Jízdní řády

Lukáš Riedel

Specifikace

Obsah

Poznámka: Text psaný *kurzívou* je velmi hrubý popis částí softwaru, které budou implementovány až v rámci bakalářské práce. V ročníkovém projektu (respektive zápočtových programech na C++ a C#) tedy tyto části zahrnuté nebudou.

1 Úvod do problému

Cílem projektu je navrhnout a implementovat sadu aplikací, přičemž uživatel bude koncový software vnímat pouze jako jednu aplikaci. Aplikace, která bude mít user-friendly grafické rozhraní a jejíž cílová platforma bude Win32, bude uživateli v offline režimu umožňovat vyhledávání spojení v rámci Pražské Integrované Dopravy a získávat informace o odjezdech z daných zastávek. Po přechodu do online režimu navíc bude moci sledovat aktuální situaci v provozu, tzn. mimořádnosti v dopravě a informace o výlukách. Online režim bude nutný pro občasnou aktualizaci jízdních řádů, kterou bude aplikace provádět sama v pravidelných intervalech (jednou za několik dnů), popřípadě, pokud dojde k překročení tohoto intervalu, kdykoliv se cílový stroj připojí k internetu. Jízdní řády budou platné na několik dnů dopředu, jakmile se bude blížit jejich expirace, uživatel bude s touto skutečností obeznámen. *Do budoucna se počítá i s vytvořením mobilní aplikace pro cílovou platformu Android, která bude pracovat online. Bude spolupracovat s výše zmíněnou aplikací tím způsobem, že v mobilní aplikaci nebude téměř žádná logika, vše potřebné bude obstarávat desktopová aplikace, mobilní aplikace bude sloužit zejména pro zadávání a zobrazování výsledků z desktopové aplikace a bude s ni komunikovat po síti prostřednictvím vhodně navrženého protokolu. Nicméně bude možnost přejít do offline módu tím způsobem, že si uživatel bude moci stáhnout jízdní řády pro jeho oblíbenou trasu, které budou dostupné na několik dní dopředu, a bude mít tedy možnost vyhledávat spojení offline.* Součástí bude webová prezentace produktu napsaná v HTML5 a CSS3, kde budou všechny důležité informace.

1.1 Dostupné zdroje

Jelikož je aplikace cílená pro Pražskou Integrovanou Dopravu, jsme nuceni jako hlavní zdroj dat použít informace o jízdních řádech v GTFS[1] formátu. Tento formát, navržený společností Google, je velmi variabilní. Cílem je optimalizovat řešení pro Prahu, nicméně aplikace také poskytne uživateli možnost jednoduše změnit cílové město - jízdní řády v tomtu formátu jsou totiž poskytovány i pro jiná velká (nejen) evropská města, například Londýn, Paříž nebo Řím. Algoritmů na hledání v jízdních řádech je celá řada, většina z nich používá nějakým způsobem modifikovaný Dijkstrův algoritmus. Toto řešení bude využívat RAPTOR [2] algoritmus, vyvinutý společností Microsoft, který nepotřebuje prioritní frontu a je dostatečně rychlý a efektivní. Algoritmus pracuje v kolech, kde v k-tém kole najde nejrychlejší spojení do všech zastávek, do nichž se můžeme dostat pomocí $k-1$ přestupů. Z toho plyne, že nehledáme spojení pouze do jedné cílové zastávky, nicméně do všech zastávek v grafu dopravní sítě. To ale nevadí, protože časová složitost zůstane stále velmi dobrá (lineární vůči trasám všech linek, každou procházíme pouze jednou) a spotřeba paměti nebude na dnešní poměry také nijak zásadní, neboť různá spojení rekonstruujeme pouze třemi pointery - odkaz na jízdu jedné konkrétní linky (tzv. trip), odkaz na nástupní stanici v daném tripu a odkaz na výstupní stanici v daném tripu. To znamená, že jedno spojení o k přestupech spotřebuje (bez

overheadů všech použitých kontejnerů a tříd) pouze $24k$ bytů paměti na x64 platformě, $12k$ bytů paměti na x86 platformě. Ze software-inženýrského hlediska bude možnost jednoduše změnit vstupní (GTFS) formát, a sice přepsáním pouze jedné části aplikace, která se stará o parsování do jakéhosi mezi-formátu. Vše ostatní nebude třeba modifikovat.

2 Architektura desktopové aplikace

Cílová platforma aplikace bude Win32. Hlavní jádro aplikace bude psané v C++, vše ostatní v C#. Aplikace bude, co se externích knihoven týče pouze funkce z STL (C++) a BCL (C#) (!!! možná ještě boost pro reprezentaci datetime, ale to -momentálně ještě nevím, to ukážou micro-benchmarky, co bude rychlejší). Aplikace by se dala rozdělit na 4 části.

