaná ve frameworku Spring, je možné ji spustit i na jiných aplikačních serverech.

[19] <http://wildfly.org/>

7.2 Implementace vyhledávacího algoritmu nad relační databází

Pro propojení aplikace s databází PostgreSQL byl zvolen BasicDataSource[20] z knihovny komponent Apache Commons[21], který nám umožňuje *poolovat*[22] připojení, čímž se vyhneme nutnosti otevírat nové spojení s databází při každém provádění SQL dotazu. Ukázka použití následuje níže:

```
<bean id="dataSource" destroy-method="close"
    class="org.apache.commons.dbcp.BasicDataSource" >
    <property name="driverClassName" value="{jdbc.driver}" />
    <property name="url" value="{jdbc.url}" />
    <property name="username" value="{jdbc.username}" />
    <property name="password" value="{jdbc.password}" />
    <property name="maxIdle" value="10" />
    <property name="maxActive" value="20" />
    <property name="poolPreparedStatements" value="true" />
</bean>
```

Pro vlastní přístup k datům při vyhledávání spojů bylo využito JDBC (Java Database Connectivity) API[23], které je standardem pro konektivitu mezi Javou a celou řadou databází (je tedy databázově nezávislé). Konkrétně bylo využito Spring JDBC implementace[24]. Pro vyhledávání spojů je vhodné nevyužívat žádného relačně-objektového mapování, které pro každý select proti databázi vyžaduje určité množství režie[25].

Postup pro vyhledávání nad relační databází byl kompletně popsán v kapitole 5. V implementaci metody *findRidesByDepartureDate()* tak začneme rekurzivně volat vyhledávací funkci, které předáme inicializační parametry. Po jejím doběhnutí budou v proměnné *ridesMap* všechny nalezené výsledky:

```
/**
* Naplni mapu ridesMap vysledky hledani cesty
* @param stationFromId stanice z
* @param stationToId stanice do
* @param departure datum odjezdu
* @param maxArrival max datum prijezdu
* @param stepNumber v kolikatem cyklu rekurze jsme
* @param ridesMap mapa RidesID(sekvence) => searchResultWrapper
    s vysledky hledani
```

[20] <http://commons.apache.org/proper/commons-dbcp/api-1.4/org/apache/commons/dbcp/BasicDataSource.html>

[21] <https://commons.apache.org/>

[22] Odkládat otevřená spojení s databází do tzv. poolu, odkud jsou při volání databáze vybírána. Odpadá tak nutnost neustálého navazování spojení s databází.

[23] <http://www.oracle.com/technetwork/java/javase/jdbc/index.html>

[24] <http://docs.spring.io/spring/docs/current/spring-framework-reference/html/jdbc.html>

[25] <http://hibernate.org/orm/what-is-an-orm/>

```
* @param visitedStops jiz navstivene stopy v ramci rekurze
* @param visitedRides jiz navstivene ridy v ramci rekurze
* @param maxTransfers max pocet prestupu
*/
protected void findRidesByDepartureDateAlgorithm(
    long stationFromId, long stationToId, Date departure,
    Date maxArrival, int stepNumber,
    Map<String, SearchResultWrapper> ridesMap,
    List<VisitedStopsWrapper> visitedStops,
    List<Long> visitedRides, int maxTransfers,
    Map<Long, Long> visitedStopTimes);
```

7.3 Implementace vyhledávacího algoritmu nad grafovou databází

Databázi Neo4j v naší aplikaci implementujeme v Embedded módu zmíněném dříve. Pro tyto účely nám výborně poslouží již zmíněná knihovna Spring Data Neo4j:

```
<bean id="graphDbService" destroy-method="shutdown"
    scope="singleton" class="org.springframework.data.
        neo4j.support.GraphDatabaseServiceFactoryBean">
    <property name="path" value="{neo4j.path}" />
</bean>
```

Pro účely vyhledávání nám nyní Neo4j poskytuje již zmíněné Traversal API. Pomocí něho jsme schopni definovat tzv. *Traversal Description*[26], pomocí kterého můžeme řídit traversování grafem dle našich potřeb.

Traversal Description vyžaduje tři základní parametry, které budou traverzaci řídit. Každý pro nás bude reprezentován jednou Java třídou. Jedná se o:

- Expander – Určuje, po kterých hranách z aktuálního vrcholu budeme dále traverzovat. Má informace o kompletní cestě z výchozího vrcholu.
- Evaluator – Určuje, jakým způsobem zpracovat aktuální vrchol. Zde budeme například kontrolovat, zda již nejsme v cílovém vrcholu.
- BranchSelector – Určuje pořadí, ve kterém jsou vrcholy zpracovávány. Bude tedy využívat námi navrženou implementaci prioritní fronty pro výběr vrcholů.

Dále samozřejmě musíme zvolit výchozí vrcholy, od kterých vlastní traverzování začne. Tyto vrcholy dostane Traversal Description jako následný parametr. Poté se již traverzace bude řídit výhradně pomocí námi definovaných pravidel a jejím výsledkem budou všechny pareto-optimální cesty z výchozího do cílového vrcholu v daném čase.

[26] <http://neo4j.com/docs/stable/tutorial-traversal-java-api.html>

7.3.1 Výběr výchozích vrcholů

Pro výběr výchozích vrcholů využijeme jazyk Cypher. Cílem je nalézt všechna zastavení na výchozí stanici pro daný rozsah času. Všechna zastavení přitom musí být v platnosti pro den výjezdu. Všechny podmínky pro nalezení výchozích zastavení jsme definovali již v kapitole 5.3.1. Připomeňme si, že v rámci vyhledávání musíme kontrolovat i připojené uzly intervalu platnosti. Uvažujme opět vstupní parametry vyhledávání z kapitoly 5.3.1:

- S_v ... 5
- S_c ... 9
- D_v ... 20.04.2015
- C_v ... 05:00
- Day_v ... pondělí
- $MaxC_p$ ... 11:00

Cypher dotaz pro nalezení výchozích zastavení pak bude vypadat následovně:

```
MATCH (s:StopNode {stationId: 5})-[IN_RIDE]->(t:TripNode)
-[IN_INTERVAL]->(i:OperationIntervalNode)
WHERE i.monday = true
and i.fromDate <= '20.04.2015' and i.toDate >= '20.04.2015'
and s.departure >= '05:00' and s.departure < '11:00'
RETURN s ORDER BY s.departure
```

Všechny vrcholy, které budou vráceny tímto dotazem, poskytneme jako výstupní vrcholy pro Traversal Description. Zjednodušená ukázka implementace je uvedena níže:

```
Iterable<Node> startNodes = getStartNodes(); //vychozi zastaveni
TraversalDescription td = graphDbService.traversalDescription()
    .order(new DepartureBranchSelector(...))
    .expand(new DepartureTypeExpander(...))
    .evaluator(new DepartureTypeEvaluator(...);

Traverser traverser = td.traverse(startNodes);
for(Path path : traverser) {
    //nalezene pareto-optimalni cesty grafem
}
```

Pro názornost si představíme i kostry jednotlivých tříd použitých pro traverzování. Všechny tyto třídy implementují rozhraní z balíku Java Traversal API.

```
/**
* Branch selector
*/
public class DepartureBranchSelector implements BranchSelector {

    @Override
    public TraversalBranch next(TraversalContext metadata)  {
        //vkladani a vyber uzlu z prioritny fronty
    }

}

/**
* Evaluator
*/
public class DepartureTypeEvaluator implements Evaluator {

    @Override
    public Evaluation evaluate(Path path) {
        //kontrola uzlu v ramci cesty
    }

}

/**
* Expander
*/
public class DepartureTypeExpander implements PathExpander<?> {

    @Override
    public Iterable<Relationship> expand(Path path) {
        //vyber hran pro dalsi traverzaci
    }

}
```

Výběr následných hran pro traverzaci i kontrolu jednotlivých uzlů budeme provádět přesně dle návrhu z kapitoly 6.

