TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Paralelizace vyhledávání spojení v MHD

Diplomová práce

Studijní program: N2612 Elektrotechnika a informatika
Studijní obor: Informační technologie

Autor práce: **Bc. Michal Křepelka**
Vedoucí práce: doc. Ing. Otto Severýn, Ph.D.
Ústav mechatroniky a technické informatiky

Liberec 2020

TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Zadání diplomové práce

Paralelizace vyhledávání spojení v MHD

Jméno a příjmení: **Bc. Michal Křepelka**
Osobní číslo: M18000147
Studijní program: N2612 Elektrotechnika a informatika
Studijní obor: Informační technologie
Zadávající katedra: Ústav mechatroniky a technické informatiky
Akademický rok: **2019/2020**

Zásady pro vypracování:

1. Proveďte analýzu výkonu aplikace vytvořené v rámci vaší bakalářské práce a identifikujte kritická místa.
2. Navrhněte možná zlepšení – optimalizace datových struktur, paralelizace vyhledávání za použití více jader procesoru, případně jiné.
3. Navržená zlepšení implementujte do aplikace, a to s důrazem na minimalizaci množství ukládaných dat.
4. Otestujte takto zlepšenou aplikaci na reálných datech, výsledky testu vyhodnoťte a analyzujte.

Rozsah grafických prací:	dle potřeby dokumentace
Rozsah pracovní zprávy:	40–50 stran
Forma zpracování práce:	tištěná/elektronická
Jazyk práce:	Čeština

Seznam odborné literatury:

[1] DEO, Narsingh. Graph theory with applications to engineering and computer science. Dover edition. Mineola, New York: Dover Publications, 2016. ISBN 978-0486807935.

[2] BAST, Hannah, CARLSSON, Erik, EIGENWILLIG, Arno, GEISBERGER, Robert, HARRELSON, Chris, RAYCHEV, Veselin, and VIGER, Fabien. Fast Routing in Very Large Public Transportation Networks using Transfer Patterns. In Mark de Berg and Ulrich Meyer, editors, ESA, volume 6346 of Lecture Notes in Computer Science, pages 290?301, Springer, 2010. ISBN 978-3-642-15775-2.

[3] KŘEPELKA, Michal. Offline vyhledávání spojů na platformě Android. Bakalářská práce, UJEP, 2017.

Vedoucí práce:	doc. Ing. Otto Severýn, Ph.D. Ústav mechatroniky a technické informatiky
Datum zadání práce:	10. října 2019
Předpokládaný termín odevzdání:	18. května 2020

L.S.

prof. Ing. Zdeněk Plíva, Ph.D.
děkan

doc. Ing. Milan Kolář, CSc.
vedoucí ústavu

V Liberci dne 10. října 2019

Prohlášení

Prohlašuji, že svou diplomovou práci jsem vypracoval samostatně jako původní dílo s použitím uvedené literatury a na základě konzultací s vedoucím mé diplomové práce a konzultantem.

Jsem si vědom toho, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu Technické univerzity v Liberci.

Užiji-li diplomovou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti Technickou univerzitu v Liberci; v tomto případě má Technická univerzita v Liberci právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Současně čestně prohlašuji, že text elektronické podoby práce vložený do IS/STAG se shoduje s textem tištěné podoby práce.

Beru na vědomí, že má diplomová práce bude zveřejněna Technickou univerzitou v Liberci v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů.

Jsem si vědom následků, které podle zákona o vysokých školách mohou vyplývat z porušení tohoto prohlášení.

31. května 2020 Bc. Michal Křepelka

Poděkování

Rád bych poděkoval své rodině za jejich podporu a doc. Ing. Otto Severýnovi, Ph.D. za jeho neocenitelné rady při vedení této práce.

Paralelizace vyhledávání spojení v MHD

Abstrakt

Tato práce formalizuje problém vyhledávání spojení ve veřejné dopravě a dále diskutuje používané grafové i negrafové algoritmy a modely. V potaz je brána použitelnost navrhovaného řešení v mobilních zařízeních, která vyhledávají spojení offline. Tato zařízení nemají stálý přístup k výkonnému serveru, který by vyhledávání provedl. Z toho důvodu musí být data jízdních řádů uchovávána v kompaktní podobě a zvolený model musí počítat s omezeným výkonem. Tato práce uvažuje možnosti využití paralelismu při vyhledávání spojení a za tímto účelem navrhuje heuristiku pro odhad optimálních časů odjezdů, která umožní paralelizaci vyhledávání několika spojení současně. Tato heuristika byla otestována nad časově závislým grafem.

Klíčová slova: Veřejná doprava, jízdní řád, vyhledávání spojení, paralelismus, heuristika

Parallelization of Connection Search in Public Transport

Abstract

This thesis formalizes the problem of connection search in public transport and further discuses available graph and non-graph based algorithms and models. The usability of the proposed solution on mobile devices, that perform the connection search offline, is taken into account. These devices do not have permanent access to powerful server that could perform the search. For this reason, timetable data must be stored in a compact form and chosen model must take into account limited performance. This thesis considers the possibility of using parallelism in connection search and for this purpose proposes a heuristic for estimating optimal departure times, which will allow parallelization of searching several connections simultaneously. This heuristic has been tested on time-dependent graph.

Keywords: Public transport, timetable, connection search, parallelism, heuristic

Obsah

Seznam obrázků

Seznam zkratek

CSA	Connection Scan Algorithm
MHD	Městská hromadná doprava
TDG	Time-dependent graph, tj. časově závislý graf
TEG	Time-expanded graph, tj. časově rozšířený graf
DAG	Directed Acyclic Graph, tj. orientovaný acyklický graf
RAPTOR	Round-Based Public Transit Optimized Router
EA	Earliest Arrival, tj. nejdřívější příjezd
EAP	Earliest Arrival Problem, tj. problém nejdřívějšího příjezdu
MNT	Minimum Number of Transfers, tj. nejmenší počet přestupů
GBR	Guidebook Route, tj. trasa z průvodce

Úvod

Veřejná doprava představuje důležitou alternativu k silniční dopravě, protože ne každý cestující vlastní řidičské oprávnění a na některá místa není vjezd povolen. Zásadním rozdílem v obou zmíněných metodách přepravy je jízdní řád. Díky němu se vyhledávání optimálního spojení stává složitější jak z pohledu cestovatele tak i z pohledu algoritmického. Dobré plánování cesty je tak o to důležitější. [1]

Oblast algoritmů pro vyhledávání spojení ve veřejné dopravě zažívá zejména od roku 2010, kdy byly publikovány přestupní vzorce, obrovský rozmach. Objevilo se několik nových algoritmů, které pro svou práci nevyžadují graf, a doplnili tak klasicky používaný Dijkstrův algoritmus. Díky tomu se daří vyhledávání spojení zrychlit a jinak vylepšit.

S příchodem nových technologií je možné vyhledávat spojení ve veřejné dopravě i na malých mobilních zařízení. Jejich výkon je však už natolik dostatečný, že ne vždy je nutné mít připojení k internetu a vyhledávání spojení realizovat na výkonném serveru, který tuto službu nabízí.

Cílem aplikací vyhledávajících optimální spojení je poskytnou ho uživateli dostatečně rychle. Jednou z možných cest, jak nějakou aplikace zrychlit, je využití paralelismu. V současné době je možné paralelizovat zejména profilovací dotazy, tj. takové dotazy, které hledají všechna optimální spojení jedoucí v nějakém časovém intervalu. Tento typ paralelizace se však svou povahou hodí spíše pro servery.

Pokusy o paralelizaci časového dotazu, tj. dotazu na nalezení optimálního spojení jedoucího mezi stanicemi po zadaném čase, byly do nedávna neúspěšné. Jedinou výjimku tvoří negrafový algoritmus RAPTOR publikovaný v [12].

Na paralelizaci aplikací vyhledávajících optimální spojení ve veřejné dopravě lze ale nahlížet i jinak a nesoustředit se přímo na vyhledávací algoritmus. Typicky totiž tyto aplikace nabízí pro vyhledání více než jedno spojení a právě zde je další možnost, jak uplatnit paralelizaci výpočtu.

V kapitole 1 bude nejprve zformalizován problém nalezení optimálního spojení ve veřejné dopravě a dále budou popsány současně používané modely a algoritmy. Probrány budou modely, které využívají pro nalezení spojení graf a Dijkstrův algoritmus, založené na TDG nebo TEG. Dále budou zmíněny i přestupní vzorce, které se od již zmíněných modelů liší tím, že vyhledávání spojení probíhá na grafu, který je třeba nalézt během předchozích výpočtů. Nakonec budou v závěru kapitoly popsány nové negrafové algoritmy CSA a RAPTOR.

V kapitole 2 bude navržena heuristika pro odhad časů odjezdu, která pro zadaný dotaz na spojení predikuje časy odjezdu optimálních spojení. Díky tomu je na základě tohoto odhadu možné spustit paralelní vyhledávání několika spojení a to pomocí

libovolného vyhledávacího algoritmu popsaného v kapitole 1. V rámci experimentu popsaného v kapitole 6 bude navrhovaná heuristika otestována při vyhledávání spojení na TDG postaveného na datech pro pražskou MHD.

Součástí experimentu bylo i vytvoření implementace CSA. Podařilo se ověřit, že tento nový přístup je oproti vyhledávání Dijkstrovým algoritmem nad TDG rychlejší, avšak se nepodařilo implementovat podporu pro modelování reálné doby na přestup a algoritmus tak nemusí nabízet optimální výsledky.

Tato práce navazuje na bakalářskou práci [7] a jejím cílem je navrhnout algoritmická zlepšení aplikace pro offline vyhledávání spojení na platformě Android, která byla v rámci [7] publikována.

1 Modely a algoritmy

Pro vyhledávání spojení je možné použít dva rozdílné přístupy. První je založen na modelování jízdního řádu do grafu, nad kterým se použije Dijkstrův algoritmus. Druhou možností je využít některý z moderních algoritmů, který sice intuitivně využívá nějaký graf, ale reálně ho pro vyhledávání spojení nepotřebuje stavět (v rámci této práce budou označovány jako *negrafové algoritmy*).

V této sekci bude nejprve zformalizován problém vyhledávání spojení v městské hromadné dopravě (MHD) respektive ve veřejné dopravě. Následně budou popsány používané grafové modely, na kterých se spojení vyhledává Dijkstrovým algoritmuem, a také novější negrafové algoritmy.

1.1 Formalizace problému

Pro zjednodušení dalšího popisu tento text navazuje na zahraniční literaturu a libovolný dopravní prostředek používaný v MHD (vlak, autobus, tramvaj, metro,...) se v něm bude označovat pouze jako vlak. Typicky je v železniční přepravě každý vlak určen jednoznačným identifikátorem, který lze intuitivně chápat jako unikátní číslo jízdy. Podle tohoto unikátního identifikátoru lze zjistit startovní zastávku a čas odjezdu vlaku z této zastávky. V případě MHD je vlak chápán analogicky, a tedy ho určuje startovní zastávka a čas odjezdu z ní.

Dopravní síť MHD je dána jízdním řádem, který představuje množinu vlaků $\mathscr{Z}$ jedoucích na některé z linek, množinu stanic $\mathscr{S}$, které tyto vlaky obsluhují a časy příjezdů a odjezdů vlaků do těchto zastávek. [7]

Příklad jízdního řádu lze vidět v tabulce 1.1 převzaté z [7]. Tato tabulka zobrazuje jízdní řád pro dvě linky MHD skládající se ze tří sloupců. Sloupec „Stanice" obsahuje vzestupně seřazené stanice, kde linka zastavuje. Symbol $*$ ve sloupci „Dojezd" označuje zastávku, ve které je tento jízdní řád vyvěšen a další údaje představují dobu v minutách, za kterou se lze do stanice zapsané na stejném řádku dostat. Intuitivně tak lze symbol $*$ chápat jako 0. Poslední sloupec „Odjezdy" obsahuje časy odjezdů dané linky ze zastávky, kde je tento jízdní řád vyvěšen. Pokud nebude řečeno jinak, budou v dalším popisu všechny jízdní řády uvádět odjezdy pro jediný den (např. pondělí). [8]

Výhodou popsané reprezentace je možnost uchovávat pouze jízdní řád ve stanici, ze které linka vyjíždí, a v ostatních stanicích této linky ho lze jednoduše odvodit pomocí časů dojezdu do ní. [7]

Tabulka 1.1: Jízdní řád MHD ve startovní stanici jednotlivých linek [7]

(a) Linka 1

Dojezd	Stanice	Odjezdy
*	A	8:45
10	B	
15	C	
18	D	
24	E	

(b) Linka 2

Dojezd	Stanice	Odjezdy
*	A	8:55
5	F	
7	C	
10	D	
14	G	

Každá jízda se skládá z *elementárních spojení*, které tvoří *množinu elementárních spojení* $\mathscr{C}$. Elementární spojení $c \in \mathscr{C}$ je uspořádaná pětice $c = (Z, S_d, S_a, t_d, t_a)$, kde $Z \in \mathscr{Z}$, $S_d, S_a \in \mathscr{S}$, t_d je čas odjezdu a t_a čas příjezdu. Navíc zastávka S_a musí bezprostředně následovat za zastávkou S_d (vlak mezi těmito zastávkami nikde nezastavuje) a $t_d \leq t_a$. Elementární spojení c se interpretuje tak, že vlak Z vyjíždí ze stanice S_d v čase t_d a přijíždí do stanice S_a v čase t_a. [9]

Čas je představován počtem minut od půlnoci prvního dne v týdnu. Každý čas t v jízdním řádu tak lze zapsat ve tvaru $t = a \times 1440 + b$, kde $a \in [0, 6]$ je pořadí dne v týdnu (pondělí až neděle) číslované od nuly a $b \in [0, 1439]$ je počet minut od půlnoci daného dne. Dělením modulo je tak možné z každého času t odvodit čas v aktuálním dnu jako $a = t \mod 1440$ a naopak pomocí celočíselného dělení $b = t \div 1440$ je možné odvodit den v týdnu. [7]

Elementární spojení, která jsou na dané lince realizována stejným vlakem, se sdružují do jízd a ty se následně sdružují do tras. Trasa je množina jízd, které obsluhují přesnou posloupnost stanic ve stejném pořadí[1]. V rámci této práce se navíc předpokládá, že vlaky obsluhující dvě elementární spojení jedoucí mezi dvěma zastávkami na jedné trase se nemohou předjíždět, tj. přepravní síť veřejné dopravy se chová jako fronta.

Zavede se značení takové, že pokud n je n-tice a x označuje její prvek, potom $x(n)$ je hodnota prvku x v n-tici n. Například pro elementární spojení $c = (Z, S_d, S_a, t_d, t_a)$ tak $S_d(c)$ označuje stanici, ze které elementární spojení c vyjíždí. [7]

Cyklický rozdíl *cycle-difference*(t_1,t_2) dvou časů t_1, t_2, kde $t_1 \leq t_2$, je definován jako nejmenší nezáporné celé číslo d takové, že $d \equiv t_2 - t_1 \mod 1440$. Dobou přepravy $d(c)$ se potom rozumí délka elementárního spojení $c = (Z, S_d, S_a, t_d, t_a)$ daná vztahem $d(c)$=*cycle-difference*($t_a(c)$,$t_d(c)$). [7, 9]

Aby se ve stanici dal umožnit přestup mezi spojeními, které nepatří do stejné jízdy, definuje se pro každou stanici $S \in \mathscr{S}$ *minimální doba přestupu transfer(S).* Formálně lze ve stanici S přestoupit z elementárního spojení c_1 na c_2 právě když $S_a(c_1) = S_d(c_2) = S$, tj. c_1 ve stanici S končí a c_2 začíná, a $transfer(S) \leq t_d(c_2) - t_a(c_1)$, tedy rozdíl mezi příjezdem vlaku $Z(c_1)$ do stanice S a odjezdem vlaku $Z(c_2)$ z této stanice je větší nebo roven minimální době přestupu. [7]

[1] Alternativně lze trasu chápat jen jako posloupnost zastávek, která je v daném pořadí obsluhována množinou jízd

V jízdním řádu uvedeném v tabulce 1.1 je možné si všimnout jisté nedokonalosti jeho reprezentace. Pravděpodobně kvůli zlepšení čitelnosti pro cestující neobsahuje informace o času příjezdu spoje do stanice. Při modelování přestupu z jednoho spoje na druhý tak může (a zpravidla bude) docházet k nepřesnostem, kdy spoj může delší dobu čekat ve stanici, ale tato doba čekání není nikde zaznamenána. Přestup tak bude namodelován chybně, přestože by byl splněn minimální čas pro přestup.