2.1 Timetables data preprocessor

Tato část aplikace bude napsaná v C#, ve finální aplikaci bude představovat jediné vlákno. Spustí se při prvním spuštění aplikace na daném stroji a následně v pravidelných intervalech, kdy bude k dispozici připojení k internetu a kdy aplikace nebude moc využívaná (jednou za několik dní, nebo pokaždé chvíli před expirací dat) kvůli aktualizacím dat. Bude kontrolovat validitu dat a její činnost by se dala rozdělit na tři části:

1. Stahování GTFS dat z předem navoleného zdroje dat a jejich následnou dekomprimaci. Z toho plyne, že podmínkou pro zdroj dat je ten, aby data byla v zip formátu. Předpokládá se, že URL ke zdroji dat nebude zadrátováno do kódu, ale bude možnost ho nastavit ručně. Předpoklad je takový, že URL ke zdroji dat bude v nějakém (pro to určeném) textovém souboru, tudíž nebude možnost měnit jízdní řády interaktivně přímo z aplikace. Tento princip dá pokročilejšímu uživateli svobodu zvolit si, v rámci jakého města bude aplikace pracovat, a neznalému uživateli nedá možnost aplikaci rozbít. (!!! možná toto částečně zahrnu do nastavení aplikace +++ mám to udělat tak, že bude možnost zadat více zdrojů dat, která se následně zmergují??? že by se v rámci jednoho spuštění aplikace daly vyhledávat jízdní řády třeba pro Prahu a pro Paříž zároveň, nebo by se daly zmergovat třeba data pro Prahu, Plzeň, Brno, Ostravu a do toho přidat data od ČD, jestli daná města a ČD data vůbec zveřejňují, to nevím, byl to jen příklad...)

2. Jak již bylo řečeno, GTFS formát je velmi variabilní. Některá povinná data jsou pro náš účel zbytečná, stejně tak se najdou nepovinná data, která potřebujeme. Příkladem nepovinných dat jsou například časy přestupů mezi jednotlivými zastávkami (tzv. footpaths). Tato část aplikace bude obecný GTFS formát přizpůsobovat tak, aby se spojení mohla vyhledávat dostatečně rychle. To znamená, že pokud bude soubor obsahující footpaths chybět, dojde k jejich dopočítání na základě GPS souřadnic zastávek. Předpokládá se, že vzhledem k tomu, že pomocí GPS souřadnic můžeme určit pouze vzdušnou vzdálenost mezi dvěma souřadnicemi, tak bude zvolena dostatečně malá průměrná rychlost chůze s tím, že budeme uvažovat pouze footpaths s dobou chůze do 10-15 minut. Dalším příkladem činnosti této části může být například to, že v GTFS

formátu jsou (povinně) uvedeny pouze údaje o jednotlivých nástupištích. Uživatel typicky nechce vyhledávat odjezdy pouze z jednoho nástupiště, ale ze všech nástupišť v rámci jedné zastávky (stanice). Aplikace tedy projde všechny nástupiště a na základě přesné shody jmen vytvoří jednotlivé zastávky (stanice) sdružující všechny nástupiště, které pod ně spadají. Do třetice uvedeme fakt, že aplikace vhodně předpočítá různé indexy a uvede data do stavu, abychom k požadovaným datům po načtení do paměti mohli efektivně přistupovat a aby nedocházelo k žádným memory leakům. Stručně řečeno, tato část načte obecná GTFS data do paměti a výstupem aplikace budou data (opět soubory), která budou plně přizpůsobena pro činnost hlavní aplikace. Předpokládá se, že na průměrně výkonném stroji poběží tato aplikace (vlákno) nejvýše 20 sekund (+ čas potřebný na stažení zip archivu) jednou za tři dny.

3. Smazání zip archivu a GTFS databáze. Tyto soubory již nebudeme potřebovat, budeme mít vytvořené vlastní.

Důvod pro zvolení C# je takový, že vzhledem k povaze aplikace není třeba takový výkon, aby musela být psaná v C++. Navíc, jelikož bude hlavní část aplikace napsaná rovněž C#, dojde ke snazšímu provázání těchto dvou aplikací. V neposlední řadě můžeme zmínit, že dekomprimace souborů lze provádět přímo z .NET kódu, čímž zlepšíme přenositelnost kódu - totiž v C++ žádná funkce na dekomprimaci souborů není a práce se soubory je tam rovněž náročnější. Na code-readability a přenositelnosti zde záleží více, když není potřeba žádný extrémní výkon. Výhoda této aplikace je rovněž ta, že ušetříme čas načítání hlavní aplikace (předpoklad zrychlení je 300%), potažmo i něco málo z paměti (!!! ono by to šlo udělat i tak, aby aplikace nevyužívala žádnou paměť a data brala z nějaké databáze, nicméně přijde mi vhodnější to mít v paměti, když na server můžou z mobilní aplikace chodit desítky požadavků za sekundu - zde jde hlavně o rychlost routovacího algoritmu a maximální rychlosti dosáhneme jen tehdy, když budeme mít data v paměti). Popřípadě, kdybychom chtěli aplikaci uzpůsobit i jiným formátům, než je GTFS, stačilo by upravit pouze tuto aplikaci a do žádné jiné bychom vůbec nemuseli zasahovat.

Během sepisováním této specifikace došlo k microbenchmarkům na datech pro Prahu u obou přístupů, a sice načítání dat pomocí knihovny z meziformátu a pak přímo z GTFS formátu. Samotný GTFS formát zabírá na disku 105 MB, zatímco meziformát pouze 43 MB. Po načtení GTFS formátu do paměti (v x86 režimu) aplikace využívá 160 MB, zatímco po načtení meziformátu pouze 120 MB - je to dáno tím, že můžeme využívat efektivnější datové struktury (kvůli předpřipravené indexaci můžeme využít vector místo mapy, nějaké struktury můžeme úplně vypustit, nemusíme si například pamatovat GPS souřadnice zastávek...). Ještě uvedeme, že čas inicializace dat z GTFS formátu je 12 sekund, zatímco čas inicializace z meziformátu pouze 3,1 sekundy - to je proto, že si spoustu věcí předpočítáme již do meziformátu, ušetříme počet alokací díky rezervaci místa ve vectorech atd. Zbývá podotknout, že čas přípravy meziformátu je společně se stahováním GTFS dat zhruba půl minuta.