7.4 Uživatelské rozhraní

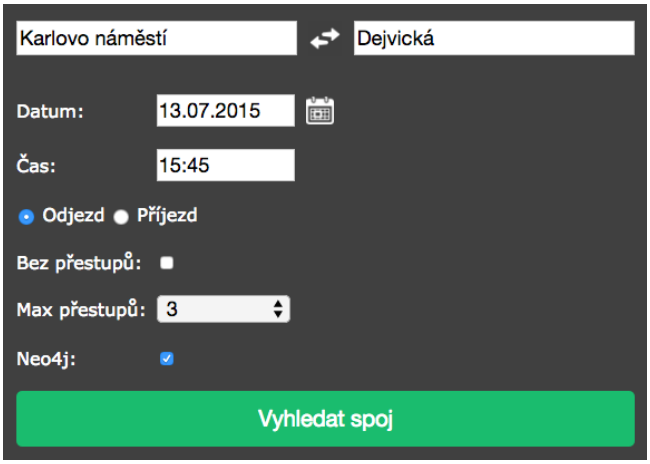
Pro tvorbu uživatelského rozhraní aplikace byl zvolen standard JSF (Java Server Faces)[27]. Kromě vyhledávacího formuláře a stránky pro zobrazní výsledků vyhledávání byla vytvořena

[27] <http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html>

i administrační část, která umožňuje správu dat z jízdních řádů. Protože hlavní cílem této práce není tvorba uživatelského rozhraní, pouze na tomto místě představíme ukázky několika základních obrazovek. Samotná aplikace je přílohou této práce a dle návodu na instalaci na přiloženém CD je možné ji spustit a vyzkoušet.

7.4.1 Vyhledávací formulář

Vyhledávací formulář musí umožňovat zadat všechny parametry definované v kapitole 4.1. Dle tohoto požadavku vznikl vyhledávací formulář na obrázku 7.2. Checkbox *Neo4j* určuje, zda cheme vyhledávat v grafové nebo relační databázi.



Obrázek 7.2: Vyhledávací formulář

7.4.2 Výsledky vyhledávání

Výsledky vyhledávání musí obsahovat informace o stanicích, časech příjezdu a výjezdu, a spojích, které k cestování využijeme. Ukázku formuláře s výsledky vyhledávání vidíme na obrázku 7.3.

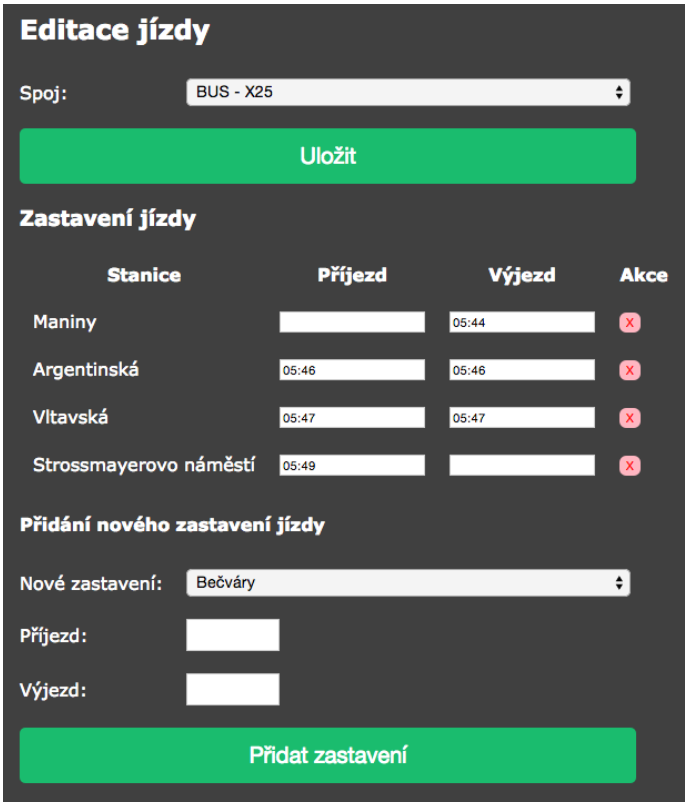
Datum	Stanice	Příjezd	Výjezd	Spoj
13.07.2015	Karlovo náměstí		17:45	B
13.07.2015	Můstek	17:47		
13.07.2015	Můstek		17:50	A
13.07.2015	Dejvická	17:56		

Doba jízdy: **0 hod 11 min**

Obrázek 7.3: Tabulka s výsledky vyhledávání

7.4.3 Administrační část

Z administrační části si ukážeme alespoň editační formulář jízd. Ten má podobu, kterou můžeme vidět na obrázku 7.4.



Obrázek 7.4: Editační formulář jízdy

Kapitola 8

Testování

Tato kapitola se zabývá testováním obou navržených algoritmů zejména z hlediska jejich rychlosti a koreknosti výsledků.

8.1 Testovací data

V České republice není snadné najít volně přístupná data o jízdních řádech ve formátu GTFS. Jak jsme si již řekli v kapitole 2.1, společnost CHAPS si svá data ve strojově zpracovatelném formátu velmi úzkostlivě střeží. Protože pro účely testování výkonnosti námi navržených algoritmů budeme pochopitelně potřebovat co největší data, odpadá i možnost využít data od menších soukromých dopravců.

Naštěstí je Dopravní podnik hlavního města Prahy (DPP) na žádost podle zákona č. 106/1999 Sb. o svobodném přístupu k informacím povinný uvolňovat data o své dopravní síti ve formátu GTFS[1]. Dopravní síť v Praze je přitom velmi rozsáhlá, a výborně se tak hodí pro účely testování rychlosti vyhledávání spojů.

8.1.1 Velikost testovacích dat

Ze zvolených testovacích dat DPP z jara roku 2013[1] byly vybrány pouze jízdy, které mají platnost minimálně v intervalu 5.7.2013 – 31.8.2013 a které provozuje přímo DPP. Vznikl tak balík dat o následující velikosti:

- 1 215 stanic
- 45 146 jízd
- 883 568 zastavení

[1] <http://www.infoprovsechny.cz/request/aktuln_verze_gtfs>

8.2 Testování rychlosti algoritmů

Testování rychlosti obou algoritmů budeme provádět s postupně se zvyšující velikostí dat. Začneme tedy testovat nejprve na menší části z celého balíku dat DPP a postupně se dopracujeme k otestování na kompletních datech. Budeme tak schopni srovnat výkonnosti obou algoritmů nad daty různé velikosti. Data budou vždy vybrána z kompletního balíku dat rovnoměrně tak, aby nemohlo dojít k situaci, že jízdní řád některých spojů zůstane kompletní, zatímco jiné spoje se nedostanou do databáze vůbec. Právě naopak dojde k poměrnému snížení počtu jízd u všech spojů rovnoměrně.