Nechť $P = (c_1, \ldots, c_k)$ je posloupnost elementárních spojení a zavede se značení:

- $dep_i(P) = t_d(c_i)$
- $arr_i(P) = t_a(c_i)$
- $S_d(P) = S_d(c_1)$
- $S_a(P) = S_a(c_k)$
- $dep(P) = dep_1(P)$
- $arr(P) = arr_k(P)$
- $d(P) = arr(P) - dep(P)$

Jestliže pro každé $i = 1, \ldots, k-1$ platí:

1. $S_d(c_{i+1}) = S_a(c_i)$.
2. $Z(c_{i+1}) = Z(c_i)$ nebo $dep_{i+1}(P) - arr_i(P) \geq transfer(S_a(c_i))$.

Potom se podle [5, 7] posloupnost P nazývá *konzistentní spojení* jedoucí ze stanice $S_d(P)$ v čase $dep(P)$ do stanice $S_a(P)$ s časem příjezdu $arr(P)$ a délkou (dobou přepravy) $d(P)$.

Cílem vyhledávání spojení je zodpovědět dotaz na nalezení konzistentního spojení ze stanice A do stanice B, který odjíždí v čase t nebo později. Takový dotaz se bude zapisovat ve tvaru $A@t \to B$ a lze ho typicky interpretovat jako řešení jednoho, ze dvou problémů. Prvním problémem je nalezení spoje, který do cílové zastávky dojede co nejdříve (Earliest Arrival Problem, EAP). Druhým problémem je nalezení takového spojení, které do cílové zastávky dorazí s co nejmenším množstvím přestupů (Minimum Number of Transfers Problem). Tato se práce se primárně věnuje nalezení řešení EAP. [7, 9]

1.2 Dijkstrův algoritmus

V případě, kdy se jízdní řád MHD modeluje jako graf, je pro nalezení spojení potřeba použít nějaký grafový algoritmus, který v takovém grafu dokáže nalézt nejkratší cestu. Přestože se v nedávné době na silničních sítích podařilo aplikovat vyhledávací algoritmus A^*, který dosáhl velmi dobrých výsledků, na sítích veřejné přepravy jeho využití nepřineslo očekávané výsledky. Nečekané problémy přinesla právě potřeba využití jízdního řádu. [1, 3] Dijkstrův algoritmus (a jeho různé varianty) tak stále představuje v současnosti jediný používaný algoritmus pro nalezení nejkratší cesty v grafu modelujícím síť veřejné přepravy a tedy nejlepšího spojení. [8]

V daném kontextu může mít termín *nejkratší cesta* různý význam. Typicky váha hrany v grafu označuje vzdálenost (udávanou například v metrech) mezi vrcholy, které s ní incidují. Pokud graf modeluje síť veřejné dopravy, označuje váha hrany dobu trvání přepravy mezi vrcholy, které hrana spojuje. [8]

Při řešení EAP tak Dijkstrův algoritmus hledá cestu mezi dvěma vrcholy reprezentujícími stanice, která trvá nejkratší dobu. Nalezená cesta představuje konzistentní spojení mezi těmito stanicemi. [8]

Nejkratší cesta se může hledat i s ohledem na více kritérií. Často používanými kritérii je doba cesty a počet přestupů. Dijkstrův algoritmus používající tato dvě kritéria je ale přibližně desetkrát pomalejší oproti algoritmu využívajícímu pouze dobu cesty. [6] Navíc jsou výsledky vyhledávání využívajícího pouze jedno kritérium z pohledu cestujícího přijatelné, i když nemusí být optimální z pohledu počtu přestupů. [5] Tato práce se tak omezí pouze na použití Dijkstrova algoritmu s jedním kritériem.

Jednou z možných uprav Dijkstrova algoritmu je, že jako priorita počátečního vrcholu se použije startovní čas místo 0. [7] Příklad zápisu algoritmu v pseoudokódu pro časově závislý graf převzatý z [6] si lze prohlédnout v algoritmu 1.2.1.

Algoritmus 1.2.1 Dijkstrův algoritmus [6]

```
G = (V, E) je graf
s je startovní (stanicový) vrchol
startTime je startovni čas
function DIJKSTRA(G, s, startTime)
    Q = ∅
    for all v ∈ V do
        d(v) = ∞
        π(v) = null
        Q = Q ∪ {v}
    end for
    d(v) = startTime
    while Q ≠ ∅ do
        u = getMin(Q)
        for all e = (u, v) ∈ E do
            newDist = d(u) + getWeight(e)
            if newDist < d(v) then
                d(v) = newDist
                π(v) = u
            end if
        end for
    end while
end function
```

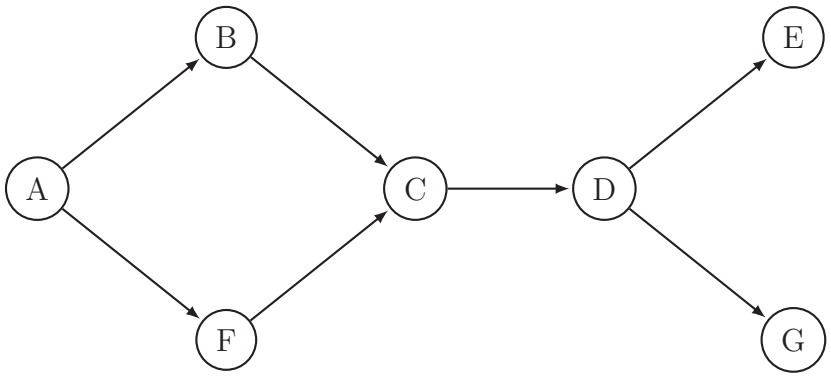
1.3 Časově závislý graf

Časově závislý graf představuje intuitivní způsob, jak modelovat síť veřejné dopravy jako orientovaný ohodnocený graf. Tento název je dán časově závislou funkcí, která slouží pro ohodnocení jeho hran. V této sekci budou popsány dvě varianty časově závislého grafu, které se liší podle tohoschopnosti modelovat dobu přestupu ve stanici. [8]

1.3.1 Bez realistické doby na přestup

Nejjednodušším grafovým modelem sítě veřejné přepravy je časově závislý graf bez realistické doby na přestup, který se pro zpřehlednění bude v rámci této práce nazývat jako *stanicový graf*. [8]

Stanicový graf se konstruuje tak, že pro každou stanici v jízdním řádu se do něj vloží jeden vrchol, který ji reprezentuje. Dva vrcholy A, B odpovídající stanicím S_A, S_B se spojí orientovanou hranou $A \to B$ právě, když existuje nějaké elementární spojení c, kde $S_d(c) = S_A$ a $S_a(c) = A_B$. Tedy přidaná hrana modeluje situaci, kdy mezi těmito dvěma stanicemi jede nějaký spoj. [7] Příklad stanicového grafu postaveného podle jízdních řádů dvou linek z tabulky 1.1 je možné si prohlédnout na obrázku 1.1 převzatém z [7].

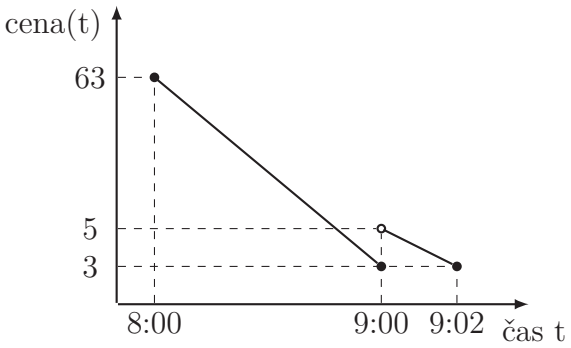


Obrázek 1.1: Stanicový graf postavený podle jízdních řádů z tabulky 1.1 [7]

Jednotlivé hrany stanicového grafu se ohodnocují váhou, která je dána časově závislou funkcí. Díky tomu jsou váhy hran vždy nezáporné. Příklad časově závislé funkce lze vidět na obrázku 1.2 převzatém z [7]. Jedná se o časově závislou funkci pro hranu $C \to D$ z graf na obrázku 1.1, která je omezená na časový interval 8:00 až 9:02. Je patrné, že váha hrany je určena nejen dobou jízdy, ale i délkou čekání ve stanici. [7, 8]

Při hledání nejkratšího spojení se musí Dijkstrův algoritmus upravit, aby uměl pracovat s časově závislou funkcí, která ohodnocuje hrany. Navíc se startovní vrchol neinicializuje nulovou vzdáleností, ale startovním časem.

Nevýhodou tohoto přístupu je, že stanicový graf neumožňuje modelovat realistickou dobu pro přestup. Tuto vlastnost lze chápat tak, že pro každou stanici S je implicitně definováno $transfer(S) = 0$, tj. minimální doba pro přestup je v ní

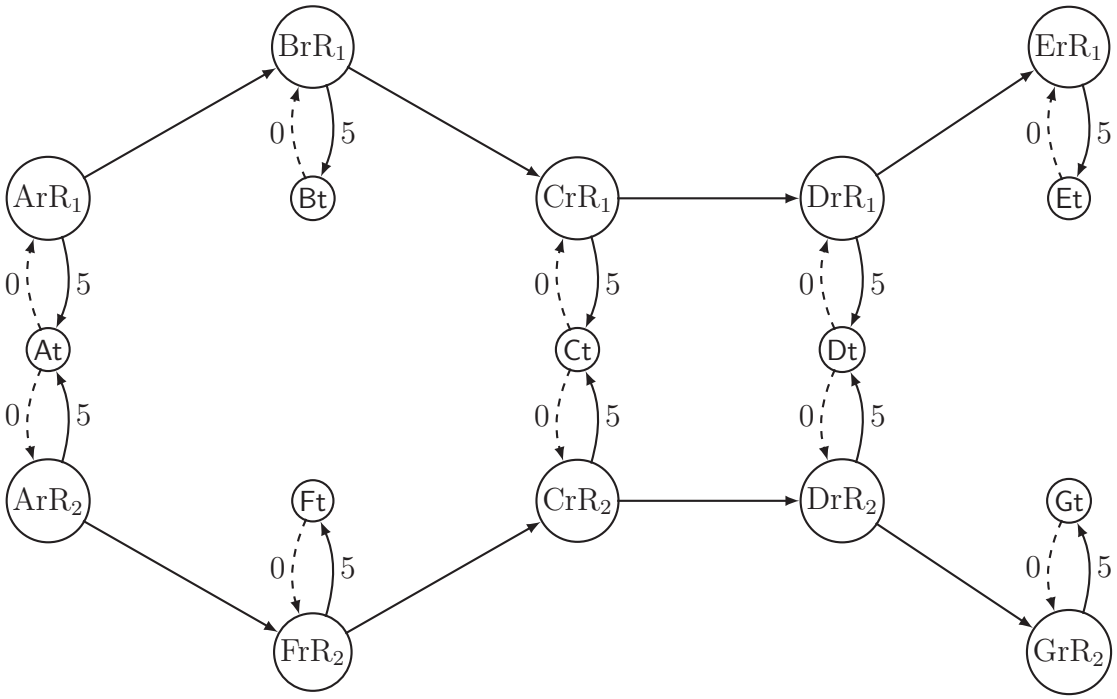


Obrázek 1.2: Náčrtek časově závislé funkce ohodnocující hranu $C \to D$ stanicového grafu z obrázku 1.1 [7]

nulová. Pokud v rámci jedné stanice existuje více nástupišť, mezi kterými cestující přechází určitou dobu, mohou být jako optimální spojení nabízena i ty, na které není kvůli vzdálenosti nástupišť reálně možné přestoupit. Navíc tak dojde k tomu, že některá skutečně optimální spojení nebudou nalezena. [7]

1.3.2 S realistickou dobou pro přestup

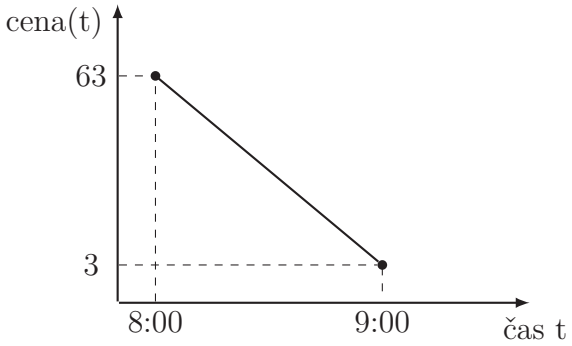
V rámci této práce se časově závislý graf s realistickou dobu pro přestup bude označovat pouze jako časově závislý graf („Time-Dependent Graph", TDG). Ve své podstatě se jedná o rozšíření stanicového grafu (respektive o úpravu jeho konstrukce) tak, aby ve stanici umožňoval modelovat nenulovou minimální dobu pro přestup.



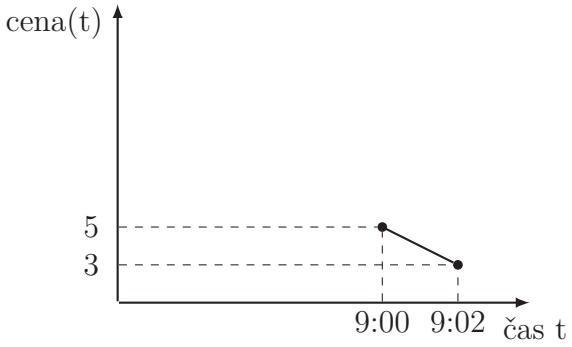
Obrázek 1.3: TDG postavený podle jízdních řádů z tabulky 1.1 [7]

TDG se konstruuje tak, že za každou stanici S, která je v jízdním řádu, se do grafu přidá jeden vrchol S_t. Dolní index t značí, že se jedná o *přestupní vrchol*. Následně se ze všech vlaků $Z \in \mathscr{Z}$ sestaví trasy[2]. Tyto trasy se postupně zpracovávají tak, že za každou stanici S na trase R_1 se do grafu přidá vrchol SrR_1, kde r označuje, že se jedná o tak zvaný *trasový vrchol*. Všechny trasové vrcholy na této trase se následně propojí orientovanou hranou ve směru jízdy, která se bude ohodnocovat časově závislou funkcí. [6]

Jako poslední se do grafu přidají *nástupní* a *výstupní hrany*, které umožňují přestup mezi spoji a tak se souhrnně označují jako *přestupní hrany*. Tyto hrany vzniknou tak, že ve stanici S se propojí hranami přestupní vrchol S_t a trasový vrchol SrR. Hrana $S_t \to SrR$ je nástupní a hrana $SrR \to S_t$ je výstupní. Nástupní hrana se ohodnotí statickou vahou 0 a výstupní hrana se ohodnotí minimální dobou nutnou pro přestup v dané stanici. [6, 8]



Obrázek 1.4: Náčrtek časově závislé funkce ohodnocující hranu $CrR_1 \to DrR_1$ TDG na obrázku 1.3 [7]



Obrázek 1.5: Náčrtek časově závislé funkce ohodnocující hranu $CrR_2 \to DrR_2$ TDG na obrázku 1.3 [8]

[2]V případě MHD je možné místo tras použít linky (vždy jeden směr), které obsluhují posloupnost zastávek ve stejném pořadí. Může tak vzniknout graf, který není optimální, protože více linek může obsluhovat jednu identickou trasu. Na správnost hledaného spojení to však nemá vliv.