Až bude mobilní aplikace, bude tato část doplněna o preprocessing základních dat pro mobilní aplikaci. Bude generovat pouze údaje o zastávkách, kde se budou tentokrát držet i GPS souřadnice - kvůli určení nejbližších zastávek na základě GPS polohy. A u každé zastávky budou vypsány linky, které jimi projíždí... Pro Prahu to bude nějakých 250 kB.

2.2 Timetables core library

Tato část aplikace bude napsaná v C++ a bude plně optimalizovaná na výkon. Důvod pro zvolení C++ je tedy zřejmý. Aplikace bude zahrnovat dva základní druhy požadavků:

1. Zobrazení prvních n odjezdů z dané zastávky v daný datum a čas. K tomu asi není co víc dodávat, najde se zastávka odpovídající danému jménu, která bude obsahovat reference na všechny nástupiště, které pod ni spadají. Každé nástupiště v sobě obsahuje údaje o všech odjezdech všech linek za jeden den. Při procházení odjezdů je nutné dbát, jestli je odjezd v daný datum relevantní, neboť o víkendy, státní svátky atd. platí jiné jízdní řády než ve všední dny. I na toto se pamatuje, totiž každý trip obsahuje položku, kdy daný trip operuje (tzv. services). Najde se prvních n odjezdů splňující kritéria a vrátí se ve formě kontejneru obsahující nějakou user-friendly datovou strukturu, jejíž obsah půjde snadno zobrazit z GUI.

2. Nalezení n nejrychlejších spojení mezi dvěma zastávkami v daný datum a čas, přičemž bude možnost navolit maximální počet přestupů a zvolit si průjezdné body. K tomu se bude využívat RAPTOR [2] algoritmus, o němž již byla řeč. Slušelo by se podotknout, že z testování společností Microsoft vyplynulo, že při vhodné implementaci a vhodných strukturách dokáže tento algoritmus nalézt libovolné spojení na grafu Londýna do 8 milisekund. Výstup bude podobný jako v posledním případě.

Předpoklad je takový, že tato část bude s hlavní částí aplikace (myšleno s tou částí s GUI) provázána pomocí interoperability C++/CLI se C#. Tato část bude sloužit pouze k počítání výsledků, veškerá logika bude napsaná v C#. V C# se bude jednat o jedno vlákno, které v sobě bude mít nekonečný cyklus, který bude zpracovávat požadavky od uživatele z GUI (!!! nebo to mám udělat tak, že bude jenom síť a požadavky z GUI budou přicházet z localhostu 127.0.0.1 ???) *a později i ze sítě, až bude hotová mobilní aplikace* a bude je umisťovat do dvou front. Fronty budou dvě, a sice pro zobrazení odjezdů ze zastávek (tzv. departure board) a vyhledání spojení (tzv. router). Každá fronta bude mít defaultně přiřazené jedno vlákno, které bude zpracovávat požadavky v ní. *V případě velkého množství požadavků se spustí (popřípadě i více) nové vlákno a požadavky se budou rovnoměrně rozdělovat. Toto se týká pouze požadavků ze sítě, předpokládá se, že z desktopové aplikace přijde pokaždé právě jeden požadavek.* (!!! Toto je velmi předčasné, o vláknech toho zatím moc nevím, tak uvidíme po přednáškách z C++ a .NET I, co se tam dozvím... Možná bude lepší na každý požadavek startovat nové vlákno, nevím...) Zpracovávání požadavků bude vypadat zhruba takto, bez újmy na obecnosti uveďme příklad pro departure board:

1. Přijde požadavek na zobrazení N odjezdů ze zastávky s názvem X v čas Y. Nekonečný cyklus to zaznamená, vytvoří objekt DepartureBoard s argumenty N, X, Y a odkazem na data. V konstruktoru dojde k inicializaci objektu, tzn. že se nastaví čas Y, počet odjezdů N a v datech se najde odkaz na zastávku s názvem X. *Pokud se jednalo o požadavek ze sítě, rovněž se uloží odchozí brána, IP adresa příjemce, cílový port a další údaje potřebné pro síť.* Pokud inicializace selže, například se nepodaří nalézt zastávku, požadavek se zruší, respektive se přesune do výstupní fronty, a bude si v sobě držet chybový stav, aby mohl informovat uživatele o příčině neúspěchu. V případě úspěšné inicializace se objekt zařadí do vstupní fronty.

2. Požadavek je ve vstupní frontě a přijde na něj řada. Zpracovávací vlákno na něm zavolá proceduru bez návratové hodnoty ObtainDepartureBoard. Tato metoda je noexcept, požadavek se po provedení této metody za všech okolností zařadí do výstupní fronty. Zpracovávaný objekt se rozšířil o časy odjezdů, které jsou uloženy v paměti ve formě vhodných kontejnerů (viz. výše). Nebo nerozšířil, když nebyl nalezený žádný odjezd z dané zastávky.

3. Požadavek je ve výstupní frontě a přijde na něj řada. Data spočítaná v předchozím bodě se odešlou na patřičný výstup (tedy *síť nebo* GUI).