Algoritmy budeme testovat vždy pro pět různých výchozích a cílových stanic. Tyto trasy byly vybrány s ohledem na jejich délku v rámci jízdního řádu tak, abychom do testování zakomponovali jak situaci, kdy cesta zabere pouze několik jednotek minut, tak situaci, kdy cesta zabere více než hodinu. Všechny testy budou probíhat pro shodné časové intervaly vyhledávání:

- DC_v ... 13.7.2013 16:30
- $MaxDC_p$ ... 13.7.2013 22:30

Rozsah hledání je tedy 6 hodin, což je v rámci MHD více než dostačující. Maximální počet přestupů P byl stanoven na 4, což je v rámci MHD poměrně vysoké číslo, nicméně si pro názornost představíme i rychlost algoritmů pro méně přestupů. Pro všechna vyhledávání byl zadán maximální počet požadovaných výsledků $n = 3$. Po nalezení 3 **nejlepších** výsledků tedy může algoritmus vyhledávání ukončit.

8.2.1 Testovací prostředí

Všechny rychlostní testy byly provedeny na zařízení MacBook Pro (13-inch, Mid 2012) s operačním systémem OS X Yosemite. Zařízení má k dispozici procesor Intel Core i5 s frekvencí 2,5 GHz, operační paměť 16 GB (1600 MHz DDR3) a SSD disk Samsung 850 PRO. Maximální velikost paměti pro JVM (Java Virtual Machine)[2], ve které aplikace běží, byla určena na 512 MB.

8.2.2 Rychlost algoritmu nad relační databází

Pro výsledky testování vždy bude uveden počet jízd, které byly v době testování v databázi přítomny. Počet zastavení ve všech případech odpovídá přibližně dvacetinásobku počtu jízd. V případě, že algoritmu trvalo nalezení výsledků výrazně déle než jednu minutu, nemusí být tento výsledek v tabulce zobrazen. V tom případě bude pro poslední zobrazený údaj pro danou trasu platit $P < 4$.

V tabulce 8.1 vidíme rychlost algoritmu pro 2 252 jízd (1/20 kompletních dat). Z pohledu balíku dat DPP se toto číslo může zdát jako malé, nicméně si připomeňme, že kompletní jízdní řády vlakové dopravy v České republice obsahují přibližně pouze 10 000 jízd [2].

[2] <http://www.java.com/en/about/>

Stanice z	Stanice do	Max. počet přestupů	Doba hledání [s]
Bělocerkevská	Dejvická	1	0.572
Bělocerkevská	Dejvická	2	2.435
Bělocerkevská	Dejvická	3	8.590
Bělocerkevská	Dejvická	4	32.427
Dejvická	Karlovo náměstí	1	1.309
Dejvická	Karlovo náměstí	2	14.775
Dejvická	Karlovo náměstí	3	52.010
Černý Most	Hybšmanka	1	1.294
Černý Most	Hybšmanka	2	73.760
Černý Most	Terminál 1	1	1.242
Černý Most	Terminál 1	2	46.562
Florenc	Anděl	1	2.278
Florenc	Anděl	2	34.659
Výstaviště Holešovice	Bělocerkevská	1	0.923
Výstaviště Holešovice	Bělocerkevská	2	24.786

Tabulka 8.1: Rychlost algoritmu pro 2 252 jízd (relační databáze)

Bohužel už pro tato data se ukazuje, že vyhledávací algoritmus nad relační databází je velmi náchylný na zvyšující se počet možných přestupů. Při volbě maximálně jednoho přestupu jsou výsledky dobré, ale pro většinu vyhledávaných spojení trvá vyhledávání se dvěma přestupy velmi dlouho. Při dalším navyšování počtu možných přestupů pak algoritmus ve většině případů nestihne nalézt výsledky ani v čase přes jednu minutu.

Pokračovat budeme testováním pro 9 017 jízd (1/5 kompletních dat). Jeho výsledky můžeme vidět v tabulce 8.2.

Stanice z	Stanice do	Max. počet přestupů	Doba hledání [s]
Bělocerkevská	Dejvická	1	7.408
Dejvická	Karlovo náměstí	1	15.650
Černý Most	Hybšmanka	1	12.565
Černý Most	Terminál 1	1	10.867
Florenc	Anděl	1	25.711
Výstaviště Holešovice	Bělocerkevská	1	16.587

Tabulka 8.2: Rychlost algoritmu pro 9 017 jízd (relační databáze)

Bohužel pro tento rozsah dat se už pro žádnou z tras nepodařilo nalézt výsledky s více než jedním přestupem v čase pod jednu minutu. V tabulce tak jsou zobrazeny pouze časy pro $P = 1$. Kvůli zvýšenému počtu rekurzivních volání z důvodu vyššího možného počtu přestupů dochází současně i k výraznému navýšení počtu zpracovávaných uzlů, kvůli čemuž je pak vyhledávání prakticky nepoužitelné. Tato skutečnost bohužel platí i pro všechna následující vyhledávání nad relační databází, kdy bude v databázi daleko více dat. Budeme tak zobrazovat pouze výsledky, kde $P = 1$.

V tabulce 8.3 vidíme rychlost agloritmu pro 22 578 jízd (1/2 kompletních dat).

Stanice z	Stanice do	Max. počet přestupů	Doba hledání [s]
Bělocerkevská	Dejvická	1	46.274
Dejvická	Karlovo náměstí	1	75.965
Černý Most	Hybšmanka	1	63.510
Černý Most	Terminál 1	1	65.046
Florenc	Anděl	1	120.610
Výstaviště Holešovice	Bělocerkevská	1	82.166

Tabulka 8.3: Rychlost algoritmu pro 22 578 jízd (relační databáze)

Bohužel na těchto datech již nedokázal algoritmus nalézt výsledky v rozumném čase. Z Florence na Anděl, což je mimochodem velmi krátká trasa, trvá vyhledávání déle než dvě minuty. Je to dáno tím, že algoritmus nejprve v rámci rekurzí začne hledat spojení na úplně jiných trasách, které k cíli nevedou.

Pro kompletní balík dat (45 146 jízd) algoritmus ani pro $P = 1$ nedokázal nalézt výsledek pro žádnou z tras ani v čase přes tři minuty. Výsledky měření rychlosti tedy nebudeme ani uvádět.

8.2.3 Rychlost algoritmu nad grafovou databází

Testování nad grafovou databází probíhalo shodně jako testování nad databází relační z kapitoly 8.2.2. Ihned si tedy můžeme představit výsledky měření. Protože nad grafovou databází je vyhledávání nepoměrně rychlejší, zobrazíme si vždy výsledky měření rychlosti jen pro $P = 1$ a $P = 4$.

V tabulce 8.4 vidíme rychlost algoritmu pro 2 252 jízd.

Stanice z	Stanice do	Max. počet přestupů	Doba hledání [s]
Bělocerkevská	Dejvická	1	0.046
Bělocerkevská	Dejvická	4	0.045
Dejvická	Karlovo náměstí	1	0.035
Dejvická	Karlovo náměstí	4	0.046
Černý Most	Hybšmanka	1	0.087
Černý Most	Hybšmanka	4	0.901
Černý Most	Terminál 1	1	0.099
Černý Most	Terminál 1	4	0.833
Florenc	Anděl	1	0.028
Florenc	Anděl	4	0.029
Výstaviště Holešovice	Bělocerkevská	1	0.093
Výstaviště Holešovice	Bělocerkevská	4	0.122

Tabulka 8.4: Rychlost algoritmu pro 2 252 jízd (grafová databáze)

Rychlost vyhledávání je téměř ve všech případech nižší, než 100 ms, což je bezesporu výborný výsledek. Současně je velmi zajímavá skutečnost, že v některých případech se zvyšující se parameter P vůbec nepodepsal na rychlosti vyhledávání.