Na obrázku 1.3 převzatém z [7] je TDG postavený podle jízdního řádu v tabulce 1.1. Tento graf obsahuje dvě trasy R_1 a R_2. Vrcholy A_t,..., G_t jsou přestupní vrcholy a ostatní vrcholy jsou trasové. Nástupní hrany s váhou 0 jsou čárkované a výstupní hrany jsou plné a ohodnocené hodnotou *transfer(S)*. [8]

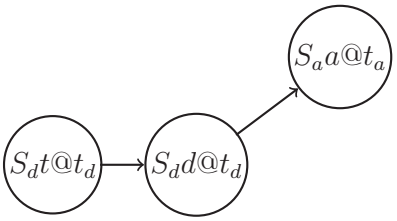
Na obrázcích 1.4 a 1.5 převzatých z [8] jsou časově závislé funkce pro dvě hrany TDG. Srovnáním s obrázkem 1.2 je zřejmé, že použitá časově závislá funkce je analogická k funkci používané ve stanicovém grafu. [8]

1.4 Časově rozšířený graf

Časově rozšířený graf („Time-Expanded Graph", TEG) je vážený orientovaný graf, který narozdíl od TDG nevyužívá pro ohodnocení hran časově závislou funkci, ale jeho hrany jsou ohodnocené staticky. [8]

Model založený na TEG zavádí koncept tzv. událostí ve stanici. Událostí se rozumí příjezd, odjezd a nebo přestup ke kterému ve stanici dojde. Každá z událostí je v grafu reprezentována jako vrchol. Název vrcholu jednoznačně určuje k čemu slouží a je ve tvaru St@9:00, kde S je název stanice, t typ vrcholu a 9:00 čas, kdy k události došlo. V grafu se vyskytují tři typy vrcholů: příjezdový (označuje se písmenem a), odjezdový (označuje se písmenem d) a přestupní (označuje se písmenem t). Hrany TEG se ohodnotí rozdílem časů událostí vrcholů, které spojují. [8]

Konstrukce TEG probíhá po jednotlivých jízdách respektive po elementárních spojeních, ze kterých se jízda skládá. Pro elementární spojení $c = (Z, S_d, S_a, t_d, t_a)$ se do grafu vloží přestupní vrchol $S_d t@t_d$, odjezdový vrchol $S_d d@t_d$ a příjezdový vrchol $S_a a@t_a$. Dále se do grafu vloží hrana $S_d t@t_d \rightarrow S_d d@t_d$ spojující přestupní vrchol s odjezdovým a následně hrana $S_d d@t_d \rightarrow S_a a@t_a$ spojující odjezdový vrchol s příjezdovým. [6] Vizualizaci konstrukce elementárního spojení c si lze prohlédnout na obrázku 1.6.



Obrázek 1.6: Elementární spojení v TEG

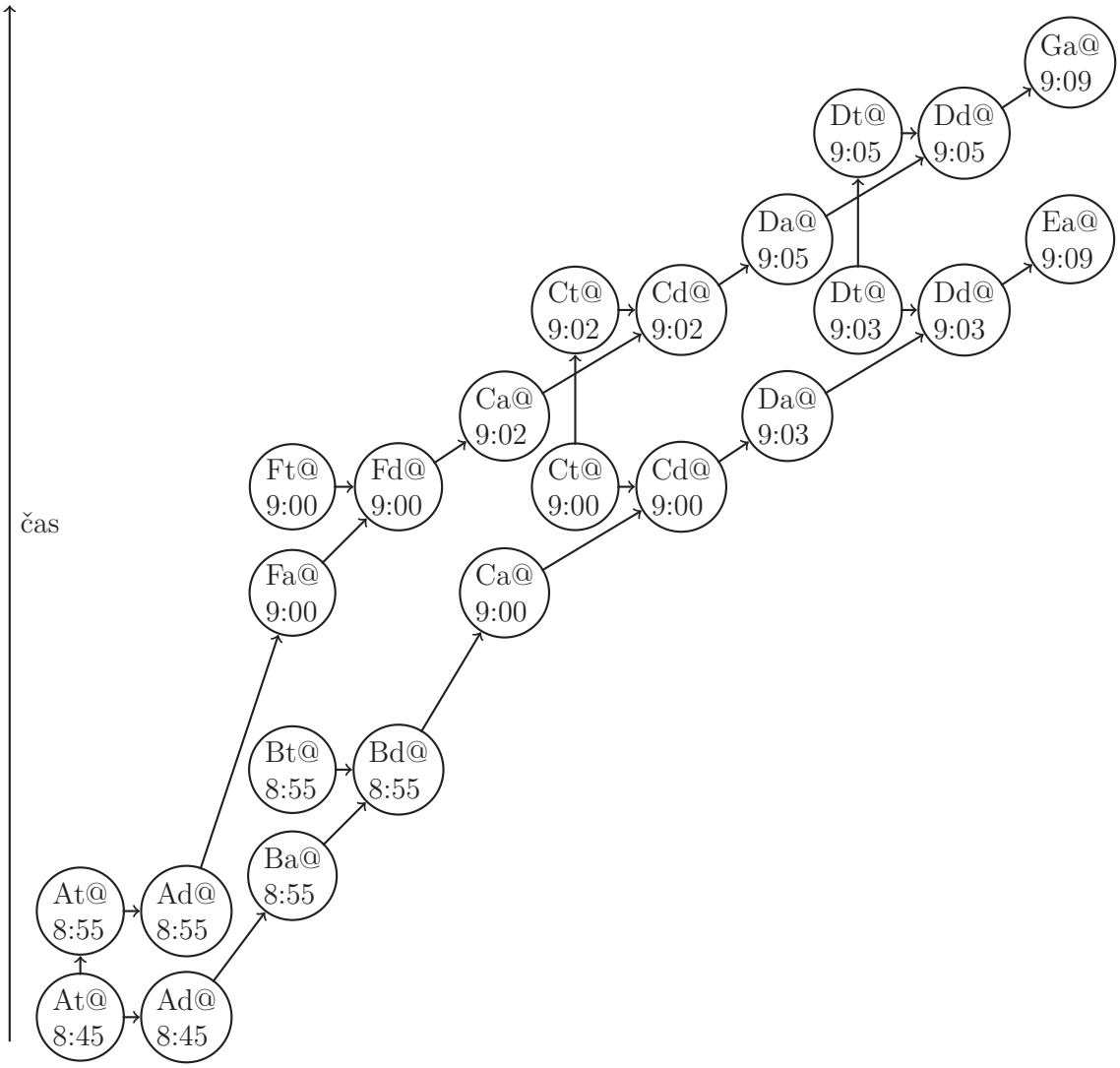
Elementární spojení, která jsou přidána v rámci jedné jízdy, je navíc třeba propojit. Pokud na elementární spojení $c_1 = (Z, S_1, S_2, t_{d_1}, t_{a_1})$ navazuje elementární spojení $c_1 = (Z, S_2, S_3, t_{d_2}, t_{a_2})$, tj. ve stanici S_2 lze pokračovat v jízdě bez nutnosti přestoupit, do grafu se přidá hrana $S_2 a@t_{a_1} \rightarrow S_2 d@t_{d_2}$. [6]

V tuto chvíli jsou namodelovány všechny jízdy, ale zatím mezi nimi není možné přestoupit. Za tímto účelem se v grafu nejprve vytvoří tzv. *čekací řetězec*. V každé stanici se seřadí přestupní vrcholy vzestupně podle času události, kterou reprezentují. Pokud mají dva vrcholy identický čas, na jejich pořadí nezáleží. Následně se

vrcholy propojí hranami ve směru růstu času tak, že z vrcholu vede hrana do bezprostředně následujícího vrcholu. Pro vrcholy $St@t_1$, $St@t_2$, $St@t_3$, kde $t_1 < t_2 < t_3$ by se přidaly hrany $St@t_1 \rightarrow St@t_2$ a $St@t_2 \rightarrow St@t_3$ [6]

Jako poslední se ve stanici S každý příjezdový vrchol $Sa@t_1$ propojí hranou $Sa@t_1 \rightarrow St@t_2$ s přestupním vrcholem $St@t_2$. Vrchol $St@t_2$ je prvním vrcholem v (seřazeném) čekacím řetězci, který splňuje podmínku $t_2 \geq t_1 + transfer(S)$. [6]

TEG sestavený podle jízdního řádu v tabulce 1.1 si lze prohlédnout na obrázku 1.7 převzatém z [8]. Minimální doba pro přestup $transfer(S)$ je ve všech stanicích 5 minut. Na první pohled je zřejmý značný nárůst počtu vrcholů i hran oproti TDG na obrázku 1.3, který je sestavený pro stejný jízdní řád.

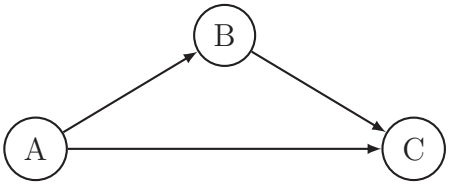


Obrázek 1.7: TEG postavený podle jízdních řádů z tabulky 1.1 [8]

1.5 Přestupní vzorce

Vyhledávání spojení pomocí přestupních vzorců (Transfer Patterns) se od již popsaných přístupů liší v tom, že mimo jízdního řádu využívají i další data, která je třeba předem vypočítat.

Základní myšlenku algoritmu lze vysvětlit na obrázku 1.8 pro dotaz $A@t \rightarrow C$. Cílem je nalézt konzistentní spojení jedoucí ze stanice A do stanice C s časem odjezdu větším nebo rovným t. Pokud nejsou o síti veřejné dopravy známy další informace, použil by se pro nalezení spojení Dijkstrův algoritmus, který by prohledával graf ve všech směrech. Pokud ale bude známa dodatečná informace, že všechny optimální cesty vedou ze stanice A do C přímým spojem nebo s jedním přestupem ve stanici B, ušetří se strojový čas. Nebude totiž potřeba vést hledání částmi grafu, kterými optimální cesta nevede. Posloupnosti vrcholů AC a ABC se nazývají *optimální přestupní vzorce* mezi stanicemi A a C. [2]



Obrázek 1.8: Myšlenka přestupních vzorců

Formálněji řečeno je přestupní vzorec tvořen startovní stanicí, posloupností přestupních stanic (v pořadí v jakém byly procházeny od startu) a cílové stanice, které se nachází na optimální cestě. [6]

Přestupní vzorce se hledají pomocí tzv. *profilovacích dotazů*. Cílem tohoto typu dotazu je nalézt všechny optimální spojení, které jedou v určitém časovém rozpětí z jedné zastávky do druhé. Právě k tomu je vhodný TEG.

Všechny jízdy v jízdním řádu se namodelují do TEG. Přestupní vzorce se počítají jako profilovací dotazy z jedné (startovní) stanice S do všech ostatních pro časový rozsah daný jízdním řádem (v případě MHD to bývá typicky týden). Za tímto účelem se použije upravený Dikstrův algoritmus, který během inicializace používá celou množinu startovních vrcholů. Jako startovní vrcholy se zvolí všechny přestupní vrcholy ve stanici S a během inicializace se jim přiřadí počáteční hodnota (vzdálenost do vrcholu) nula. Po skončení Dijkstrova algoritmu je možné zpětným trasováním získat množinu přestupních vzorců ze stanice S do libovolné stanice v grafu. [8]

Výše popsaným postupem se sice nalezne množina přestupních vzorců $\mathscr{P}$ pro spoje jedoucí ze stanice A do stanice B, ale tato množina nemusí být optimální. Množina $\mathscr{P}$ se označí jako optimální, pokud pro každý dotaz $A@t \rightarrow B$ je přestupní vzorec optimálního spojení v $\mathscr{P}$ a každý přestupní vzorec v $\mathscr{P}$ je optimální v nějakém čase t pro dotaz $A@t \rightarrow B$. Tedy $\mathscr{P}$ obsahuje právě všechny optimální přestupní vzorce pro spoje jedoucí ze stanice A do stanice B [5]

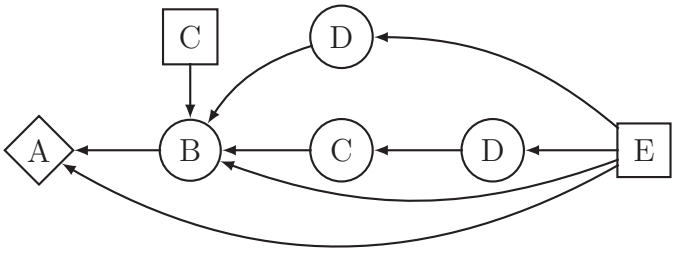
Problém spočívá v tom, že příjezdové vrcholy nejsou propojené jako přestupní vrcholy, které v rámci zastávky tvoří čekací řetězec. Cílem použitého profilovacího

Dijkstrova algoritmu je v podstatě najít nějakou nejkratší cestu do každého příjezdového vrcholu v cílové zastávce. To ale znamená, že některé cesty mohou být zbytečně dlouhé a bylo by výhodnější využít optimální spojení, které přijelo dříve a v zastávce počkat. [3]

Neoptimální přestupní vzorce není možné odstranit tak, že se do grafu přidají další hrany propojující přestupní vrcholy, podobně jako se vytvářel čekací řetězec. „Příjezdový řetězec" je však možné simulovat. V cílové stanici B se vzestupně seřadí všechny příjezdové vrcholy podle času události, kterou reprezentují. Postupně se prochází od vrcholu s nejnižším časem události a porovnává s vrcholem s bezprostředně vyšším časem události. Pokud pro dva vrcholy $Ba@t_1$ a $Ba@t_2$, kde $t_1 \leq t_2$ je $d(Ba@t_2) > d(Ba@t_1)+(t_2-t_1)$, potom se cesta do vrcholu $Ba@t_2$ nahradí cestou do $Ba@t_1$ a hodnota $Ba@t_2$ se zvýší o $t_2 - t_1$. [8]

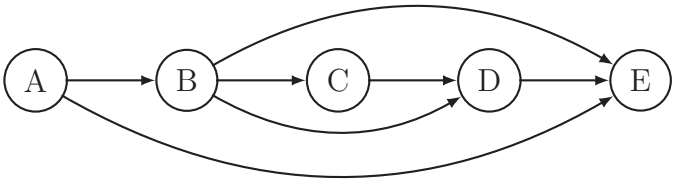
Nalezené přestupní vzorce se uchovávají jako orientovaný acyklický graf (Directed acyclic graph, DAG). Pro startovní stanici A se do DAG vloží všechny unikátní přestupní vzorce vedoucí do cílové stanice B. Stanice tvořící vzorec se do grafu vloží jako vrcholy a propojí se hranami orientovanými proti směru jízdy. Pro přestupní vzorec $AC_1 \ldots C_nB$ se do DAG vloží cesta $B \to C_n \to \ldots \to C_1 \to A$. Takový DAG se vytvoří pro každou startovní stanici A. [2]

Na obrázku 1.9 převzatém z [2] je příklad DAG vytvořeného pro stanici A ze vzorců AE, ABE, ABC, ABDE a ABCDE. V grafu se využívají tři typy vrcholů. V každém DAG je právě jeden startovní vrchol, který se označí kosočtvercem. Vrcholy označující cílové stanice jsou označeny čtvercem. Posledním typem jsou prefixové vrcholy označené kolečkem, které reprezentují přestupní stanice. [2] Na DAG tak lze nahlížet jako na prefixový strom. [6]



Obrázek 1.9: DAG pro stanici A postavený z přestupních vzorců AE, ABE, ABC, ABDE a ABCDE [2]

Dotaz $A@t \to E$ na nalezení spojení se vyhodnocuje na *dotazovacím grafu*, se musí získat z uloženého DAG pro stanici A. V DAG se vyhledá vrchol odpovídající cílové stanici E (označený čtvercem) a průchodem ke startovní stanici se naleznou všechny cesty, do startovní stanice A. Otočením orientace hran v těchto cestách se získá dotazovací graf reprezentující přestupní vzorce ze stanice A do stanice E. [2] Na obrázku 1.10 převzatém z [2] je dotazovací graf pro dotaz na spojení $A \to E$ vytvořený z z DAG na obrázku 1.9



Obrázek 1.10: Dotazovací graf pro dotaz $A@t \rightarrow E$ postavený z DAGu na obrázku 1.9 [2]

Dotazovací graf je velmi podobný stanicovému grafu s tou výhodou, že v něm není třeba modelovat přestup. Každá cesta ze startovní do cílové stanice totiž představuje jeden přestupní vzorec, který v sobě uchovává informaci o přestupu ve svých vrcholech.