Deklarace třídy DepartureBoard tedy může vypadat například takto.

```
class DepartureBoard {
private:
  std::vector<Timetables::Structures::Departure> foundDepartures;
  const Timetables::Structures::Station& station;
  const Timetables::Structures::DateTime& earliestDeparture;
  const std::size_t count;
public:
  DepartureBoard(const Timetables::Structures::DataFeed& feed,
  const std::wstring& stationName, const Timetables::Structures::DateTime&
  earliestDeparture, const size_t count) : earliestDeparture(earliestDeparture),
  count(count), station(feed.Stations().Find(stationName)) {}

  void ObtainDepartureBoard();

  inline const std::vector<Timetables::Structures::Departure>& ShowDepartureBoard()
  { return foundDepartures; }
};
```

Pro třídu Router to bude dosti podobné.

2.3 Timetables main application

Aplikace bude zcela jistě naprogramována v C# a bude využívat buď technologie WinForms nebo WPF (!!! nevím, co bude lepší, předmět UI .NET mám zapsaný, tak uvidím - zatím znám jen WinForms). Bude nějakým (již popsaným) způsobem obsahovat obě výše zmíněné aplikace. Aplikace bude uživateli poskytovat příjemné grafické rozhraní, kde jak pro zobrazování odjezdů, tak pro vyhledávání spojení, bude k dispozici "našeptávač"pro nalezení zastávky. Bude k dispozici hezky formátovaný RSS feed pro mimořádnosti v dopravě a informace o výlukách, který bude k dispozici pouze v online módu. Pokud bude aplikace v offline módu a bude vyžadována aktualizace dat, aplikace na tuto skutečnost upozorní. *V online módu se bude chovat jako server pro mobilní aplikaci.* V nastavení bude možnost zvolit si barevný motiv (předpokládají se základní barvy) *a do budoucna zde bude i možnost lokalizace aplikace, tzn. přepínání mezi anglickým a českým jazykem.*

2.4 Timetables mobile data preprocessor

Toto bude rozšíření desktopové aplikace. Aplikace bude v C# a v rámci hlavní aplikace se bude jednat o jediné vlákno. Vstupem bude požadavek na vytvoření dat pro jednu trasu. Aplikace

spustí vyhledávání spojení, typicky by se měla vyhledat spojení ve všech dnech v týdnu, ve všech denních dobách a i nějaké alternativy. Na základě výsledků tato aplikace vytvoří data pro danou trasu - na základě vygrepování všeho potřebného z hlavních dat, aby se na těchto datech mohl routovací algoritmus spouštět a vždycky (v každý den, každou denní dobu) mezi dvěma zadanými zastávkami nalezl několik optimálních spojení. Tato data se následně po síti pošlou do mobilní aplikace. Opět se předběžně počítá s nějakou frontou, aby bylo možné, v případě velkého množství požadavků, data nějak uspokojivě zpracovávat a na požadavky odpovídat.

3 Architektura mobilní aplikace

Aplikace napsaná v C# pro cílovou platformu Android bude velmi podobná jako ta desktopová s tím rozdílem, že bude obsahovat pouze aplikaci popsanou v sekci 2.3. To znamená, že veškeré požadavky na zobrazení informací o zastávkách nebo spojeních se budou řešit online a to tím způsobem, že prostřednictvím vhodně navrženého protokolu, předběžně se počítá s aplikačním protokolem nad TCP, pracujícím na nějakém volném portu s komunikací (návratové kódy) podobné protokolu HTTP, bude posílat data do desktopové aplikace, kde se budou vyhledávat spojení a po síti se vracet zpět do aplikace mobilní. Mobilní aplikace bude v její základní verzi pouze takový zobrazovač, prohlížeč... Bude také obsahovat našeptávač, RSS feed i nastavení - stejně jako desktopová aplikace.
Bude zde i možnost uložit si oblíbené trasy, které se budou moci vyhledávat offline. To bude velmi pravděpodobně realizováno tak, že se po síti pošle požadavek do desktopové aplikace a aplikace ze sekce 2.4 ho zpracuje. Nalezne nejvhodnější spojení i nějaké alternativy, z dat vygrepuje nutné údaje o odjezdech z potřebných zastávek a ve formátu ze sekce 2.1 je pošle do mobilní aplikace. Výhodou tohoto přístupu je fakt, že kompletní data jsou poměrně hutná, pro Prahu mají zhruba 100 MB. Pokud se vygrepuje pouze to potřebné pro jednu trasu, budeme se pohybovat ve velikostech jednotek MB (spíše stovky kB), možná i méně. V mobilní aplikaci, pokaždé když budeme chtít vyhledat nějaké oblíbené spojení, se on-demand načtou tato data do paměti a spustí se odlehčená verze routovacího algoritmu, který bude zahrnut přímo v mobilní aplikaci a bude velmi pravděpodobně přepsán do C#. Jelikož budeme procházet velmi odlehčený graf, většinou to bude jen cesta, strom nebo graf s hodně málo cykly, bude C# výkonnostně dostačující a vyhneme se problémům s interoperabilitou. I tato data budou mít nějaký datum expirace, po expiraci dat již nebude oblíbená trasa k dispozici a bude nutné data zaktualizovat. Na to bude připravené tlačítko, které zařídí vše potřebné. Popřípadě se bude moci nastavit automatická aktualizace oblíbených tras...