V tabulce 8.5 vidíme rychlost algoritmu pro 9 017 jízd.

Stanice z	**Stanice do**	**Max. počet přestupů**	**Doba hledání [s]**
Bělocerkevská	Dejvická	1	0.182
Bělocerkevská	Dejvická	4	0.358
Dejvická	Karlovo náměstí	1	0.115
Dejvická	Karlovo náměstí	4	0.136
Černý Most	Hybšmanka	1	0.731
Černý Most	Hybšmanka	4	4.381
Černý Most	Terminál 1	1	0.261
Černý Most	Terminál 1	4	1.932
Florenc	Anděl	1	0.067
Florenc	Anděl	4	0.075
Výstaviště Holešovice	Bělocerkevská	1	0.505
Výstaviště Holešovice	Bělocerkevská	4	0.397

Tabulka 8.5: Rychlost algoritmu pro 9 017 jízd (grafová databáze)

Doba vyhledávání nám pro jeden z výsledků přeskočila 4 s, nicméně většina ostatních výsledků byla nalezena do 0.5 s, což je pořád velmi dobré. Pomalejší nalezení výsledku z Černého Mostu na Hybšmanku je dáno tím, že zatímco z Černého Mostu spousta spojů vyjíždí, na Hybšmanku jich jede jen velmi málo. Tím pádem jsme nuceni prohledávat větší prostor.

V tabulce 8.6 vidíme rychlost algoritmu pro 22 578 jízd.

Stanice z	**Stanice do**	**Max. počet přestupů**	**Doba hledání [s]**
Bělocerkevská	Dejvická	1	0.169
Bělocerkevská	Dejvická	4	0.254
Dejvická	Karlovo náměstí	1	0.136
Dejvická	Karlovo náměstí	4	0.196
Černý Most	Hybšmanka	1	0.422
Černý Most	Hybšmanka	4	3.304
Černý Most	Terminál 1	1	0.843
Černý Most	Terminál 1	4	5.530
Florenc	Anděl	1	0.091
Florenc	Anděl	4	0.094
Výstaviště Holešovice	Bělocerkevská	1	1.927
Výstaviště Holešovice	Bělocerkevská	4	0.836

Tabulka 8.6: Rychlost algoritmu pro 22 578 jízd (grafová databáze)

Všimneme si, že rychlost vyhledávání se u spousty tras téměř nezměnila, někde se vyhledávání

dokonce zrychlilo. Tím, že v databázi máme více spojů, jich také více jede mezi výchozí a cílovou stanicí. Přitom čím dříve nalezneme tři nejlepší výsledky, tím dříve můžeme celé vyhledávání ukončit. Toto pozorování je nesmírně zajímavé. Námi navržený algoritmus se totiž ukazuje nebýt příliš náchylný na skokové zvýšení počtu dat v databázi.

Nakonec zbývá otestovat rychlost nad kompletními daty pražské MHD, což je 45 146 jízd (a 883 568 zastavení). Výsledky můžeme vidět v tabulce 8.7.

Stanice z	**Stanice do**	**Max. počet přestupů**	**Doba hledání [s]**
Bělocerkevská	Dejvická	1	0.261
Bělocerkevská	Dejvická	4	0.973
Dejvická	Karlovo náměstí	1	0.164
Dejvická	Karlovo náměstí	4	0.133
Černý Most	Hybšmanka	1	0.478
Černý Most	Hybšmanka	4	4.259
Černý Most	Terminál 1	1	1.283
Černý Most	Terminál 1	4	14.920
Florenc	Anděl	1	0.138
Florenc	Anděl	4	0.129
Výstaviště Holešovice	Bělocerkevská	1	1.721
Výstaviště Holešovice	Bělocerkevská	4	0.908

Tabulka 8.7: Rychlost algoritmu pro 45 146 jízd (grafová databáze)

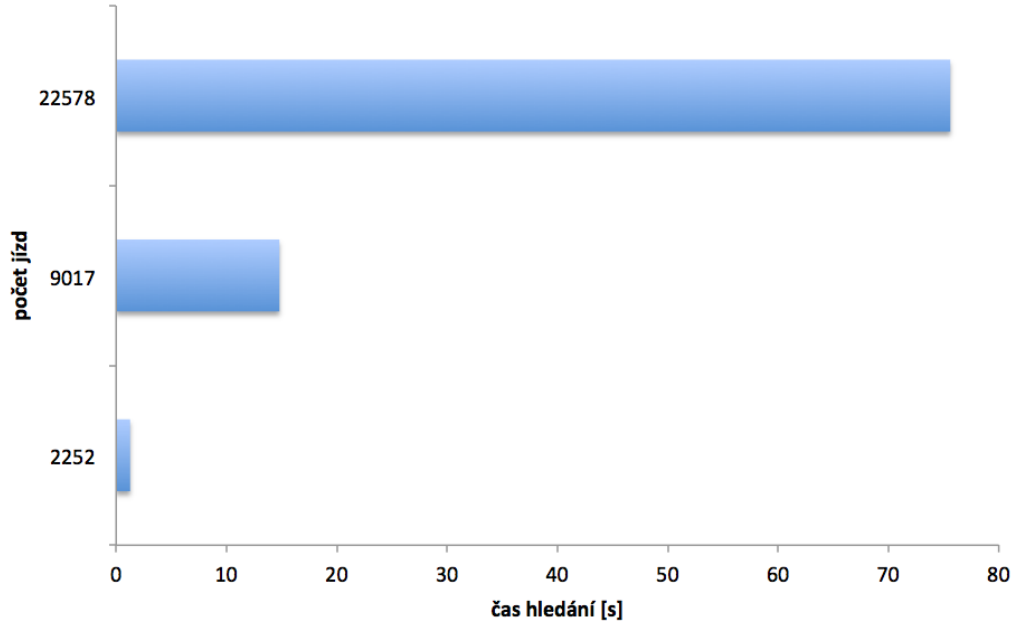
Většina časů se opět vešla do vteřiny. Výrazně déle trvalo pouze vyhledání spojení z Černého Mostu na Terminál 1 (Letiště). Opět je to dáno faktem, že z Černého Mostu vyjíždí spousta spojů, zatímco na Terminál 1 jezdí pouze několik autobosů. Pro vyhledání je tedy třeba prohledat poměrně velký prostor dat. V případě, že se pokusíme vyhledat spojení v opačném směru (z Terminálu 1 na Černý Most), jsou výsledky nalezeny výrazně rychleji. Toto zjištění pro nás nicméně není při bližším studiu závislostí v jízdních řádech a znalosti principu vyhledávacího algoritmu příliš překvapivé.