Hrany dotazovacího grafu jsou ohodnoceny časově závislou funkcí, která využívá toho, že mezi dvěma vrcholy (stanicemi) jede vždy přímé spojení a na hraně není možné přestoupit. Aby bylo ohodnocení hrany rychlé, je třeba si vytvořit speciální strukturu, která umí ohodnotit přímá spojení. Jízdy se sdruží podle svých tras do linek, tj. všechny jízdy na takto vzniklé lince obsluhují stejnou posloupnost stanic[3]. Pro každou stanici se vytvoří seznam uspořádaných dvojic (L, O), kde L označuje linku, která v ní zastavuje a O je pořadí stanice na trase linky. [5, 6]

Při ohodnocování hrany $A \rightarrow B$ pro dotaz na přímé spojení $A \rightarrow B$ se pro stanice A a B udělá mezi seznamy uspořádaných dvojic průnik podle linek. Ze získané množiny se vyřadí ty uspořádané dvojice, které mají vyšší pořadí na trase ve stanici A než ve stanici B. Z linek v množině se vybere ta s nejnižším časem odjezdu z A, který je v čase t nebo později. Váha hrany se určí jako rozdíl doby příjezdu cílové stanice a času t. [5, 6]

1.5.1 Optimalizace a heuristiky

Přestupní vzorce jsou velmi náročné na výpočet, a tak je vhodné zavést optimalizace a také heuristiky, které náročnost snižují.

Jedna z optimalizací spočívá v použití kompaktnějšího grafového modelu než je navrhovaný TEG. Přestupní vrcholy je možné z grafu odstranit a jejich účel mohou zastat odjezdové vrcholy, tj. přesměrují se do nich hrany vedoucí z příjezdových do přestupních vrcholů a vytvoří se z nich čekací řetězec umožňující přestupy ve stanici. Dále je možné využít toho, že má veřejná doprava dán denní jízdní řád na týden a v některé dny se tento jízdní řád opakuje. Graf tak lze „složit" modulo 24 hodin a každému vrcholu přidat masku, která určuje, jestli spoj v daný den jede. [5, 6]

Další optimalizací je použití *významných stanic* (HUB). V grafu se vybere 1% stanic (například těch, kde se protíná nejvíce tras) a označí se za HUB. Při hledání přestupních vzorců Dijkstrovým algoritmem se z HUB provádí tzv. globální vyhledávání, tj. vyhledávání do všech stanice, které již bylo popsáno v předchozí sekci. [3, 6]

[3]Pokud by některý spoj tzv. zatahoval, musí se vyčlenit do samostatné linky

Z ostatních stanic se provádí už jen tzv. lokální vyhledávání, které spočívá v tom, že Dijkstrův algoritmus ukončí prohledávání dané větve výpočtu, jakmile narazí na nějakou významnou stanici. Tento HUB se označí jako *přístupová stanice* pro danou startovní stanici. [3, 6]

Dotazový graf pro dotaz $A@t \rightarrow B$ se pak skládá z přestupních vzorců mezi stanicemi $\{(A, B)\} \cup (\{A\} \times \mathscr{X}) \cup (\mathscr{X} \times \{B\})$, kde $\mathscr{X}$ je množina přístupových stanic pro stanici A. [2].

Důležitým zrychlením je heuristika tří přestupů (3-legs heuristic), která se společně s významnými stanicemi využívá při řešení tzv. problému 15ti hodin do vedlejší vesnice. V oblastech s horší dopravní obslužností totiž i lokální vyhledávání prohledá značnou část grafu a trvá dlouho než narazí na HUB. Heuristika tří přestupů při hledání přestupních vzorců ze startovní stanice umožňuje na každé cestě použít nejvýše dva přestupy. Pokud by se na nějaké cestě musely použít více než tři vlaky, je tato větev výpočtu ukončena a už se v ní nepokračuje. [3] V praxi tak dojde ke ztrátě pouze malého množství optimálních přestupních vzorců, ale omezí velikost prohledávané části grafu, což je důležité u velmi velkých grafů (např. spoje v rámci kontinentu). [2, 3]

1.6 Connection Scan Algorithm

Connection Scan Algorithm (CSA) je jedním z nových přístupů k vyhledávání spojení ve veřejné dopravě. Jedná se o tzv. negrafový algoritmus, který narozdíl od Dijkstrova algoritmu nepracuje nad grafem. Dochází tak k úspoře potřebné paměti, protože grafy, jako například TEG, bývají značně rozsáhlé. Další výhodou je, že není potřeba využívat haldu nebo prioritní frontu, nad kterou se provádí mnoho operací a snižuje se tak výkon Dijkstrova algoritmu. [8, 10]

Pro elementární spojení c se jako $trip(c)$ označí jízda, do které toto spojení patří. Jako c_{next} se označí elementární spojení, které bezprostředně následuje za c v rámci jízdy $trip(c)$. Elementární spojení c_{next} je dosažitelné, pokud cestovatel jede předcházejícím elementárním spojení c stejné jízdy nebo se již nachází ve stanici S, ze které elementární spojení odjíždí, v čase jeho odjezdu. Formálněji musí být splněny podmínky $trip(c) = trip(c_{next})$ a nebo $S(t) \leq c_{next}(t_d)$, kde $S(t)$ je čas příchodu respektive příjezdu do stanice S. [10]

Základní myšlenka algoritmu vychází z použití Dijkstrova algoritmu na TEG, kde je výsledkem optimální cesta mezi dvěma vrcholy (stanicemi), která se skládá z posloupnosti dosažitelných elementárních spojení. Pokud se TEG postaví nad aperiodickým jízdním řádem, potom díky tomu, že každý vrchol v něm odpovídá nějaké události a hrany vedou pouze z vrcholu s nižším časem do vrcholu s vyšším časem události, je tento graf acyklický. Jeho hrany tak lze topologicky uspořádat podle času události ve vrcholu, ze kterého vychází. Právě v tom pořadí by hrany zpracovával i Dijkstrův algoritmus. [10]

Všechna elementární spojení z jízdního řádu se vloží do jednoho pole (Connection Array), které se vzestupně setřídí podle času odjezdu (analogicky, jako by se topologicky třídili hrany TEG). Pro každou stanici S se udržuje záznam s dobou příjezdu $S(t)$ elementárního spojení do stanice a elementární spojení $S(c)$, kterým se do stanice přijelo. [10]

Před vyhodnocením dotazu na spojení $A@t \rightarrow B$ se provede inicializace. Pro všechny záznamy stanic S se nastaví $S(t) = \infty$ a $S(c) = null$. Pro počáteční stanici se navíc položí $A(t) = t$. [10]

Samotné vyhledávání optimálního spojení probíhá tak, že algoritmus postupně prochází (skenuje) pole elementárních spojení od nejlevějšího, tj. toho s nejmenším časem odjezdu. Algoritmus pro každé elementární spojení $c_i = (Z, S_d, S_a, t_d, t_a)$ testuje, jestli je toto spojení dosažitelné. Pokud navíc toto spojení zlepší cestu do stanice $S = S_a(c_i)$, tj. $t_a(c_i) < S(t)$, potom bude spojení c_i relaxováno (analogicky jako v Dijkstrově algoritmu). Záznam pro stanici S se upraví jako $S(t) = t_a(c_i)$ a $S(c) = c_i$. [8, 10]

Po zpracování celého pole elementárních spojení obsahuje záznam pro cílovou stanici B dobu nejkratší cesty $B(t)$ do ní. Zpětným průchodem podle spojení se nalezne seznam optimálních elementárních spojení. [8, 10]

Pro zrychlení běhu algoritmu je vhodné použít následující optimalizace. Pokud je doba odjezdu aktuálně zpracovávaného spojení vyšší než doba příjezdu již nalezeného spojení do cílové stanice, skenování se ukončí. Všechna následující elementární spojení už zřejmě na nejkratší cestě nebudou. Analogicky není třeba vyhodnocovat elementární spojení před zadaným časem odjezdu. Záznam, od kterého začne algoritmus skenovat spojení, sice lze dohledat sekvenčně, ale mnohem vhodnější je hledání půlením. [8, 10]

1.6.1 Rozšíření o realistickou dobu pro přestup

Výše popsaný algoritmus nabízí rychlé nalezení spojení, ale podobně jako stanicový graf neumožňuje modelovat realistickou dobu pro přestup. Pouhým upravením podmínky dosažitelnosti elementárního spojení c ve stanici S na $S(t) + transfer(S) \leq c(t_d)$ nemusí být vždy nalezeno optimální spojení. Jako modelový příklad poslouží jízdní řád v tabulce 1.2 s minimální dobou nutnou na přestup 2 minuty.

Tabulka 1.2: Jízdní řád, podle kterého je nalezeno suboptimální spojení

(a) linka A

dojezd	stanice	čas
*	A	10:00
3	B	
5	C	
8	D	

(b) linka B

dojezd	stanice	čas
*	A	10:01
4	C	

(c) linka C

dojezd	stanice	čas
*	C	10:10
5	D	

Při letmém pohledu na jízdní řád v tabulce 1.2 je zřejmé, že optimální spojení pro dotaz A@10:00 $\rightarrow D$ je:

A@10:00 — Linka A $\rightarrow D$@10:08

Místo tohoto spojení, ale algoritmus nalezne suboptimální spojení, které se skládá ze dvou spojů:

A@10:01 — Linka B $\rightarrow C$@10:05
C@10:10 — Linka C $\rightarrow D$@10:15

Tabulka 1.3: Pole elementárních spojení

Spojení	c_1	c_2	c_3	c_4	c_5
Linka	Linka A	Linka B	Linka A	Linka A	Linka C
Odjezd	A@10:00	A@10:01	B@10:03	C@10:05	C@10:10
Příjezd	B@10:03	C@10:05	C@10:05	D@10:08	D@10:15
Použito	✓	✓	×	×	✓

Tabulka 1.4: Záznamy pro stanice po jednotlivých krocích výpočtu

(a) Inicializace

stanice	čas	spojení
A	10:00	null
B	∞	null
C	∞	null
D	∞	null

(b) Po zpracování c_1

stanice	čas	spojení
A	10:00	null
B	10:03	c_1
C	∞	null
D	∞	null

(c) Po zpracování c_2

stanice	čas	spojení
A	10:00	null
B	10:03	c_1
C	10:05	c_2
D	∞	null

(d) Po zpracování c_2

stanice	čas	spojení
A	10:00	null
B	10:03	c_1
C	10:05	c_2
D	∞	null

(e) Po zpracování c_2

stanice	čas	spojení
A	10:00	null
B	10:03	c_1
C	10:05	c_2
D	∞	null

(f) Po zpracování c_2

stanice	čas	spojení
A	10:00	null
B	10:03	c_1
C	10:05	c_2
D	10:15	c_5

Příčina neoptimálního výsledku spočívá v procházení pole elementárních spojení a použití jednoho záznamu pro každou stanici. V tabulce 1.3 je sestaveno pole elementárních spojení pro jízdní řád 1.2. V této sekci bude pro zjednodušení popisu

záměnný identifikátor linky a jízdy, protože každá linka má právě jednu jízdu, a tedy nemá smysl je odlišovat. V tabulce 1.4 jsou pak shrnuty záznamy pro jednotlivé stanice po každé zpracovaném elementárním spojení.

Je zřejmé, že nedojde k použití spojení c_4 přestože leží na optimální cestě. Důvodem k tomu je, že záznam pro stanici C už relaxovalo spojení c_2, které má sice stejný čas dojezdu jako spojení c_3, ale bylo zpracováno dříve. Navíc není možné přestoupit, protože minimální doba nutná pro přestup je 2 minuty.

Jako řešení se nabízí uchovávání informace o tom, jestli už některé předchozí spojení dané jízdy bylo použito na cestě, tj. bylo dosažitelné, a tedy jsou dosažitelná i následující spojení. Pro každé elementární spojení se použije příznak *isReachable*, který se během inicializace nastaví na *false*. Pokud je během výpočtu nalezeno dosažitelné spojení c, nastaví se příznak $isRechable$ následujícího elementárního spojení c_{next} na *true*. Během výpočtu je pak elementární spojení c_{next} automaticky považováno za dosažitelné. [10]

1.7 Round-Based Public Transit Optimized Router

Dalším negrafovým algoritmem, který stejně jako CSA nepotřebuje pro svoji funkci graf ani prioritní frontu, je Round-Based Public Transit Optimized Router (RAPTOR). Oproti ostatním algoritmům je už i jeho základní verze navržena pro hledání Paretovsky optimálního spojení z pohledu minimalizace doby příjezdu (Earliest Arrival, EA) do cílové stanice a počtu přestupů (Minimum Number of Transfers, MNT), tj. optimalita spojení je hodnocena podle dvou kritérií. [9, 12]

V souvislosti s Paretovsky optimálním spojením je třeba si zavést relaci dominance $\preceq$. Uspořádaná dvojice (d_1, p_1) dominuje uspořádané dvojici (d_2, p_2), pokud $d_1 \leq d_2$, $p_1 \leq p_2$ a jedna z těchto nerovností je ostrá. Množina uspořádaných dvojic, které se vzájemně nedominují, se nazývá Paretova množina. Pokud se tedy spojení vyhodnocuje podle dvou kritérií (EA a MNT) a $J_1 \preceq J_2$, potom spojení J_1 není v žádném kritériu horší než J_2. [5, 12]

Narozdíl od CSA, RAPTOR pracuje hlavně s trasami a jízdami, které je obsluhují. Princip spočívá v tom, že se algoritmus prohledáváním tras snaží postupně najít spojení ze startovní do cílové stanice pomocí $1, 2, \ldots, k$ jízd (spojů).

Každá stanice S navštívená v rámci jízdy T tak má přiřazený čas příjezdu $t_{arr}(T, S)$ a odjezdu $t_{dep}(T, S)$, kde $t_{arr}(T, S) \leq t_{dep}(T, S)$. [12]

Tento algoritmus pracuje v tzv. rundách (kolech), přičemž v i-té rundě nalezne optimální spojení s maximálně $i-1$ přestupy. V každé stanici S se uchovává záznam $t_0(p), t_1(p), \ldots, t_K(p)$, kde $t_i(p)$ je čas nejdřívějšího (známého) příjezdu do stanice s maximálně i použitými jízdami (spoji) a K je dané horní omezení počtu rund. Cílem k-té rundy je nalézt $t_k(p)$ pro všechny stanice S. Jednotlivé rundy algoritmu se skládají ze tří fází. [12]

V první fázi k-té rundy se pro všechny stanice S nastaví doba příjezdu s nejvýše k spoji $t_k(p)$ na $t_k(p) = t_{k-1}(p)$. Tato hodnota slouží jako horní hranice pro nejdřívější příjezd do stanice S pomocí nejvýše k jízd, tj. na cestě dojde nejvýše ke $k-1$ přestupům. V nově nalezeném spojení je vždy více přestupů než v tom z mi-

nulé rundy. Pokud nově nalezené spojení trvá déle, potom je dominováno spojením z minulé rundy, a tedy se nepoužije. [12]

V druhé fázi se právě jednou zpracuje každá trasa vytvořená z jízdního řádu. Pro trasu r se označí $\mathcal{T}(r) = (T_0, T_1, \ldots, T_{|\mathcal{T}(r)|-1})$ posloupnost jízd na trase r setříděná vzestupně podle času odjezdu. Při zpracování trasy r se uvažují ty cesty (spojení), jejichž poslední, tj. k-tá, jízda je na trase r. Nechť $ET(r, S_i)$ je nejdříve jedoucí jízda na trase r, na kterou se dá ve stanici S_i nastoupit[4], a tedy $t_{dep}(T, S_i) \geq t_{k-1}(S_i)$. [12]

Zpracování trasy probíhá tak, že se v daném pořadí ve směru jízdy postupně navštěvují její stanice dokud se nenalezne stanice S_i, pro kterou je $ET(r, S_i)$ definováno. V takové stanici S_i je možné na trasu „nastoupit" a daná jízda T se označí za *aktuální jízdu* pro k. Trasa se dále prohledává a pro každou následující stanici S_j se aktualizuje čas příjezdu do ní $t_k(S_j)$ za pomocí této jízdy. Pro rekonstrukci cesty (spoje) se nastaví ukazatel na zastávku, na které se na jízdu T nastoupilo.