4 Návrh datových struktur

4.1 Datové struktury vytvořené preprocessingem dat

Data se z GTFS formátu načtou do paměti v přirozené formě. Musíme tak činit sekvenčně, neboť na sobě jednotlivá data různě závisí. GTFS formát je velmi obecný, pro nás mají význam pouze tyto položky:

1. Calendar - Obsahuje údaje o jednotlivých services. Díky tomu víme, kdy daný trip operuje. Každá service je indexovaná svým ID. Dále obsahuje binární hodnoty od pondělí do neděle, kde 1 signalizuje, že daná service v daný den operuje a 0 analogicky pro opačný případ. Můžeme reprezentovat buď polem boolovských proměnných, nebo pomocí bitů například v charové proměnné. Jelikož je počet všech services poměrně malý (většinou max 100-200), obětujeme 1 kB paměti pro reprezentaci v poli, abychom dosáhli lepší code-readability. Rovněž zde nalezneme dvě data, která nám udávají, v jakém rozmezí jsou jízdní řády platné.

2. Calendar Dates - Jeden záznam v tomto souboru nám říká něco o případných mimořádnostech v provozu dané service. Vždycky musí obsahovat ID dané service, datum mimořádnosti a typ mimořádnosti. Typ bude v našem případě pouze přidána/odebrána v daný datum, proto to můžeme reprezentovat boolovskou proměnnou. Při čtení tohoto souboru nevytváříme žádnou novou strukturu, pouze rozšiřujeme strukturu vytvořenou v předchozím bodě o mimořádnosti.

3. Routes - Obsahuje nejstručnější informace o provozu jednotlivých linek, jako je krátký název dané línky (například A, 22, 135...), dlouhý název linky (například Depo Hostivař - Nemocnice Motol), barvu linky v HEX formátu a typ linky, přičemž my si vystačíme s osmi typy (Tram = 0, Subway = 1, Rail = 2, Bus = 3, Ship = 4, CableCar = 5, Gondola = 6, Funicular = 7). Každá linka je rovněž indexována nějakým stringovým identifikátorem, proto je pro linky nejvhodnější strukturou unordered map/Dictionary, zatímco pro services to může být vector/List.

4. Stops - Obsahuje údaje o jednotlivých nástupištích, indexace pomocí stringů. Pro nás hrají roli pouze název zastávky a GPS souřadnice, konkrétně zeměpisná šířka délka. Opět by bylo vhodné uchovávat si informace v unordered mapě/Dictionary. Při čtení dat je nutné dbát položku o typu lokace, kde uvažujeme pouze lokace typu 0, což je právě nástupiště. Zbylé dvě možnosti jsou stanice (ty si vytvoříme dle svých kritérií sami) a vchody/východy do/ze stanic.

5. Trips - Trip je trasa linky s časy odjezdu. Typicky máme daleko více tripů než linek. Například odjezd metra A ze stanice Depo Hostivař v 7:00:00 je jiný trip než odjezd téže linky ze stejné stanice v čas 7:10:00. Tripy se indexují přirozenými čísly, tudíž nám jakožto struktura stačí vector/List. Dále zde máme odkazy na jednotlivé linky v podobně jejich identifikátorů, to stejné platí pro services. U tripů nám stačí si uchovávat jen headsign, což je název cílové stanice ("to, co je na přijíždějící tramvaji/metru/autobusu"). Trasu určíme z následujícího bodu.

6. Stop Times - Toto je nejrozsáhlejší položka všech dat a obsahuje údaje o odjezdech. Každá položka obsahuje identifikátor tripu, identifikátor zastávky, o kolikátou zastávku v řadě se jedná a časy příjezdu a odjezdu z dané zastávky. Ke každé zastávce přidáme ukazatel na každý stop time, který je asociován k dané zastávce (snadné vyhledávání departure board) a to stejné učiníme pro každý trip, jemuž daný stop time přísluší. Tak jsme schopni zrekonstruovat trasu tripu.

Struktura pro GPS souřadnice bude navíc obsahovat statickou metodu pro počítání doby chůze mezi dvěma body. Defaultní průměrná rychlost je nastavena na 0,8 m/s a metoda využívá Haversine formuli.
Pro správné fungování RAPTOR algoritmu ještě potřebujeme vytvořit trasy pro tripy, které sdílí stejné zastávky. Údaje o linkách přímo použít nemůžeme, protože například ačkoliv linka 22 v trase Vypich - Nádraží Strašnice sdílí stejné zastávky, tak tatáž linka v trase Bílá Hora - Nádraží Hostivař je v podstatě linka úplně jiná. Toto uděláme v rámci preprocessingu. Položku routes si označíme jako routes info a vytvoříme skutečnou položku routes, kde jedna route bude obsahovat posloupnost zastávek in-order, dále odkaz na route info a odkazy na všechny tripy, seřazené podle času odjezdu. To si přetřídíme již v rámci preprocessingu dat. Pro algoritmus také potřebujeme vědět, jaké routes prochází jednotlivými stops. Budeme procházet všechny route a u každé zastávky do ni přidáme odkaz na danou route. Struktura pro tento přístup může být klidně vector/List. Toto si ale nemusíme předpočítávat, protože by to bylo stejně efektivní, jako kdybychom to počítali přímo při inicializaci dat z C++. Proto to budeme dělat pokaždé při spuštění aplikace.
Rovněž musíme dopočítat footpaths, tedy dobu přestupu mezi jednotlivými zastávkami. Původní plán byl takový, že GTFS data, která tuto volitelnou položku obsahují, ji budou také využívat. Nakonec vyšlo najevo, že bude lepší vše předpočítat ručně, aby to bylo konzistentní a měli jsme časy mezi všemi zastávkami, které potřebujeme. Jediná nevýhoda tohoto přístupu je, že čas běhu se společně s rostoucími zastávkami kvadraticky zvyšuje. Haversine formule potřebuje ke své činnosti goniometrické funkce, proto je tato část algoritmu skutečně velmi časově náročná. Údaje o footpaths budeme uchovávat u každého nástupiště (stop) ve formě mapy/SortedDictionary a to tak, že klíčem bude intová hodnota, za jak dlouho jsme schopni při chůzi průměrnou rychlostí dojít do zastávky, která se schovává pod tímto klíčem (v sekundách). Pro ušetření paměti (a konec konců i času) budeme do paměti ukládat pouze hodnoty nepřevyšující 600 sekund.

