VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

PLÁNOVÁNÍ CEST V LETECKÉ DOPRAVĚ

ROUTE PLANNING IN AIR TRANSPORT

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. MAREK SYCHRA**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. ZBYNĚK KŘIVKA, Ph.D.**
SUPERVISOR

BRNO 2017

Zadání diplomové práce/21282/2017/xsychr05

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2017/2018

Zadání diplomové práce

Řešitel: **Sychra Marek, Bc.**
Obor: Inteligentní systémy
Téma: **Plánování cest v letecké dopravě**
Route Planning in Air Transport
Kategorie: Algoritmy a datové struktury

Pokyny:

1. Seznamte se s problematikou plánování cest v různých typech dopravy se zaměřením na leteckou.
2. Nastudujte existující algoritmy pro plánování cest a grafové reprezentace používané při řešení. Analyzujte také různé relevantní vlastnosti grafových reprezentací (např. míra dynamičnosti grafu, možnosti předzpracování apod.).
3. Dle konzultací s vedoucím navrhněte úpravu/optimalizaci vybraného algoritmu pro specifika letecké dopravy.
4. Proveďte experimentální porovnání s existujícím algoritmem na reálných datech a zhodnoťte jeho přínos. Pokuste se také nový algoritmus srovnat oproti existujícím (typicky proprietárním) vyhledávačům letů.

Literatura:

- Bast H. et al. (2016) Route Planning in Transportation Networks. In: Kliemann L., Sanders P. (eds) Algorithm Engineering. Lecture Notes in Computer Science, vol 9220. Springer, Cham
- Dibbelt J., Pajor T., Strasser B., Wagner D. (2013) Intriguingly Simple and Fast Transit Routing. In: Bonifaci V., Demetrescu C., Marchetti-Spaccamela A. (eds) Experimental Algorithms. SEA 2013. Lecture Notes in Computer Science, vol 7933. Springer, Berlin, Heidelberg
- dle doporučení vedoucího

Při obhajobě semestrální části projektu je požadováno:

- Body 1, 2 a částečně bod 3.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Křivka Zbyněk, Ing., Ph.D.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 23. května 2018

L.S.

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Problematika plánování cest v letecké dopravě (v hromadné dopravě obecně) je podobná hledání nejkratší cesty v grafu. Hlavními rozdíly jsou však časová závislost vstupního grafu a fakt, že cena cesty je určena více kritérii. Cílem této práce bylo vytvořit komplexní systém, který je schopen po načtení databáze elementárních letů odpovídat na uživatelské dotazy cestou spojenou z více samostatných letů. Výsledku je docíleno pomocí dvou algoritmů pro plánování cest v hromadné dopravě, CSA a RAPTOR, které byly upraveny pro specifika letecké dopravy. Experimenty, které probíhaly na reálných datech, ukázaly masivní zrychlení původních algoritmů při použití navržených optimalizací. Celý systém byl také porovnán s existujícím proprietárním řešením.

Abstract

The problem of route planning in air transport (in public transport in general) is similar to the shortest path problem. The main differences are the time dependency of the input graph and the multicriterial aspect of the path costs. The aim of this thesis was to create a complex system that would be able to load elementary flights from database and then respond to user requests with combined journeys made from single flights. It was achieved using two state of the art algorithms, CSA and RAPTOR, which were adapted for the flight graph. The experiments which were run on real world data showed massive speedup of the algorithms when using the proposed optimisations. The whole system was also tested against an existing proprietary solution.

Klíčová slova

plánování cest, letecká doprava, nejkratší cesta, hromadná doprava, časově závislý graf

Keywords

route planning, air transport, shortest path, public transport, time dependent graph

Citace

SYCHRA, Marek. *Plánování cest v letecké dopravě*. Brno, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Zbyněk Křivka, Ph.D.

Plánování cest v letecké dopravě

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Zbyňka Křivky, Ph.D. Další informace mi poskytl Mgr. Jan Plhák. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Marek Sychra
19. května 2018

Poděkování

Rád bych poděkoval Ing. Zbyňku Křivkovi, Ph.D. za vedení při vypracování práce a Mgr. Janu Plhákovi za odborné konzultace.

Obsah

Kapitola 1

Úvod

V dnešní době je doprava něčím naprosto neoddělitelným od našich životů. Ať už jde o cestu mezi domovem a zaměstnáním, pracovní cestu nebo prázdninový výlet. K tomu ale také patří jisté plánování cesty. Dnes už není potřeba si cestu vyhledávat sám ručně v atlase, lze použít jeden z mnoha plánovačů cesty na internetu. Stejná situace je při použití hromadné dopravy; již neplatí, že by si člověk musel pamatovat, kudy a čím se dostane přes městskou část. Toto je umožněné díky faktu, že hromadná doprava, jak ji známe, je centralizovaná a jízdní řády bývají zpravidla neměnné po delší dobu.

Typem hromadné dopravy, jejíž plánování je však stále v počátcích, je letecká doprava. Vzhledem k tomu, že pokrývá celý svět a informace o letech jsou velmi proměnlivé, existuje pouze málo společností, které se odhodlaly řešit problém plánování tras v letecké dopravě. Tato práce vznikla ve spolupráci s firmou, která se zabývá řešením kombinování letů. Cílem je vytvořit systém, který je schopen v reálném čase odpovídat na dotazy uživatelů (odkud, kam a kdy chtějí letět) nakombinovanou cestou spojenou z elementárních letů.

V kapitole 2 je více přiblížena problematika plánování cest společně s bližší specifikací zadání. Dále v kapitole 3 jsou popsány přístupy k řešení problému hledání nejkratší cesty ve statickém grafu (v silniční dopravě). Ačkoliv se nejedná o stejný problém, pochopení přístupů má veliký přínos při hledání nejkratší cesty ve složitějším grafu - v dynamickém (v hromadné dopravě). Současné přístupy k řešení plánování cest v hromadné dopravě jsou shrnuty v kapitole 4. Následuje kapitola 5 s technickou analýzou letového grafu a stanovení předpokladů pro výběr optimální metody. V kapitole 6 je popsán návrh komplexního systému, který je schopen plánovat cesty, společně s několika vylepšeními. Implementační detaily jsou představeny v kapitole 7. Práci uzavírá kapitola 8, kde jsou popsány experimenty, které vyhodnocují úspěšnost navržených optimalizací, stejně jako přínos celého systému oproti stávajícímu řešení.

Kapitola 2

Motivace a úvod do problematiky

Doprava jako taková tu byla již od dob, kdy si člověk sedl na koně a vyjel do vedlejší vesnice. Postupně se vyvíjela. Nejprve existovalo pouze pár typů dopravy a ta na větší vzdálenost byla dostupná pouze pro malé procento obyvatelstva. O nějakém systému či běžných spojích se taktéž nedalo mluvit. V poslední čtvrtině minulého tisíciletí (současně s průmyslovou revolucí) pravidelných spojů přibylo, ale stále jich bylo velice málo, takže cestující stál především o to, aby se do destinace vůbec dostal. Nároky byly malé. Ve 20. století dopravních možností na světě přibývalo, ale teprve s příchodem pokročilejších technologií a internetu se objevil pojem *plánování cest* (angl. *route planning*). Důvod byl jasný - s lepším dopravním pokrytím světa se zároveň zvyšovaly požadavky lidí na efektivní cestování. Bylo tedy zapotřebí mít možnost v takovém množství dat nalézt požadované informace.

Plánování cest může být rozděleno podle mnoha kritérií. Pro tuto práci je podstatné rozdělení na základě časové závislosti [5]:

- plánování v pozemní silniční dopravě: Je časově nezávislé, neexistují jízdní řády, podle kterých by se člověk musel řídit. Výsledné nejlepší cesty jsou tedy zpravidla nezávislé na tom, kdy se je člověk rozhodne uskutečnit. Jistou výjimkou může být cestování v noci či v zimě, ale v zásadě se přístup k plánování výrazně nemění.

- plánování v hromadné dopravě: Je časově závislé, podléhá existujícím jízdním řádům a pasažér má podstatně menší volnost - je plně závislý na vozidlech hromadné dopravy. Vzniklé problémy jako např. porucha vozidla, zácpa, aj., které mají za následek zpoždění vozidla mohou mít za následek zmeškání následujícího spoje a tím pádem úplnou změnu naplánovaného itineráře.

Letecká doprava se řadí do druhé sekce. Avšak je velmi rozdílná od ostatních typů hromadné dopravy. Mezi hlavní rozdíly patří:

- Plánování probíhá v rámci celého světa, ne pouze státu či města.

- Neexistují zde linky s více zastávkami. Pasažér nastoupí (= první zastávka) a po příletu letadlo ihned opustí (nebereme v potaz mezipřistání, z pohledu pasažéra to pro něj neznamená, že by mohl opustit let).

- Zastávek je neúměrně méně vzhledem k počtu hran v grafu.

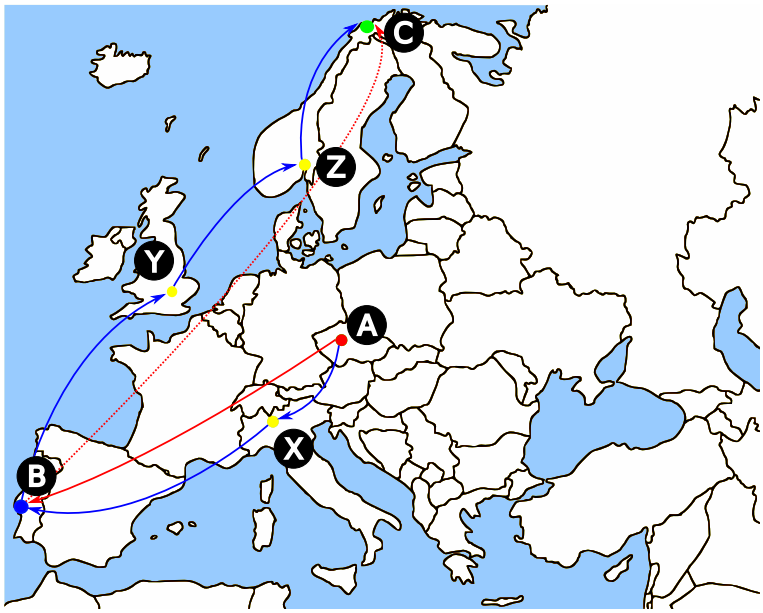
- Neexistuje jedna společnost, která by dopravu zaštiťovala, nikdo „nevlastní“ všechna data.

- Kratší cesta nemusí být levnější.
- ... a mnoho dalších, detailněji budou popsány v kapitolách 4 a 5.

2.1 Prostředí vývoje

Hlavní motivací této práce je pokus o zlepšení současného vyhledávacího systému jedné z firem, která se zabývá plánováním cest v letecké dopravě. Tato firma je jedna z mála, která se specializuje právě na kombinování letů různých nespolupracujících aerolinek s cílem nabízení lepších nabídek zákazníkům. Lepší nabídky se dají představit jako:

- levnější spojení - mezi dvěma destinacemi sice může existovat přímé spojení, ale levnější je to při kombinaci s jedním přestupem,
- spojení, které by jinak nevzniklo - mezi dvěma destinacemi žádná aerolinka nelétá, ale s nakombinováním přes jiné místo už je cesta reálná. Obě možnosti jsou znázorněny na obrázku 2.1.



Obrázek 2.1: Příklad dvou hlavních přínosů kombinování letů. První bod je znázorněn při plánování z Prahy (A) do Lisabonu (B). Přímý let sice existuje, ale levnější je zkombinovat dva různé lety s přestupem v Miláně (X): Praha - Milán a Milán - Lisabon. Druhý bod je ukázán na cestě z Lisabonu (B) do Tromsø (C). Na této lince žádná aerolinka neobsluhuje, ale využitím třech různých letů se přes Londýn (Y) a Oslo (Z) dá do cíle dostat.

Současné řešení spoléhá na značné předzpracování, díky němuž je schopno během okamžiku odpovědět i na velmi složité dotazy. Toto řešení s sebou nese i jistá úskalí, jako například velkou časovou a prostorovou náročnost. Úkolem této práce je řešení zlepšit. Je jasné, že každý přístup má své výhody a nevýhody, proto je potřeba si vytyčit kritéria, která náš systém bude muset splňovat a dále pak taková, která nebudou tolik podstatná a bude je možné opomenout.

Hlavními požadovanými kritérii v řešení budou:

- schopnost adaptace - data je potřeba aktualizovat častěji, předzpracování tedy nemůže trvat dlouho,
- rychlost - je nutné, aby program byl schopen na dotazy odpovídat v únosném čase,
- vysoká variabilita dotazu - uživatel by měl mít možnost si nastavit filtry výsledků co nejpodrobněji.

Výše zmíněná kritéria na první pohled působí jako výběr těch nejlepších vlastností, které by člověk po kombinátoru mohl chtít. Nyní bude zapotřebí najít takové vlastnosti systému, které buď obětujeme, nebo kde bude možno dojít k určitému kompromisu. Cílem následujících navržených změn bude udělat řešení co nejvíce „reálné“ z hlediska implementace při zachování kvality výsledků.

- Filtrování duplikovaných letů - reálná data posbíraná z různých zdrojů mohou obsahovat lety se stejnými nebo objektivně horšími vlastnostmi (vyšší cena, větší počet přestupů, aj.)
- Zmenšení časového rámce, ve kterém se dá vyhledávat - současný vyhledávač dovoluje vyhledávat lety téměř na rok dopředu. To s sebou ale nese větší potřebu operační paměti na uchování dat. Je ale možné využít faktu získaného ze statistik spolupracující firmy, že více než polovina dotazů na vyhledávač je právě na následující měsíc.
- Omezit složitost dotazů - můžeme předpokládat, že většina dotazů je jednoduchých.
- Aplikování vhodných heuristik - je možné zanedbat některé kombinace kvůli zrychlení algoritmu.

Kapitola 3

Plánování cest ve statickém grafu

V této kapitole budou blíže přiblíženy přístupy k řešení plánování cest ve statickém grafu. Jedná se především o automobilovou dopravu, ale lze problém vztáhnout i na plánování pěší cesty, apod. Důležitým faktem je, že graf je v čase neměnný. Nezáleží na tom, jestli se člověk rozhodne plánovanou cestu jet nyní nebo za týden. Výjimku mohou tvořit uzávěrky a objížďky, ale i tehdy se jedná o statický fakt (uzávěrka buď je, či není).

Tento problém se dá i dále dělit na detailnější variace. Nejjednodušším případem je prosté hledání cesty mezi dvěma uzly $(1 : 1)$. Dále existují složitější variace, kdy hledáme cestu z jednoho uzlu do všech nebo naopak $(1 : n)$. Nejsložitějším zadáním je hledání nejkratších cest mezi množinami, přesněji řečeno mezi každými dvěma uzly z obou množin $(m : n)$. Vzhledem k tomu, že i zadání hlavního problému práce bylo specifikováno jako problém $1 : 1$, v popisu algoritmů se zaměříme především na tuto variaci.

Tato kapitola bude sloužit pouze jako drobné přiblížení tohoto odvětví problematiky. Přestože cílový problém je podstatně jiného základu, lze zde najít inspiraci pro vylepšené algoritmy nebo použití zajímavých heuristik. Srovnání metod a zdůraznění možného využití pro hromadnou dopravu je popsáno v podkapitole 3.9.

3.1 Definice termínů

Plánování cest ve statickém grafu ve své základní formě představuje pouze hledání nejkratší cesty mezi dvěma uzly v neorientovaném grafu. Všechny algoritmy tedy mají stejné předpoklady. Uvažme graf $G = (V, E)$ kde V je konečná množina vrcholů a E množina hran mezi nimi. Každé hraně $(u, v) \in E$ je přiřazena nezáporná váha $w(u, v)$.

Cestu v grafu P lze definovat následovně:

$$P = (v_0, v_1, \ldots, v_n) \in V \times V \times \cdots \times V \wedge \forall i \in \{0, 1, \ldots, n-1\} : (v_i, v_{i+1}) \in E \tag{3.1}$$

A stejně tak její celkovou váhu $W(P)$.

$$W(P) = \sum_{i=0}^{n-1} w(v_i, v_{i+1}) \tag{3.2}$$

Zadáním problému je potom hledání cesty s minimální celkovou vahou z počátečního uzlu s do uzlu t mezi všemi cestami ψ, které vedou mezi těmito uzly, jak je vidět na rovnici 3.3.

$$dist(s,t) = min\{W(p)|p \in \psi \wedge v_0 = s \wedge v_n = t\} \tag{3.3}$$

3.2 Prohledávání do šířky

Prohledávání do šířky (angl. *breadth-first search* - BFS) je jedním ze základních způsobů procházení grafu a na rozdíl od prohledávání do hloubky (angl. *depth-first search* - DFS), které používá zásobník, je založeno na datové struktuře fronta. Pro hledání nejkratší cesty v ohodnoceném grafu je potřeba jej drobně upravit. V klasické variantě existuje množina *closed_set*, kam se vkládají všechny zpracované uzly a už se znovu neprocházejí. Tento detail zaručuje úplnost algoritmu, ale zároveň eliminuje možnost dostat se do stejného uzlu levnější cestou. Algoritmus totiž spoléhá na stejnou váhu všech hran a na fakt, že uzly expanduje postupně - cesta se tedy pořád monotónně zvětšuje.

Při hledání cesty z uzlu s do uzlu t by se dal algoritmus popsat následujícím postupem:

1. Umísti dvojici $(s, 0)$ do prázdné fronty a nastav *minprice* na ∞.
2. Pokud je fronta prázdná, skonči a vrať *minprice*, jinak jdi na bod 3.
3. Vytáhni první uzel u a příslušnou cenu p z fronty a
   (a) pokud $u == t$ a $p < minprice$, nastav $minprice = p$ a pokračuj bodem 2,
   (b) pokud $u == t$ a $p \geq minprice$, pokračuj bodem 2 (nelze nic zlepšit),
   (c) jinak pokračuj bodem 4.
4. Vezmi aktuální uzel u a pro všechny uzly v, do kterých vedou hrany z u vlož do fronty dvojici $(u, p + w(u, v))$. Dále se vrať na bod 2.

Tento algoritmus má v základní verzi při použití množiny *closed_set* časovou složitost lineární vzhledem k velikosti grafu $O(|V| + |E|)$. V naší pozměněné variaci (bez množiny zpracovaných uzlů) se však tato složitost zhoršuje na exponenciální a algoritmus se stává téměř nepoužitelným. Tvoří však základ pro další algoritmy.

3.3 Dijkstrův algoritmus

Nejznámějším algoritmem pro hledání nejkratší cesty v ohodnoceném grafu je Dijkstrův algoritmus [1, 5, 11, 16]. Podstatným rozdílem oproti BFS je výběr následujícího uzlu. Nebere se první ve frontě, ale ten s nejnižší aktuální cenou. Tento fakt umožňuje každý uzel projít právě jednou a jakmile se zpracuje cílový uzel t, algoritmus končí s korektním výsledkem.

Rychlost algoritmu závisí na použité datové struktuře při implementaci výběru uzlu s nejnižší cenou [5, 11]. Složitost samotného procházení uzlů je $O(|V|)$. V případě použitého jednoduchého pole pro ukládání průběžných cen uzlů se musí při každém výběru pole lineárně procházet, z čehož plyne složitost $O(|V|)$, tedy výsledná $O(|V|^2)$. Za předpokladu použití prioritní fronty implementované Fibonacciho haldou se tato složitost zlepší na $O(|V| \log |V| + |E|)$. Ovšem při aplikaci algoritmu s prioritní frontou na velké dopravní sítě člověk zjistí, že přínos prioritní fronty se zmenšuje, a to kvůli stále častějším případům nenalezení potřebných dat v paměti (angl. *cache miss*) [11].

Postup algoritmu je následující:

1. Vytvoř pole průběžných vzdáleností uzlů od počátečního uzlu s $dist[|V|]$ a nastav všechny hodnoty na inf, až na hodnotu u počátečního uzlu, tu nastav na 0. Dále vytvoř bitovou masku $visited[|V|]$ se všemi bity nastavenými na $false$.