8.2.3.1 Grafické znázornění rychlostí

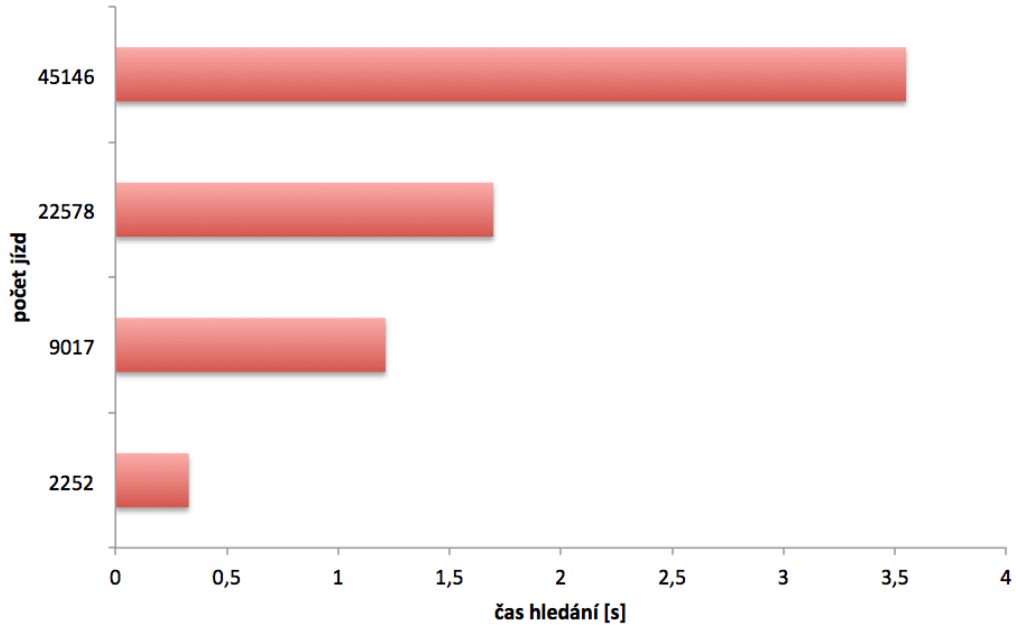
Pro přehlednost si ještě výsledky měření rychlosti představíme v grafickém znázornění. Vždy tak v grafu zobrazíme průměrný čas nutný k vyhledání na všech námi otestovaných trasách pro jednotlivé velikosti dat. Výsledky rychlosti vyhledávání nad relační databází zobrazíme pouze pro $P = 1$, protože pro $P > 1$ nebyl nad většími daty nalezen výsledek ani v čase nad jednu minutu. Naopak v grafu rychlosti vyhledávání nad grafovou databází si zobrazíme výsledky měření pro $P = 4$. Nakonec si rychlosti obu algoritmů porovnáme přehledně v jednom grafu. Pro tyto účely musíme opět zobrazit data pouze pro $P = 1$.

Na obrázku 8.1 můžeme vidět grafický přehled rychlosti vyhledávání nad relační databází pro $P = 1$.

Na obrázku 8.2 můžeme vidět grafický přehled rychlosti vyhledávání nad grafovou databází pro $P = 4$.

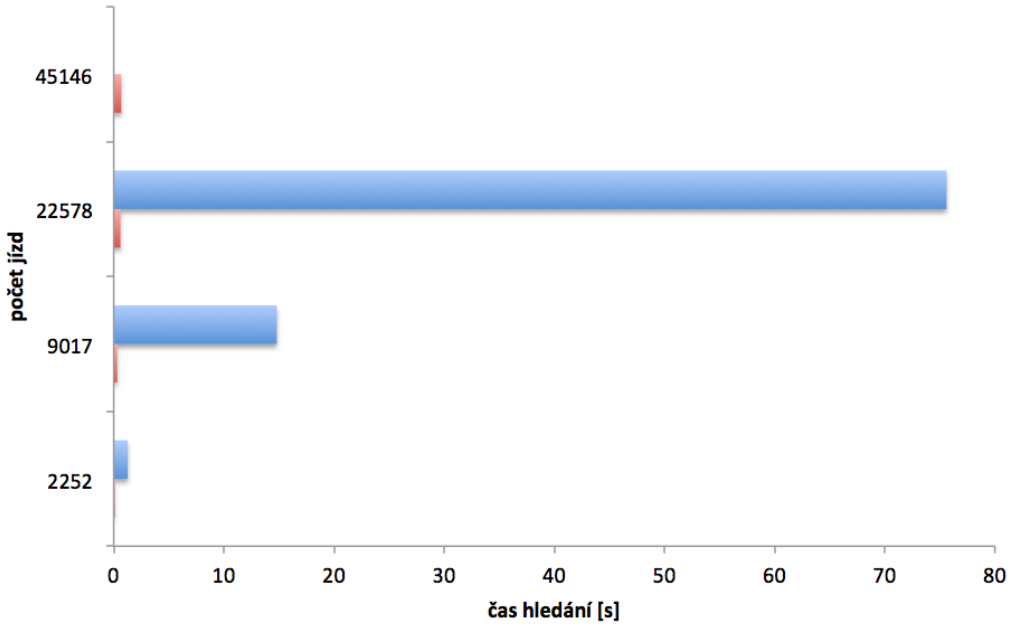


Obrázek 8.1: Rychlost vyhledávání nad relační databází pro $P = 1$. Osa x reprezentuje čas vyhledávání v sekundách, osa y reprezentuje počet jízd v databázi.



Obrázek 8.2: Rychlost vyhledávání nad grafovou databází pro $P = 4$. Osa x reprezentuje čas vyhledávání v sekundách, osa y reprezentuje počet jízd v databázi.

Na obrázku 8.3 můžeme vidět grafické porovnání rychlostí vyhledávání nad grafovou i relační databází pro $P = 1$. Připomeňme, že nad kompletními daty se nepodařilo pomocí algoritmu nad relační databází nalézt výsledek ani v čase nad jednu minutu, tato data tedy v grafu nejsou zobrazena.



Obrázek 8.3: Srovnání rychlostí vyhledávání nad relační (modře) a grafovou (červeně) databází pro $P = 1$. Osa x reprezentuje čas vyhledávání v sekundách, osa y reprezentuje počet jízd v databázích.

8.2.4 Srhnutí testů rychlosti

Algoritmus nad relační databází se bohužel příliš neosvědčil. Ukazuje se, že je vhodný pouze pro menší data a zejména pro menší počet přestupů.

Naopak algoritmus nad grafovou databází nalezl výsledky i nad kompletními daty z pražské MHD téměř vždy do jedné vteřiny. Pouze ve výjimečných případech trvalo nalezení déle, zejména pak ve specifických případech, kdy hledání probíhá z velmi frekventované výchozí stanice do cílové stanice, kam jezdí jen malé množství spojů.

Markantní rozdíl mezi oběma přístupy pro vyhledávání spojů je dán nejen faktem, že přístup k datům v grafu je rychlejší, ale také tím, že v grafu hledáme opravdu jen pareto-optimální cesty. Po nalezení n nejlepších výsledků tak můžeme vyhledávání ukončit, protože lepší výsledky zcela jistě neexistují. Naopak algoritmus nad relační databází data prochází v *náhodném* pořadí (viz kapilota 5.5.4.3) a i po nalezení n prvních výsledků musíme v hledání pokračovat dále, protože stále mohou existovat výsledky lepší. Vyhledávání můžeme ukončovat pouze na těch zastaveních, jejichž absolutní čas je vyšší než absolutní čas alespoň n již nalezených výsledků.

Nabízelo by se proto nové otestování algoritmu nad relační databází pro menší časový interval vyhledávání. To by skutečně přineslo zrychlení vyhledávání (protože by algoritmus vyhledával nad menšími daty), bohužel zkrácení časového intervalu pro vyhledávání nelze doporučit. I v rámci MHD, kde jsou spojení poměrně pravidelná a častá, totiž zejména v nočních hodinách může dojít k situaci, kdy dlouho mezi S_v a S_c žádné spojení neexistuje. V těchto případech by tedy vyhledávač nevrátil žádné výsledky, což je pro nás nežádoucí stav. Navíc úpravou vstupních parametrů pouze jedné implementace vyhledávání bychom znehodnotili možnost srovnání obou implementací.