Dále je třeba pro aktuální jízdu projít stanice S_i na trase r, ve kterých může být možné nastoupit na dřívější jízdu, protože cesta nalezená v předchozí rundě mohla být rychlejší cesta. Pro každou stanice S_i je tak třeba ještě ověřit, jestli platí $t_{k-1}(S_i) < t_{arr}(T, S_i)$ a případně aktualizovat T pomocí $ET(r, S_i)$. [12]

Třetí a poslední fáze rundy slouží pro pěší přechody mezi zastávkami, tj. uvažují konstantní vzdálenost mezi stanicemi, kterou je třeba překonat chůzí. Pěší přechody však nejsou součástí této práce, a tedy nebudou popisovány. [12]

Zpětným průchodem je možné pro každé k najít spojení, které není dominováno, ze startovní stanice S_s do cílové stanice S_t s nejmenším možným časem příjezdu skládající se z nejvýše k jízd (spojů). [12]

Výhodou RAPTORu je, že jednotlivé cesty je možné zpracovávat víceméně nezávisle, a tak je možné tuto činnost paralelizovat. Pokud se dvě trasy protínají, použije se při počínání toho vrcholu běžné synchronizační primitivum nebo lze použít přístup jako v precedenčním (konfliktním) grafu, který se předem vypočítá. [12]

1.8 Vyhodnocení

Ze zmíněných grafových modelů se pro použití v prostředí s omezeným výkonem i pamětí hodí pouze model založený na TDG, který byl použit i v [7]. Stanicový graf je sice velice kompaktní, ale neumožňuje modelovat reálné přestupy. TEG je naproti tomu detailní grafová reprezentace jízdního řádu, ale je velmi paměťově náročná.

Přestupní vzorce se od již zmíněných grafových modelů liší tím, že pro vyhledávání spojení potřebují postavit dotazový graf, který je třeba předem vypočítat. Tento přístup tak vyžaduje uložení nejen jízdních řádů, ale i dalších dat, a proto je vhodný spíše pro serverové řešení.

Negrafové algoritmy přináší nový potenciál využitelný v mobilních zařízeních, protože pro svou funkci nepotřebují graf ani haldu (jako Dijkstrův algoritmus). Zejména CSA nabízí jednoduchou implementaci, nízkou paměťovou i výpočetní náročnost, a proto se jeví být ideální pro použití v aplikacích pro zařízení s omezeným výkonem.

[4] Pokud taková jízda $ET(r, S_i)$ existuje.

2 Heuristika

Jednou z možností jak zrychlit nějaký výpočet je jeho paralelizace, kterou ale není vždy možné realizovat. V této kapitole bude nevržena heuristika pro odhad optimálních časů odjezdu ze stanice. Takový odhad je výhodný v případě, kdy je uživateli nabízeno několik po sobě jedoucích spojení. Výpočet se může paralelizovat tak, že se každé spojení vyhledává nezávisle na ostatních. Díky tomu je navíc možné použít libovolný z algoritmů popsaný v kapitole 1.

2.1 Paralelizace výpočtu

Při hledání optimálního spojení ve veřejné dopravě je možné vyhodnocovat dva druhy dotazů. První možností je hledat nejrychlejší spojení (time-query), tj. pro dotaz $A@t \rightarrow B$ je cílem nalezení konzistentního spojení ze stanice A do stanice B, které odjíždí v čase t nebo později, tedy je dosažitelné. Druhou možností jsou profilovací dotazy, kdy se hledají všechna spojení v nějakém intervalu (profile-query). [11]

Paralelní výpočet optimálního spojení ve veřejné dopravě je z kombinatorického pohledu velmi náročný problém. Pro grafové modely založené na časově závislé funkci dokonce ani není známý algoritmus, který by dokázal pro výpočet jednoho optimálního spojení využít paralelismu jinde než při relaxaci hran a práci s prioritní frontou a přitom nevykonal podstatně více práce než sekvenční implementace Dijkstrova algorimu. [11]

Při vyhledávání spojení v grafu silniční dopravy je možné paralelizaci provést přímočaře tak, že se spustí dvě instance Dijkstrova algoritmu proti sobě, tj. jedna bude vyhledávat ze startu do cíle a druhá v opačném směru. Síť veřejné dopravy se ale značně liší od sítě silniční dopravy, protože využívá jízdní řád. V časově závislém grafu jsou hrany ohodnoceny dynamicky, a tak není možné odhadnou váhy hran, které by měly být použity druhou instancí v již zmiňovaném přístupu se dvěma instancemi Dijkstrova algoritmu. [1, 3]

Další možností je paralelizovat vyhledávání spojení na TEG. Tento grafový model má staticky ohodnocené hrany a přístup, kdy se využijí dvě instance Dijkstrova algoritmu je možné použít. Bohužel TEG je typicky velmi rozsáhlý, a tak se hodí spíše pro profilovací dotazy a pro vyhledávání jediného spojení nemá jeho použití smysl.

Na grafových modelech veřejné dopravy tak lze využít paralelismu pouze k výpočtu profilovacích dotazů na TEG nebo TDG. [3, 11]

V případě negrafových algoritmů je situace optimističtější, protože alespoň algoritmus RAPTOR, popsaný v kapitole 1, umožňuje paralelní vyhledávání jediného spojení.

2.2 Návrh heuristiky

Aplikace pro vyhledávání optimálního spojení pro nějaký dotaz $A@t \to B$ obvykle nenabízí pouze jedno ale několik spojení ze stanice A do stanice B. Všechna tato spojení odjíždí v čase t nebo později a typicky jsou spojení setříděná vzestupně podle času odjezdu. Pokud by tedy pro zadaný dotaz bylo aplikací nabídnuto n spojení s časy $t_1, t_2, t_3, \ldots, t_n$, potom $t_1 < t_2 < t_3 < \ldots < t_n$.

Jednotlivé po sobě jedoucí spojení se musí dohledávat sekvenčně. Algoritmus nejprve pro dotaz $A@t \to B$ nalezne spojení s časem odjezdu t_1 a spustí nové hledání pro dotaz $A@t' \to B$, kde $t' = t_1 + t_{min}$ a t_{min} je nejmenší jednotka času, se kterou daná aplikace (respektive její jízdní řád) pracuje. V českém prostředí se běžně používá $t_{min} = 1$ minuta. Analogicky se vyhledávání opakuje, dokud není nalezen požadovaný počet spojení.

Pokud by bylo možné odhadnout časy odjezdů $t_1, t_2, t_3, \ldots, t_n$, potom by se dal tento sekvenční výpočet paralelizovat a každé spojení vyhledávat nezávisle na ostatních. Pro vyhledávání spojení by se tak používala samostatná instance vyhledávacího algoritmu a tedy je možné použít i libovolný z algoritmů popsaný v kapitole 1.

Navrhovaná heuristika pro odhad časů odjezdu vychází z myšlenky přestupních vzorců, které pomocí předchozích výpočtů naleznou nad TEG všechny optimální cesty ze startovní do cílové zastávky a uloží si z nich postavený DAG. Vyhodnocování dotazu už probíhá pouze na dotazovém grafu vytvořeném z tohoto DAGu.

Heuristika stejně jako přestupní vzorce potřebuje pro svoji funkci provést předchozí výpočty. Během nich se profilovacími dotazy na TEG najdou všechna optimální spojení ze startovní stanice A do všech ostatních stanic. Pro každou cílovou stanici B se uloží identifikátory linek l, které jsou první na nějaké optimální cestě z A do B, tj. l je první spoj daného spojení.

Pro dotaz $A@t \to B$ se odhaduje čas odjezdu tak, že se načtou uložené časy odjezdu linek l pro cesty z A do B. Z těchto časů se vybere n nejnižších a unikátních časů t_i, že $t_i \geq t$. Časy t_i jsou odhadem odjezdu optimálních spojení ze stanice A do B.

Tabulka 2.1: Jízdní řád pro demonstraci heuristiky

(a) linka A

dojezd	stanice	čas
*	A	10:00
2	B	10:05
4	C	10:10
8	D	10:15

(b) linka B

dojezd	stanice	čas
*	A	10:10
4	E	10:20
8	D	10:30

Odhad časů odjezdu pomocí heuristiky lze demonstrovat na jízdním řádu z tabulky 2.1. Optimální spojení ze stanice A do D představují právě obě linky. Pro dotaz $A@10:05 \rightarrow D$ pro který by se měly odhadnout čtyři optimální časy odjezdu heuristika vrátí: 10:05, 10:10, 10:15 a 10:20.

2.3 Přestupní vzorce

Výše popsané hledání optimálních linek pro heuristiku pomocí profilovacích dotazů, které prochází celý graf, je velmi náročné. Jako vhodné řešení se jeví využít výpočet přestupních vzorců a během fáze, kdy se z grafu extrahují optimální cesty, si navíc ukládat i optimální linky pro heuristiku. Optimalizace a heuristiky totiž algoritmus zrychlují oproti přímočarému počítání profilovacích dotazů. Další výhodou tohoto přístupu by byla možnost používat heuristiku a společně s přestupními vzorci.

2.3.1 Problém významných stanic

Stěžejní optimalizací zrychlující výpočet přestupních vzorců představují významné stanice, tj. HUBy. Profilovací dotazy se rozdělí na globální, které se provádí z HUBů, a lokální, jež se provádí z ostatních stanic. Pokud během lokálního vyhledávání narazí algoritmus na HUB, daná větev výpočtu se ukončí. Aby nedošlo ke ztrátě některých optimálních spojení jedoucích přes HUB, při stavění dotazového grafu se tato větev „doplní" připojením přestupních vzorců získaných z odpovídajícího globálního profilovacího dotazu. Do dotazového grafu se tímto způsobem může vložit neoptimální cesta, která ale díky Dijkstrovu algoritmu hledajícímu optimální spojení nemůže ovlivnit výsledek hledání.

Přímočaré použití této optimalizace může způsobit „zarušení" informací o optimální lince a heuristika pro odhad časů odjezdu se potenciálně stane nepoužitelnou. Do seznamu optimálních linek by se musela přidat každá linka jedoucí do HUBu a to bez ohledu na to, jestli přes HUB vůbec nějaké optimální spojení jede.

Problém si lze dobře ukázat na jednoduchém jízdním řádu zapsaném v tabulce 2.2. Ze stanice A a do C jezdí přímé spojení, které je optimální. Sousedící stanice B je HUB a jezdí do ní spoje z A. Naopak z HUBu B do stanice C nejezdí žádné spojení.

Tabulka 2.2: Jízdní řád pro demonstraci problému s použitím optimalizace významných stanic

(a) linka A

dojezd	stanice	čas
*	A	8:00
5	C	8:30
		9:00
		9:30

(b) linka B

dojezd	stanice	čas
*	A	8:05
10	B	8:10
		8:15
		8:20

Lokální prohledávání z A se ukončí v B a jako optimální linky pro cestu ze stanice A do C se označí *linka A* i *linka B*. Pokud by měla heuristika odhadnout časy odjezdu pro dotaz $A@8{:}00 \rightarrow C$, potom by vrátila: 8:00, 8:05, 8:10 a 8:15. Optimálním časem odjezdu je ale jen první zmíněný.

Optimalizace v podobě významných stanic se používá s heuristikou tří přestupů a díky jejich kombinaci lze výpočet přestupních vzorců zrychlit. Pokud není implementováno ani jedno ze zmíněných vylepšení, heuristika pro odhad časů odjezdu se stává náchylnou na problém 15-ti hodin do příští vesnice.

2.3.2 Guidebook Routing

Trasa z průvodce (Guidebook route, GBR) je posloupnost spojů pro cestu mezi dvěma stanicemi, která není závislá na čase. Například pro cestu ze stanice A do C by to mohl být popis v následujícím tvaru: Ve stanici A nastup na *linku 15* nebo *linku 19*, podle toho, která přijede dřív, a pokračuj do stanice B. Tam přestup na *linku 3* a pokračuj až do stanice C. [4]

Princip směrování podle průvodce (Guidebook Routing) je tak velmi podobný přestupním vzorcům. Rozdíl spočívá v tom, že zatímco přestupní vzorce pokrývají všechny trasy, které jsou optimální v nějakou dobu, směrování podle průvodce pokrývá jenom jejich podmnožinu. Zjednodušeně řečeno trasy, které jsou optimální jen výjimečně, nejsou označeny jako GBR. [4]

Cílem tohoto přístupu je pokrýt všechna optimální spojení jedoucí v nějakém časovém úseku, například jednom dnu, pomocí malého množství GBR. Pokrytím se intuitivně rozumí situace, že optimální spojení jede podle popisu daném v GBR. Počet spojení, které GBR pokrývá, se označí jako jeho *frekvence*. Dále se, analogicky k přestupnímu vzorci, zavede pojem *vzorec spojení*, který se skládá ze startovní stanice, posloupnosti stanic, ve kterých dojde k přestupu a cílové stanice. [4]

Algoritmus pro nalezení GBR využívá toky v síti. Pro dvojici stanic A a B, pro které se hledají GBR z A do B, se zkonstruuje orientovaný kondenzovaný graf z paretovsky optimálních vzorců spojení[1]. Nejprve se do grafu vloží všechny stanice ve vzorcích z A do B. Za každý vzorec $s_1, s_2, \ldots, s_k$ se pro $i = 1, 2, \ldots, k$ do grafu vloží orientovaná hrana (s_i, s_{i+1}). Hrany spojující stanici s_i se stanicí s_{i+1} se zkondenzují do jediné orientované hrany, které se přiřadí kapacita rovná počtu těchto hran. [4]

Na kondenzovaném grafu se nalezne cesta s nejvyšším tokem, která představuje GBR s nejvyšší frekvencí. Pro hledání další GBR se jednoduše od kapacit hran odečte velikost toku na cestě s nejvyšším tokem. Tímto způsobem lze opakovaně nalézt množinu GBR s požadovaným pokrytím. [4]

Jak již bylo naznačeno, přestupní vzorce mohou posloužit jako základ pro výpočet GBR. Jedinou informací, kterou je třeba si navíc zapamatovat, je násobnost přímých spojení. [4]

[1] V původním článku se používají vzorce spočtené pomocí dvou kritérií - délky jízdy a počtu přestupů. Podobně jako v případě algoritmu přestupních vzorců lze ale předpokládat, že daný přístup bude použitelný i pro jedno kritérium.

Výhodou směrování podle průvodce je, že snižuje počet linek, které jsou označeny jako optimální, a jako GBR jsou označeny pouze ty nejčastěji používané trasy. Odstranění linek, které jsou optimální jen výjimečně, by mohlo mít pozitivní vliv na navrhovanou heuristiku. Navíc lze Guidebook Routing implementovat nad přestupními vzorci nebo třeba RAPTORem. Použitelnost GBR v heuristice pro odhad časů odjezdu je však třeba potvrdit experimentem. [4]

2.4 Další možnosti použití

Navrhovaná heuristika může nabídnout i další možnosti použití, protože v sobě skrývá informaci o topologii sítě veřejné dopravy. Pokud pro nějaký dotaz $A@t \to B$ nevrátí žádný čas odjezdu, potom stanice B není dostupná žádnou cestou vedoucí z A. Není tak potřeba provádět další zdlouhavé prohledávání a tím zrychlí vrácení odpovědi.

V případě silniční přepravy se pro nalezení optimální cesty používá algoritmus A^*, který při prohledávání využívá heuristiku odhadující vzdálenost do cíle a tím se omezí prohledávání neperspektivních větví výpočtu. Podobně je možné pomocí heuristiky pro odhad časů odjezdu rozšířit Dijkstrův algoritmus tak, aby v nějaké stanici S pokračoval v prohledávání pouze vrcholů (a hran) vytvořených pro linku, která je heuristikou označena jako optimální.