Po naparsování GTFS formátu do paměti aplikace dojde ke vhodnému výpisu dat do nových souborů, aby se daly použít k výpočtům. To bude probíhat v podstatě tak, že původní soubory okleštíme na nutné minimum (viz. předchozí text), popřípadě vytvoříme soubor s footpaths a rozhodně vytvoříme soubor se stanicemi (stations). Při výstupu nástupišť (stop) tedy bude navíc každý záznam obsahovat intový identifikátor, kterým budou indexovaný stanice v nově vzniklém souboru. Dále soubor s routes přeměníme na routes info, soubor routes vytvoříme podle výše zmíněných pravidel a do každého tripu navíc přidáme identifikátor na danou route. Každá route navíc bude obsahovat ID na příslušné route info. Dále na první řádku každého nového souboru, pro nějž budeme později vytvářet v paměti kontejner, připíšeme počet záznamů, abychom si na danou velikost mohli kontejner před načítáním dat přizpůsobit a ušetřit tak čas i počet alokací. Nově vytvoříme soubor s jediným záznamem, v němž bude na první řádce napsáno, kdy jízdní řády expirují (konec platnosti první service v datech). Variabilitu GTFS při parsování ošetříme tím způsobem, že využijeme první řádky souboru, kde je pokaždé napsáno, jaká data daný sloupec obsahuje. Již na prvním řádku tedy ověříme, jestli daný soubor obsahuje vše, co potřebujeme. Následně, protože nikde není psáno, v jakém pořadí mají sloupce vystupovat, vytvoříme Dictionary s klíčem string a hodnotou int , kde string je název daného sloupce a int je index, tedy o kolikátý sloupec se jedná.

K datům budeme přistupovat přes tento slovník. Vytvoříme interface IDataFeed, který bude obsahovat seznam všech souborů, které musí výsledná data obsahovat (viz. níže) a metodu CreateDataFeed, která tyto soubory vytvoří. Pro každý soubor vytvoříme abstraktní třídu, která bude definovat datové položky v daném souboru a způsob, jakým se mají vypisovat. Parsování GTFS souborů vyřešíme tak, že pro každý nový soubor vytvoříme třídu GtfsXXX, kde XXX je název abstraktní třídy, od níž se bude dědit. Takže abstraktní třídy budou obsahovat vše potřebné pro výpis nového formátu, zatímco zděděné třídy budou obsahovat vše potřebné pro parsování GTFS formátu. Děláme to takto proto, abychom umožnili snadné rozšíření o parsování nějakého nového formátu do formátu potřebného pro práci knihovny pro jízdní řády. Celou práci aplikace bude obsluhovat jedna generická funkce, kde se jako typ funkce zadá formát, z kterého se mají data parsovat. Respektive, tento formát bude reprezentován třídou, v našem případě GtfsDataFeed, která bude implementovat rozhraní IDataFeed. Konkrétně by vše popsané mohlo vypadat třeba takto:

```
public interface IDataFeed {
    Calendar Calendar { get; }
    CalendarDates CalendarDates { get; }
    RoutesInfo RoutesInfo { get; }
    Stops Stops { get; }
    Stations Stations { get; }
    Footpaths Footpaths { get; }
    Trips Trips { get; }
    StopTimes StopTimes { get; }
    Routes Routes { get; }
    string ExpirationDate { get; }
    void CreateDataFeed(string path);
}

public static class DataFeed {
    public static void GetAndTransformDataFeed<T>() where T : IDataFeed {
        Downloader.GetDataFeed("temp_data/");
        IDataFeed data = (T)Activator.CreateInstance(typeof(T), (string)"temp_data/");
        data.CreateDataFeed("data/");
        Downloader.DeleteTrash("temp_data/");
    }
}
```

Následuje seznam souborů, které bude nový formát obsahovat s názvy sloupců (ty ovšem nikde v souborech napsané nejsou, jedná se o "implementační tajemství"). Pro připomenutí platí, že na první řádce každého souboru je vždy počet záznamů, které následují. Záznamy mají vždy fixní velikost (fixní počet sloupců). Je to tak pro ušetření počtu alokací při načítání do vectorů a také proto, abychom mohli načíst všechna data pomocí for cyklu. Záznamy od sebe nejsou odděleny newliny, vše je napsané v jedné řádce. Jednotlivé sloupce jsou od sebe odděleny středníkem.