2. Pokud má uzel u s nejnižší hodnotou $dist[u]$, pro který platí $visited == false$ hodnotu inf, skonči.

3. Vyber z pole $dist$ uzel u s nejnižší průběžnou vzdáleností $dist[u]$, pro který platí $visited[u] == false$ a označ ho jako aktuální.

4. Pokud $u == t$, vrať $dist[u]$ a skonči.

5. Jinak postupně pro každý uzel v, pro který existuje hrana z u (u, v), porovnej hodnoty $dist[u] + w(u, v)$ a $dist[v]$ a do $dist[v]$ ulož tu menší z nich.

6. Nakonec nastav $visited[u]$ na $true$ a jdi na krok 2.

3.4 Obousměrné vyhledávání

Obousměrné vyhledávání (angl. *bidirectional search*) je spíše optimalizační technika než samostatný algoritmus [5, 11, 16]. Jeho hlavní myšlenkou je střídavé prohledávání hran jak z počátečního, tak z koncového uzlu. Jakmile je jednou stranou nalezen vrchol, který byl zpracován druhou stranou, algoritmus končí s nalezenou cestou (spojení obou částí).

3.5 A*

Předchozí algoritmy byly bez jakékoli snahy o predikci umístění cílového uzlu. Algoritmus A* patří do kategorie hladových (angl. *greedy*) algoritmů, které se pokouší určit směr, kudy by se mělo samotné zpracovávání uzlů ubírat [16].

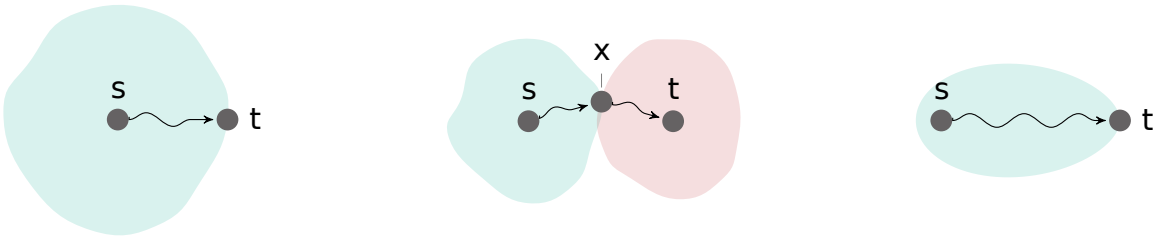
Nejdůležitější částí algoritmu je zvolit vhodnou heuristickou funkci $\pi : V \to \mathbb{R}$, která bude určovat spodní odhad ceny cesty z uzlu u do koncového uzlu v. Za předpokladu, že funkce π je konzistentní (splňuje nerovnost $w(u, v) \geq \pi(u) - \pi(v)$), je zaručeno, že v grafu nebudou záporné hrany. Z toho plyne, že je možno jako základ použít klasický Dijkstrův algoritmus (viz sekce 3.3) s tím rozdílem, že cena $f(u)$ uzlu u se určuje jako součet dosavadní provedené cesty a spodní odhad získaný danou funkcí, jak je vidět na rovnici 3.4.

$$f(u) = g(u) + \pi(u) \tag{3.4}$$

Jako funkce π bývá v dopravě nejčastěji volena právě Eukleidovská vzdálenost, což je nejkratší vzdálenost mezi dvěma body (vzdušná vzdálenost), neboť je to ta teoreticky nejkratší. Srovnání prohledávaného prostoru v případě A* a dvou předchozích metod je vidět na obrázku 3.1.

3.5.1 ALT

A* se dá vylepšit za předpokladu, že v grafu platí trojúhelníková nerovnost. Zároveň je potřeba ve fázi předzpracování určit malou množinu uzlů $L \subset V$ a pro každý z nich se spočítá vzdálenost mezi ním a každým uzlem z V. Tato vylepšená metoda, která spojuje A*, význačné body a trojúhelníkovou nerovnost se nazývá ALT (z angl. *A*, landmarks, triangle inequality*) [16].



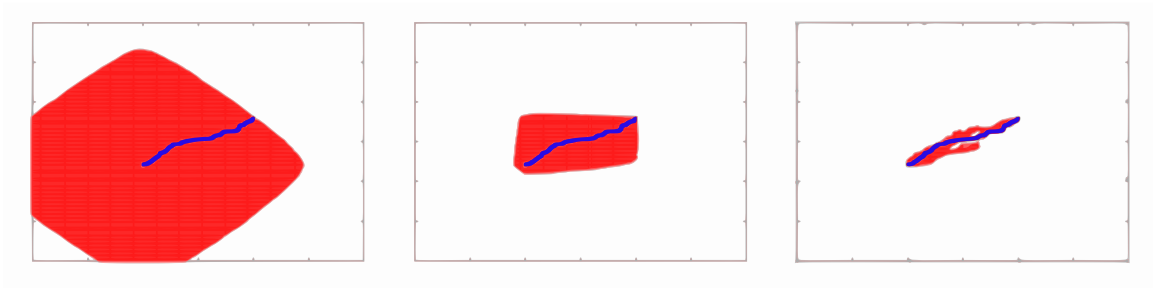
Obrázek 3.1: Srovnání prohledávaného prostoru při použití klasického Dijkstrova algoritmu (vlevo), obousměrného prohledávání (uprostřed) a A* (vpravo). Převzato z [5].

Pro každou dvojici uzlů (s, t) a každý bod $l_i \in L$ musí platit následující nerovnosti:

$$dist(s,t) \geq dist(s,l_i) - dist(t,l_i) \tag{3.5}$$

$$dist(s,t) \geq dist(l_i,t) - dist(l_i,s) \tag{3.6}$$

Tímto získáváme validní spodní odhad ceny cesty (s, t), který drasticky zmenšuje proskenovaný prostor při hledání nejkratší cesty, jak je vidět na obrázku 3.2. Pokud je použito více bodů l_i, je možné vzít maximum ze všech spodních odhadů. V grafech silniční dopravy bylo zjištěno, že nejlepší výsledky dávájí ty body, které se nacházejí právě kolem hranic grafu [17].



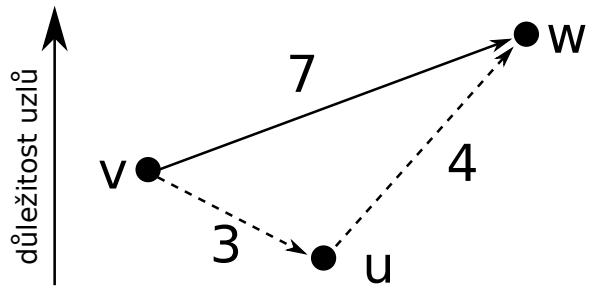
Obrázek 3.2: Srovnání prohledávaného prostoru při použití klasického Dijkstrova algoritmu (vlevo), A* (uprostřed) a ALT (vpravo). Převzato z [16].

3.6 Hierarchické přístupy

Hierarchické přístupy využívají hierarchických vlastností silničních sítí. Když se člověk rozhodne jet z místa A do místa B, s ubíhající cestou je očekávatelné, že úroveň (důležitost) hran se bude zvětšovat až do chvíle, kdy zůstane konstatní (dálnice). S blížícím se koncem cesty se bude úroveň naopak analogicky zmenšovat. Algoritmy opět používají předzpracování, které probíhá zpravidla ve dvou úrovních [11]. V první fázi se uzly rozdělí do úrovní podle důležitosti. V druhé fázi se zkoumají přestupní hrany mezi jednotlivými úrovněmi.

Algoritmus *Contaction Hierarchies* spadá do této kategorie [5, 11, 15]. V první fázi předzpracování se vytvoří $|V|$ úrovní, tedy jedna pro každý uzel. Dále se uzly seřadí podle důležitosti, podle které se postupuje při kontrakci. Důležitost je určena několika faktory, z nichž nejdůležitější je tzv. *edge difference*, tedy rozdíl v počtu hran, pokud se daný uzel vyjme. Dále je nutné uzly vybírat rovnoměrně napříč celým grafem, aby výsledné vyhledávání bylo co nejefektivnější. Následující kontrakce spočívá v dočasném vyjmutí nejméně

důležitého uzlu u z grafu a v případě, že nejkratší cesta vedoucí mezi dvěma sousedními uzly v a w uzlu u vedla právě přes u, je vytvořena zkratka (v, w) (nová hrana). Příklad je vidět na obrázku 3.3.



Obrázek 3.3: Ukázka operace kontrakce. Čárkovaně jsou znázorněny původní hrany v grafu; je vidět, že nejkratší cesta z v do w vede právě přes u. Jako první se vybere uzel u, kvůli nejnižší důležitosti a dvě hrany jsou nahrazeny jednou zkratkou.

Operací kontrakce se podoba grafu co se týče uzlů nezmění (finální graf se sestává z původního a zkratek), tudíž výsledné vyhledávání je možno provést základním Dijkstrovým algoritmem nad pozměněným grafem.

3.7 Štítkovací algoritmy

Hlavní myšlenkou algoritmů založených na označování (angl. *labeling*) je vytváření virtuálních zkratek [1, 5]. V rámci fáze předzpracování se pomocí vzdáleností mezi jednotlivými uzly vytvoří nová vrstva vodítek, podle kterých se už skutečně vyhledává. Nejjednodušším příkladem může být předpočítání všech nejkratších vzdáleností mezi každými dvěma uzly a uložení do tabulky. Každý další dotaz už má konstantní složitost. Toto ovšem připadá v úvahu pouze v těch nejmenších grafech.

Algoritmus *Hub labeling* je založen na vytvoření štítku (angl. *label*) $L(u)$ pro každý uzel u a předpokladu, že pro získání nejkratší cesty $dist(s, t)$ stačí pouze štítky $L(s)$ a $L(t)$. Každý štítek $L(u)$ obsahuje seznam *hub* uzlů a jejich vzdálenosti od u. Množina *hub* uzlů H musí být vybrána tak, aby v průniku mezi štítky každých dvou uzlů byl alespoň jeden *hub*, viz rovnice 3.7.

$$\forall s, t \in V : L(s) \subset H \wedge L(t) \subset H \wedge L(s) \cap L(t) \neq \emptyset \tag{3.7}$$

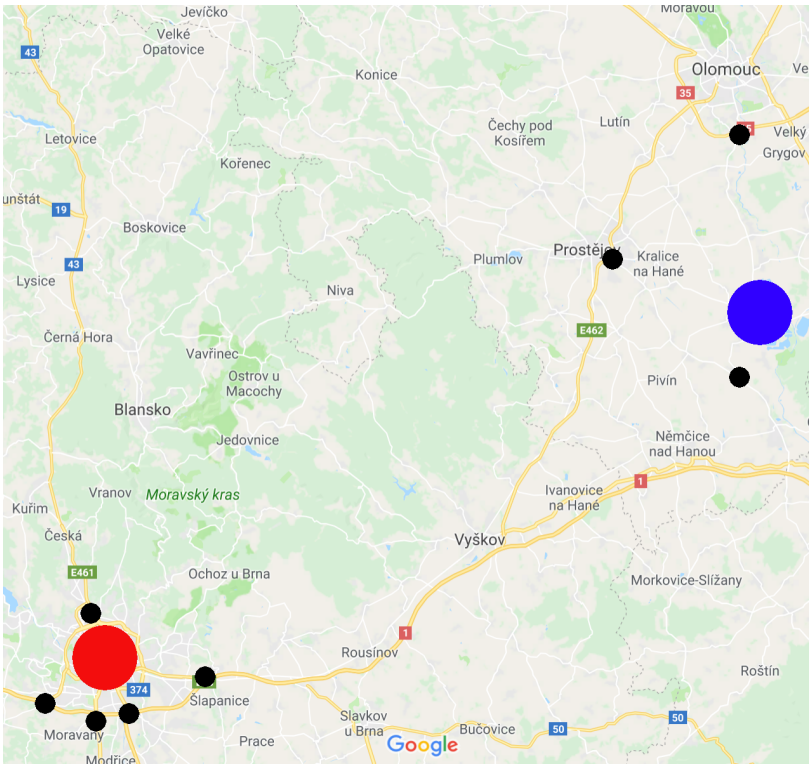
Výsledná nejkratší cesta se potom určí nalezením minima v průniku dvou štítku, jak je vidět na rovnici 3.8.

$$dist(s, t) = min\{dist(s, u) + dist(u, t) | u \in L(s) \wedge u \in L(t)\} \tag{3.8}$$

3.8 Metoda přestupních bodů

Podobnou metodou využívající jistým způsobem hierarchii dopravních sítí je metoda přestupních bodů (angl. *transit node routing*) [6, 7]. Zavádí pojem přestupního bodu (angl. *transit node*), který představuje místo, přes které člověk s velkou pravděpodobností na cestě ze startu do cíle projede. Přestupní body pro cestu z velkého města do dostatečně vzdáleného cíle si lze představit např. jako jednotlivé nájezdy na dálnici.

Algoritmus dělí dotazy na lokální a nelokální, což spočívá v určení Eukleidovské vzdálenosti mezi startem a cílem a porovnání s referenčním prahem. Tento práh je předem empiricky určen. V případě datasetu sestávajícího ze silniční sítě USA byl poměr mezi lokálními a nelokálními téměř 1:99 [6]. Vzhledem k tomu, že lokální dotazy nesplňují předpoklad přestupních bodů a vzdálenost mezi uzly je velmi malá, autoři uvádějí, že je lze vyhodnocovat kterýmkoli algoritmem pro nejkratší cestu. Kvůli tomuto faktu i kvůli poměru mezi dotazy se proto dále zaměříme na zpracování nelokálních dotazů.



Obrázek 3.4: Na mapě lze vidět podklady pro dotaz na cestu z Brna (červený kruh) do Tovačova (modrý kruh). Černé menší kruhy představují jejich nejbližší přestupní uzly. Zde je vidět například to, že z Brna ven člověk určitě pojede přes jeden z jeho výjezdů na větší silnice/dálnice. Dále je vidět, že dotaz si vyžaduje pouze 5 * 3 * 3 přístupů do paměti.

Náročné předzpracování předchází stavu, kdy je systém připraven odpovídat na dotazy. V prvním kroku se nalezne co nejmenší množina přestupních bodů, přičemž tato množina je definována následovně: Pro každou dvojici uzlů z nelokálního dotazu musí nejkratší cesta mezi těmito dvěma uzly vést přes alespoň jeden bod v této množině. Dále se pro každý uzel v grafu (který není sám přestupním bodem) zjistí co nejmenší množina nejbližších přestupních bodů (angl. *closest transit nodes*), tak, aby platilo, že každá nejkratší cesta vedoucí z tohoto uzlu prochází přes alespoň jeden bod v této množině. Příklad přestupních bodů je vidět na obrázku 3.4.

Poté se vytvoří dvě tabulky. První obsahuje předpočítané vzdálenosti mezi všemi přestupními body navzájem a v druhé jsou vzdálenosti z každého uzlu do všech uzlů v jeho množině nejbližších přestupních bodů.

Vyhodnocení nejkratší cesty mezi startem *src* a cílem *dst* je pouze hledání minima mezi předpočítanými hodnotami. Pro všechny páry vytvořené mezi nejbližšími přestupními body startu a cíle se zjistí cena cesty přes právě tyto body (všechny vzdálenosti jsou známy: start

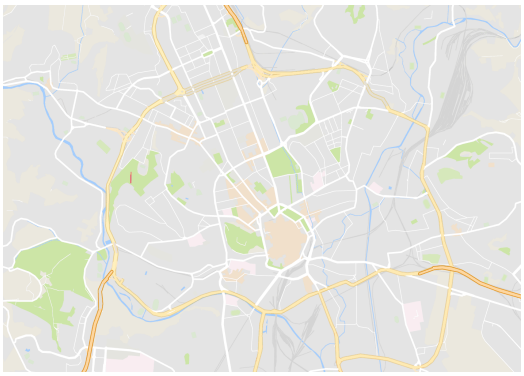
- blízký přestupní bod startu - blízký přestupní bod cíle - cíl). Tím se vytvoří množina (viz zápis 3.9), ve které se nalezne minimum.

$$\{dist(src, t_{src}) + dist(t_{src}, t_{dst}) + dist(t_{dst}, dst) | t_{src} \in T_{src} \wedge t_{dst} \in T_{dst}\} \tag{3.9}$$

3.9 Využití pro plánování v hromadné dopravě

Výše byly popsány různé metody od nejobecnějších (Dijkstrův algoritmus) až po velmi konkrétně zaměřené na pozemní dopravu (hierarchické přístupy). Je samozřejmé, že více specifické algoritmy optimalizované pro dopravní síť jsou výkonnější než právě ty obecné, nicméně i ty působí jako základ pro některé algoritmy pro hledání cest v rámci hromadné dopravy. Základní metody pro hledání cest v hromadné dopravě nejdříve změní časově závislý (dynamický) graf na statický a poté aplikují Dijkstrův algoritmus (popsáno dále v kapitole 4.4). Vzhledem k podobnosti základu Dijkstrova algoritmu a A* je jasné, že A* lze taktéž aplikovat na dynamický graf (stejně tak ALT).

Zajímavější je však úvaha nad těmi specifickými. Při použití hierarchických metod u hromadné dopravy se musí brát v potaz, že tato metoda je založena právě na přítomnosti striktní hierarchie. Když porovnáme silniční síť v Brně na obrázku 3.5 se sítí hromadné dopravy na obrázku 3.6, zjistíme, že úroveň hierarchie je velmi rozdílná. Zatímco v silniční síti jsou již na první pohled vidět jednotlivé úrovně silnic (městský okruh, hlavní spojovací silnice, dálnice), u hromadné dopravy je pouze slabý náznak, který ani nemusí znamenat, že se skutečně jedná o lepší cestu - úsek s více linkami nemusí být nutně rychlejší, důležitější při hledání cesty je spíše to, jak dobře na sebe spojení navazují. Vzhledem k tomu, že hlavní zrychlení algoritmu závisí na tom, jaká je vzdálenost od další úrovně hierarchie, je v hromadné dopravě tato metoda téměř nepoužitelná [3].



Obrázek 3.5: Dopravní síť města Brna[1].

Obrázek 3.6: Síť hromadné dopravy v Brně[2].

Metoda přestupních bodů, která je při použití v silniční dopravě extrémně rychlá, je teoreticky použitelná i pro hromadnou dopravu. I v hromadné dopravě ve městě lze nalézt velmi malou množinu přestupních bodů, které splňují potřebné vlastnosti (podrobněji viz kapitola 3.8). Problém ovšem nastává při výpočtu potřebných vzdáleností, který už je podstatně složitější než u dopravních sítí (kvůli časovému faktoru a případně i více kritériím). V posledních letech se však objevil nový přístup, který tento problém překonává a bude detailněji popsán v kapitole 4.8.

[1] zdroj: https://www.google.cz/maps
[2] zdroj: http://www.dpmb.cz

Bylo naznačeno, že v případě, že některý z algoritmů pro statický graf lze použít pro dynamický, je potřeba vždy udělat nějakou změnu navíc. Dalo by se říct, že hledání cesty v dynamickém grafu je ve všech ohledech těžší než hledání cesty ve statickém. Z tohoto faktu plyne, že analýza statických algoritmů je často velmi inspirativní a vede k lepšímu porozumění při řešení problému plánování v hromadné dopravě.

Kapitola 4

Plánování cest v hromadné dopravě

V této kapitole blíže přiblížíme problém plánování cest v hromadné dopravě. Nejdříve definujeme termíny používané v problematice a specifikujeme zadání řešených problémů. Dále budou popsány možné přístupy pro práci s časově závislým grafem a konkrétní algoritmy pro plánování cest. V závěru určíme předpoklady, které by graf spojení musel splňovat pro použití jednotlivých algoritmů.

4.1 Definice termínů v problematice

Pro správnou orientaci v problematice je třeba definovat správné názvosloví. Nejdříve budou definovány prvky, které se v dopravních sítích nacházejí a se kterými algoritmy pracují [5].

Zastávka (*stop*) představuje místo zastavení dopravního prostředku, kde člověk může nastoupit či vystoupit. Zastávka p může mít definovaný minimální čas na přestup $t_{ch}(p)$ [12]. Lze si je představit jako autobusové zastávky, železniční stanice nebo letiště.