3 Analýza aplikace a návrh zlepšení

3.1 Současný stav

Aplikace MHDApp pro offline vyhledávání spojů na platformě Android publikovaná v bakalářské práci [7] využívá grafový model založený na časově závislém grafu s realistickými dobami pro přestup (viz sekce 1.3.2). Tento model se jeví jako vhodný vzhledem ke svému použití v mobilní aplikaci. Jeho primární výhodou je nízká náročnost na množství distribuovaných dat jízdních řádů, která jsou potřebná pro vyhledávání spojení. Díky tomu je tak možné tato data přenášet i prostřednictví mobilního internetu. Stejně tak se jedná o model, který je poměrně nenáročný na množství operační paměti. Nejedná se sice o nejrychlejší dostupný algoritmus, avšak jeho flexibilita (možnost ho jednoduše rozšířit o podporu pro pěší přechody mezi stanicemi, rozšíření na dotazy lokace-lokace) a výše popsané vlastnosti z něj pro tento případ činí vhodným modelem.

Aplikace MHDApp je celá vytvořená v jazyku Java a tu využívá i pro uložená data. Jako databáze pro uchování a distribuci dat jízdních řádů pro aplikaci byl zvolen serializovaný javovský objekt. Pro menší sítě veřejné dopravy se jedná o vhodnou reprezentaci, nicméně s nárůstem velikosti sítě bude narůstat i velikost tohoto objektu, což může způsobit lag při startu aplikace, kdy se databáze načítá do operační paměti.

Celá aplikace využívá pro svoje fungování pouze jedno vlákno. To je velmi nevhodné, protože se jedná o UI vlákno, které operační systém Android používá pro obsluhu uživatelského rozhraní. Jedná se tak o primární příčinu výše popsaného lagu. Zároveň nastává podobný lag i během vyhledávání spojení, které brání vykreslení aktivity s nalezeným spojením. Kvůli druhému zmíněnému lagu aplikace nabízí pro zadaný dotaz pouze jedno optimální spojení.

3.2 Návrh zlepšení

Cílem této práce je navrhnout vylepšení aplikace pro vyhledávání spojení z pohledu použitých algoritmů. Zjevné nedostatky, jako je realizace načítání databáze a provádění výpočtu v UI vlákně, tak v dalším textu nebudou diskutovány.

Problém pomalé deserializace je z velké míry dán použitím nevhodných prostředků pro ni. Pro deserializaci je použita instance třídy *ObjectInputStream*, která rovnou objekt s databází načítá i deserializuje. Značného zrychlení načítání tak lze poměrně

nenáročně získat pouze použitím mezikroku, kde se soubor pomocí *FileInputStream* načte do bytového pole a teprve potom se provede jeho deserializace.

Možným vylepšením současného stavu by mohlo být využití rychlejšího algoritmu pro vyhledávání spojení. V současné chvíli se jako vhodný kandidát jeví Connection Scan Algorithm (viz sekce 1.6), který nevyužívá pro svoji funkci graf a ani prioritní frontu.

V současné chvíli nabízí aplikace pro zadaný dotaz pouze jedno optimální spojení. Paralelním vyhledáváním několika spojení najednou by mohla nabídnout nalezení více optimálních spojení za stejný čas. Za tímto účelem je vhodné použít heuristiku pro odhad časů odjezdu (viz sekce 2).

4 Návrh implementace

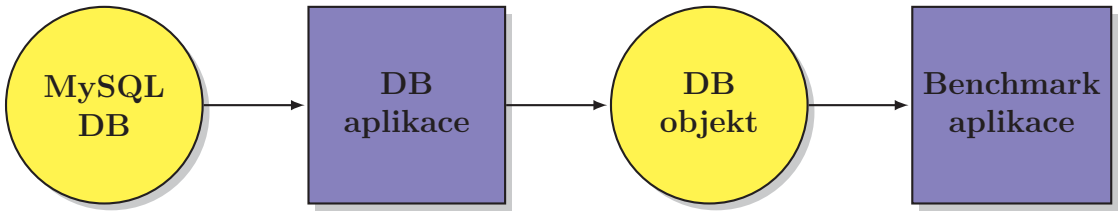
4.1 Typ a architektura aplikace

V rámci bakalářské práce [7] vznikly dvě aplikace - stolní DBBuilder pro vytvoření databáze a mobilní MHDApp, která nad zmíněnou databází vyhledává spojení v MHD. Tato diplomová práce navazuje na zmíněnou bakalářskou práci a navrhuje možnosti jejího vylepšení.

V rámci této práce zůstala aplikace DBBuilder z [7] zachována a navíc byla rozšířena tak, aby umožňovala výpočet dat používaných v heuristice pro odhad časů odjezdu (viz kapitola 2). Hlavním a jediným úkolem této aplikace je načít data z existující MySQL databáze, provést nad nimi výpočet dat pro heuristiku a tato data společně s jízdním řádem serializovat do souboru. K výpočtů ani exportu dat není třeba grafické rozhranní, proto byl DBBuilder implementován jako stolní konzolová aplikace.

Cílem této práce je navrhnout zlepšení použitelná v mobilní aplikaci MHDApp. Z této aplikace bylo zachováno pouze její jádro sloužící k vyhledávání spojení na časově závislém grafu, které bylo použito jako základ benchmarkovací aplikace ConnectionSearchExperiement. Tato aplikace slouží, jak už název napovídá, k porovnání rychlosti naprogramovaných vyhledávacích algoritmů. Pro svoji jednoduchost tak byla i v tomto případě zvolena implementace v podobě stolní konzolové aplikace.

Byla zachována architektura využívající dvě aplikace, které si mezi sebou „předávají" databázi, jako v [7]. Jedním z důvodů pro oddělení aplikací je, že databázová aplikace DBBuilder provádí výpočty náročné na čas, a tak je výhodné je provést pouze jednou a výsledek uložit do databáze. Dále byla jako požadavek stanovena snadná přenositelnost naimplementovaných vylepšení vyhledávacího algoritmu zpět do MHDApp.



Obrázek 4.1: Architektura aplikace

Na obrázku 4.1 je vizualizace použité architektury znázorňující databáze jako žlutá kolečka a aplikace jako fialové čtverce. Šipky označují směr předávání dat.

4.2 Volba programovacího jazyka

Implementovaná vylepšení je třeba jednoduše přenést zpět do aplikace MHDApp, která je určená pro platformu Android. Vhodnými kandidáty tak primárně jsou jazyky Java a Kotlin.

Při návrhu vylepšení byl kladen důraz na co největší opětovné použití již hotových komponent. Obě aplikace tak vychází z kódu použitého v pracích [6] a [7], který je napsán v jazyku Java. Veškerý kód by bylo třeba převádět do Kotlinu, což představuje dodatečnou práci, a tedy je výhodnější použít Javu.

4.3 Data a jejich uložení

Obě implementované aplikace vyžadují pro svoji funkci databázi. Aplikace DBBuilder jako zdroj dat, nad kterými vypočte data potřebná pro heuristiku pro odhad časů odjezdu, využívá MySQL databázi převzatou (v podobě tzv. dumpu) z [6].

MySQL databáze byla zvolena jako jednorázový zdroj dat, který obsahuje sice neaktuální, ale reálná a dobře zdokumentovaná data jízdních řádů pražské MHD. Nad touto databází se v rámci [6] hledali přestupní vzorce, a tedy některé její tabulky nebylo pro současné použití potřeba.

Níže je uvedena struktura používaných tabulek databáze převzatá z [6]:

- Lines (LineID, LineName, fromStationID, toStationID)

  fromStationID $\subseteq$ Stations.StationID
  toStationID $\subseteq$ Stations.StationID
- Stations (StationID, StationName)
- LineStations (LineID, StationID, Order, TimeToStop, WaitTime)

  StationID $\subseteq$ Stations.StationID
  LineID $\subseteq$ Lines.LineID
- LineStartTimes (LineID, StartTime, Day)

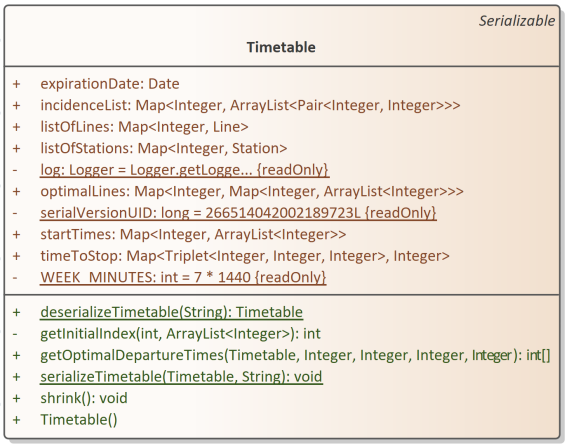
  LineID $\subseteq$ Lines.LineID
- ExpirationDate (Date)

Každý záznam seznamu označuje jednu tabulku v databázi, u které jsou v závorce uvedeny sloupce této tabulky. Podtržený název sloupce označuje primární klíč a symbol $\subseteq$ označuje cizí klíč. [6]

Řádek v tabulce Lines odpovídá jedné lince jízdního řádu. Pokud stejně pojmenovaná linka obsluhuje nějakou trasu v obou směrech, je do tabulky vložen záznam pro každý směr. Tabulka Stations obsahuje seznam stanic jízdního řádu. LineStations uchovává informaci o tom, jaká linka staví v které stanici, kolikátá je stanice na trase a čas dojezdu do této stanice ze startu. Záznam v tabulce LineStartTimes odpovídá jednomu času odjezdu linky. ExpirationDate pak obsahuje jediný záznam a tím je datum expirace databáze.

Z dat získaných výpočtem a jízdních řádů z MySQL databáze se následně vytvoří instance Třídy Timetable, která se serializuje do souboru. Ten je poté možné předat do benchmarkovací aplikace. Použitý model práce s daty tak přímo vychází z [7].

V [7] byly diskutovány i další možnosti uložení dat, aby je bylo možné jednoduše distribuovat a používat (na mobilním zařízení s operačním systém Android). Zároveň se ale úspěšně ověřilo, že serializovaný objekt představuje vhodný způsob uchovávání dat. Na obrázku 4.2 třída Timetable zapsaná v UML.



Obrázek 4.2: UML diagram třídy Timetable

„Instance této třídy se naplňuje pomocí dat získaných z MySQL databáze následovně:

- **listOfLines** se naplní daty z tabulky Lines, kde jeden záznam v tabulce odpovídá jedné lince. Jako klíč pro HashMap se použije Lines.LineID.
- **listOfStations** se naplní daty z tabulky Stations, kde jeden řádek v tabulce odpovídá jedné stanici. Jako klíč pro HashMap se použije Stations.StationID.
- **expirationDate** se naplní jediným záznamem uloženým v tabulce ExpirationDate.
- **startTimes** se naplní daty odjezdů linek z tabulky LineStartTimes. Časy odjezdů se ukládají do struktury HashMap, kde klíčem je Lines.LineID a hodnotou je seznam odjezdů dané linky, který je vzestupně setříděný. Uložené časy se upravují tak, aby odpovídaly skutečnému dnu v týdnu tak, jak bylo popsáno ve formálním popisu. MySQL databáze si totiž pamatuje čas v intervalu [0,1439] a příznak dne v intervalu [0,3], kde 0 označuje pondělí až čtvrtek, 1 označuje pátek, 2 označuje sobotu a 3 označuje neděli.
- **timeToStop** se naplní z tabulky LineStations. Klíčem této HashMapy je uspořádaná dvojice (lineID, stationID) a hodnotou je čas dojezdu linky do zastávky.
- **incidenceList** se naplni z tabulky LineStations. Klíčem HashMapy je lineID a hodnotou je seznam uspořádaných dvojic (stationID, order), které jsou vzestupně setříděné podle order.” [7]

Jediným rozdílem oproti implementaci popsané v [7] je, že jako klíč hašovací tabulky timeToStop se použije uspořádaná trojice (LineID, StationID, Oder), kde Order je pořadí stanice na trase.

4.4 Objektový návrh

Objektový návrh obou aplikací vychází z [7], které díky jeho modularitě bylo možné jednoduše rozšířit o další funkce a přitom zachovat co nejvíce původního kódu. Kompletní objektový návrh obou aplikací ve formě UML diagramu vytvořeném v programu Enterprise Architect je k dispozici na přiloženém DVD.

4.4.1 Databázová aplikace

Databázová aplikace DBBuilder vychází ze stejnojmenné aplikace, která vznikla v rámci [7]. Tato aplikace nabízela pouze export dat MySQL databáze do serializovaného Java objektu. Do této aplikace byla vložena část aplikace MHDDatabaseApp publikované v [6], která sloužila pro výpočet přestupních vzorců a po úpravě provádí profilovací dotazy na TEG nezbytné pro heuristiku navrženou v kapitole 2. Cílem bylo opětovné použití existujících komponent a jejich propojení s minimem nezbytných úprav v objektovém návrhu.

Aplikace se tak skládá z jednoduchého (konzolového) uživatelského rozhraní, které zastřešuje několik částí pro práci s MySQL databází, výpočet dat pro heuristiku a serializaci a export dat do souboru.

Jelikož je objektový návrh přiložen na DVD, v této sekci bude uveden pouze zevrubný seznam použitých tříd. Černě označané třídy jsou převzaty z [7] a šedě označené z [6]:

- **DataLoader** - Načítá data z MySQL databáze do paměti.
- **DBHelper** - Metody pro obsluhu databáze.
- **Dijkstra** - Implementace Dijkstrova algoritmu pro nalezení nejkratší cesty v grafu.
- **FibonacciHeap** - Otevřená implementace Fibonacciho haldy v Javě z projektu JGraphT.
- **Graph** - Třída reprezentující graf.
- **GraphBuilder** - Postaví TEG z dat načtených DataLoaderem.
- **Line** - Třída reprezentující linku v jízdním řádu.
- **Main** - Vstupní bod do aplikace. Obsluhuje uživatelské rozhraní.
- **Pair** - Uspořádaná dvojice.
- **PropertiesHelper** - Umožňuje manipulaci s uživatelskými nateveními a jejich uložení do souboru.
- **Station** - Třída reprezentující stanici v jízdním řádu.
- **Timetable** - Třída v jejíž instanci se uchovává jízdní řád a data heuristiky pro odhad časů odjezdu.
- **TransferPatternComputer** - Provádí profilovací dotazy nad TEG.
- **Triplet** - Uspořádaná trojice.
- **Utils** - Třída obsahující pomocné metody.