Calendar - ServiceID, Monday ... Sunday, ValidSince, ValidUntil
CalendarDates - ServiceID, Date, TypeOfExtraordinaryEvent
RoutesInfo - RouteInfoID, ShortName, LongName, MeanOfTransport, Color
Stops - StopID, ParentStationID
Stations - StationID, Name

Footpaths - Duration, FirstStop, SecondStop
Trips - TripID, ServiceID, RouteID, DepartureTime (relative time since midnight, in seconds)
StopTimes - TripID, StopID, ArrivalTime, DepartureTime (both seconds since departure from the first stop in the trip)
Routes - RouteID, RouteInfoID, NumberOfStops, Headsign
ExpirationDate - ValidUntil

Co se týče nutného offline fungování mobilní aplikace, budou k dispozici pouze názvy zastávek a linky, které jimi projíždějí. V mobilní aplikace tedy bude, co se dat týče, soubor routes info. Budeme pro ni muset vytvořit speciální soubor stations, kde jen přidáme identifikátory všech routes info, které projíždí nějakým nástupištěm dané stanice. Toto vše vytvoří aplikace na preprocessing dat v rámci aktualizace jízdních řádů. Těmto souborům vytvoří nějaký timestamp, do kdy jsou data validní. Mobilní aplikace požádá o tato data při prvním spuštění a poté pokaždé, když dojde k expiraci dat na základě timestampu. Pokud nebude k dispozici připojení k internetu, mobilní aplikace upozorní na možnou expiraci základních dat.

Později, na základě profilace výkonnosti algoritmu, se ukáže, jestli bude nutné si do paměti předem připravit i různé indexy, například kolikátá je zastávka X v pořadí route, nebo jestli bude dostatečně efektivní využít lineární přístup hledání optima/minima/maxima, popř. binární vyhledávání (pro nějaké jiné položky, například hledání ideálního času odjezdu). Toto teď vůbec nejsem schopen odhadnout, bude to hrát roli až při existenci mobilní aplikace, kdy může na server chodit několik stovek požadavků za sekundu. Pak by samozřejmě bylo ideální mít algoritmus co nejrychlejší.

4.2 Problém s časem

Je zde problém s časem v GTFS formátu, a sice že můžou obsahovat i data převyšující čas 23:59:59. V GTFS formátu je čas 25:15:00 a 1:15:00 ekvivalentní, a oba to jsou validní časy. Je to tak proto, že v GTFS se o datech (téměř) vůbec nemluví, pouze o času a dnech v týdnu. Podle dne v týdnu určíme, kdy daný trip operuje. Nicméně může existovat trip, který vyjíždí v operační den ve 23:50:00 a do cílové stanice dojíždí další den v 0:10:00. Kdyby onen trip v ten další den neoperoval, byl by zde problém, protože by nikdy nedojel do cílové stanice. Proto je toto řešeno tak, že trip odjíždí v operační den ve 23:50:00 a dojíždí v "ten samý"operační den ve 24:10:00, což je v reálné situace samozřejmě nedefinovaný čas. Při operacích s časem v C# používáme klasickou a přirozenou strukturu DateTime, zatímco při práci v C++ si vytvoříme kopii DateTime z .NET, která bude obsahovat podobné funkce a čas se bude reprezentovat jako Unix timestamp, tedy v proměnné time_t si budeme uchovávat počet sekund od 1.1.1970 0:00. Časy reprezentujeme relativně. Konkrétně, čas odjezdu tripu (tzn. odjezd z první zastávky) je počet sekund od půlnoci daného dne. Takže například odjezd ve 12:00:00 bude mít hodnotu 43200 sekund. V rámci tripu, odjezd z jednotlivých zastávek určujeme relativně vůči odjezdu tripu. Takže například, pro stejný trip, který odjíždí ve 12:00:00 a ze třetí zastávky v pořadí odjíždí v 12:10:00, bude čas odjezdu u této zastávky

nastavený na 600 sekund. Děláme to takto, protože chceme, aby knihovna byla v budoucnu použitelná i pro nějakou větší oblast, například celou republiku nebo Evropu. Jinak bychom mohli časy uchovávat absolutně, nicméně pak by délka jízdy jednoho tripu nesměla převýšit 24 hodin, což typicky v MHD nenastane. Toto je ale čistě programátorský problém, uživateli se vše bude zobrazovat v "normálních"časech.

(veřejné, neformální) poznámky mě, abych nezapomněl, co probrat na konzultaci:

1) Code convention - v C++ psát C# stylem, když to později budu propojovat (aby to bylo konzistentní), nebo klasickým C++ stylem? tzn. pojmenovat metodu v C++ FindSomething() nebo find_something(), třídu Router nebo router, (privátní) datovou položku someVariable nebo some_variable_?

2) Circular dependency v headerech... existuje nějaký návod (algoritmus), jak to vyřešit? Třeba pro ty datové položky, kde mám v hpp čistě jen referenci, využívat forward declaration a když s tím nějak pracuji i v hpp, tak include?