Spojení (*connection*) je elementární spojovací část sítě. Spojení c je definováno počáteční zastávkou $p_{dep}(c)$ s časem odjezdu $t_{dep}(c)$ a koncovou zastávkou $p_{arr}(c)$ s časem příjezdu $t_{arr}(c)$.

Pěší přestup (*footpath*) je pěší zkratka mezi dvěma zastávkami. Definuje ji dvojice zastávek a délka chůze. Všechny pěšiny v síti jsou tranzitivně uzavřené. Kvůli tomuto faktu nemohou být ve výsledné trase dva po sobě jdoucí pěší přestupy.

Spoj (*trip*) je posloupnost několika spojení, která na sebe navazují a lze je projet v právě jednom vozidle.

Linka (*route*) je seskupení spojů obsluhujících stejné zastávky bez ohledu na příjezdové a odjezdové časy.

Trasa (*journey*) je výsledná cesta j, kterou hledáme a která je definována posloupností spojení. Aby na sebe mohly dvě spojení c_a a c_b, které nejsou součástí stejného spoje navazovat, minimální čas přestupu v $p_{arr}(c_a)$ musí být menší než rozdíl odjezdového a příjezdového času v dané zastávce: $t_{ch}(p_{arr}(c_a)) < t_{dep}(c_b) - t_{arr}(c_a)$.

Tyto pojmy jsou společné pro všechny typy hromadné dopravy, avšak letecká doprava některé z nich razantně mění. Největší změny se týkají právě *spoje* a *spojení*, které v podstatě splývají v jedno. V letecké dopravě totiž neexistují lety, kde by si člověk mohl vybrat, jestli z letadla na letišti vystoupí, nebo jestli zůstane a poletí dál. Výjimkou se mohou zdát mezipřistání, ale tam člověk nemá možnost získat nazpět své zavazadlo a zadruhé není zaručené, že to bude člověku umožněno. Dalším rozdílem jsou vlastnosti spojení. Kde při hromadné dopravě ve městech nehraje roli cena (zpravidla bývá řešena formou tarifních pásem), v letecké dopravě je to mnohdy to nejdůležitější kritérium při plánování výsledné cesty.

4.2 Definice problémů

V plánování cesty ve statické pozemní automobilové síti je formulace problému jednoduchá - nalezení nejkratší/nejrychlejší cesty z A do B. Vzhledem k variabilitě a časové závislosti plánování v hromadné dopravě mají řešené problémy jiné formulace [5, 12, 14, 9].

Nejjednodušším případem je takzvaný problém nejdřívějšího příjezdu (angl. *earliest arrival problem*). Zadáním tohoto problému je počáteční zastávka p_{dep}, příjezdová zastávka p_{arr} a čas t. Cílem je najít takovou trasu j, která odjíždí ze zastávky p_{dep} nejdříve v čase t a přijíždí do zastávky p_{arr} nejdříve jak je to možné, viz rovnice 4.1.

$$journey = min_{t_{arr}}\{j | p_{dep}(j) = p_{dep} \wedge p_{arr}(j) = p_{arr} \wedge t_{dep}(j) \geq t\} \tag{4.1}$$

Rozšířením je potom intervalový problém (angl. *range/profile problem*). Zadání, co se týče zastávek, zůstává stejné: vstupem jsou počáteční zastávka p_{dep} a příjezdová zastávka p_{arr}. Avšak namísto jednoho času t je vstupem celý časový interval t_r. Cílem je poté pro každou jednotku času t z t_r najít cestu s nejdřívějším příjezdem. Rozdělení časového intervalu lze brát jako skutečné dělení po určitých úsecích, nebo to lze přetransformovat na „pro každý odjezd“ z daného intervalu. Výsledná množina cest je definována v rovnici 4.2.

$$journeys = \{min_{t_{arr}}\{j | p_{dep}(j) = p_{dep} \wedge p_{arr}(j) = p_{arr} \wedge t_{dep}(j) = t\} | t \in t_r\} \tag{4.2}$$

Vzhledem k tomu, že v hromadné dopravě nejde vždy pouze jen o délku jízdy (přicházejí v úvahu počet přestupů a cena, ale je možné sem zahrnout i další kritéria - například dopravce, třída vozu, apod.), objevuje se další problém - problém více kritérií (angl. *multicriteria problem*). V rámci definice problému je potřeba určit optimalizační kritéria, která jsou pro nás podstatná a která se nemohou vzájemně dominovat (např. jet dobu t za cenu c, nebo dobu $2t$ za cenu $c/2$?). Výsledkem řešení problému je pak Pareto množina (viz část 4.2.1) cest.

Z přehledu problémů je jasné, že v sítích hromadné dopravy je nejběžnějším problémem právě ten poslední, problém více kritérií.

4.2.1 Pareto optimalita

Pareto optimalita (efektivita) je pojem, který pochází z odvětví teorie her [13]. Definujme si hru $\Gamma = (Q, S, U)$, kde Q je množina všech hráčů, S je množina množin jejich strategií $S = \{S_i\}_{\forall i \in Q}$ a U je množina jejich užitkových funkcí $U = \{U_i\}_{\forall i \in Q}$. Užitková funkce má tvar $U_i(s) = u$, kde $s \in P$ reprezentuje strategický profil všech hráčů - tedy množinu aktuálně hraných strategií (strategických profilů P pak může být až 2^Q).

Mějme dva strategické profily $s \in P$ a $s' \in P$. Profil s pareto dominuje nad profilem s', jestliže platí:

$$\forall i \in Q : U_i(s) \geq U_i(s') \tag{4.3}$$

Strategický profil $s \in P$ je potom pareto optimální, pokud neexistuje žádný jiný profil $s' \in P$, který by ho dominoval. Jinými slovy užitek plynoucí z daného profilu s musí být pro každého hráče větší nebo stejný, než by byl za použití profilu s'.

Transformace na cesty v hromadné dopravě je následující: strategické profily představují jednotlivé cesty, hráči představují kritéria a užitky hráčů jsou konkrétní hodnoty jednotlivých kritérií výsledných cest. Tím pádem jedna cesta dominuje druhou, pouze pokud je lepší v každém kritériu.

Q_0/Q_1	S_{00}	S_{01}
S_{10}	1;3	**3;1**
S_{11}	**1;2**	2;10

Tabulka 4.1: Ukázka výsledků hry mezi dvěma hráči, kdy každý má dvě strategie. Užitková funkce je nyní převrácená, takže čím nižší hodnoty, tím lépe.

cena;trvání	–	–
–	€1;3h	**€3;1h**
–	**€1;2h**	€2;10h

Tabulka 4.2: Transformace hry na situaci při porovnávání cest v dopravě.

Na tabulkách výše lze vidět příklad transformace hry na porovnávání cest. Co jsou konkrétní strategie není podstatné, jde nám pouze o možnost získat výslednou množinu profilů (cest), které jsou pareto optimální.

4.3 Modelování jízdního řádu

Vzhledem k dalšímu rozměru hran (časové závislosti) je potřeba náležitým způsobem reflektovat tento fakt, pokud s nimi člověk chce efektivně pracovat a používat klasické grafové metody. V minulosti se ustálily dva hlavní přístupy k modelování časově závislého jízdního řádu dopravních spojení, časově rozvíjené a časově závislé modelování, které budou v následujících sekcích popsány [5].

Jako příklad uveďme jízdní řád o třech vlacích a pěti zastávkách. Minimální čas na přestup v zastávce je stanoven na 5 minut. Názorně budou ukázány oba výsledné grafy.

zastávka		čas
A	odj.	8:05
B	příj.	9:55
	odj.	10:02
C	příj.	11:57
	odj.	12:00
D	přij.	13:20

(a) vlak č. 1

zastávka		čas
C	odj.	12:00
E	přij.	13:00

(b) vlak č. 2

zastávka		čas
C	odj.	13:00
E	přij.	14:00

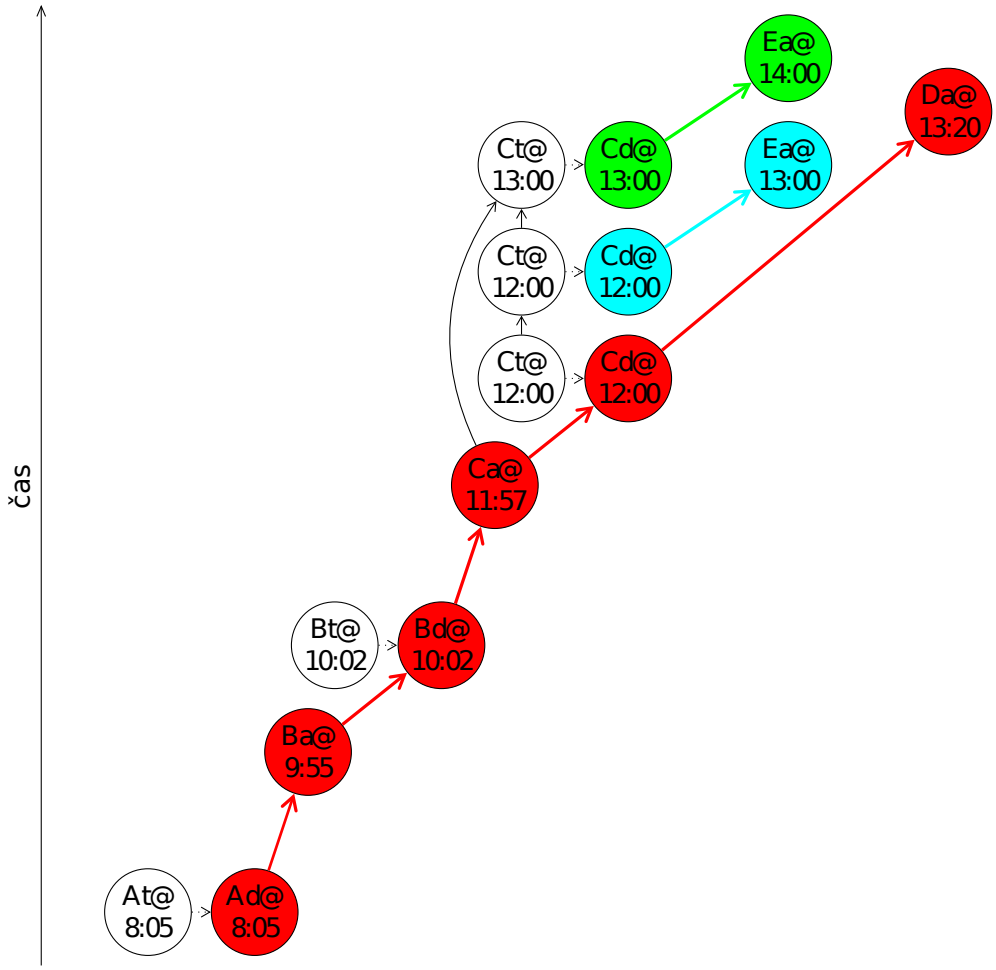
(c) vlak č. 3

Tabulka 4.3: Příklad jízdního řádu třech vlaků.

4.3.1 Časově rozvíjené modelování

Tento typ modelování grafu (angl. *time-expanded*) vytváří pro každou událost v jízdním řádu uzel s časem [5]. Typy uzlů jsou: odjezd vozidla ze zastávky, příjezd vozidla na zastávku a přestup na zastávce. Každé dvě události, které na sebe mohou navazovat (jak časově, tak pozičně), jsou spojeny hranou.

Každá dvojice *(příjezd,odjezd)* ve stejné zastávce, která je součástí jednoho spoje, je hranami spojená implicitně. Pro každý příjezd se dále uzel pokusí spojit se všemi přestupovými uzly pro danou příjezdovou zastávku. Zde je možné zavést realistický detail, že lze propojit pouze ty uzly, mezi kterými je časový rozdíl větší, než potřebný čas pro přestup na zastávce. Všechny přestupové uzly jsou propojeny s dalšími přestupovými uzly na též zastávce, ovšem pouze s těmi, které mají čas větší nebo rovno času daného uzlu.



Obrázek 4.1: Ukázka časově rozvíjeného typu modelování grafu. Jednotlivé vlakové spoje jsou znázorněny barvami. Notace má tvar: <uzel><typ události(**t**ransfer, **d**eparture, **a**rrival)>@<čas>. Například u uzlu *Ca@11:57* je vidět, že 1) je implicitně spojen s uzlem *Cd@12:00*, protože je součástí stejného spoje, 2) není spojen s prvním uzlem *Ct@12:00*, protože je rovnou spojen s *Cd@12:00*, viz 1), 3) nemůže být spojen s uzlem *Ct@12:00*, protože mezi nimi není alespoň 5 minut a 4) je spojen s uzlem *Ct@13:00* jakožto jediným validním přestupovým uzlem, který je možné stihnout.

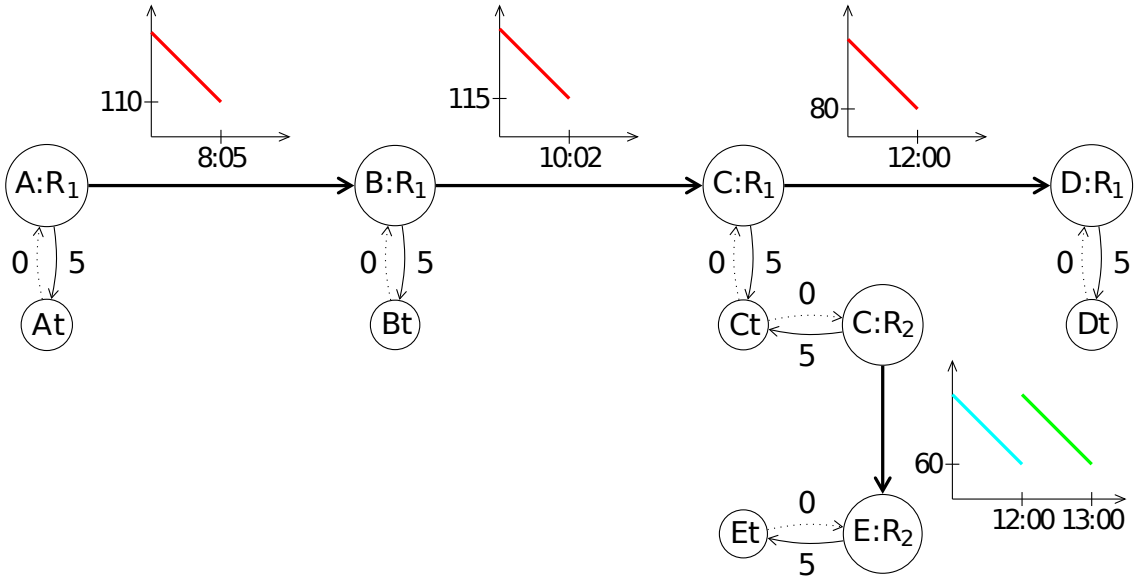
4.3.2 Časově závislé modelování

Velkou nevýhodou předchozího přístupu je obrovská velikost výsledného grafu. Proto byl zaveden nový způsob modelování, časově závislé (angl. *time-dependent*), které taktéž zohledňuje časovou závislost [5, 8].

Nyní se spoje a spojení agregují do obecných linek (zastávky, které je možno projet bez vysednutí z vozidla) a pro každou zastávku z linky je vytvořen uzel reprezentující odjezd i příjezd. Tedy pro zastávku, která je obsluhována více linkami je také vytvořeno více uzlů. Dále pro každou zastávku, nehledě na to, kolika linkami je obsluhována, je vytvořen jeden přestupní uzel, který modeluje čas potřebný na přestup v zastávce. Hrana vedoucí do něj je ohodnocena časem potřebným na přestup v dané zastávce.

Hlavní změna oproti klasickému modelování přichází v určení ceny hrany mezi dvěma uzly zastávek. Každá hrana (u, v) má funkci času na cestě (angl. *travel-time function*) $\tau_{(u,v)}$ určující čas, za jak dlouho se člověk může dostat z u do v v závislosti na čase. Funkce má skokový charakter, má spojité pouze částečné intervaly, na kterých je monotónně klesající se směrnicí $k = -1$. Spojité intervaly symbolizují čekací dobu na zastávce, v čase odjezdu $t_{dep}(c)$ každého spojení c pak funkce vrací délku samotného spojení, viz rovnice 4.4. Konkrétní příklad modelování jízdního řádu z tabulky 4.3 je vidět na obrázku 4.2.

$$\tau_{(p_{dep}(c),p_{arr}(c))}(t_{dep}(c)) = t_{arr}(c) - t_{dep}(c) \tag{4.4}$$



Obrázek 4.2: Příklad modelování časově závislého typu. Hrany do přestupních uzlů mají cenu 5, což představuje pět minut potřených na přestup. Dále je vidět, že nehledě na to, že jízdní řád obsahoval tři vlaky, spojení se zagregovala do pouze dvou linek (R_1: A-B-C-D a R_2: C-E). Na hraně C-E je vidět, že funkce času obsahuje dva spojité intervaly, které představují dvě spojení, které jsou na dané trase. V bodě 12:00 vrací délku spojení (60), ale o minutu později už první spoj ujel, funkce tedy vrací čekací dobu + délku následujícího spojení (59 + 60). Spojité intervaly v časových funkcích jsou zvýrazněny barvou daného vlaku (stejnou jako v předchozí kapitole). Inspirováno [14].

4.4 Variace Dijkstrova algoritmu

Pro nejjednodušší řešení všech problémů zmíněných v sekci 4.2 je možné použít variace Dijkstrova algoritmu popsaného v sekci 3.3.

Pro řešení základního problému nejdřívějšího příjezdu je možné spustit klasický Dijkstrův algoritmus nad grafem, pokud se jedná o *time-expanded* model [5, 18, 9]. Jako výchozí uzel se zvolí kombinace výchozí zastávky a času a algoritmus se zastaví jakmile narazí na uzel, který patří k cílové zastávce. Pokud jde o *time-dependent* model, je nutné udělat jedinou modifikaci. Při dotazu na $dist(s, t, time)$, když algoritmus skenuje hranu (u, v), je potřeba evaluovat její hodnotu s parametrem $time + dist(s, u)$. Algoritmus končí, jakmile se projde cílová zastávka.

Intervalový problém lze taktéž řešit modifikovaným Dijkstrovým algoritmem, ale již je potřeba, aby šlo o *time-dependent* model [5]. Namísto jedné hodnoty si algoritmus v každém uzlu pamatuje celou funkci, která mapuje všechny časy odjezdů z výchozí zastávky na čas, kdy je možné se do dané zastávky dostat nejdříve. Pokaždé, když se zpracovává hrana (u, v), tak se napojí funkce τ_uv uzlu u s funkcí $\tau_{(u,v)}$ hrany (u, v), čímž se získá mapovací funkce pro cestu z s do v přes u. Ta se následně sloučí s funkcí τ_v uzlu v, přičemž se pro každý odjezdový čas vezme minimum. Vzhledem k tomu, že funkce τ již nejde objektivně porovnávat, pokaždé, když se funkce nějakého uzlu vylepší, je znovu umístěn do fronty na zpracování (neuchovává se znak zpracování uzlu). Tento algoritmus se anglicky nazývá *Time Dijkstra*.

Pokud v problému více kritérií optimalizujeme kromě příjezdového času pouze jedno další kritérium, které je navíc diskrétní, lze použít další variaci tohoto algoritmu. Toto rozšíření, které používá *time-dependent* modelování grafu, se anglicky nazývá *Layered Dijkstra* [8]. Předpokládejme, že druhým kritériem je počet přestupů, pro které existuje horní hranice K. Před samotným procházením grafu se vytvoří K kopií grafu, které představují jednotlivé úrovně a přestupní uzly se propojí s odpovídajícími přestupními uzly ve vyšší úrovni. Dále se nad nově propojeným grafem spustí *Time Dijkstra* začínající na první úrovni. Abychom brali v potaz dominanci týkající se druhého kritéria, lze uvést, že štítky na nižších úrovních automaticky dominují ty ve vyšších (avšak pouze pokud je splněna dominance prvního kritéria).