Data pro vyhledávání spojení a heuristiku pro odhad časů odjezdu se načítají do instance třídy Timetable, kterou lze vidět na UML diagramu na obrázku 4.3. Aby bylo možné tento objekt serializovat do souboru, musí každá třída, kterou Timetable využívá, implementovat rozhranní *Serializable*. [7]

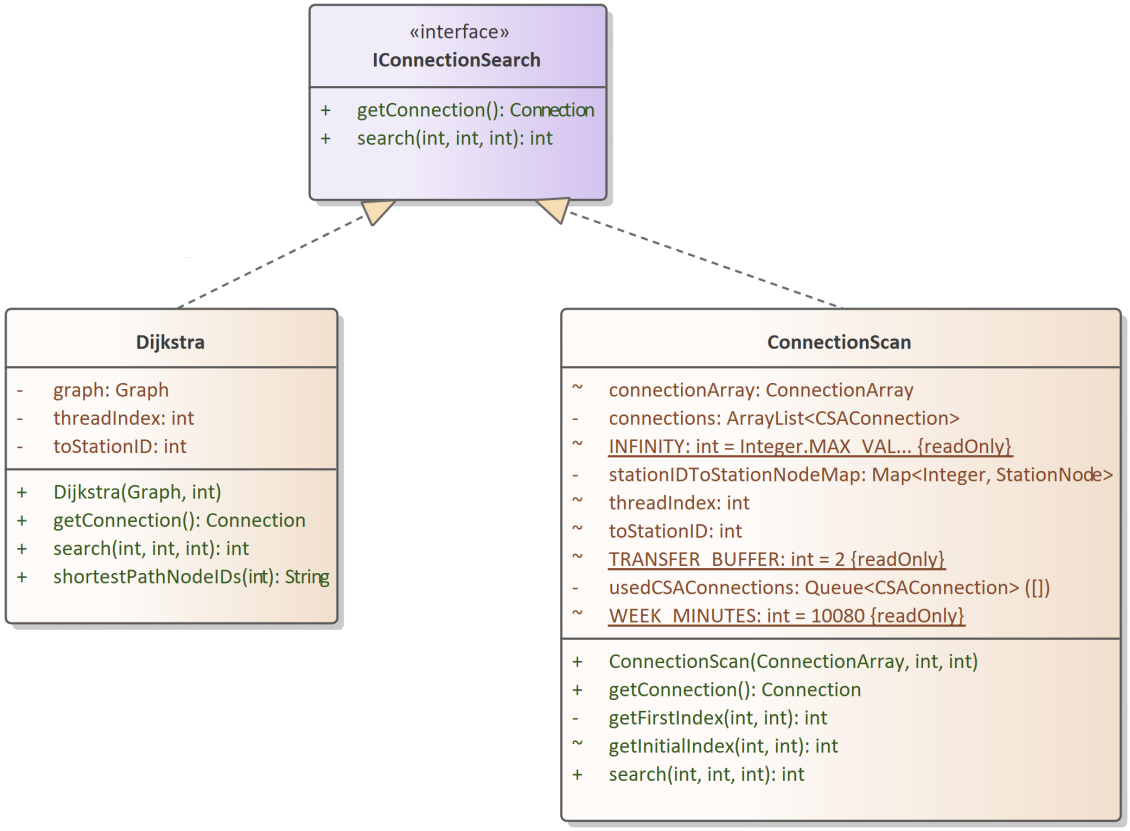
Obrázek 4.3: UML diagram tříd uchovávajících data jízdních řádů

4.4.2 Benchmarkovací aplikace

Benchmarkovací aplikace slouží pro určení rychlosti vyhledávání spojení a skládá se ze tří logických částí, které tvoří tzv. balíčky (package): ConnectionSearchExperiment, Time DependentSearch a ConnectionScan. TimeDependentSearch slouží k vyhledávání spojení na TDG. Za tímto účelem využívá kód převzatý z aplikace MHDApp. Druhý zmíněný, ConnectionScan, umožňuje vyhledávání spojení pomocí CSA. Posledním balíčkem je ConnectionSearchExperiement, který onen benchmark spouští.

Programování bylo vedeno vůči rozhranní IConnectionSearch, které poskytuje dvě metody. První metodou je *search(int, int, int)*, které se jako parametr předá id startovní a cílové stanice a čas odjezdu. Ta na základě těchto parametrů nalezne nejkratší cestu mezi zadanými stanicemi a vrátí čas příjezdu. Následně je možné zavolat metodu *getConnection()*, která vrátí seznam spojení použitých na nalezené optimální cestě.

Použitím rozhranní IConnectionSearch je možné v aplikaci jednoduše zaměnit implementaci vyhledávacího algoritmu bez nutnosti jakkoli modifikovat kód, který nad ním pracuje. Příklad jeho použití pro vyhledávání pomocí Dijkstrova algoritmu na TDG a pomocí CSA je na obrázku 4.4.



Obrázek 4.4: UML diagram rozhraní IConnectionSearch

Balíček TimeDependentSearch vychází z [7]. Ze zmíněné práce byl převzat model vyhledávání spojení na TDG pomocí Dijkstrova algoritmu a byl upraven tak, aby využíval rozhranní IConnectionSearch a novou verzi Timetable. Zároveň byla přídana podpora pro realizaci paralelního výpočtu. Níže je uvedený zevrubný seznam použitých tříd (bez vnitřních tříd):

- **Connection** - Třída obalující seznam elementárních spojení. Její instance slouží pro popis nalezené optimální cesty.
- **ConnectionSearchExecutor** - Umožňuje spuštění výpočtu nejkratší cesty v samostatném vlákně.
- **Dijkstra** - Implementace Dijkstrova algoritmu pro nalezení nejkratší cesty v grafu.
- **ElementaryConnection** - Elementární spojení mezi dvěma stanicemi.
- **Graph** - Třída reprezentující graf.
- **IConnectionSearch** - Rozhraní proti kterému se implementují jednotlivé vyhledávací algoritmy.
- **Query** - Třída reprezentující jeden zadaný dotaz na nalezení spojení.
- **Utils** - Třída obsahující pomocné metody.

V rámci balíčku ConnectionScan je implementován pro vyhledávání spojení implementován CSA, který se skládá z následujících tříd:

- **ConnectionArray** - Pole spojení CSAConnection.
- **ConnectionScan** - Implementace CSA.
- **CSAConnection** - Elementární spojení.
- **StationNode** - Záznam pro stanici.

Jedinou funkcí ConnectionSearch je načíst ze souboru dotazy (query), spustit nad nimi vyhledávání optimálních spojení pomocí algoritmů implementovaných v TimeDependentSearch a ConnectionScan a zapsat výsledky společně s dobou trvání výpočtu.

4.5 Návrh uživatelského rozhraní

Obě implementované aplikace slouží primárně pro ověření navrženého konceptu pro zvýšení výkonu vyhledávání spojení, a proto bylo zvoleno nejjednodušší možné uživatelské rozhraní využívající příkazovou řádku.

4.5.1 Databázová aplikace

Aplikace, která vznikla v rámci této práce, rozšiřuje DBBuilder publikovaný v [7] o funkci výpočtu dat potřebných pro heuristiku pro odhad časů odjezdu a jejich kompresi. Uživatelské rozhraní tak umožňuje upravovat nastavení, exportovat databázi do souboru a provést její kompresi. Zvolená funkce se spustí předaním parametru na příkazové řádce.

Nastavení

„Pro spuštění nastavení se předá programu parametr **settings**. Po spuštění bude uživatel vyzván, aby zadal příkaz. Nastavení podporuje následující příkazy:

- **save** - Uložení provedených nastavení.
- **file** - Nastavení jména souboru (včetně cesty), do kterého se budou data exportovat.
- **mysql-db** - Nastavení jména MySQL databáze.
- **mysql-user** - Nastavení uživatelského jména pro přístup do MySQL databáze.
- **mysql-pass** - Nastavení uživatelského hesla pro přístup do MySQL databáze.
- **help** - Vypsání všech dostupných příkazů.
- **end** - Ukončení programu.

Pokud bude jako vstup zadáno cokoli, co neodpovídá klíčovému slovu v seznamu výše, bude uživateli zobrazena nápověda, tj. vypíšou se všechny příkazy, které aplikace v Nastavení podporuje jako v případě zadání příkazu **help**.” [7]

Export databáze

Spuštěním této funkce se z MySQL databáze načtou data jízdních řádů, uloží se do instance třídy Timetable a spustí se výpočet dat heuristiky pro odhad časů odjezdu. Po skončení výpočtu se objekt Timetable naplní výslednými daty a serializuje se do nastaveného souboru (viz Nastavení). Pro spuštění exportu dat se programu na příkazové řádce předá parametr **export**.

Komprese databáze

Pro spuštění komprese se předá programu parametr **shrink**. Program načte databázi uloženou v souboru, do kterého se data exportovali, a provede kompresi. Zkomprimovanou databázi zapíše do souboru *shrunken-timetable.bin*.

4.5.2 Benchmarkovací aplikace

Benchmarkovací aplikace se podobá spíše jednoduchému skriptu. Do adresáře, kde se aplikace nachází, se vloží vstupní soubory:

- **timetable.bin** - Exportovaná databáze (serializovaný objekt Timetable).
- **queries.txt** - Dotazy na spojení (viz kapitola 6).

Aplikace se spouští bez jakýchkoliv parametrů a výsledky výpočtu se opět uloží do několika souborů v adresáři, kde se aplikace nachází:

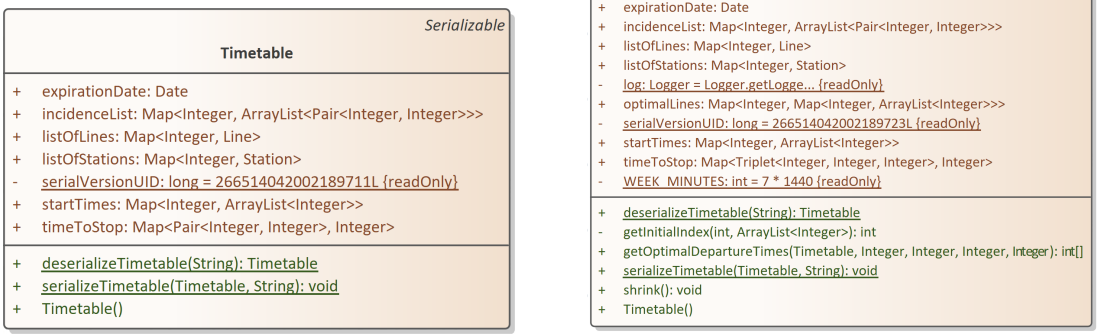
- **statistics.txt** - statistika výpočtu.
- **serial.txt** - Výsledky sekvenčního vyhledávání pomocí TDG.
- **parallel-dirty.txt** - Výsledky paralelního vyhledávání pomocí TDG.
- **parallel.txt** - Výsledky paralelního vyhledávání pomocí TDG očištěné od duplicitních spojení.
- **parallel-csa-dirty.txt** - Výsledky paralelního vyhledávání pomocí CSA.
- **parallel-csa.txt** - Výsledky paralelního vyhledávání pomocí CSA očištěné od duplicitních spojení.

5 Implementační detaily

V této kapitole budou zmíněny některé detaily, které úzce souvisí s návrhem implementace aplikace.

5.1 Chyba v Timetable

Použitá implementace třídy Timetable vychází z [7], ale bylo ji třeba upravit, protože obsahovala chybu. Hašovací tabulka timeToStop využívala klíče v podobě uspořádané dvojice (LineID, StationID). Pro zadaný klíč vracela čas dojezdu linky s daným LineID do zastávky s odpovídajícím StationID. Taková implementace neumožňovala správně uložit jízdní řády pro spoje, které na trase obsahovaly cyklus, tj. během jedné jízdy projeli některou stanicí vícekrát.



(a) Původní implementace Timetable **(b)** Opravená implementace Timetable

Obrázek 5.1: Srovnání původní a nové implementace třídy Timetable

Na obrázku 5.1 je pro porovnání zobrazena původní a nová implementace třídy Timetable. Chyba se odstranila tak, že se nově jako klíč hašovací tabulky timeToStop používá uspořádaná trojice (LineID, StationID, Oder), kde Order je pořadí dané zastávky na trase.

5.2 Shrink

Databáze optimálních linek je uchovávána jako serializovaný javovský objekt a tak je možné se pokusit snížit jeho velikost. Při ukládání nového seznamu optimálních linek

mezi dvěma stanice (ArrayList) se zkontroluje, jestli už identický seznam neexistuje. Pokud ano, nevloží se nový, ale pouze reference (ukazatel) na už existující seznam. To může být výhodné v případě odlehlých stanic, kam vede jediná dlouhá trasa, protože v těchto případech se bude uložený seznam optimálních linek shodovat v několika stanicích.

Tato funkce provádějící kompresi byla pro použití v databázové aplikaci pojmenována jako *Shrink*.

5.3 Deserializase databáze

Deserializace databáze, která je uložena jako serializovaný Java objekt, je vzhledem k velikosti souboru zdlouhavá. Tento problém je z velké míry dán použitím nevhodných prostředků pro ni. Konkrétně pro deserializaci je použita přímo instance třídy *ObjectInputStream*, která využívá metodu *readObject()*. Značného zrychlení načítání tak lze poměrně nenáročně získat pouze použitím mezikroku v podobě bytového pole. Konkrétně tedy načtení souboru pomocí *FileInputStream* a jeho metody *readAllBytes()* a následně předat takto načtené pole bytů prostřednictvím instance *ByteArrayInputStream* k deserializaci do *ObjectInputStream*.

5.4 Jízdní řád pro CSA

CSA umožňuje velmi rychlé nalezení spojení nad aperiodickým jízdním řádem. Běžně se však jízdní řád MHD stanovuje pro jeden týden, který se opakuje. Řešením je ho „nagenerovat" na každý den zvlášť. [8]

Pokud by se například opakoval jízdní pro pondělí až čtvrtek, do aperiodického jízdního řádu se přidají časy odjezdu, ke kterým se podle dne odjezdu přičte $n \times 1440$, kde n je pořadí dne v týdnu (pondělí je nultý den, tj. $n = 0$) a 1440 počet minut v jednom dni.

V případě týdenní periody se do aperiodického jízdního řádu přidají ještě jednou odjezdy pro pondělí, ke kterým se přičte 7×1440, tj. počet minut v týdnu.

6 Experiment

Cílem experimentu bylo implementovat navrhovanou heuristiku pro paralelní vyhledávání spojení a ověřit kvalitu poskytované predikce časů odjezdu, tj. její úspěšnost. Důraz byl přitom kladen na celkové zrychlení aplikací nabízejících více spojení pro jeden dotaz. Na základě tohoto pokusu pak bylo provedeno zhodnocení, zda-li je heuristika vhodná pro praktické použití.

Za účelem zrychlení vyhledávání spojení byl implementován také CSA. V případě tohoto algoritmu se však nepodařilo modelovat reálnou dobu pro přestup.

Druhotným cílem experimentu bylo ověřit předpoklad, že se v navržené struktuře pro optimální linky často opakuje identický seznam linek. Testovaná metoda *Shrink* byla zhodnocena z pohledu minimalizace velikosti databáze.

6.1 Popis experiemntu

Pro ověření funkčnosti a měření praktické časové náročnosti byl jako referenční zařízení použit notebook Dell Latitude E6410 s touto konfigurací:

- Windows 10
- CPU Intel Core i3 M380 (2 jádra s max. frekvencí 2,53GHz, 4 „virtuální" vlákna)
- 8GB RAM
- 250 GB SSD

Testovací aplikace byla naimplementována a spouštěna v open-source variantě Javy verze 12 od OpenJDK.

Jako základ pro experiment posloužila data publikovaná v [7]. Jedná se o data jízdních řádů pražské MHD bez příměstských a atypických (např. lanovka) spojů, tj. obsahuje pouze metro, tramvaje a autobusy. Přestože data již nejsou aktuální (expirovala v roce 2014), poskytují věrný obraz dopravní sítě z reálného života, na které by mohla být heuristika pro odhad časů odjezdu v praxi nasazena, a jsou dobře zpracována do dostupné MySQL databáze. Nevýhodou této databáze je naopak absence spojů, které „zatahují", tj. spojů, které nejedou až na konečnou zastávku dané linky, ale končí v dřívější stanici.

Za účelem ověření úspěšnosti heuristiky byly ze zvolených dat vytvořeny dvě databáze. První databáze se skládala pouze z linek metra a tramvajových linek. Druhá databáze obsahovala všechny linky dostupné v převzaté databázi, tj. metro, tramvaje i autobusy.

V rámci experimentu bylo aplikaci zadáno 50 dotazů (viz tabulka 6.4) na vyhledání spojení ve tvaru *„start;čas;den;cíl"* a aplikace tento počet zdvojnásobila tak, že přidala dotazy s prohozenou startovní a cílovou stanicí. Celkem tedy bylo vyhledávacímu algoritmu předáno 100 dotazů na nalezení spojení. Pro každý dotaz se pomocí heuristiky predikovaly 4 časy odjezdu a pro tyto časy se paralelně ve 4 vláknech vyhledalo spojení. Celkem tedy bylo zadáno a vyhledáno 400 spojení nad každou ze zvolených databází. Nalezená spojení se následně očistila od duplicit.

Nejprve byla vyhledáno všech 100 dotazů spojení sekvenčně. Výpočet probíhal tak, že na zadaný dotaz bylo nalezeno jedno spojení. Čas odjezdu tohoto spojení inkrementovaný o 1 minutu následně posloužil jako čas odjezdu zadaný v dalším dotazu. Celkem byly tímto způsobem vyhledány 4 unikátní spojení pro každý dotaz.