8.3 Testování správnosti výsledků

Otestovat, zda algoritmy vrací *správné* výsledky, je poměrně obtížné. Je totiž nejasné, co vůbec můžeme nazvat správným výsledkem. Algoritmus totiž vrácené výsledky vybírá na základě porovnávání jednotlivých cest, které jsme definovali v kapitole 5.5.4. Může se tedy stát, že algoritmus bude preferovat jiné spojení, než které by v ideálním případě preferoval cestující. Toto je bohužel skutečnost, která bude vždy platit napříč všemi existujícími vyhledávači.

8.3.1 Testování platnosti všech zastavení

Otestovat v každém případě můžeme, zda jsou všechny vrácené výsledky validní. Tedy zda jsou všechna zastavení, kterými nalezené cesty vedou, platná pro daný den. Otestovat můžeme i správné pořadí vrácených výsledků a jejich počet.

V aplikaci k tomuto účelu vznikla sada jednotkových a integračních testů, které tyto vlastnosti kontrolují. Jednotkovými testy jsou pokryty techniky pro závěrečné řazení a výběr výsledků. V rámci intergračních testů je pak v cyklech vždy kontrolováno nalezení výsledků mezi náhodnými stanicemi. Pro všechny vrácené výsledky pak můžeme zkontrolovat platnost všech zastavení pro daný den.

8.3.2 Testování kvality výsledků

Protože v rámci vyhledávání nad grafovou databází máme k dispozici kompletní data z pražské MHD, můžeme vrácené výsledky pro konkrétní zadání porovnat s výsledky, které vrací vyhledávač IDOS. Bohužel pro účely této práce byla použity data z roku 2013, nicméně většina jízdních řádů zůstává od té doby velmi podobná.

Všechny nalezené výsledky pro vyhledávání, které jsme prováděli v kapitole 8.2, tedy byly porovnány s výsledky pro shodné zadání ve vyhledávači IDOS. Po porovnání obou vyhledávačů můžeme konstatovat, že náš algoritmus vždy nalezne velmi podobné výsledky. Volba spojů je téměř ve všech případech shodná, rozdíl ovšem nastává u přestupů. Vyhledávač IDOS totiž nechává časový interval pro přestup výrazně vyšší než náš vyhledávač, který má pevně daný minimální časový interval pro přestup (1 minuta). Můžeme tak říci, že náš vyhledávač v některých případech vrátí lepší výsledky (protože kratší časový interval na přestup opravdu stačí), zatímco v některých případech může vrátit výsledky, kdy se nemusí podařit přestup stihnout. Tohoto si ale jsme vědomi již od kapitoly 4.2.1, kde jsme pro účely této práce zavedli konstatní počet minut nutných na přestup.

Kapitola 9

Závěr

Na závěr si shrneme výsledky této práce a navrhneme možné další pokračování ve výzkumu na téma využití grafových databází pro vyhledávání spojů veřejné dopravy.

9.1 Současný stav

Nejprve jsme podrobně analyzovali existující implementace vyhledávačů spojů v České republice, abychom mohli navrhnout vlastní vyhledávač, který bude umožňovat vyhledávání na základě vybraných vstupních parametrů. Tyto parametry byly vybrány právě z již existujících vyhledávačů a podrobně jsme si je přestavili v kapitole 4.

Současně jsme se seznámili s detaily implementací vybraných vyhledávačů a zjistili tak, že vlastní vyhledávání řeší v rámci operační paměti. Toto zjištění velkým způsobem podpořilo motivaci této práce.

9.2 Návrh algoritmů pro vyhledávání v jízdních řádech

Díky podrobné analýze vyhledávačů a jízdních řádů jsme mohli navrhnout vlastní postup pro vyhledávání. Jako výchozí data, která jsme využili pro naplnění databází, jsme zvolili data ve standardizovaném formátu GTFS.

Nejprve jsme navrhli detailní postup vyhledávání nad relační databází. Již při návrhu tohoto algoritmu jsme ale navrhli i několik postupů, které poté byly využity i při návrhu vyhledávání nad grafovou databází. Struktura grafu, nad kterým vyhledávání probíhá, byla z velké části ovlivněna v minulosti již představenými modely. Strukturu jsme pouze mírně pozměnili a přizpůsobili více našim požadavkům.

Již při navrhování řešení se ukázalo, že grafová struktura je pro ukládání dat jízdních řádů, ale i pro následné vyhledávání, mnohem vhodnější než relační databáze. Zejména proto, že nad grafovou strukturou je možné implementovat algoritmus pro hledání pouze pareto-optimálních cest, který jsme také navrhli.

9.3 Implementace

Implementace vyhledávače nad relační databází byla velmi přímočará, protože jsme všechny aspekty vyhledávání definovali již při návrhu algoritmu.

Implementace grafové databáze a vyhledávání nad touto databází pak byly také poměrně jednoduché. Využili jsme totiž možností dobře zpracovaného Traversal API, které slouží pro vyhledávání nad databází Neo4j. Díky tomuto API jsme byli schopni do vyhledávání jednoduchým způsobem zakomponovat všechny námi požadované podmínky a omezení a přitom jsme si vystačili pouze s pár Java třídami. Tento fakt hodnotíme nadmíru pozitivně. Existující vyhledávače, které vyhledávají v rámci operační paměti, totiž musí vytvářet složité struktury dat, ve kterých musí držet všechny závislosti mezi daty. Nad těmito daty teprve poté probíhá vlastní vyhledávání. V našem případě ale máme data stále na jednom místě a traverzování grafem nám velkým dílem předurčuje Traversal API. Pozor jsme si tedy museli dát zejména na správnou implementaci prioritní fronty a správné kontrolování zpracovávaných vrcholů.

Výsledkem implementace je funkční aplikace, která vyhledávání spojů umožňuje. Tato aplikace bezezbytku splňuje požadavky, které jsme definovali v kapitole 4.

9.4 Testování

Nakonec jsme se věnovali testování obou navržených algoritmů. Bohužel se ukázalo, že algoritmus pro vyhledávání nad relační databází selhává pro velká data nebo větší počet přestupů. Můžeme jej tak doporučit k nasazení pouze u menších soukromých dopravců, kteří neprovozují větší množství spojů.

Naopak vyhledávací algoritmus nad grafovou databází se ukázal být velmi rychlý. Nad kompletními daty z pražské MHD dokázal vyhledat většinu spojů do jedné vteřiny. Pro některé specifické trasy mu sice vyhledávání zabralo delší dobu, i tak ale výsledky měření rychlosti můžeme označit za velmi dobré.

Zároveň je nutné poukázat na fakt, že pro testování rychlosti byla použita data z jízdních řádů, která jsou svým rozsahem druhá největší v České republice [2]. Pokud si zároveň uvědomíme velkou hustotu pražské dopravní sítě, nezbývá než výslednou rychlost vyhledávání označit za výbornou a velmi nadějnou pro budoucí uplatnění.

Databáze Neo4j se tedy ukázala jako výborná volba pro tuto práci a nezbývá než konstatovat, že její možnosti jsou již velmi pokročilé, a je tedy možné její využití i v rámci složitých aplikací a velkých dat.