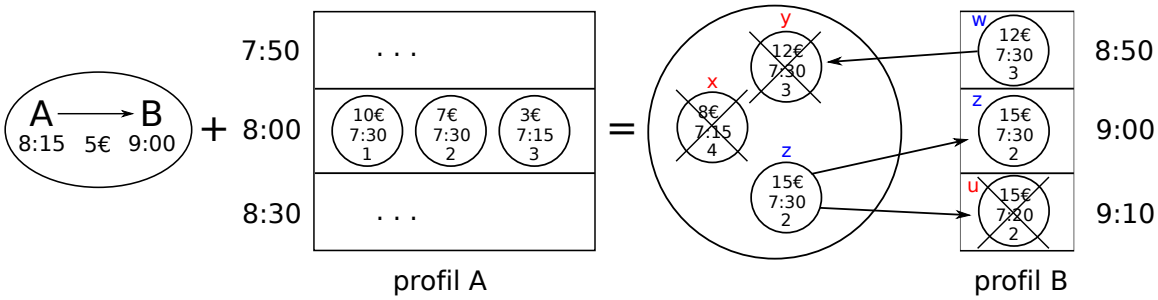
4.5 Algoritmus CSA

Vzhledem k tomu, že grafy hromadné dopravy velmi rostou ve velikosti, konvenční grafově založená řešení přestávají stačit. Proto byl vymyšlen *Connection Scan Algorithm* (CSA), který nevytváří žádný graf, ale pracuje přímo se seznamem základních spojení [12]. Jediným předzpracováním, které se vykonává, je seřazení seznamu spojení podle odjezdového času. Při zpracování dotazu se pole prochází maximálně jednou (ne nutně celé), díky čemuž se docílí skvělé cache lokality při čtení hran.

Pro první problém a zadání spočívající v zastávkách p_{dep}, p_{arr} a odjezdovém čase t se postupuje následovně. Každá zastávka si uchovává hodnotu τ s časem nejdřívějšího příjezdu do zastávky (u výchozí zastávky je to nastaveno na t). Algoritmus potom prochází seřazené pole a u každého spojení c zjišťuje, zdali ho může být dosaženo, tedy jestli platí $t_{dep}(c) \geq \tau(p_{dep}(c))$. Pokud ano, porovnají se hodnoty $\tau(p_{arr}(c))$ a $t_{arr}(c)$ a do $\tau(p_{arr}(c))$ se uloží ta menší. Po projití celého pole je ve všech štítcích čas nejdřívějšího příjezdu do dané zastávky. Případně si lze pamatovat nejdřívější příjezd u cílové zastávky a jakmile se načte spojení s odjezdovým časem větším, může se skončit.

Řešení zbylých dvou problémů je natolik provázané, že bude popsáno společně. Příjezdový čas se dá totiž označit jen jako další optimalizační kritérium. Každá zastávka si teď uchovává profil - seznam dvojic (t_{arr}, bag), kde bag obsahuje seznam vzájemně se nedominujících štítků a t_{arr} představuje čas, jaký nejmenší může být příjezdový čas $t_{arr}(c)$ určitého spojení c, aby se dalo napojit na tyto štítky. Každý štítek se skládá z kombinace optimalizačních kritérií takových, aby bylo možné tento štítek prodloužit o libovolné spojení (například cena, počet přestupů, ...).

Ve chvíli, kdy se skenuje spojení c, se v profilu zastávky $p_{dep}(c)$ nalezne dvojice $(time, bag)$ s takovým nejvyšším možným časem, aby platilo $time \geq t_{dep}(c)$. Tím se získá množina štítků, na které lze navázat daným spojením a zároveň mají nejkratší čekací čas. Spojení c se pokusí napojit na všechny štítky dané množiny a tím se vytvoří nový bag. Při každém prodlužování stávajících spojení se musí sledovat, zdali se nepřesáhl nějaký limit (např. počet přestupů). Nyní přichází na řadu profil zastávky $p_{arr}(c)$, ze kterého se vyberou všechny dvojice, pro jejichž čas platí $time \geq t_{arr}(c)$. Zde je možné, že nové spojení by je mohlo vydominovat svým příjezdovým časem, tedy že odjíždí později a zároveň přijíždí dříve. Díky seřazenému seznamu spojení se bude přistupovat na velmi malý počet dvojic. Nyní je potřeba vyzkoušet dominanci mezi novým bagem a těmi stávajícími. Všechny se musí projít a vymazat ty, které jsou alespoň jedním z nových dominované. Jako další krok se z profilu vezme první bag s nižším odletovým časem a proces dominance se spustí ještě jednou, ale naopak. Nyní jsou nově vytvořené štítky testovány oproti těm, které přijely dříve a tím pádem je mohou dominovat. Na konci se (pokud daný příjezdový čas ještě neexistuje) vytvoří nová dvojice t_{arr}, bag a vloží do profilu.



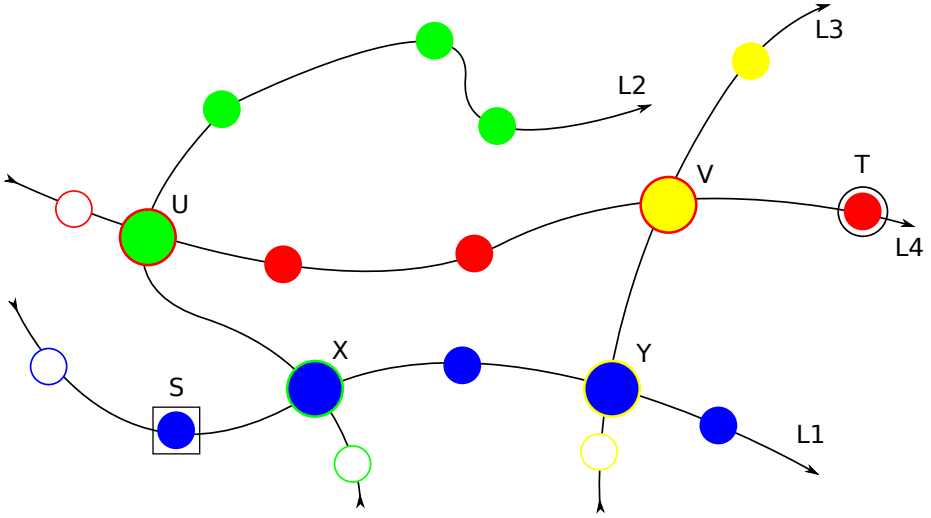
Obrázek 4.3: Příklad napojování spojení a následné ukládání do profilu. Předpokládejme, že maximální počet tolerovaných přestupů je 3 a štítky ukládáme jako trojici *(cena, odjezdový čas, počet přestupů)*. Skenované spojení letí v 8:15 z A do B, kam doletí v 9:00 a stojí 5€. Nalezne se záznam s časem 8:00 (nejpozdější příjezd, který umožňuje stihnout dané spojení) a napojí se na všechny štítky v něm. Zde je vidět, že se rovnou zahodí štítek `x` kvůli počtu přestupů. Dále se nový bag pokouší dominovat štítky s pozdějším příjezdem, viz dominance `z` nad `u` kvůli odjezdovému času. Poté se dominuje nový bag předchozím (s časem 8:50), výsledkem čehož štítek `w` eliminuje `y`. Nakonec zbývající `z` vytvoří nový záznam s časem 9:00 a uloží se tam.

Procházení končí ve chvíli, kdy načtené spojení má odjezdový čas vyšší, než je horní hranice vstupního intervalu. Nyní se projde profil cílové zastávky a z uložených štítků se rekonstruují výsledné trasy.

4.6 Algoritmus RAPTOR

Podobným přístupem jako CSA je algoritmus založený na kolech (angl. *Round bAsed Public Transit Optimized Router* - RAPTOR) [10]. Stejně jako CSA, nereprezentuje data grafem, ale pracuje pouze s efektivně uloženými daty. Hlavním rozdílem mezi RAPTORem a CSA je skutečnost, že RAPTOR funguje nad celými linkami a ne pouze s jednotlivými spojeními. Je tedy mnohem vhodnější pro hromadnou dopravu. Má také fixní optimalizační kritérium - počet přestupů. Proto také pracuje v kolech a celý proces se opakuje tolikrát, kolik je maximální počet segmentů. Rozšíření ze schopnosti řešit první problém na schopnost řešit zbylé dva problémy je analogický jako u CSA, popíšeme tedy rovnou silnější verzi algoritmu.

RAPTOR si ukládá data jako celé linky a v každé zastávce je uložena informace, na které linky je možné nasednout. Každá zastávka si opět pamatuje profil příjezdů, ale nyní pro každé kolo jeden. Na začátku zpracování dotazu se odjezdová zastávka nastaví jako aktivní a započne první kolo. V prvním (a stejně tak i v každém dalším) kole se procházejí pouze ty linky, které odjíždí z aktivních zastávek. V prvním kole se do profilů dalšího kola všech příjezdových zastávek proskenovaných spojení uloží bagy vzájemně se nedominujících štítků. Pokaždé, když se podaří takový bag vytvořit a zároveň lze z příjezdové zastávky přesednout na jinou linku, tak se tato zastávka označí jako aktivní pro další kolo. V každém následujícím kole se opět procházejí pouze aktivní zastávky, které byly nastaveny předchozí kolo. Procházené linky vycházející z aktivních zastávek se pokouší prodloužit štítky zastávek na lince uložené v profilu předchozího kola. Proces procházení linek je znázorněn na obrázku 4.4.



Obrázek 4.4: Příklad hledání cesty ze zastávky S do zastávky T. V prvním kole se projde pouze linka L1 a zastávky X a Y se označí jako aktivní. V druhém kole se zpracují linky L2 a L3 a zastávky X a Y se obě označí jako aktivní. Ve třetím kole se prochází linka L4 od zastávky X a v zastávce Y se daná linka začne pokoušet napojit i na linku L3. Prázdné kruhy představují zastávky, které nejsou nikdy skenovány.

Dalším rozdílem oproti CSA je moment, kdy je potřeba nově vytvořený bag vložit do současného profilu. Následující procedura se provádí pro každý štítek z bagu. Při zkoumání vztahu nového štítku k současnému profilu mohou nastat tři možnosti: 1) nový štítek dominuje nějaký jiný, 2) nový štítek je dominován a 3) ani jedno z předchozích. V případě 1) je jasné, že v současném profilu není žádný záznam se štítkem, který by ten nový dominoval,

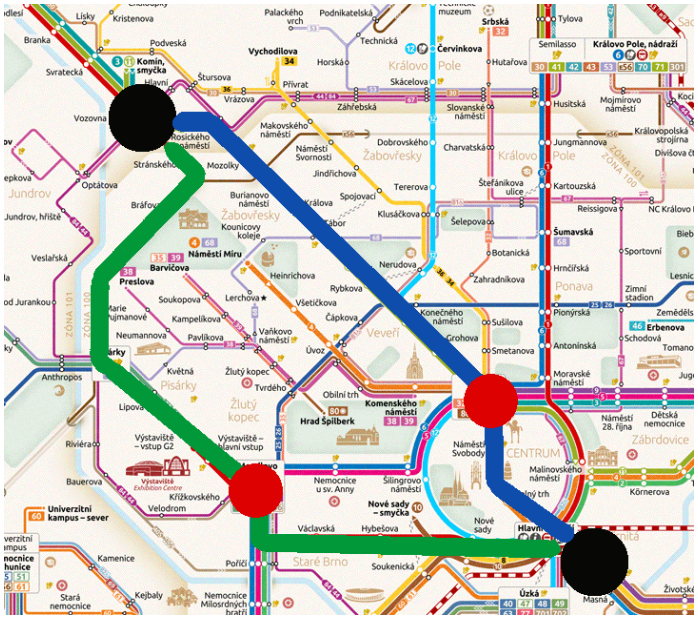
ale mohou existovat další, které jsou dominovány. Proto se v této situaci projde daný profil a vyhodí se všechny dominované a nakonec se vloží nový štítek. Pokud nastane možnost 2), nový štítek se zahodí a pokračuje se dalším. V případě, že nenastane ani jedna z možností 1) a 2), stále existuje možnost, že by byl nový štítek některým z nižších úrovní dominován, ale už ne naopak (ty s méně přestupy mohou být nejhůř neporovnatelné). Je proto nutné, aby se s daným štítkem prošly všechny nižší úrovně a přesvědčilo se, že štítek není nikým dominován. Pouze v takovém případě je možné jej vložit do současného profilu.

Algoritmus končí buď s posledním kolem nebo ve chvíli, kdy už nebyly označeny žádné aktivní zastávky. V poslední části se musí projít profily cílové zastávky ve všech úrovních, aby se zrekonstruovaly všechny výsledné cesty.

4.7 Metoda přestupních vzorů

Metoda přestupních vzorů (angl. *Transfer Patterns*) je založena na jednoduché myšlence, že všechny optimální cesty mezi dvěma místy splňují jistý vzor [4]. Vzorem je myšlen seznam linek a zastávek, kde člověk přesedá (pro ilustraci viz obrázek 4.5). Jak z toho vyplývá, této metodě předchází obrovské předpočítání všech optimálních cest mezi všemi možnými destinacemi, aby se zjistily potřebné vzory. V případě, že by byl graf spojení tak obrovský, že by předpočítání všech vzorů bylo téměř nemožné, je možné vypočítat pouze části těchto vzorů tak, aby výsledné vzory šly z těchto částí poskládat. Většinou se zvolí množina důležitých destinací jako jádro grafu a vzory se hledají právě dvoufázově. Nejdříve mezi všemi destinacemi a všemi uzly z jádra a dále pak z jádra do všech destinací.

Vyhodnocování dotazu je pak velmi jednoduché. Prochází se pouze ta potenciální spojení, která splňují dané přestupní vzory a která na sebe v daných zastávkách navazují časově.



Obrázek 4.5: Příklad přestupních vzorů na trase Vlhká - Vozovna Komín v rámci MHD Brno. Je vidět, že existují pouze dva optimální vzory cest (modrá a zelená). Červené body znázorňují místa, kde je potřeba přesednout.

4.8 Algoritmus TRANSIT

Jak již bylo zmíněno v kapitole 3.8, i metoda přestupních bodů byla adaptována pro hromadnou dopravu. Tato výsledná metoda se jmenuje TRANSIT [2] a modeluje graf spojení modifikovaným *time-expanded* přístupem.

Stejně jako v případě metody přestupních bodů, i TRANSIT se dá rozdělit na dvě části - předzpracování a provádění dotazů. Ve fázi předzpracování je potřeba získat množinu přestupních bodů. Naivní přístup by počítal přestupní body přímo z *time-expanded* grafu jako události v čase, kdy se přestupuje v jedné zastávce. Avšak vzhledem k tomu, že graf vytvořený tímto způsobem je velmi velký, autoři jako přestupní body neuvažují konkrétní události v čase, ale přímo zastávky. Zde ovšem vyvstává problém, že některá zastávka je validním přestupním bodem pouze v některou část dne. Den je proto segmentován do menších částí po několika hodinách a v každé se hledají tyto přestupní zastávky. Výsledkem je několik množin přestupních bodů odpovídající různým částem dne. V těchto menších množinách už není takový problém vypočítat potřebné vzdálenosti mezi 1) zastávkami a jejich přestupními body, 2) přestupními body navzájem a 3) přestupními body a jejich zastávkami.

Ve chvíli zpracování dotazu na cestu mezi zastávkami A a B v čase t se vyberou přestupní body T_A T_B a analogicky ke statické verzi se hledá minimum mezi možnými nakombinovanými cestami. Jediným rozdílem je indexovaní do tabulek pomocí příjezdových časů do jednotlivých bodů. Problém více kritérií autoři řeší redukcí více kritérií na jedno pomocí lineární utilizační funkce.

4.9 Srovnání metod a jejich použití pro leteckou dopravu

V této kapitole byly představeny přední algoritmy, které se v současnosti používají pro plánování cest v hromadné dopravě. První dvě z prezentovaných ke svému fungování nepotřebují žádné předzpracování, což z nich dělá ideálního kandidáta pro dynamické prostředí letecké dopravy. CSA je z těchto metod nejuniverzálnější, vzhledem k tomu, že v letecké dopravě neexistuje nic jiného než základní spojení (narozdíl od linek v hromadné dopravě). Zároveň také neklade žádné nároky na pravidla v grafu (hierarchie, vzory) a zároveň slibuje efektivní zpracovávání paměti. Oproti tomu je RAPTOR specializovaný na hromadnou dopravu s linkami. Nicméně zde je možné stále pracovat se spojeními jako s linkami a využít pouze konceptu více úrovní a dobrého rozdělení dat mezi odletová letiště.

Zbylé dvě metody vyžadují značné předzpracování a kladou jisté nároky na strukturu grafu. Metoda přestupních vzorů pro svou efektivitu vyžaduje, aby jich nebylo přespříliš, což se analyzuje dále v kapitole 5.3. Poslední metoda využívající přestupní body předpokládá, že existuje malé jádro, přes které vede každá optimální cesta. S tím souvisí také nárok na minimum lokálních dotazů (které jsou definovány právě jádrem). V následující kapitole se ukáže, že graf letů je sice částečně strukturovaný, nicméně stále nevykazuje optimální prostředí pro použití této metody.

Kapitola 5

Analýza grafu letů

V této kapitole popíšeme analýzu leteckého grafu. Přiblížíme skutečnou velikost, hustotu a speciální vlastnosti samotného grafu. V závěru kapitoly také srovnáme graf letů oproti grafům reprezentujících spojení jiných typů dopravy.

5.1 Základní informace

Data, která jsou zpracovávána na vstupu byla dodána spolupracující firmou. Data obsahují pouze jednosměrné lety, jak přímé, tak i spojené kombinace (zpravidla akce aerolinek) více letů. Vzhledem k názvosloví, které bylo zavedeno v sekci 4.1 je nutno zmínit, že ačkoliv se na první pohled zdá, že se jedná o spoje (posloupnost několika letů), tak je to pouze elementární spojení - nelze v průběhu cesty opustit let (většinou se jedná pouze o mezipřistání). Jediným rozdílem, který z toho tedy pro algoritmus plyne je ten, že je potřeba si pamatovat, kolik každé elementární spojení obsahuje přestupů (jelikož je to jedno z optimalizačních kritérií).

Jednotlivé lety obsahují následující informace:

src	odletové letiště
dst	příletové letiště
price	cena letu
dep_time	čas odletu
arr_time	čas přistání
segments	počet fyzických letů ve spojení

Tabulka 5.1: Vlastnosti letů.

časový interval	240 dní	30 dní
# vstupních letů	2 420 981 824	672 823 550
# unikátních	1 143 572 376	178 508 449
# periodicky redukovaných	23 391 267	-
# letišť	2 935	2 935

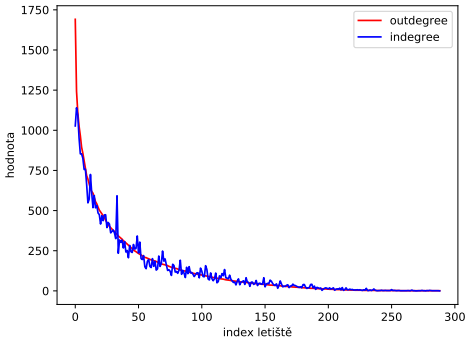
Tabulka 5.2: Základní informace o grafu letů získané v listopadu 2017.

Z tabulky 5.2 je vidět, že ve vstupních datech se nachází mnoho duplikovaných letů, které je možné bezeztrátově eliminovat. Dále je zajímavá periodická redukce. Při tomto

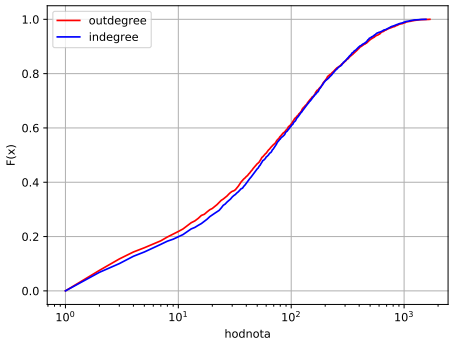
postupu se nebrala v potaz cena a dny, kdy let odlétá, ale pouze přesný čas v rámci dne. Tímto způsobem se lety, které létají periodicky v různé dny transformovaly v jeden. Jak se zdá na první pohled, zdánlivě to výrazně zmenšuje výsledný graf, nicméně vzhledem k tomu, že cena je důležitým parametrem, jedná se stále o naprosto jiná spojení (z hlediska plánování). Tato redukce by maximálně mohla dopomoci při ukládání do paměti, neboť pro každý let a jeho opakování by se tak musely základní informace pamatovat pouze jednou a dále už jen vektor s cenami a informace, který den je let aktivní.