V další fázi byly vyhledány všechny dotazy na spojení paralelně za pomocí navrhované heuristiky pro odhad časů odjezdu. Pro každý zadaný dotaz byly heuristikou predikovány 4 časy odjezdu po čase zadaném v dotazu. Pro tyto dotazy byla vyhledána spojení, která ale nemusela být unikátní, tj. v nalezených spojeních se mohly vyskytovat duplicity. Tyto duplicity byly následně odstraněny. Z toho důvodu nenalezl paralelní algoritmus stejné množství spojení jako sekvenční a poměr počtu paralelně nalezených spojení a sekvenčně nalezených určuje úspěšnost heuristiky.

Primárním cílem experimentu bylo ověřit, jestli dojde použití heuristiky pro odhad časů odjezdu umožní zrychlení vyhledávání více spojení na TDG. Pro porovnání byla implementováno i vyhledávání pomocí CSA, avšak tento algoritmus nevracel vždy optimální spojení, a tedy na něm nebyla úspěšnost heuristiky testována.

Jako součást pokusu byla navržena metoda *shrink* optimalizující velikost databáze, konkrétně tedy datovou strukturu uchovávající seznamy optimálních linek jedoucích mezi dvojící stanic. Tato metoda byla aplikována na obě použité databáze.

6.2 Výsledek experimentu

6.2.1 Heuristika pro odhad časů odjezdu

Pro obě zvolené databáze byly použity stejné stejné dotazy na nalezení optimálního spojení pomocí Dijkstrova algoritmu na TDG. Seznam zadaných dotazů je uveden v tabulce 6.4. Uvedené dotazy byly aplikací rozšířeny o další dotazy tak, že byl otočen směr cesty, tj. došlo k prohození startovní a cílové stanice, ale čas a den odjezdu zůstal zachován.

Na první databázi bylo dosaženo následujících výsledků:

- Sekvenční výpočet trval 2265 milisekund
- Paralelní výpočet trval 718 milisekund
- Pomocí heuristiky nalezeno 298 unikatních spojení z 376. Úspešnost 80.5%

Z výsledeků je zřejmé, že zatímco sekvenčně trvalo nalezení jednoho spojení průměrně 5,66 milisekundy, paralelně stačilo aplikaci jen 2,23 milisekundy. Došlo tak ke zrychlení 2,5krát. Oproti předpokládaným 400 spojením nalezla aplikace jen 376 spojení. To je dáno tím, že 6 zadaných dotazů (celkem se tedy vyhedávalo 24 spojení) bylo na spojení mezi zastávkami, kam jezdí pouze autobusy, a tedy pro ně (v rámci redukované databáze) žádná cesta neexistuje. Tyto spoje tak bylo nutné vzít v úvahu a dodatečně upravit statistiku generovanou programem.

Na tomto příkladu je vidět i sekundární výhoda heuristiky, která již během odhadování časů odjezdu odhalí, že nějaké spojení není vůbec dostupné, zatímco standardní algoritmus k tomu musí dojít zdlouhavým výpočtem.

Na druhé databázi bylo dosaženo následujících výsledků:

- Sekvenční výpočet trval 5890 milisekund
- Paralelní výpočet trval 2531 milisekund
- Pomocí heuristiky nalezeno 291 unikatních spojení z 400. Úspešnost 72.75%

Sekvenčně trvalo nalezení jednoho spojení průměrně 14,73 milisekundy, paralelně stačilo aplikaci jen 8,7 milisekundy. Přestože klesla úspěšnost heuristiky, došlo tak ke zrychlení 1,7krát. Heuristika tak dosahuje dobré úspěšnosti a zároveň velmi uspokojivě zrychluje nalezení více spojení. Pro porovnání jsou výsledky experimentu shrnuty v tabulce 6.1.

Tabulka 6.1: Výsledky experimentu

	Redukovaná databáze	Plná databáze
Celkem sekvenčně	2265 ms	5890 ms
Celkem paralelně	718 ms	2531 ms
Průměr sekvenčně	5,66 ms	14,73 ms
Průměr paralelně	2,23 ms	8,7 ms
Zrychlení (na 1 spojení)	2,5x	1,7x
Úspěšnost heuristiky	80.5%	72.75%

6.2.2 Metoda shrink

Metoda shrink dosáhla překvapivě dobrých výsledků. Při její aplikaci na první (redukovanou) databázi byla její velikost zredukována na 32,25% původní velikosti. Ještě lepších výsledků dosáhla na druhé (plné) databázi, kdy se její velikost podařilo zredukovat na pouhých 23,54% původní velikosti. Experimentem se tak podařilo potvrdit předpoklad, že se v databázi často opakuje množina optimálních linek pro více zastávek. Díky této optimalizaci je možné velmi znatelně snížit velikost databáze a tím také urychlit její načítání. Souhrn výsledků je možné si prohlédnout v tabulce 6.2.

Tabulka 6.2: Porovnání velikosti databází

	Před kompresí (kB)	Po kompresi (kB)	Poměr
Redukovaná databáze	4989	1609	32,25%
Plná databáze	77107	18154	23,54%

6.2.3 CSA

Implementace CSA, která vznikla v rámci této práce, neumožňuje modelovat reálnou dobu na přestup. Veškeré snahy o implementaci algoritmu popsaného v sekci 1.6 byly neúspěšné. Vytvořený vyhledávací algoritmus tak nemusí vždy vrátit optimální spojení. Z tohoto důvodu není možné spolehlivě vyhodnotit jeho výkon v porovnání s Dijkstrovým algoritmem vyhledávajícím na TDG, protože výsledkem dotazu mohou být naprosto rozdílná spojení. Pro úplnost je však v tabulce 6.3 uvedeno alespoň porovnání doby, kterou algoritmy potřebovaly pro paralelní vyhledávání.

Tabulka 6.3: Srovnání doby běhu Dijkstrova algoritmu na TDG a CSA

	Redukovaná databáze	Plná databáze
Dijkstra paralelně	718 ms	2531 ms
CSA paralelně	343 ms	624 ms

Tabulka 6.4: Zadané dotazy

Ze stanice	Čas odjezdu	Den odjezdu	Do stanice
Florenc	9:00	středa	Letňany
Florenc	20:00	čtvrtek	Letňany
Florenc	9:00	středa	Náměstí Republiky
I. P. Pavlova	7:00	úterý	Depo Hostivař
I. P. Pavlova	7:00	úterý	Gercenova
I. P. Pavlova	7:00	úterý	Háje
I. P. Pavlova	17:00	středa	Hlavní nádraží
I. P. Pavlova	17:00	středa	Karlovo náměstí
Malostranská	11:00	pátek	I. P. Pavlova
Malostranská	11:00	pátek	Malostranské náměstí
Malostranské náměstí	9:50	pondělí	Břevnovský klášter
Malostranské náměstí	9:50	pondělí	Dejvická
Malostranské náměstí	9:50	pondělí	Hlavní nádraží
Malostranské náměstí	9:50	pondělí	Kuchyňka
Malostranské náměstí	11:50	pondělí	Národní divadlo
Malostranské náměstí	8:00	pátek	Nádraží Holešovice
Malostranské náměstí	10:00	pondělí	Náměstí Míru
Malostranské náměstí	10:00	pondělí	Pankrác
Malostranské náměstí	10:00	pondělí	Petřiny
Malostranské náměstí	17:00	pondělí	Skalka
Malostranské náměstí	17:00	pondělí	Strossmayerovo náměstí
Malostranské náměstí	17:00	pondělí	Větrník
Malostranské náměstí	17:00	pondělí	Vinohradská vodárna
Nádraží Holešovice	14:00	sobota	Černý Most
Nádraží Holešovice	11:00	pondělí	Dělnická
Nádraží Holešovice	7:00	pondělí	I. P. Pavlova
Nádraží Holešovice	8:00	pondělí	Křižíkova
Nádraží Holešovice	8:00	pondělí	Lazarská
Nádraží Holešovice	5:00	pátek	Lazarská
Nádraží Holešovice	8:00	pondělí	Větrník
Náměstí Míru	20:00	pondělí	Kubánské náměstí
Náměstí Republiky	9:50	pondělí	Hlavní nádraží
Pankrác	7:00	úterý	Gercenova
Pankrác	15:15	úterý	Pražská tržnice
Sparta	6:00	pondělí	Strossmayerovo náměstí
Sparta	20:00	pondělí	Strossmayerovo náměstí
Větrník	9:10	pondělí	Čechovo náměstí
Větrník	9:10	pondělí	Dejvická
Větrník	9:10	pondělí	Divoká Šárka
Větrník	19:10	pondělí	Sparta
Větrník	19:10	pondělí	Strossmayerovo náměstí
Vozovna Pankrác	19:10	pondělí	Malostranské náměstí
Vychovatelna	8:30	pondělí	Křižíkova
Výstaviště Holešovice	9:10	pondělí	Pankrác

7 Závěr

Cílem práce bylo navrhnout zlepšení aplikace pro operační systém Android publikované v [7]. Tato aplikace umožňuje offline vyhledávání spojení ve veřejné dopravě, tj. bez přístupu k nějaké internetové službě, která by realizovala vyhledávání na serveru. Kvůli tomu je třeba brát v potaz potenciálně omezený výkon a paměť mobilního zařízení s Androidem.

V kapitole 1 byl formalizován problém vyhledávání spojení ve veřejné dopravě a následně byly popsány používané modely a algoritmy. Pozornost byla věnována jak Dijkstrovu algoritmu, který umožňuje vyhledávání nad různými modely, ale i novým negrafovým algoritmům. Ukazuje se, že právě negrafové algoritmy jako CSA jsou vhodné pro použití v zařízeních s omezeným výkonem. Pro svou funkci totiž nepotřebují graf, který může spotřebovat hodně operační paměti, ani haldu (nebo prioritní frontu), nad kterou se v případě vyhledávání na grafech dělá mnoho operací, což vyhledávací algoritmus zpomaluje.

Následující kapitola 2 diskutovala použití paralelismu při vyhledávání spojení. Tato technika by umožnila zrychlení vyhledávání a tím i zlepšení zážitku z uživatelského rozhraní. Byla navržena heuristika pro odhad časů odjezdu optimálního spojení, která by umožnila paralelizovat vyhledávání několika spojení současně. Rovněž byly navrženy její další zlepšení.

V závěru práce byl v kapitole 6 úspěšně proveden experiment pro ověření použitelnosti navrhované heuristiky. Na pražské dopravní síti (pro jízdní řády metra, tramvají a autobusů) dokázala heuristika využívající pro vyhledávání model založený na TDG nabídnout jedno spojení v průměru 1,7krát rychleji oproti sekvenčnímu vyhledávání (na TDG) s úspěšností predikce 72,75%.

V rámci experimentu byla testovaná i implementace CSA, která dosahovala výborné rychlosti vyhledávání spojení oproti Dijkstrovu algoritmu na TDG, avšak nepodařilo se implementovat podporu reálné doby pro přestup. Nalezená spojení tak ne vždy byla optimální.

Pro zlepšení zážitku z uživatelského rozhraní se tak ukázalo jako vhodné, aby aplikace pro zadaný dotaz vyhledávala více spojení. Jako vhodné se pro tento účel ověřilo použití navržené heuristiky pro odhad časů odjezdu, která umožňuje využití toho druhu paralelismu.

Práci lze hodnotit jako úspěšnou, protože navrhovaná heuristika pro odhad časů odjezdu umožňuje pro zadaný dotaz paralelně vyhledávat několik spojení a to pomocí libovolného popsaného algoritmu. Pokud by se navíc podařilo opravit chybu v implementaci CSA, nabízí uvedené algoritmy zlepšení a zrychlení aplikace z [7].

7.1 Budoucí práce

Během práce vzniklo mnoho podnětů pro další zkoumání. Zajímavým podnětem byla problémová implementace CSA a do budoucna by tak bylo vhodné ověřit, jestli je skutečně možné modelovat přestupy s reálnou dobou.

Guidebook Routing, tj. směrování podle průvodce, se jevilo být zajímavým řešením, jak navrhovanou heuristiku pro odhad časů odjezdu očistit od vlivu vzorců, které jsou optimální pouze v malém počtu časů odjezdu.

Neméně zajímavá je možnost experimentálně ověřit realizaci heuristiky pro odhad časů odjezdu nad daty získanými z výpočtu přestupních vzorců a zjemněnými pomocí Guidebook routing. Použití heuristiky společně s přestupními vzorci by umožňovalo implementaci velmi rychlého řešení pro vyhledávání spojení pro servery.

Literatura

[1] Bast, Hannah. *Car or Public Transport – Two Worlds*. In Susanne Albers, Helmut Alt, and Stefan Näher, editors, Efficient Algorithms, volume 5760 of Lecture Notes in Computer Science, pages 355–367. Springer, 2009. ISBN 978-3-642-03456-5.

[2] Bast, Hannah, Carlsson, Erik, Eigenwillig, Arno, Geisberger, Robert, Harrelson, Chris, Raychev, Veselin, and Viger, Fabien. *Fast Routing in Very Large Public Transportation Networks using Transfer Patterns*. In Mark de Berg and Ulrich Meyer, editors, ESA, volume 6346 of Lecture Notes in Computer Science, pages 290–301, Springer, 2010. ISBN 978-3-642-15775-2.

[3] Bast, Hannah. *Efficient Route Planning*. [přednáška]. Freiburg: Albert-Ludwigs-Universiät Freiburg, 2012. [vid 5.3.2020]. Záznam dostupný z: https://ad-wiki.informatik.uni-freiburg.de/teaching/EfficientRoutePlanningSS2012

[4] Bast, Hannah, Storandt, Sabine. *Flow-Based Guidebook Routing*. In Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments (ALENEX), pages 155-165, 2014. DOI 10.1137/1.9781611973198.15.

[5] Geisberger, Robert. *Advanced Route Planning in Transportation Networks*. PhD thesis, Karlsruhe Institute of Technology, 2011.

[6] Křepelka, Michal. *Off-line vyhledávání spojení na platformě Google Android*. Bakalářská práce, MFF UK, 2015

[7] Křepelka, Michal. *Offline vyhledávání spojů na platformě Android*. Bakalářská práce, UJEP, 2017

[8] Křepelka, Michal. *Rešerše možností vyhledávání spojení v MHD*. Magisterský projekt, TUL, 2019

[9] Pyrga, Evangelia, Schulz, Frank, Wagner, Dorothea, Zaroliagis, Christos. *Efficient Models for Timetable Information in Public Transportation Systems*. ACM Journal of Experimental Algorithmics, 12(2.4):1–39, 2008.

[10] Strasser, Ben, Dibbelt, Julian, Pajor, Thomas, Wagner, Dorothea. *Intriguingly Simple and Fast Transit Routing*. In Experimental algorithms. 12th international symposium, SEA 2013, Rome, Italy, 2013.

[11] Delling, Daniel, Katz, Bastian, Pajor, Thomas. *Parallel Computation of Best Connections in Public Transportation Networks*. Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing (IPDPS), Atlanta, pages 1-12, 2010. DOI 10.1109/IPDPS.2010.5470345.

[12] Delling, Daniel, Pajor, Thomas, Werneck, Renato. *Round-Based Public Transit Routing*. Transportation Science, volume 49, 2012. DOI 10.1287/trsc.2014.0534.

A Obsah přiloženého DVD

soubor/adresář	obsah
diplomová-prace.pdf	text diplomové práce v pdf
eclipse-source.zip	zdrojové kódy obou aplikací importovatelné do Eclipse
full-shrunken-timetable.bin	databáze (serializovaný objekt Timetable), komprimovaná metodou Shrink
full-timetable_reduced.bin	databáze (serializovaný objekt Timetable)
full-timetable-experiment.zip	výsledky experimentu nad databází
javadoc.zip	vygenerovaný javadoc
queries.txt	dotazy na vyhledání spojení
rebuild_db.sql	skript pro import databáze použité v [6]
rebuild_reduced_db.sql	skript pro import redukované databáze
reduced-shrunken-timetable.bin	redukovaná databáze (serializovaný objekt Timetable), komprimovaná metodou Shrink
reduced-timetable_reduced.bin	redukovaná databáze (serializovaný objekt Timetable)
reduced-timetable-experiment.zip	výsledky experimentu nad redukovanou databází
uml-diagramy.zip	UML diagramy pro databázovou i benchmarkovací aplikaci vytvořené v programu Enterprise Architect