9.5 Budoucí možnosti

Na základech této práce je možné vybudovat velmi rychlý a chytrý vyhledávač spojů hromadné dopravy, který může nalézt uplatnění jak u soukromého dopravce, tak pro účely vyhledávání napříč větším počtem jízdních řádů.

Pro tyto účely je nutné umožnit větší počet vstupních parametrů algoritmu. Dále je možné přidat doplňující informace o spojích, na základě kterých mohou být výsledky vyhledávání lépe určeny. Pro budoucí rozšíření je aplikace i vyhledávací algoritmus velmi dobře uzpůsoben.

Díky robustnímu návrhu datové struktury totiž není problém přidat další vrcholy do grafu. Uzpůsobení chování vyhledávače (tedy porovnávání jednotlivých cest, přidání podmínek výběru vrcholů...) je také velmi jednoduché, protože v rámci traverzování máme k dispozici vždy všechny potřebné údaje o cestě a již nalezených výsledcích vyhledávání.

Literatura

[1] ALLAIN, A. *Recursion in C* [online]. 2011. [cit. 7. 5. 2015]. Dostupné z: <http://www.cprogramming.com/tutorial/c/lesson16.html>.

[2] BLAŽEK, J. Algoritmus pro vyhledávání dopravních spojení v jízdních řádech hromadné dopravy na kapesních zařízeních. Master's thesis, České vysoké učení technické v Praze, Fakulta elektrotechnická, Česká republika, 2009.

[3] BRODAL, G. S. – JACOB, R. Time-dependent Networks as Models to Achieve Fast Exact Time-table Queries. *Electronic Notes in Theoretical Computer Science*. 2004, 92, 0, s. 3 – 15. ISSN 1571-0661. doi: http://dx.doi.org/10.1016/j.entcs.2003.12.019. Dostupné z: <http://www.sciencedirect.com/science/article/pii/S1571066104000040>. Proceedings of {ATMOS} Workshop 2003.

[4] CITIES, S. *INFORMAČNÍ DOPRAVNÍ SYSTÉM VERSUS OTEVŘENÁ DATA* [online]. 2014. [cit. 23. 4. 2015]. Dostupné z: <http://www.scmagazine.cz/article/view/44>.

[5] CROSMAN, P. *Cloud Computing Begins to Gain Traction on Wall Street* [online]. 2009. [cit. 5. 5. 2015]. Dostupné z: <http://www.wallstreetandtech.com/infrastructure/cloud-computing-begins-to-gain-traction-on-wall-street/d/d-id/1261032>.

[6] DIJKSTRA, E. W. A Note on Two Problems in Connexion with Graphs. *NUMERISCHE MATHEMATIK*. 1959, 1, 1, s. 269–271.

[7] EVROPSKÁ KOMISE. *Doprava: Vítězem první soutěže inteligentní mobility se stává ...* [online]. 2012. [cit. 23. 4. 2015]. Dostupné z: <http://europa.eu/rapid/press-release_IP-12-233_cs.htm>.

[8] FINLEY, K. *5 Graph Databases to Consider* [online]. 2011. [cit. 11. 5. 2015]. Dostupné z: <http://readwrite.com/2011/04/20/5-graph-databases-to-consider>.

[9] FREDMAN, M. L. – TARJAN, R. E. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *J. ACM*. July 1987, 34, 3, s. 596–615. ISSN 0004-5411. doi: 10.1145/28869.28874. Dostupné z: <http://doi.acm.org/10.1145/28869.28874>.

[10] Google, Inc. *Google developers: What is GTFS?* [online]. 2012. [cit. 24. 4. 2015]. Dostupné z: <https://developers.google.com/transit/gtfs/>.

[11] KAINULAINEN, P. *Understanding Spring Web Application Architecture: The Classic Way* [online]. 2014. [cit. 11. 5. 2015]. Dostupné z: <http://www.petrikainulainen.net/software-development/design/understanding-spring-web-application-architecture-the-classic-way/>.

[12] KALLA, T. Analýza integrace systému vyhledávání spojů se systémem kolizních informací a její využití. Master's thesis, Masarykova univerzita, Fakulta informatiky, Česká republika, 2015.

[13] KUČERA, F. *Java na serveru: úvod* [online]. 2010. [cit. 11. 5. 2015]. Dostupné z: <http://www.zdrojak.cz/clanky/java-na-serveru-uvod/>.

[14] MALÝ, M. *Školení: PostgreSQL efektivně* [online]. 2010. [cit. 11. 5. 2015]. Dostupné z: <http://www.zdrojak.cz/zpravicky/skoleni-postgresql-efektivne/>.

[15] MIČKA, P. *Java pro začátečníky (21) - Kolekce* [online]. 2011. [cit. 5. 5. 2015]. Dostupné z: <http://www.algoritmy.net/article/34009/Kolekce-21>.

[16] Neo Technology, Inc. *Intro to Cypher* [online]. 2015. [cit. 5. 5. 2015]. Dostupné z: <http://neo4j.com/developer/cypher-query-language/>.

[17] Neo Technology, Inc. *What is a Graph Database?* [online]. 2015. [cit. 5. 5. 2015]. Dostupné z: <http://neo4j.com/developer/graph-database/>.

[18] Neo Technology, Inc. *Top Ten Reasons for Choosing Neo4j* [online]. 2015. [cit. 5. 5. 2015]. Dostupné z: <http://neo4j.com/top-ten-reasons/>.

[19] ORDA, A. – ROM, R. Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length. *Journal of the ACM (JACM)*. 1990, 37, 3, s. 607–625.

[20] ORDA, A. – ROM, R. Minimum weight paths in time-dependent networks. *Networks*. 1991, 21, s. 295–319.

[21] PALLOTTINO, S. – SCUTELLà, M. G. Shortest Path Algorithms in Transportation Models: Classical and Innovative Aspects, 1998.

[22] PASTUCHOVÁ, M. *Open source přebírá v oblasti softwaru klíčovou roli* [online]. 2011. [cit. 11. 5. 2015]. Dostupné z: <http://www.ictmanazer.cz/2011/11/open-source-prebira-v-oblasti-softwaru-klicovou-roli/>.

[23] PEVNÝ, J. *Idos.cz - Jízdní řády na internetu snadno a rychle* [online]. 2010. [cit. 23. 4. 2015]. Dostupné z: <http://www.swmag.cz/718/idos-cz-jizdni-rady-na-internetu-snadno-a-rychle/>.

[24] POLESNÝ, D. *Bileto: Inteligentní nákup vlakových jízdenek, který se zalíbil i Seznamu* [online]. 2014. [cit. 23. 4. 2015]. Dostupné z: <http://www.zive.cz/bleskovky/bileto-inteligentni-nakup-vlakovych-jizdenek-ktery-se-zalibil-i-seznamu/sc-4-a-173047/default.aspx>.

[25] PYRGA, E. et al. Efficient Models for Timetable Information in Public Transportation Systems. *J. Exp. Algorithmics*. June 2008, 12, s. 2.4:1–2.4:39. ISSN 1084-6654. doi: 10.1145/1227161.1227166. Dostupné z: <http://doi.acm.org/10.1145/1227161.1227166>.

[26] Přispěvatelé populárně naučného portálu. *Grafy a grafové algoritmy (4): Hledání nejkratší cesty* [online]. 2015. [cit. 5. 5. 2015]. Dostupné z: <http://popular.fbmi.cvut.cz/it/Stranky/Grafy-a-grafove-algoritmy-4-Hledani-nejkratsi-cesty.aspx>.