5.2 Hustota grafu a shluky

Hustota grafu byla též analyzována. Cílem bylo získat informaci o celkové propojenosti grafu, jakožto i o důležitosti jednotlivých uzlů (letišť). Na prvním grafu 5.1 jsou vidět oba stupně jednotlivých letišť. V tomto případě je zanedbán čas, jde pouze o informaci, jestli mezi dvěma uzly existuje orientovaná hrana. Je vidět, že obě metriky spolu pozitivně korelují. Průměrný stupeň uzlu je potom 130. Na druhém grafu 5.2 je lépe zobrazena propojenost grafu letů. Je vidět, že 20 % letisť má stupeň menší nebo rovno 10 - jedná se tedy o velkou okrajovou část. Na druhou stranu téměř 40 % z nich má stupeň větší než 100, což svědčí o existenci velkého jádra.



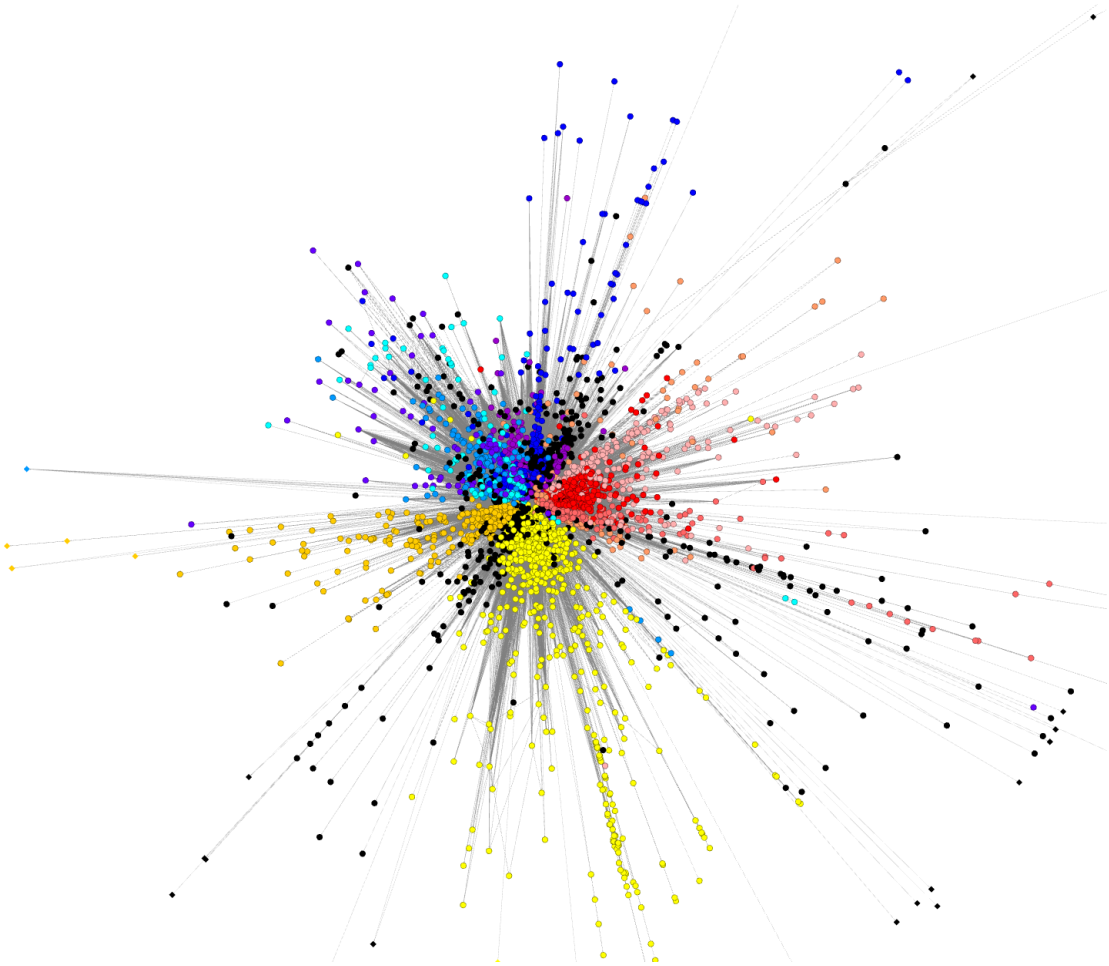
Obrázek 5.1: Graf ukazující hodnoty indegree a outdegree pro jednotlivá letiště (vzorkováno s faktorem 10 kvůli přehlednosti).



Obrázek 5.2: Distribuční funkce stupňů letišť.

Shluková analýza se realizovala pomocí programu Gephi[1], vizualizačního nástroje pro analýzu grafů. Cílem analýzy shlukování grafu by mělo být zjištění, zdali by šly aplikovat hierarchické přístupy pro plánování cest. Na grafu 5.3 jsou na první pohled jasně vidět dvě vlastnosti. Zaprvé, že se jedná o velmi hustě propojený graf. Průměrná nejkratší cesta mezi dvěma uzly je 2.4 a průměr celé sítě je 9. Na druhou stranu lze však vidět výborně propojené (avšak ne už tolik segmentované) jednotlivé kontinenty. Je tedy možné, že využití lokality dílčích uzlů grafu by mohlo dopomoci při hledání výsledné cesty.

[1] https://gephi.org/



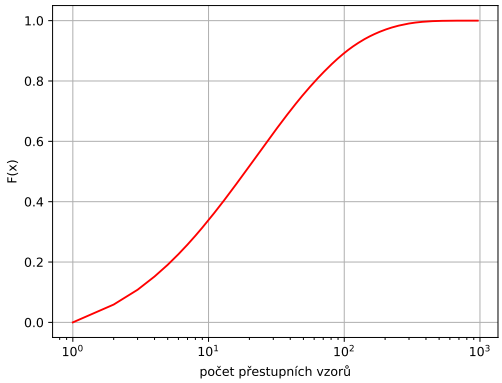
Obrázek 5.3: Graf vygenerovaný program Gephi. Jednotlivé uzly jsou obarveny podle kontinentů (modrá - Evropa (a její části), žlutá - Severní Amerika, oranžová - Jižní Amerika, červená - Asie, černá - zbytek). Program se snažil zobrazit graf takovým způsobem, aby se co nejméně hran křížilo.

5.3 Přestupní vzory

Pomocí algoritmu poskytnutého zmíněnou firmou jsme byli schopni získat všechny pareto optimální cesty mezi všemi letišti ve výše zmíněném grafu. Pro každou dvojici jsem si zaznamenal počet unikátních přestupních vzorů a výsledek pak zanesl do grafu v podobě distribuční funkce viz obrázek 5.4. Je vidět, že množství přestupních vzorů pro jednu trasu pro většinu tras nepřesahuje počet 200. Dalším mnohem praktičtějším zjištěním je fakt, že skoro polovina tras má méně než 20 různých vzorů. Toto by mohlo být dostatečně malé množství na to, aby metoda přestupních vzorů stála za vyzkoušení.

5.4 Srovnání s jinými typy dopravy

Hlavním rozdílem grafu letů oproti jiným typům dopravy je především počet unikátních spojení. Vzhledem k tomu, že při hledání letů je cena výrazným faktorem, nedá se říct, že by lety byly periodické a člověk mohl generalizovat plánování na pouze pár po sobě jdoucích



Obrázek 5.4: Graf distribuční funkce počtu přestupních vzorů na trase mezi dvěma letišti.

dní. Velmi se může lišit cena letů odlétajících současný týden a těch, které odlétají až za půl roku. Proto nelze v tomto grafu vyhledávat pouze v rámci krátkého období a zmenšit jej co do počtu hran.

Dalším rozdílem je počet uzlů, který je v porovnání s ostatními grafy velmi malý (navzdory tomu, že letová data pokrývají celý svět). Předkládáme alespoň základní srovnání počtů (viz tabulka 5.3) s datovými sadami použitými ve zvolených článcích.

dataset	Londýn [10]	Německo [12]	Evropa [12]	Svět [9]
typ dopravy	MHD	vlaky	vlaky	lety
časový interval	1 den	2 dny	3 dny	1 den – 1 měsíc
# zastávek	20 843	6 822	30 517	1 172
# spojů	133 011	94 858	463 887	-
# spojení	5 132 672	976 678	4 654 812	29 490

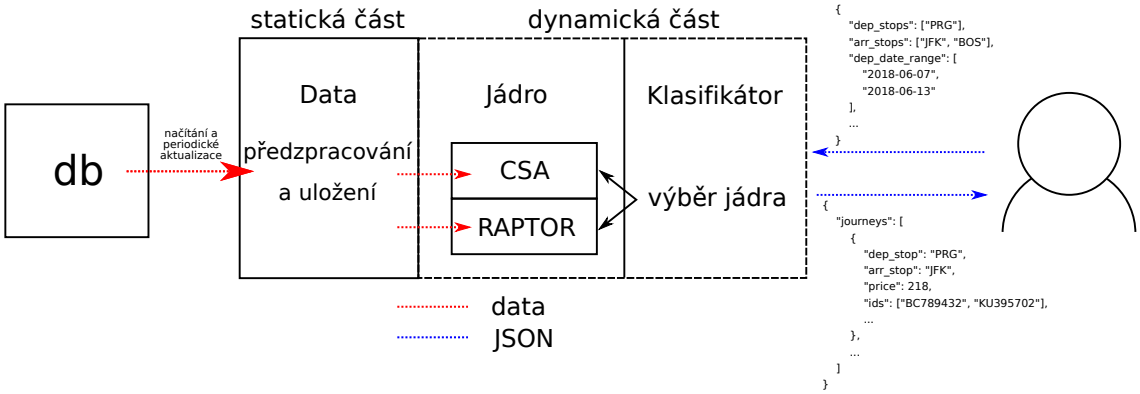
Tabulka 5.3: Srovnání velikosti grafů mezi různými typy dopravy.

U veškeré pozemní přepravy je vidět, jak lze využít periodické jízdní řády a pracovat pouze s pár dny. Údaje o letovém grafu pocházející z článku [9] předkládáme spíše ze zajímavosti, jelikož nebylo jasné, zdali data pokrývají celý měsíc (je to možné, dataset obsahuje data pouze od dvou aerolinek), nebo pouze jeden den. V každém případě je vidět, že ani jeden z popsaných datasetů nedosahuje velikosti našeho letového grafu. Tento fakt je způsobem absencí dokonalé periodicity letů (kvůli měnícím se cenám).

Kapitola 6

Návrh vyhledávacího systému

V této kapitole popíšeme návrh komplexního systému, který je schopen načítat data z různých zdrojů, ukládat je do svých struktur a předzpracovat je, aby byl připraven odpovídat na klientské dotazy. Pro výpočetní jádro jsme zvolili dva algoritmy popsané v kapitole 4, CSA a RAPTOR. Zvolil jsem oba, protože experimenty prokázaly, že každý z nich se lépe vypořádává s jiným typem dotazů (viz kapitola 8). Pro schopnost rozhodování, který algoritmus kdy použít jsme natrénovali klasifikátor, který se stará právě o toto rozhodování. Celé schéma je vidět na obrázku 6.1. Celý systém by se dal rozdělit na dvě části a to statickou (načítání dat) a dynamickou (vyhodnocování dotazů). Při příchozím dotazu se provede klasifikace daného dotazu a vyhodnotí se, které z výpočetních jader se má spustit. Po získání nakombinovaných cest se výsledek vráti klientovi ve formátu JSON. Jednotlivé části systému budou blíže popsány v následujících podkapitolách.



Obrázek 6.1: Schéma navrhovaného systému.

6.1 Formát komunikačního rozhraní

V této podkapitole budou popsány formáty klientského dotazu a serverové odpovědi ze strany systému.

6.1.1 Klientský dotaz

Níže je vidět formát dotazu, který je přijímán navrženým systémem. Uživatel jako hlavní kritérium specifikuje, jaká má být řadící metrika pro vrácené kombinace. Jde buď o cenu

(uživatel chce nejlevnější), trvání (nejkratší), nebo kvalitu (nejkvalitnější cesty). Kvalita letu a potažmo celé cesty je určena funkcí poskytnutou spolupracující firmou. Dále musí specifikovat dvě data, která určují interval odletu (data jsou brána jako čas lokální půlnoci předchozího dne v odletových městech). Odletová a příletová letiště se v rámci svých množin spojí, tím pádem se nevyhledávají všechny dvojice, ale pouze ty nejvýhodnější (dle zadané metriky), které je možné mezi těmito množinami získat. Dále pak lze specifikovat řadu nepovinných filtrů, které ovlivňují kombinování výsledné cesty. Mezi ně patří maximální čas na přestup, maximální cena, maximální počet přestupů a maximální trvání cesty. Posledním parametrem je počet vrácených výsledků. Některé nepovinné filtry mají předem nastavený strop: maximální počet přestupů je 3 a maximální trvání cesty je 60 hodin.

```
{
    "order": "price" | "duration" |  "quality",
    "dep_date": [
        <date>,
        <date>
    ],
    "dep_stop_ids": [
        <stop_id>, ...
    ],
    "arr_stop_ids": [
        <stop_id>, ...
    ],
    "filters": {
        "max_transfer_time": <uint>?,
        "max_price": <uint>?,
        "max_transfers": <uint>?,
        "max_duration": <uint>?
    },
    "limit": <uint>
}
```

6.1.2 Odpověď serveru

Odpověď systému, jejíž formát je níže, je také zapsán ve formátu JSON. Obsahuje všechny potřebné informace (čísla letů) a základní popisné informace dávající jasný náhled na danou cestu. Výsledné cesty jsou seřazené podle kritéria v dotazu. Odpověď by mohla být obohacena o přesné přestupní časy, nicméně čísla letů obsahují všechny informace, ze kterých je možné přesné časy zjistit (a o to se stará navazující systém).

```
"journeys": [
    {
        "dep_stop_id": <stop_id>,
        "arr_stop_id": <stop_id>,
        "dep_time_local": <date> <time>,
        "arr_time_local": <date> <time>,
        "price": <uint>,
        "segment_count": <uint>,
        "duration": <uint>,
```

```
        "legs": [
          {
                "id": <flight_number>,
                "segment_count": <uint>
          }, ...
        ]
    },
    ...
]
```

6.2 Správce dat

Jedinou součástí statické části systému je právě správce dat. Hlavním úkolem tohoto modulu je periodické načítání dat z určeného datového zdroje. Na začátku je nutné načíst všechna data a uložit je, ale s postupujícím časem je možné načítat pouze změny a uložená data tak aktualizovat. Nicméně po úvaze a srovnání potřebných operací jsme rozhodli, že podstatně jednodušší a zároveň nepříliš náročné na zdroje je celá data načíst znovu, zpracovat je a pouze zaměnit. Celý proces pak netrvá déle než 20 minut a zachovává plnou konzistenci dat. Systém je nastaven tak, že vyhledávání se bude vždy realizovat nad statickým obrazem dat, jádro tak nebude muset řešit problém s nekonzistencí dat.

Ve fázi načítání se rozhoduje, ze kterého zdroje se lety načtou. Na výběr je z následujících:

CSV Jedná se o datový formát, který ukládá data v textovém formátu, přičemž každý záznam je na jednom řádku a dílčí položky jsou odděleny čárkami. V tomto datovém formátu lze jednoduše vytvořit umělá data (například pro testování) pro kombinační systém. Na každém řádku se musí nacházet informace potřebné pro kombinování.

Dump Tento zdroj představuje binární soubor, který byl vytvořen přečtením celé databáze, kompresí a uložením dat na disk. Smyslem formátu je ve fázi testování omezit načítání z databáze a ve fázi experimentování zachovat stejná data (pro potřeby porovnávání).

Cassandra NoSQL databáze Cassandra je hlavní zdroj, odkud se načítají reálná aktualizovaná data. Jedná se o spojení více databází, které jsou dostatečně stabilní na to, aby byly schopny vydržet celkový sken dat a přitom ještě zvládnout zpracovávat příchozí aktualizace.

Po načtení jednosměrných spojení z jednoho ze zdrojů se seznam letů profiltruje od duplikátů (klasifikace duplikátů byla zmíněna v kapitole 5.1) a načtené lety se transformují. Odletová a příletová letiště se převedou na indexy pro lepší práci a lokální časy se převedou na koordinovaný světový čas (UTC) s tím, že je potřeba si zapamatovat v jaké časové zóně se které letiště nachází. Pro kombinování je nutné mít časy právě v unifikovaném formátu. Dále se jednotlivá spojení uloží do dvou struktur podle potřeb algoritmů. V případě CSA se jedná pouze o jedno velké pole letů seřazených podle času a u RAPTORa jsou lety rozdělené podle odletového letiště (a také seřazené podle času).

Během načítání letů se zároveň budují tři statické (časově nezávislé) grafy potřebné při fázi předzpracování. Následující procedura je analogická pro cenu, počet segmentů a trvání letu, popíšeme ji proto jen pro cenu. Při načítání se pro každou dvojici letišť, mezi

kterými existuje alespoň jeden let, uloží nejlevnější cena napříč celým datovým rámcem a z těchto hodnot se poté vytvoří nový statický graf. Zároveň s tím se pro každé letiště uloží seznam letišť, která jsou s ním přímo spojeny (alespoň jedním letem) - pojmenujme jej seznam sousedů. Dále se v tomto grafu naleznou nejkratší cesty (např. Dijkstrovým algoritmem) mezi každými dvěma letišti a uloží se do tabulky. Tyto hodnoty budou sloužit jako spodní odhad ceny cesty mezi každými dvěma letišti. Toto je jediné předzpracování, které je prováděno.

Ve chvíli, kdy předzpracování skončí, nahradí se aktuální data (pokud existují) a systém může načítat data znovu. Případně se může rozhodnout, že se bude nějakou dobu před další aktualizací čekat, vše záleží na tom, jak čerstvá data jsou potřeba. Stejně tak je systém připraven odpovídat na dotazy.

6.3 Klasifikátor

První částí dynamické části je klasifikátor. Po obdržení dotazu je potřeba rozhodnout, které z výpočetních jader se má zavolat, neboť jak experimenty ukázaly (viz kapitola 8), klientské dotazy se dají rozdělit na dvě části podle toho, který algoritmus danou skupinu dotazů vyhodnocuje rychleji.

Postupovali jsem standardně, a to shromážděním velkého množství reálných uživatelských dotazů (poskytnutých zmíněnou firmou) a testováním, pro který dotaz je který algoritmus rychlejší. Jak jsme zjistili tuto informaci, bylo ještě potřeba zvolit vhodnou množinu rysů, které by měly dobré rozhodovací vlastnosti. Následuje výčet rysů s drobným popisem:

arr_stops_len Počet odletových letišť.

dep_stops_len Počet příletových letišť.

date_range Časové rozmezí pro odlet (v počtu dnů).

src_outdegree Kumulativní hodnota *outdegree* pro odletová letiště.

dst_indegree Kumulativní hodnota *indegree* pro příletová letiště.

transfer_dist Vzdálenost odletové a příletové množiny ve statickém grafu modelujícím minimální počty přestupů.

routing_points Počet letišť, která se budou při plánování skutečně brát v potaz (pro metodiku výběru letišť viz kapitola 6.4.1).

Rozhodli jsme se klasifikátor modelovat pomocí rozhodovacího stromu. Shromážděná data jsme rozdělili v poměru 3:1 na trénovací a testovací, trénovací použili jako vstup do modelu a na testovacích zjišťovali, jaké jsou optimální parametry modelu. Podrobné výsledky klasifikace, jako například její úspěšnost a důležitost jednotlivých rysů budou rozebrány dále v kapitole 8.3.

Po separátním naučení klasifikátoru je v navrhovaném systému vytvořen nový se zjištěnými parametry, který je dále připraven na rozhodování. Žádné nové doučování není žádoucí, neboť by bylo nutné, aby v běžném provozu systém vyhodnocoval dotazy oběma způsoby za cílem zjištění, jestli predikce byla správná. Nicméně toto by výrazně zatížilo výpočetní prostředky. Mnohem praktičtější je proto jednou za čas shromáždit nový seznam dotazů, za pomoci kterých se klasifikátor znovu doučí.

6.4 Výpočetní jádro

Hlavní částí systému je dynamické výpočetní jádro. Jeho úkolem je na validní dotaz odpovědět seznamem nakombinovaných letů. Nachází se přesně uprostřed systému, kdy z jedné strany dostává informace od klasifikátoru (společně s výběrem jádra také klientský dotaz) a z druhé strany, od správce dat, dostává datové podklady.

Výpočetní jádro obsahuje implementaci dvou algoritmů: CSA a RAPTOR, popsaných v kapitolách 4.5 a 4.6. Nicméně zdroje, ze kterých jsme čerpali, samozřejmě nepopisují všechny detaily, které je potřeba zohlednit pro efektivní implementaci. Některé již byly popsány ve zmíněných kapitolách. Stále však bylo potřeba vymyslet vhodná vylepšení, aby výsledné řešení bylo dostatečně rychlé.

Následuje popis vylepšení, která jsme pro oba algoritmy navrhli a která využívají informací, které jsme zjistili během analýzy (graf obsahuje odlehlá letiště).

První vylepšení si klade za cíl omezit prostor, který je potřeba projít a zpracovat. Hlavní prerekvizitou pro něj je vytvoření časově nezávislých grafů spodních odhadů ve fázi předzpracování. Další je analogické, jen se používá v jiném místě a s jinými parametry. Poslední vylepšení využívá faktu, že víme, podle kterého kritéria uživatel chce mít výsledky seřazené a umožňuje tak znovu omezit prohledávaný prostor.

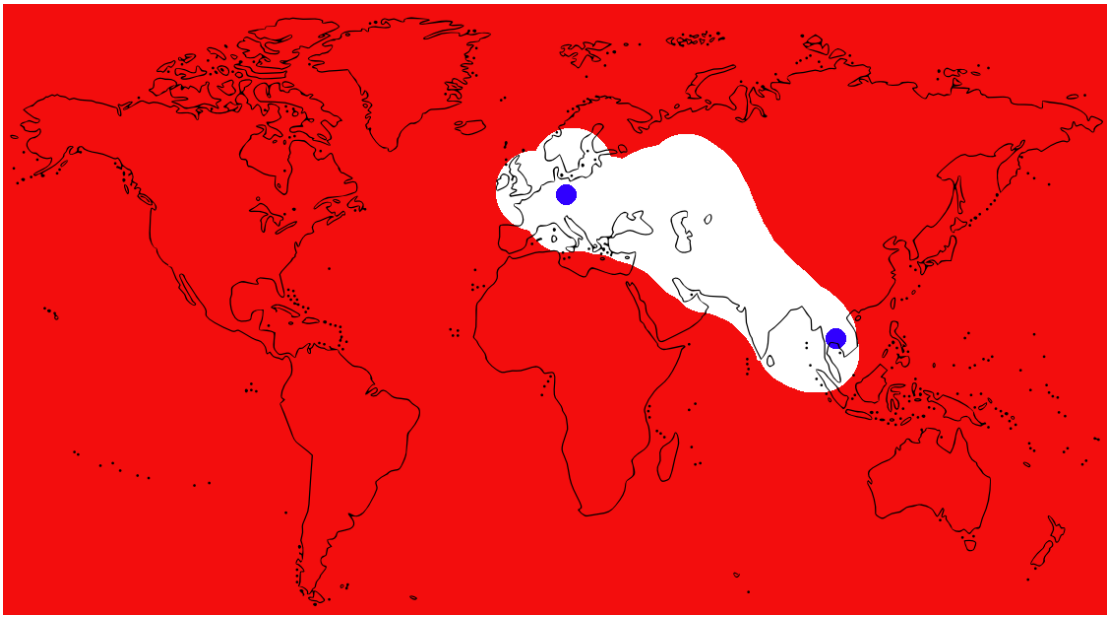
6.4.1 Redukce prohledávaného prostoru

Nejslabší z ořezávání (co se týče filtrování destinací) je ale zároveň výpočetně nejméně náročné a lze jej použít pro oba algoritmy. Ořezávání vzdálených či nevýhodných destinací se provádí pro každý dotaz individuálně. Pokaždé při přijetí dotazu se pro každé letiště x profiltruje seznam jeho sousedů $N(x)$ pomocí omezujících údajů v dotaze - konkrétně pomocí omezení na cenu, počet přestupů a celkovou dobu cesty. Pro každé z kritérií je následující procedura stejná. Aby bylo letiště v seznamu sousedů zachováno, musí se přes něj dát dostat do alespoň jedné z cílových destinací T při zachování limitu daného kritéria. Pro každé kritérium se sestrojí množina $N'_{crit}(x)$ (viz rovnice 6.1) a nakonec se udělá průnik dílčích množin.

$$N'_{crit}(x) = \{n | n \in N(x) \wedge \exists t \in T : dist_{crit}(x, n) + dist_{crit}(n, t) \geq limit_{crit}\} \tag{6.1}$$

Tímto způsobem se pro velkou část spojení odřízne zpracovávání už v první chvíli, kdy se spojení vůbec načte. Odříznou se tak dvě nejnáročnější části zpracovávání: přístup do paměti pro cílovou multimnožinu (doposud se procházelo jedno velké pole prvků kontinuálně uložených za sebou v paměti) a spojování stávajících štítků s novým letem. Toto odříznutí se sice vykonává bez jakékoli hlubší informace o daném letu (bere se v potaz pouze odletové a příletové letiště), nicméně o to víc je možné test provést rychleji. Z principu toto oříznutí vychází pouze ze statických informací o spodních odhadech, aby nebylo potřeba žádných informací získaných v průběhu. Příklad redukce prohledávaného prostoru je vidět na obrázku 6.2.

Pro algoritmus RAPTOR lze odřezávání dovést ještě dále, a to zmenšováním prostoru individuálně pro každé z kol, avšak pouze odřezáváním na bázi přestupů. Lze využít toho, že v každém kole přesně víme, kolik přestupů budou mít všechny rozvíjené štítky. Tímto způsobem pro každé místo a jeho sousedy budeme požadovat, aby bylo možné se přes ně dostat do cíle na počet přestupů získaný vztahem $celkovy\_pocet\_kol - aktualni\_kolo$. Tímto se daný prostor ještě dále zmenší.



Obrázek 6.2: Příklad redukce prohledávaného prostoru na trase Praha - Bangkok. Spojení, jejichž příletové letiště leží v červené oblasti se vůbec nebudou dále rozvíjet.

Tento způsob ořezávání je optimální a nikdy nemůže odříznout místo, které by mohlo vést k reálné cestě. Nicméně stále je možné, že například pro cenu nebude nastaven cenový strop nebo pro krátké spojení se bude uplatňovat vysoký časový limit, což povede k rozvíjení spojení, která budou naprosto převyšovat (v negativním slova smyslu) nejlepší výsledek. Proto je možné toto ořezávání provést ještě jednou, ale namísto daného limitu použít odhad, který nechceme přesáhnout. Rozhodli jsme se jako nový limit zvolit dvojnásobek nejnižší ceny mezi výchozím letištěm a cílovou destinací. Toto má za efekt odříznutí dalšího prostoru za cenu rizika neprozkoumání určitých kombinací. Úspěšnost tohoto vylepšení a stejně tak výsledné zrychlení je popsáno v kapitole 8.1.

6.4.2 Dynamické odmítání spojení

Ve chvíli, kdy se načtené spojení dostane přes výše zmíněnou kontrolu, se načte příslušná multimnožina a dané spojení se začne napojovat na štítky vevnitř. V této chvíli už máme informaci o tom, jaká by byla současná cena (míněno agregovaná ze všech kritérií) a je možné provést dynamické rozhodnutí, jestli má smysl prodlužovat štítek do dané destinace. Toto rozhodnutí se vykonává na základě heuristické funkce 6.2. Tato funkce znovu volí jako maximální toleranci dvojnásobek nejlepší ceny. Nyní se ovšem musí dát pozor, aby dvojnásobek nepřesáhl faktický limit pro danou metriku.

Vzhledem k tomu, že není možné zjistit, ke kterému z cílových destinací se daný štítek blíží, je potřeba rozhodnutí provést v souladu se všemi cíli - pokud se do alespoň jednoho dá ještě dostat (pokud je nerovnost splněna), je potřeba danou extenzi povolit. Opět se rozhodování musí provádět přes všechna optimalizační kritéria, tentokrát jsou konkrétní funkce 6.3, 6.4, 6.5 mírně pozměněné, každé z kritérií se chová jinak, když se napojuje.

$$f(x) = min(2 * x, limit) \tag{6.2}$$

$$\exists dst \in T : label.price + c.price \geq f(min\_price(label.src, dst)) \quad (6.3)$$

$$\exists dst \in T : c.arr\_time - label.src\_time \geq f(min\_duration(label.src, dst)) \quad (6.4)$$

$$\exists dst \in T : label.transfers + c.transfers + 1 \geq f(min\_transfers(label.src, dst)) \quad (6.5)$$

Toto vylepšení už je podstatně výpočetně náročnější (sčítání hodnot kritérií se štítkem, více porovnávání) oproti vyzkoušení, zdali se přes určitou destinaci má smysl ubírat. Nicméně má také větší potenciál detekovat slepou uličku.

6.4.3 Prořezávání

Jako další vylepšení jsme navrhli prořezávání možného prostoru za použití již dokončených cest. Protože bereme v potaz více kritérií a obecně je možné, že dvě cesty budou neporovnatelné, je v dotazu nutné specifikovat, podle kterého z nich se budou finální výsledky řadit. Tím získáme nápovědu, která dovoluje obejít Pareto-optimalitu a umožňuje nám tak exaktně určit, která z dvou cest je lepší. Myšlenka je taková, že pokud už máme nakombinovanou cestu, která vede ze startu do cíle a má cenu x, tak pokud při kombinování narazíme na moment, kdy rozdělaná kombinace má cenu $x' \geq x$, můžeme ji zahodit a přestat rozvíjet tuto větev.

Problém nastává ve chvíli, kdy systém obecně podporuje více výsledků. Finální výsledky je potřeba si uchovávat seřazené. V případě, že počet již nakombinovaných cest nepřesahuje limit specifikovaný v dotazu, žádné prořezávání se nekoná. Pouze v případě, že již máme požadovaný počet výsledků, nové potenciální prodloužení se porovnávají s nejhorším výsledkem a pokud jsou horší, vůbec se jimi nepokračuje. V případě vkládání nového finálního výsledku v situaci, kdy už máme požadovaný počet, se nový výsledek znovu porovná s nejhorším a pokud je lepší, starý výsledek se vyhodí a nový se zařadí na svoje místo. Strukturu finálních štítků lze modelovat pomocí prioritní fronty.

Kapitola 7

Implementace plánovače cest

V následujících sekcích budou popsány použité technologie a struktura implementovaného systému.

7.1 Použité technologie

V rámci implementace byly použity následující knihovny, nástroje a programovací jazyky.

Python Jde o vysokoúrovňový programovací jazyk, který byl použit pro veškerou administrativu programu a síťovou komunikaci. Byla použita verze 3.6, která již obsahuje podporu asynchronního zpracování úloh (popsáno v dalším bodě). Taktéž pro výpočetně nenáročné části je Python ideální volbou, vzhledem k jednoduchosti a přívětivosti jazyka.

Asyncio Jedná se o knihovnu[1] v jazyce Python s verzí 3.4 a vyšší. Umožňuje programátorovi asynchronní zpracování vstupně výstupních úloh. Nejedná se o paralelismus jako takový v pravém slova smyslu (ten ani Python nedovoluje), jako spíše využití času, kdy část kódu čeká na I/O operaci efektivním způsobem. Především právě pro serverové řešení je Asyncio dobrou volbou.

scikit-learn Další větší knihovnou[2] pro Python, která byla použita je scikit-learn. Jedná se o knihovnu, která je zaměřena na strojové učení, byla proto použita pro implementaci klasifikátoru.

C++ Jde o programovací jazyk nižší úrovně ideální pro výpočetně náročné pasáže. Při srovnání s jazykem Python je C++ podstatně rychlejší (samozřejmě vždy záleží na konkrétním případě). Dovoluje také lepší správu paměti - např. v případě, že je potřeba uložit obrovské množství struktur (v našem případě letů) a záleží na každém bajtu (případně i bitu!), v C++ lze ovlivnit, kolik má která položka zabírat místa. Nicméně vyšší režie a složitost vývoje v C++ je neoddiskutovatelná. Proto bylo zapotřebí nalézt vhodný způsob, který by dovoloval kombinaci těchto dvou jazyků.

Pybind11 Jedním z možných řešení spojení jazyků Python a C++ je právě knihovna Pybind11[3]. Tato knihovna umožňuje zpřístupnění celých struktur a funkcí jazyka

[1] https://docs.python.org/3/library/asyncio.html
[2] http://scikit-learn.org/stable/
[3] https://github.com/pybind/pybind11

C++ v Pythonu vytvořením sdílené knihovny, která se v Pythonu načte jako klasická knihovna. Nejdůležitějším použitím této knihovny je v našem případě právě možnost volání složitých výpočtů přímo z kódu interpretovaného Pythonem. Pybind11 zároveň pro C++ funkce dovoluje odemknout globální zámek v Pythonu (angl. *Global Interpreter Lock* - GIL), což umožňuje zpracovávat více dotazů skutečně paralelně. Vzhledem k tomu, že dotazy nijak nemění strukturu programu, není ani potřeba brát v potaz synchronizaci dílčích vláken.

Boost Jedná se o velkou knihovnu[4] pro jazyk C++, která obsahuje řešení mnoha běžných problémů v podobě struktur a algoritmů. V systému je použita na práci s časy a daty a na grafové algoritmy potřebné pro zbudování statických grafů a hledání nejkratších cest v nich.

Git Známý verzovací systém, který umožňuje zálohu zdrojových kódů, separátní vývoj a celkově správu vývoje.

7.2 Struktura programu

V této kapitole budou popsány podstatné třídy celého programu.

7.2.1 Python část

V této části budou popsány hlavní třídy té části programu, která byla implementována v jazyce Python. Jak je vidět, jde spíše o administrativní a rozhodovací části.

Node Hlavní třída programu, která se stará o inicializaci pomocných objektů a přijímá příchozí dotazy. Při vlastní inicializaci načte konfigurační soubor a v závislosti na obsažených informacích vytvoří instanci třídy `Data`. Dále si vytvoří objekt třídy `Classifier`, který se bude starat o vybírání jádra. Ve chvíli, kdy přijme příchozí dotaz a je již inicializovaná datová struktura `QueryData`, naplní strukturu `Request` a zavolá svůj objekt klasifikátoru, který provede klasifikaci dotazu. V závislosti na výsledku se zvolí instance jedné z tříd `CSASolver` a `RAPTORSolver` a provede se její metoda `run(Request, QueryData)`. Návratová hodnota výsledku jádra má typ `Result`, která se dále převede do formátu JSON.

Data Třída, která se stará o periodickou aktualizaci dat. Při inicializaci obdrží zdroj, ze kterého data bude získávat. Ve chvíli, kdy jsou data načtena, se vytvoří objekt třídy `QueryData`, kde započne předzpracování.

Classifier Význam této třídy byl již popsán výše, stará se pouze o klasifikaci dotazu a rozhodování, které z výpočetních jader je pro daný dotaz ideální.

7.2.2 C++ část

Zde budou krátce popsány třídy a struktury, které byly implementovány v jazyce C++. Jde především o výpočetní jádro a s ním spjaté entity.

Solver Každý z algoritmů má svou vlastní výpočetní třídu `Solver` ve svém prostoru jmen, tedy `csa::Solver` a `raptor::Solver`. Obě musí implementovat metodu `run(Request,`

[4] https://www.boost.org/

`QueryData)`, která vrací výsledek typu `Result`. Tím nutná podobnost končí, a přestože jsou některé části podobné, oba algoritmy jsou implementovány zcela odděleně.

Request Rozsáhlá třída, která se stará jak o načtení vstupního dotazu a transformace informací, tak i o rozhodování o chodu obou algoritmů. Obsahuje totiž kvalifikační metody, které musí spojení a případné extenze stávajících štítků splňovat.

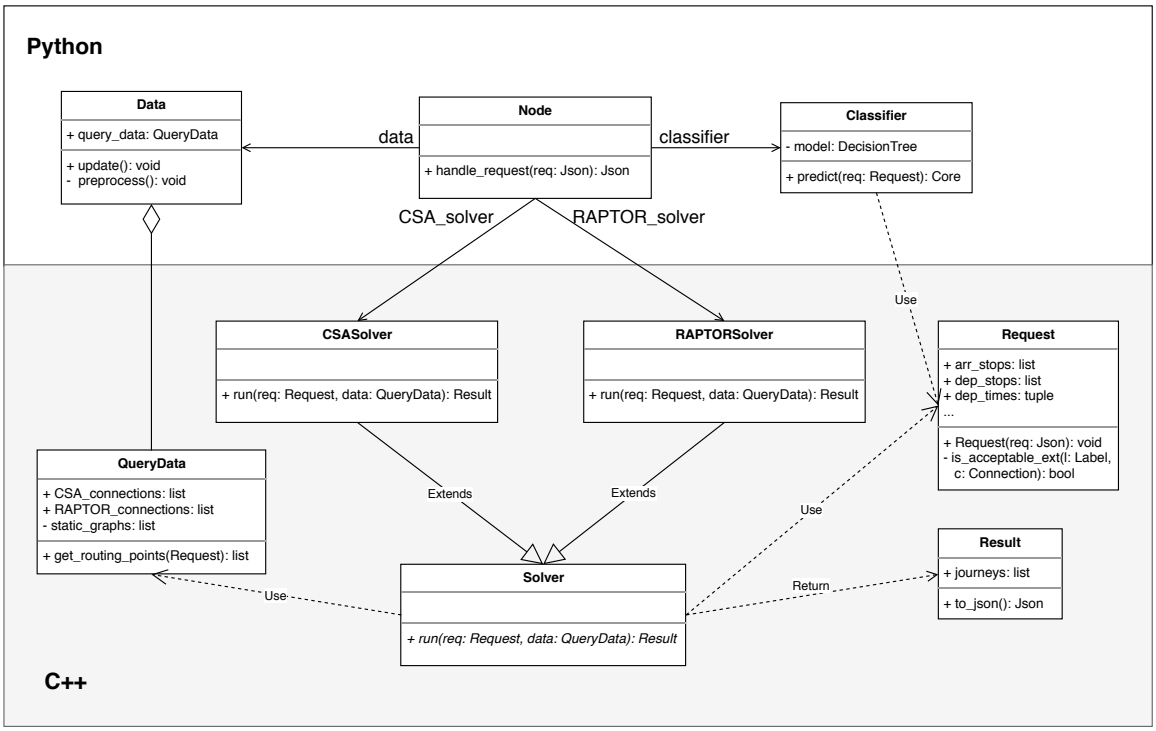
Result Třída, která se stará o uchovávání výsledných cest a správné formátování odpovědi.

QueryData Jde o protějšek třídy `Data` z Python části. Tato třída se stará o efektivní uložení dat potřebných pro oba použité algoritmy a počáteční předzpracování.

7.2.3 Diagram tříd

Pro lepší pochopení interakce jednotlivých tříd a rozdělení programu na obě části předkládáme diagram tříd na obrázku 7.1. Je vidět jasné rozdělení celého programu na dvě části. Třídy `Data` a `Classifier` jsou zcela nezávislé na hlavní třídě `Node`. Jak je vidět, obě výpočetní jádra musí implementovat metodu `run(req,data)` a používají stejnou strukturu `QueryData`, která obsahuje data ve strukturách vhodných pro oba algoritmy.