[27] Přispěvatelé Wikipedie. *IDOS* [online]. 2014. [cit. 23. 4. 2015]. Dostupné z: <http://cs.wikipedia.org/wiki/IDOS>.

[28] RAMBA, J. *Přehled grafových databází* [online]. 2013. [cit. 11. 5. 2015]. Dostupné z: <http://www.zdrojak.cz/clanky/prehled-grafovych-databazi/>.

[29] RAMBA, J. *Grafová terminologie a dostupné technologie* [online]. 2013. [cit. 5. 5. 2015]. Dostupné z: <http://www.zdrojak.cz/clanky/grafova-terminologie-a-dostupne-technologie/>.

[30] REGIOJET. *RegioJet zahájil přípravy na vstup na Slovensko na trasu Bratislava - Košice* [online]. 2014. [cit. 23. 4. 2015]. Dostupné z: <http://www.studentagency.cz/pro-media/tiskove-zpravy/RegioJet_zahajil_pripravy_na_vstup_na_Slovensko.html>.

[31] SCHULZ, F. Timetable Information and Shortest Paths. Master's thesis, der Universität Fridericiana zu Karlsruhe (TH), der Fakultät für Informatik, Deutschland, 2005.

[32] SCHULZ, F. – WAGNER, D. – WEIHE, K. Dijkstra's Algorithm On-Line: An Empirical Case Study from Public Railroad Transport. *JOURNAL OF EXPERIMENTAL ALGORITHMICS*. 2000, 5, 12, s. 2000.

[33] SEDLÁK, J. *České Bileto chce revoluci v prodeji jízdenek* [online]. 2015. [cit. 23. 4. 2015]. Dostupné z: <http://e-svet.e15.cz/internet/ceske-bileto-chce-revoluci-v-prodeji-jizdenek-online-1170003>.

[34] SKŘIVAN, J. *Databáze a jazyk SQL* [online]. 2000. [cit. 5. 5. 2015]. Dostupné z: <https://www.interval.cz/clanky/databaze-a-jazyk-sql/>.

[35] solid IT gmbh. *DB-Engines Ranking of Graph DBMS* [online]. 2015. [cit. 13. 5. 2015]. Dostupné z: <http://db-engines.com/en/ranking/graph+dbms>.

[36] STĚHULE, P. *Jemný úvod do jazyka PL/pgSQL PostgreSQL* [online]. 2009. [cit. 5. 5. 2015]. Dostupné z: <http://postgresql.ok.cz/doc/plpgsql.html>.

[37] TERRILL, G. *Neo4j - an Embedded, Network Database* [online]. 2008. [cit. 11. 5. 2015]. Dostupné z: <http://www.infoq.com/news/2008/06/neo4j>.

[38] The PostgreSQL Global Development Group. *Connection Pools and Data Sources* [online]. 2015. [cit. 5. 5. 2015]. Dostupné z: <http://www.postgresql.org/docs/7.4/static/jdbc-datasource.html>.

[39] VYLEŤAL, M. *Tomáš Chlebničan (CHAPS): Data, která má Bileto a Seznam, pocházejí od nás* [online]. 2014. [cit. 23. 4. 2015]. Dostupné z: <http://www.lupa.cz/clanky/tomas-chlebnican-chaps-data-ktera-ma-bileto-a-seznam-pochazeji-od-nas/>.

[40] H. DARWEN:, C. J. D. *A Guide to the SQL standard : a users guide to the standard database language SQL*. Addison Wesley, 4th edition, 1997. In English.

[41] ZAJÍČKOVÁ, L. – BŘEČKA, P. Datový model dopravní sítě pro správu dat a řízení veřejné hromadné dopravy. [online], 2013. Dostupné z: <gis.tuzvo.sk/tiki-download_file.php?fileId=140>.

Příloha A

Obsah přiloženého CD

- `abstract` ..... abstrakt práce
  - `abstract-cz.txt`
  - `abstract-en.txt`
- `readme.txt` ..... popis obsahu CD
- `src`
  - `db`
    - `neo4jDatabase.zip` ..... data pro databázi Neo4j
    - `postgreDatabase.zip` ..... data pro databázi PostgreSQL
  - `searchApplication.zip` ..... zdrojové kódy aplikace
  - `chaluja7-bachelor-thesis-2015.zip` ..... zdrojová forma práce ve formátu LaTeX
- `text`
  - `chaluja7-bachelor-thesis-2015.pdf` ..... text práce ve formátu PDF

Příloha B

GTFS formát dat

V této příloze jsou ukázky jednotlivých CSV souborů z balíku GTFS. Pro lepší čitelnost jsou tato data již zformátována do sloupců. V originálních souborech jsou záznamy odděleny čárkou.

	A	B	C	D	E	F
1	stop_id	stop_name	stop_lat	stop_lon	location_type	parent_station
2	U953Z102	Skalka	50.068435	14.507169	0	U953N1
3	U713Z102	Strašnická	50.073336	14.490091	0	U713N2
4	U921Z102	Želivského	50.07854	14.474891	0	U921N3
5	U118Z102	Flora	50.078288	14.461886	0	U118N4

Obrázek B.1: Soubor stops.txt z GTFS balíku dat

	A	B	C	D	E	F
1	agency_id	agency_name	agency_url	agency_timezone	agency_lang	agency_phone
2	1	Dopravni podnik hl. m. Prahy, akciova spolecnost	http://www.dpp.cz	Europe/Prague	cs	296191817
3	6	Jaroslav Stepanek	http://www.ropid.cz	Europe/Prague	cs	284821024
4	11	CSAD POLKOST, spol. s r.o.	http://www.ropid.cz	Europe/Prague	cs	321697274
5	63	ABOUT ME s.r.o.	http://www.ropid.cz	Europe/Prague	cs	267290540

Obrázek B.2: Soubor agency.txt z GTFS balíku dat

	A	B	C	D	E	F
1	route_id	agency_id	route_short_name	route_long_name	route_type	route_color
2	L991D1	1	A		1	408000
3	L992D1	1	B		1	FFFF00
4	L993D1	1	C		1	FF0000
5	L1D1	1	1		0	

Obrázek B.3: Soubor routes.txt z GTFS balíku dat

	A	B	C	D	E
1	route_id	service_id	trip_id	trip_headsign	shape_id
2	L991D1	1	1	Dejvická	1
3	L991D1	1	2	Skalka	2
4	L991D1	1	3	Dejvická	1
5	L991D1	1	4	Depo Hostivař	3

Obrázek B.4: Soubor trips.txt z GTFS balíku dat

	A	B	C	D	E
1	trip_id	arrival_time	departure_time	stop_id	stop_sequence
2	1	6:37:10	6:37:40	U953Z102	1
3	1	6:39:55	6:40:25	U713Z102	2
4	1	6:42:05	6:42:35	U921Z102	3
5	1	6:44:00	6:44:20	U118Z102	4

Obrázek B.5: Soubor stop_times.txt z GTFS balíku dat

	A	B	C	D	E	F	G	H	I	J
1	service_id	monday	tuesday	wednesday	thursday	friday	saturday	sunday	start_date	end_date
2	1	1	1	1	1	1	0	0	20130701	20130831
3	2	0	0	0	0	0	1	0	20130625	20130831
4	3	0	0	0	0	0	0	1	20130625	20130831
5	4	1	1	1	1	0	0	0	20130625	20130630

Obrázek B.6: Soubor calendar.txt z GTFS balíku dat