V třídě `Node` je podstatná hlavní metoda `handle_request(json)`, která zapouzdřuje všechny dílčí akce: transformaci dotazu, klasifikaci, získání dat pro dotaz, spuštění vybraného jádra a návrat transformovaného výsledku ve formátu JSON.



Obrázek 7.1: Diagram tříd zachycující jak interakci jednotlivých tříd mezi sebou, tak i rozdělení tříd do dvou hlavních skupin dle jazyka, ve kterém jsou implementovány.

Kapitola 8

Experimenty a výsledky

V této kapitole popíšeme experimenty, kterými jsme otestovali různé aspekty systému. V první části se věnujeme vyhodnocení úspěšnosti vybraných vylepšení a popisujeme, na čem je úspěšnost založena. V dalších částech se zaměříme na porovnaní obou algoritmů. Ukážeme, na který typ dotazů se hodí které jádro a následně vyhodnotíme klasifikační model, který se o tuto predikci snaží. V poslední části kapitoly pak srovnáme vlastní řešení s existujícím proprietárním řešením firmy, která podobný systém vlastní a navrhneme případná vylepšení, která by mohla dopomoci k lepším výsledkům.

Data, která byla použita jako podklad pro všechny experimenty, pochází od výše zmíněné firmy. Dataset obsahuje 952426415 letů mezi 3122 letišti v časovém rozmezí 13. 4. – 2. 5. 2018. Dotazy, kterými jsme systém testovali, byly taktéž poskytnuty danou firmou a představují reálné uživatelské dotazy na jednosměrné kombinace zaznamenané na webovém portálu. Vybraná množina dotazů byla zbavena příliš složitých dotazů (počet odletových/-příletových letišť ≥ 5, časové rozmezí odletu větší jak týden).

Testování probíhalo na stroji s konfigurací Intel(R) Xeon(R) CPU E5-1650 v3 @ 3.50GHz s 12 CPU, 256GB RAM a operačním systémem Ubuntu 16.04.

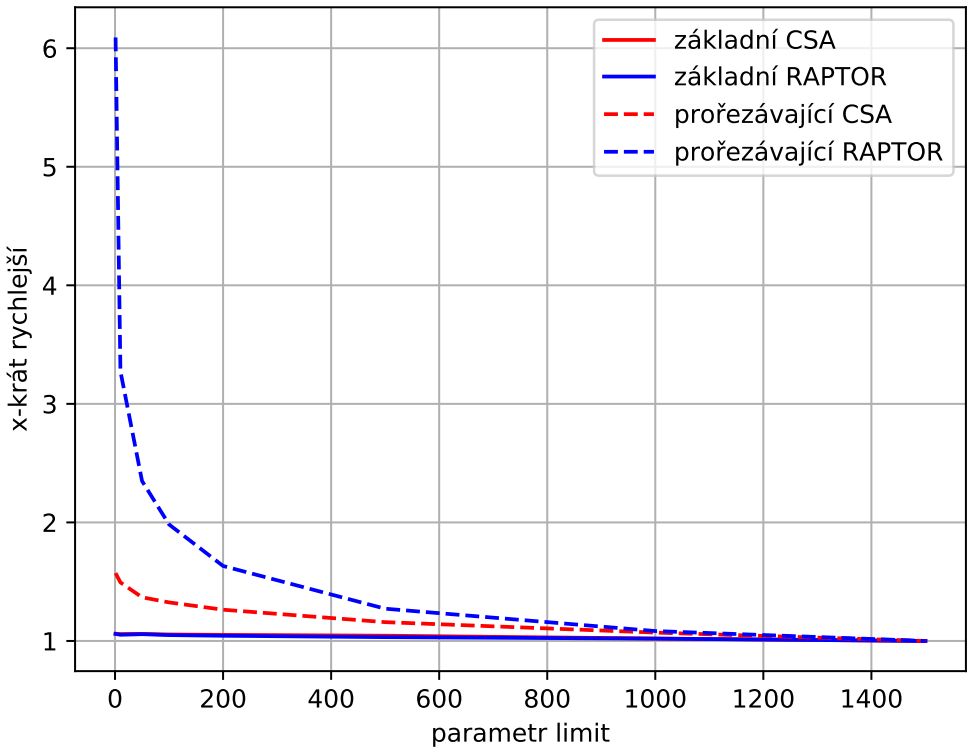
8.1 Vyhodnocení zvolených optimalizací

V této kapitole předvedeme experimenty ukazující úspěšnost jednotlivých vylepšení.

8.1.1 Prořezávání

Jako první vyhodnotíme přínos prořezávání popsaného v kapitole 6.4.3. Jak lze tušit, vylepšení je velmi závislé na parametru *limit* v dotazu, který určuje, kolik výsledků si uživatel přeje vrátit. Čím více výsledků bude povoleno, tím menší je šance, že počet potenciálních výsledků přesáhne tento limit a prořezávání začne účinkovat. Například pokud je počet všech možných kombinací na jedné trase 200, ale limit je nastaven na 300, nikdy k prořezávání nedojde, protože struktura nebyla naplněna.

Experiment jsme vyhodnocovali pro obě výpočetní jádra, pokaždé s i bez prořezávání (tedy čtyři verze celkem). Nejdříve jsme rovnoměrně vybrali 1000 dotazů a každý jsme vyhodnocovali s jiným zadaným parametrem *limit*. Vzorky pro tento parametr jsme brali ze seznamu $[1500, 1000, 500, 200, 100, 50, 10, 1]$. V rámci jedné verze jsme pak agregovali dílčí zrychlení mezi jednotlivými hodnotami parametru a zanesli do grafu. Výsledek je vidět na obrázku 8.1.



Obrázek 8.1: Výsledné zrychlení algoritmu při použití techniky prořezávání v závislosti na parametru *limit* v dotazu. Výsledné zrychlení (hodnoty y) jsou vztažené vůči času zpracovávání při nastavení limitu na 1500.

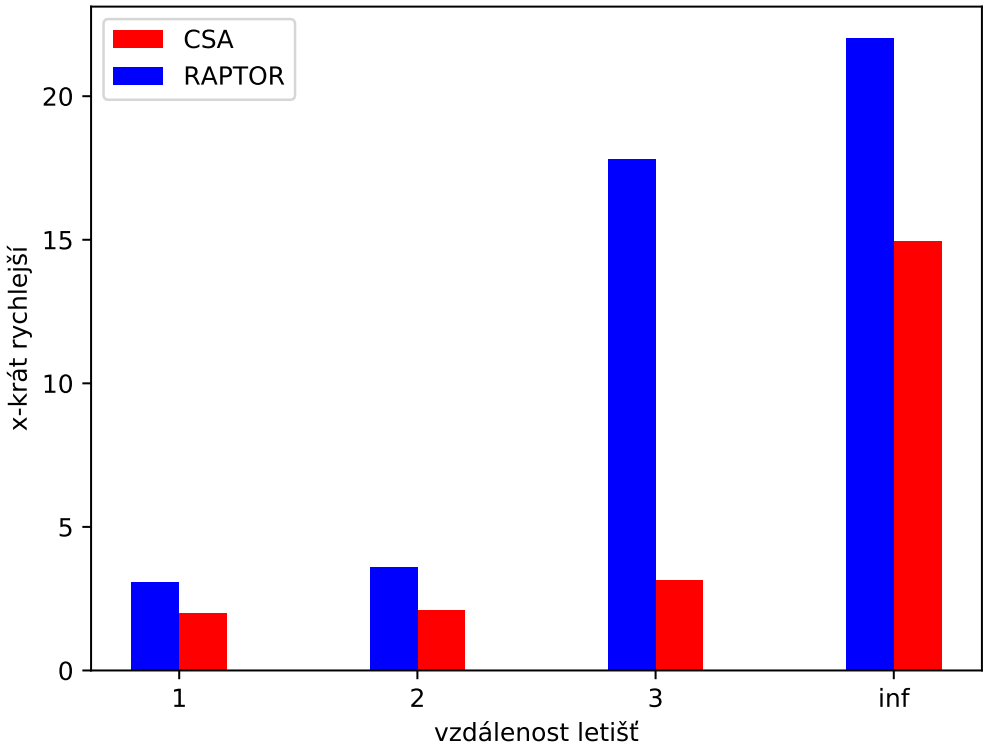
Jak je vidět, bez ořezávání tento parametr nemá téměř žádný dopad na rychlost. Avšak při ořezávání a nižších hodnotám parametru je zrychlení značné. V případě RAPTOR je zrychlení větší než u CSA, neboť neprovedení extenze stávajícího štítku nemá za efekt jen neuložení nové kombinace, ale může znamenat nezařazení nového uzlu mezi seznam aktivních (které se budou znovu rozvíjet).

Vzhledem k tomu, že toto vylepšení neurčuje směr, kudy se prohledávání bude ubírat, není potřeba přemýšlet nad správností tohoto řešení. Výsledky se mazat nemohou a všechny metriky (cena, kvalita, trvání) mají rostoucí charakter. V případě, že by se extenzí již existující kombinace mohla nějaká z metrik zmenšit, nebylo by možné tuto optimalizaci použít.

8.1.2 Redukce prostoru pomocí limitů

Cílem dalšího experimentu bylo vyhodnotit vylepšení provádějící redukci prohledávaného prostoru založené na fixních limitech popsané v kapitole 6.4.1. Při tomto vylepšení se berou v potaz tři limity, které mohou být nastaveny v dotazu (v závorkách jsou uvedeny výchozí hodnoty): maximální počet přestupů (3), maximální délka letu (60 hodin) a maximální cena (bez výchozí hodnoty). V dostupných dotazech je minimum těch, které nastavují tyto hodnoty, budeme tedy nyní brát v potaz pouze výchozí hodnoty. Hned je vidět, že největší

dopad na redukci bude mít eliminace vzdálených míst co se týče počtu přestupů. Cena totiž nemá nastavený žádný limit a málokterá kombinace letů, která trvá více než 60 hodin má méně než 3 přestupy.



Obrázek 8.2: Výsledné zrychlení algoritmů při použití redukce prostoru. Hodnoty na vertikální ose představují poměr kolikrát je vylepšená verze rychlejší než prostý algoritmus. Hodnoty na horizontální ose představují teoretickou nejmenší vzdálenost (co do počtu segmentů) mezi letišti získanou při předzpracování. Mezi destinacemi existovalo také několik dvojic, které nebyly propojeny vůbec, jejich vzdálenost je zobrazena jako *inf*.

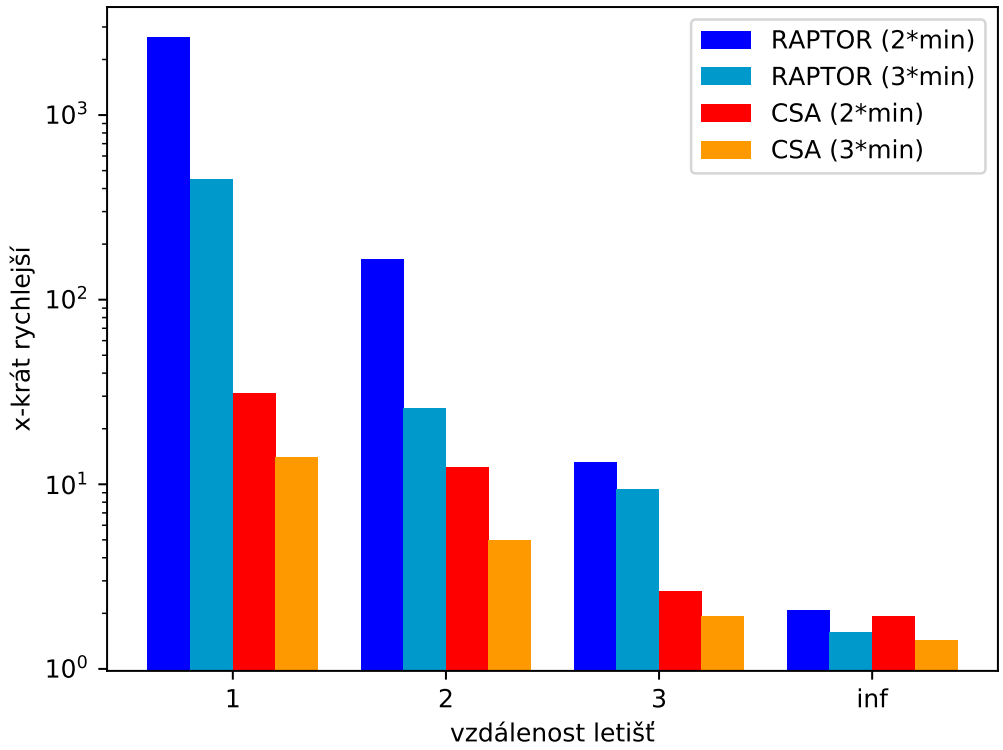
Experiment jsme opět vyhodnocovali pro oba algoritmy v porovnání s verzemi bez jakékoliv optimalizace. Do jádra jsme poslali 1000 dotazů a zaznamenávali dílčí časy, ze kterých jsme získali finální zrychlení. Atribut trasy, který by mohl toto vylepšení ovlivňovat nejvíce, je právě teoretická nejmenší vzdálenost (co do počtu segmentů) mezi výchozí a cílovou destinací. Vynesli jsme proto výsledné zrychlení do grafu v závislosti na této vzdálenosti, viz obrázek 8.2.

První fakt, který je v grafu vidět, je, že RAPTOR reaguje na optimalizaci lépe než CSA. Důvod je podobný jako u předchozího experimentu, a to, že při odříznutí destinace se odříznou i procházení všech letů odlétajících z dané destinace. Dále je vidět, že při hledání cesty mezi vzdálenějšími destinacemi je zrychlení podstatně větší. Čím blíže totiž zadané destinace jsou, tím je prostor, kde může samotné hledání probíhat, určen s menší přesností. Čím vyšší je minimální vzdálenost, tím méně je možných míst, přes která se lze dostat do cíle v rámci omezení na počet segmentů. Toto vylepšení stejně jako předchozí nemění stav výsledku, opět zvládne naleznout všechny, které nalezla neoptimalizovaná verze.

8.1.3 Redukce prostoru pomocí heuristiky

Předchozí vylepšení má i druhý stupeň, a to odřezávání za použití heuristiky. Jde o to, že se zbavujeme míst, přes která když člověk poletí, tak nabude nepřijatelného navýšení některého z kritérií (cena, počet přestupů, trvání letu) vzhledem k nejlepší možné cestě na této trase. Nicméně toto vylepšení již může mít za efekt vynechání některých kombinací za cenu vyšší rychlosti.

Tento experiment probíhal podobně jako předchozí, s tím, že se porovnávalo odřezávání pomocí heuristiky proti odřezávání pomocí limitů. Celý experiment jsme opakovali dvakrát, a to pro různé heuristické funkce: první vracela dvojnásobek minima a druhá trojnásobek.

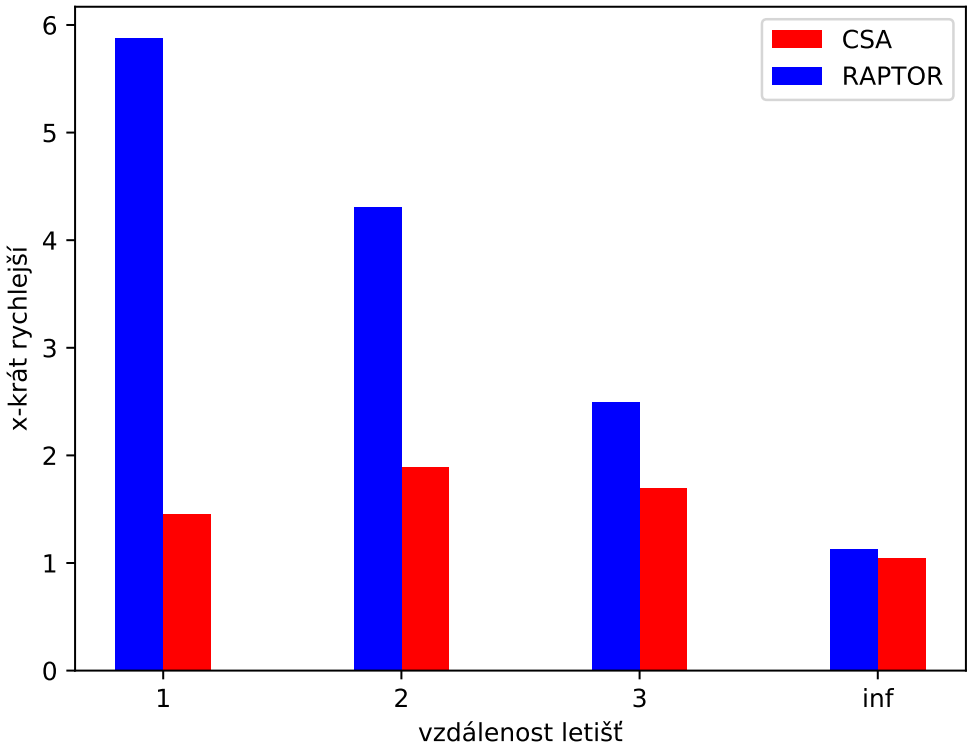


Obrázek 8.3: Výsledné zrychlení algoritmů při použití redukce prostoru s heuristikou. Výsledná zrychlení jsou v poměru k verzi s použitím redukce bez heuristiky. Popis os v grafu je stejný jako u obrázku 8.2.

Znovu je vidět, že RAPTOR je vylepšením ovlivněn více. Dále je vidět naprostý opak trendu vzhledem ke vzdálenosti letišť oproti předchozímu experimentu. Tato optimalizace už skutečně zmenšuje prostor proporcionálně ke vzdálenosti mezi oběma destinacemi. Avšak velkým faktorem zde je právě heuristická funkce, která rozhoduje o odřezávání krajních destinací. Uvádíme proto porovnání dvou heuristických funkcí. V případě tvrdší heuristiky (tolerance maximálně dvojnásobku minima) je zrychlení téměř o řád větší než při použití slabší (trojnásobek minima). Avšak cena za rychlost je určité množství kombinací, které se nenaleznou. V případě tvrdší verze šlo o téměř 20 % dotazů, které nevrátily stejný počet kombinací jako neoptimalizovaná verze, u slabší už jich bylo pouze 10 %.

8.1.4 Dynamické ořezávání pomocí heuristiky

Posledním vylepšením, které bylo implementováno, je dynamické odmítání prodloužených štítků (viz kapitola 6.4.2). V tomto experimentu jsme porovnávali tuto verzi s verzí z předchozího experimentu s tvrdší heuristickou funkcí (neboť je to volné rozšíření). Toto vylepšení má však potenciál být přesnější díky tomu, že v momentě odřezávání má dokonalejší informaci než při statické redukci prostoru. Experiment probíhal stejně jako v předchozím případě.

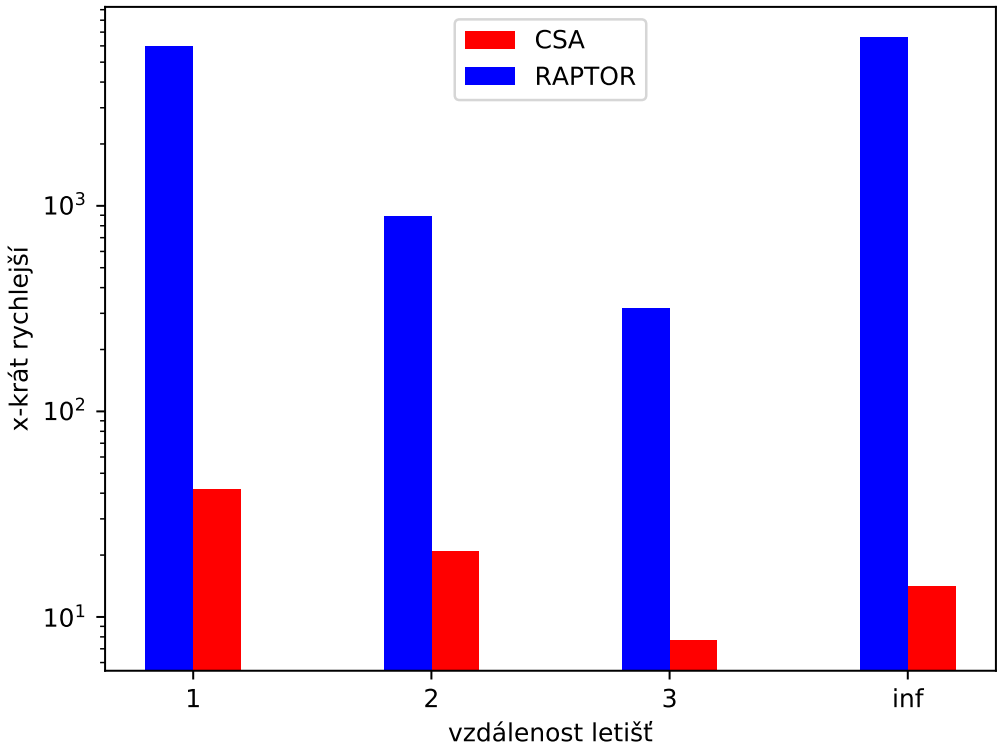


Obrázek 8.4: Výsledné zrychlení algoritmů při použití dynamického ořezávání. Výsledná zrychlení jsou v poměru při použití statické redukce s heuristikou. Popis prvků v grafu je stejný jako u obrázku 8.2.

Z obrázku 8.4 je vidět, že přestože už předchozí optimalizace algoritmy výrazně zrychlila, toto vylepšení zvládlo posunout hranici ještě dál. Opět je vidět stejný trend pro vzdálenost destinací jako v předchozím případě. Podle hodnot pro CSA je možné soudit, že pro tento algoritmus jsme narazili na určitou hranici, kdy algoritmus již nelze výrazně zrychlovat. Nevýhodou této optimalizace je, že výsledky velmi trpí na chybějící kombinace. Více než 50 % odpovědí nevrátilo stejný počet kombinací jako referenční verze. Toto samo o sobě nemusí znamenat horší výsledek, vylepšení mohlo pouze odmítnout velmi špatné kombinace. Nicméně při detailnějším zkoumání výsledných kombinací jsme zjistili, že v téměř 30 % optimalizovaná verze nevrátila stejnou kombinaci s nejnižší cenou.

8.1.5 Spojení všech optimalizací

Náplní posledního experimentu bylo sjednocení všech optimalizací a porovnání se základními verzemi obou algoritmů. Výsledek je vidět na obrázku 8.5. Jak bylo již průběžně vidět, vylepšení měla obrovský přínos zejména pro metodu RAPTOR, nicméně i CSA se povedlo výrazně zrychlit. Odpovědí, které nenalezly všechny kombinace jako základní verze bylo 69 %. Z této části necelá půlka neobsahovala stejnou nejlevnější kombinaci jako v referenční verzi. Závěr z těchto výsledků stejně jako návrhy na zlepšení budou popsány dále v kapitole 8.5.



Obrázek 8.5: Výsledné zrychlení algoritmů při spojení všech optimalizací. Výsledná zrychlení jsou v poměru při použití základních verzí algoritmů. Popis prvků v grafu je stejný jako u obrázku 8.2.

8.2 Srovnání implementovaných metod

Jak bylo popsáno v kapitole 6, zjistili jsme, že ani jedna ze zvolených metod nedominuje nad druhou, ale naopak. Dokonce se nám podařilo zkonstruovat model, který klasifikuje příchozí dotazy a vybírá algoritmus, který bude pro daný dotaz ideální. V této kapitole budou popsány výsledky porovnání obou algoritmů.

Tentokrát jsme vybrali 7500 dotazů a vyhodnotili je oběma jádry. Dále jsme dotaz klasifikovali a do výsledků pro model uložili dobu výpočtu predikovaného jádra. Jako referenční model jsme zvolili takový, který v každém případě zvolí rychlejší jádro. Výsledné časy jsme

seřadili a provedli nad nimi základní analýzu (průměr, medián, 95% percentil, maximum). V tabulce 8.1 jsou vidět výsledky tohoto experimentu.

	avg	median	95 perc	max
CSA	247,78	93,00	947,50	16472,67
RAPTOR	256,23	31,33	1426,08	8378,00
model	178,17	31,84	836,25	8404,33
dokonalý model	165,86	31,33	813,17	8378,00

Tabulka 8.1: Tabulka zobrazující výsledky vzájemného porovnání obou algoritmů a klasifikačního modelu. Jednotlivé buňky ukazují odpovídající časy výpočtu v milisekundách.

První řádek představuje situaci, kdy bychom všechny dotazy vyhodnocovali pomocí CSA, druhý pak, když by se použil RAPTOR. Třetí řádek ukazuje časy při použití našeho modelu - tedy při vybírání predikovaného algoritmu. Poslední řádek pak znázorňuje výběr toho rychlejšího algoritmu ve všech případech.

Z tabulky je vidět, že oba algoritmy jsou neporovnatelné, ani jeden není ve všech metrikách lepší. Podle sloupce median je vidět, že ve většině z případů je RAPTOR až třikrát rychlejší, ale při dotazech, kdy je CSA rychlejší (sloupec 95 % percentil), se absolutní rozdíl rychle zvětšuje.

Velmi pozitivní jsou výsledky klasifikačního modelu. Již z těchto výsledků je dobře vidět, že model klasifikuje dobře a hodnoty výrazně zlepšuje (průměr, 95% percentil), nebo jsou velmi podobné (nízké odchylky u mediánu a maxima).

8.3 Vyhodnocení modelu

Jak již bylo popsáno v kapitole 6.3, jako klasifikační model byl zvolen rozhodovací strom. Při validaci modelu jsme postupovali následovně: nejdříve jsme shromáždili 30000 reálných uživatelských dotazů a časy výpočtů obou algoritmů pro všechny dotazy. Dále jsme promazali ty, kde časy byly velmi podobné (čas jedné verze musel být alespoň o 10 % lepší) a zbytek rozdělili v poměru 3:1 na trénovací a testovací. Model jsme naučili na trénovací části a na testovací jsme zjišťovali přesnost predikce.

Vzhledem k tomu, že predikujeme pouze dvě třídy, můžeme model vyhodnotit jako binární. Z predikovaných pravděpodobností náležitostí k oběma třídám a posouvání určujícího prahu lze získat hodnoty *false positive* a *true positive*, pomocí kterých jsme vykreslili ROC křivku. Výsledná hodnota AUC (angl. *area under curve*), která se používá jako objektivní metrika pro úspěšnost modelu, nám vyšla **0.95** (maximum je 1).

Rysem s největší rozhodovací schopností se ukázalo být počet letišť (**routing_points**), přes která má cenu hledat cestu. Čím menší byl tento počet, tím větší byla šance, že RAPTOR bude rychlejší. Tento fakt opět dokazuje náš předpoklad, že RAPTOR je optimální pro jednodušší dotazy a CSA je robustnější vůči velikosti prohledávaného prostoru.

8.4 Srovnání s existujícím řešením

V předešlých kapitolách jsme vyhodnotili navržená vylepšení, stejně jako samotné algoritmy proti sobě. Nyní se pokusíme implementovaný systém srovnat oproti stávajícímu řešení firmy, která se danou problematikou zabývá. Vzhledem k tomu, že neexistuje jedna

objektivní metrika, která by určovala, které řešení je lepší, předkládáme seznam kvalit (viz tabulka 8.2), které mohou dát detailnější představu o úspěšnosti řešení.

	Náš systém	Firma
Doba předzpracování	< 10 min	4 hod
Medián výpočetního času	31 ms	3 ms
Počet serverů potřebných pro předzpracování	1	20
Počet serverů potřebných pro uložení dat	1	8
Maximální časové rozmezí dat (v rámci počtu serverů)	60 dní	240 dní
Podpora jednoduchých dotazů	✓	✓
Podpora složitých dotazů	✗	✓
Podpora komplexních omezení	✓	✗

Tabulka 8.2: Kvalitativní porovnání našeho systému s existujícím proprietárním systémem.

Naše řešení je celkově mnohem jednodušší oproti stávajícímu řešení. To s sebou nese jak pozitiva, tak i negativa. Mezi pozitiva se řadí především velmi nízký čas předzpracování. Dále systém podporuje volitelné omezení komplexnějšího charakteru, než u stávajícího řešení. Například omezení na určité aerolinky, třídy letu nebo počítání ceny včetně odbaveného zavazadla. Také díky jednoduchosti se nám podařilo snížit potřebné zdroje (jak výpočetní CPU čas, tak i zabranou paměť RAM), které kvantifikujeme počtem serverů.

Na stranu druhou jsme za tyto vylepšení museli zaplatit určitou cenu. Jako první se musí vyzdvihnout neschopnost realizovat složité dotazy (například z kruhové oblasti o průměru 500 km do celého státu, nebo z jednoho letiště do všech letišť po celém světě). S tímto je však počítáno, náš systém vychází z předpokladu, že tento typ dotazů je mnohem vzácnější než běžnější dotaz typu 1:1. Dalším omezením je menší časový rámec, který jsme aktuálně schopni pokrývat. Řešením by však mohla být distribuce datových úseků mezi jednotlivé servery a vytvoření nové vrstvy, která by dotaz přeposílala podle datumu v dotazu. Poslední velkou slabinou našeho systému je stále rychlost. Přestože se nám podařilo mnoha vylepšeními rychlost podstatně zvýšit, referenční systém je stále minimálně o řád rychlejší.

8.5 Zhodnocení výsledků

V předchozích podkapitolách jsme experimentálně porovnali navržený systém. Ukázalo se, že naše vylepšení zvládla několikanásobně zrychlit základní verze algoritmů, pouze v některých případech za cenu nekompletních výsledků. Dalším poznatkem bylo, že implementovaná vylepšení zrychlovala algoritmus RAPTOR podstatně více než CSA.

Přestože se náš systém velmi přiblížil možnosti být zapojen do produkčního prostředí, rychlost stále není dostatečná. Nepočítáme, že by mohl být rychlejší než referenční systém, ale aby se vyplatilo tento systém používat, musí být rychlejší. Jak jsme také ukázali v experimentech o redukci prostoru pomocí heuristiky, systém za rychlost platí nepřesnými výsledky. Dospěli jsme k několika směrům budoucího vývoje, které by mohly dopomoci k vylepšení celého systému.

Přestupní vzory V kapitole 5.3 jsme ukázali, že využití přestupních vzorů nemusí být slepou uličkou. Kvůli složitosti metody jsme se ji rozhodli nezahrnout do našeho porovnání navzdory tomu, že tato metoda skýtá velký potenciál. V první fázi by bylo

potřeba analyzovat, jak často se přestupní vzory mění a jak často by tedy bylo potřeba je znovu přepočítávat. V případě, že by byly relativně neměnné, mohlo by se vyplatit jednou za čas provést náročný výpočet všech vzorů, které by se mohly pro určitá letiště (ta s menším počtem vzorů) využívat.

Cachování výsledků Prostor možných kombinací destinací a dat není natolik rozmanitý, aby se uživatelské dotazy neopakovaly. Naše zběžná analýza databáze dotazů ukázala, že během časového období jedné hodiny téměř 10 % dotazů již bylo před méně než hodinou vyhodnoceno. V případě, že by šlo o těžší dotazy, cachování výsledků by mohlo dopomoci k lepšímu výkonu.

Předpočítání části cest V kapitole 5.2 jsme přiblížili hustotu grafu a vyjádřili předpoklad, že jádro není na první pohled tak malé, aby bylo jisté, že bude možné použít metodu přestupních bodů. Bylo by však možné metodu využít jen z části, například předpočítat cesty pouze uvnitř jádra a cesty z a do jádra plánovat dynamicky. Předpokladem je však podrobnější výzkum a výpočet, kolik by takových uložených cest bylo.

Zpřesnění heuristické funkce V experimentu popsaném v kapitole 8.1.3 jsme ukázali, jaký vliv má horní limit tolerance na zrychlení algoritmu a úplnost výsledků. Vždy jsme určovali limity společně pro všechna kritéria (cena, trvání letu, počet přestupů), nicméně tato kritéria nemusí mít stejnou váhu. Například dvojnásobný nárůst počtu přestupů nemusí být tak dramatický jako dvojnásobná cena. Z referenčních výsledků by šlo analyzovat rozptyl hodnot jednotlivých kritérií a dle toho určit limit pro heuristické funkce.

Segmentace časového období při předzpracování Ve fázi předzpracování se momentálně vytvoří pouze jeden statický graf reprezentující spodní odhady cen vzdáleností mezi letišti. V současné situaci je tedy možné, aby existoval velmi dobrý let (levný, přímý, ...), který odlétá až za několik měsíců, ale stanovoval velmi nízký limit pro celý časový rámec. Avšak je nepopiratelné, že lety odlétající v nejbližším týdnu budou s velkou pravděpodobností mnohokrát dražší než ty odlétající za několik měsíců. Tímto se dostáváme do situace, kdy je nastaveno velmi tvrdé síto a může se stát, že kvůli heuristice odmítneme více výsledků. Možné vyřešení této situace by bylo rozdělit předzpracování po několika vzájemně se překrývajících intervalech (např. týdnech).

Kapitola 9

Závěr

V této práci byl popsán komplexní problém hledání optimálních cest v hromadné dopravě se zaměřením na leteckou dopravu. Z popsaných metod byly v souladu s analýzou grafu letů vybrány dva moderní algoritmy, CSA a RAPTOR, které představují jádro navrženého systému. Dále byla implementována různá vylepšení využívající strukturu leteckého grafu. Výsledný systém funguje jako serverová aplikace, která je schopna udržovat data o letech a odpovídat na uživatelské dotazy. Ve chvíli, kdy přijde uživatelský dotaz na naplánování cesty, systém dotaz klasifikuje a vybere vhodnější z obou algoritmů, který požadované cesty vyhledá.

Podkladem pro veškeré testování byly reálné lety a uživatelské dotazy získané od spolupracující firmy. Nejdříve byla vyhodnocena úspěšnost navržených optimalizací. Experimenty ukázaly, že vylepšení týkající se redukce prohledávaného prostoru ve své základní formě zrychlily původní verze algoritmů až 15x bez negativních dopadů na kvalitu výsledků. Podstatou dalšího vylepšení bylo zamítnutí cesty ve chvíli, kdy by výsledkem byla nevýhodná kombinace. Toto vylepšení zrychlilo algoritmy ještě výrazněji, avšak za cenu nenalezení všech potenciálních výsledků. Dále byly porovnávány oba algoritmy mezi sebou. Přestože se ukázalo, že algoritmus RAPTOR reaguje mnohem lépe na veškerá vylepšení, nelze stoprocentně říci, že by byl pro výraznou část dotazů rychlejší než algoritmus CSA.

Při porovnání systému s existujícím řešením výše zmíněné firmy již nešlo výsledek určit jednoznačně. Systém této složitosti obsahuje mnoho vlastností a zpravidla nelze mít takový, který by splňoval všechny požadavky. Tento systém však cílil na velkou flexibilitu za cenu neschopnosti pokrýt všechny uživatelské dotazy. Čas vyžadovaný pro předzpracování vstupních dat, který navržený systém potřebuje, je skutečně o mnoho nižší než u porovnávaného řešení (zrychlení z řádu hodin do řádu minut). Podařilo se také zmenšit potřebné zdroje pro předzpracování a uložení dat. Na druhou stranu je tento systém o řád pomalejší a není schopen vyhodnocovat složitější dotazy, které jsou ovšem mnohem vzácnější než ty jednoduché.

Aby byl systém připraven pro použití v produkčním prostředí, je potřeba jej ještě vylepšit z hlediska rychlosti i přesnosti. V závěru kapitoly 8 byly popsány možné směry budoucího vývoje, mezi kterými je jak vyzkoušení jiných metod (přestupní vzory), tak optimalizace současných vylepšení. Při zdokonalování současného řešení půjde především o zpřesnění heuristické funkce, která určuje, který výsledek může být ještě atraktivní pro uživatele. Toho může být docíleno podrobnější analýzou databáze uchovávající informace o prodeji letenek.

Literatura

[1] Abraham, I.; Delling, D.; Goldberg, A. V.; aj.: A hub-based labeling algorithm for shortest paths in road networks. In *International Symposium on Experimental Algorithms*, Springer, 2011, s. 230–241.

[2] Antsfeld, L.; Walsh, T.: Finding multi-criteria optimal paths in multi-modal public transportation networks using the transit algorithm. In *Proceedings of the 19th ITS World Congress*, 2012, str. 32.

[3] Bast, H.: Car or public transport—two worlds. In *Efficient Algorithms*, Springer, 2009, s. 355–367.

[4] Bast, H.; Carlsson, E.; Eigenwillig, A.; aj.: Fast routing in very large public transportation networks using transfer patterns. In *European Symposium on Algorithms*, Springer, 2010, s. 290–301.

[5] Bast, H.; Delling, D.; Goldberg, A.; aj.: Route planning in transportation networks. In *Algorithm Engineering*, Springer, 2016, s. 19–80.

[6] Bast, H.; Funke, S.; Matijević, D.: Transit: ultrafast shortest-path queries with linear-time preprocessing. In *9th DIMACS Implementation Challenge—Shortest Path*, 2006.

[7] Bast, H.; Funke, S.; Sanders, P.; aj.: Fast routing in road networks with transit nodes. *Science*, ročník 316, č. 5824, 2007: s. 566–566.

[8] Brodal, G. S.; Jacob, R.: Time-dependent networks as models to achieve fast exact time-table queries. *Electronic Notes in Theoretical Computer Science*, ročník 92, 2004: s. 3–15.

[9] Delling, D.; Pajor, T.; Wagner, D.; aj.: Efficient route planning in flight networks. In *OASIcs-OpenAccess Series in Informatics*, ročník 12, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2009.

[10] Delling, D.; Pajor, T.; Werneck, R. F.: Round-based public transit routing. *Transportation Science*, ročník 49, č. 3, 2014: s. 591–604.

[11] Delling, D.; Sanders, P.; Schultes, D.; aj.: Engineering Route Planning Algorithms. *Algorithmics of large and complex networks*, ročník 5515, 2009: s. 117–139.

[12] Dibbelt, J.; Pajor, T.; Strasser, B.; aj.: Intriguingly simple and fast transit routing. In *International Symposium on Experimental Algorithms*, Springer, 2013, s. 43–54.

[13] Fudenberg, D.; Tirole, J.: Game theory. Technická zpráva, The MIT press, 1991.

[14] Geisberger, R.: Advanced route planning in transportation networks. *Diss. Karlsruher Instituts fr Technologie*, 2011.

[15] Geisberger, R.; Sanders, P.; Schultes, D.; aj.: Contraction hierarchies: Faster and simpler hierarchical routing in road networks. *Experimental Algorithms*, 2008: s. 319–333.

[16] Goldberg, A. V.; Harrelson, C.: Computing the shortest path: A search meets graph theory. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, Society for Industrial and Applied Mathematics, 2005, s. 156–165.

[17] Goldberg, A. V.; Werneck, R. F. F.: Computing Point-to-Point Shortest Paths from External Memory. In *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments (ALENEX'05)*, SIAM, 2005, s. 26–40.

[18] Schulz, F.; Wagner, D.; Weihe, K.: Dijkstra's algorithm on-line: an empirical case study from public railroad transport. *Journal of Experimental Algorithmics (JEA)*, ročník 5, 2000: str. 12.

Příloha A

Obsah CD

Přiložené CD obsahuje:

- `src/`: obsahuje zdrojové kódy rozdělené do dílčích složek:
  - `combain/`: zdrojové kódy pro část programu napsanou v jazyce Python
  - `include/`: hlavičkové soubory a implementace šablon v C++
  - `src/`: zdrojové kódy pro část programu v jazyce C++
- `doc/`: obsahuje zdrojové kódy v jazyce LaTeX pro tuto diplomovou práci
- `README.txt`: informační soubor blíže popisující přiložené soubory
- `xsychr05-dp.pdf`: diplomová práce ve formátu PDF