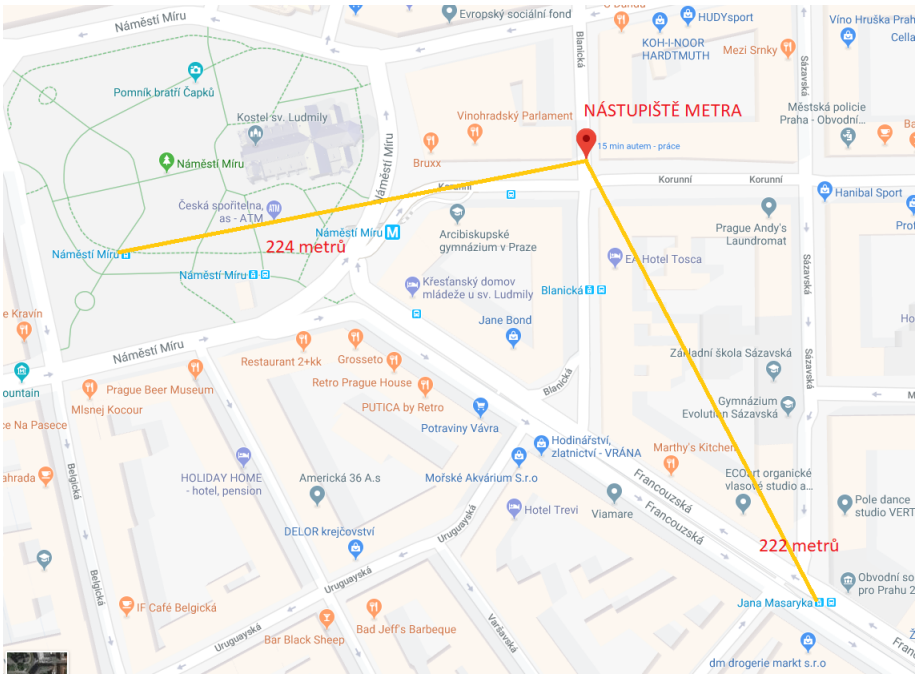
3) Server aplikace - navrhnout GUI desktop app nad tím, nebo zaintegrovat do GUI app? že by server app byla nějaká konzolová aplikace bez UI (jen zobrazování nějakých hlášek) + by k tomu byl jednoduchý konfigurátor ve WinForms (pro nastavení serveru, které by bylo třeba v nějaké xml), to by umožnilo pustit server někde vzdáleně třeba na Linuxu, které nepodporují WPF (v kterých nejspíš budu dělat GUI pro desktop)... kdybych tohle měl, mohl bych na tom postavit tu desktop app, jinak bych to přímo zaintegroval do té desktopové aplikace a server by např. na Linuxu, nebo vzdáleně přes konzoli, nešel pustit... upřímně by se mi asi líbilo více, mít oddělenou desktopovou aplikaci od serveru, ale to bych pak musel přepsat skoro celou specifikaci... mohl bych na tom pak snadno postavit i nějakou aplikaci v ASP.NET podobnou té desktop aplikaci, chci se ASP.NET přes léto naučit...

Pak bych měl: 1) server konzolovou aplikaci (která by zahrnovala preprocessor, core library a logiku pro server, byla by v C# kvůli preprocessoru a lepší přenositelnosti serveru + interop C++/CLI s core library), 2) easy konfigurátor pro server (v C#, WinForms, opravdu jen jedno okénko s možností nastavení portu pro síť, zdroje dat a podobných blbin, nastavení uloženo v nějakém XML, z kterého by si to z první aplikace brala, bylo by to fakt maximálně 300 řádků kódu...), 3) desktop aplikaci s GUI ve WPF, ta je komplet popsána v sekci 2.3, jen by teda byla možnost ji využívat i online - kdyby jel server na stejném stroji jako tahle aplikace, jen by se nastavila adresa 127.0.0.1, 4) online aplikaci v ASP.NET (což by byla volitelná část, myslím, že čím více toho bude do bakalářky, tím lépe), což by bylo to stejné jako předchozí desktop aplikace jen s tím rozdílem, že kdyby jel server někde vzdáleně, mohl bych i vzdáleně vyhledávat spojení přímo z prohlížeče, 5) mobilní aplikaci v C#, viz sekce 2.4

Uznávám, že je to celkem zmatečný popis, raději bych to probral na konzultaci...

4) Problém s přestupy - je to měřeno podle vzdušené vzdálenosti, někdy nastávají problém viz. obrázek (tzn. že v tomhle případě mě to při přestupu na/z metra někdy pošle na Jana Masaryka, přitom Náměstí Míru je logicky blíž...). Bohužel fakt nevím, jak to ošetřit, když to chci dělat pro nějaká obecná data a ne jen pro Prahu, protože kdyby byl výstup z metra i do Francouzské ulice (což by v nějaké jiném městě takhle klidně mohlo být), bylo by to opravdu rychlejší. Pak je tu třeba i problém, když chci přestoupit na Muzeu z C na A směr Nemocnice Motol. Tak podle nějaké pevné průměrné rychlosti (mám tam tuším 0,8 m/s) přestup na Muzeu z C na A trvá 1,5 minuty, zatímco přestup mezi dvěma nástupišti na

Náměstí Míru trvá minuty dvě... To ale nemůžu napevno zvolit, že je tam téměř nulový přestup, protože někde v nějakém městě může třeba opravdu existovat stanice, kde je ten dvouminutový přestup mezi dvěma nástupišti reálný :(



5) Proč je unordered_map tak pomalá? Respektive operace find mi v profileru přijde strašně pomalá, ačkoliv jsem měl za to, že by to mělo být O(1). Četl jsem, že je to nějak vnitřně implementované pomoci linked listů, a ty způsobují výpadky cache, je to pravda? Zkoušel jsem nějakou strukturu dense_hashmap z knihovny sparsehash od Googlu, kde sice samotné operace byly o dost rychlejší, ale celková doba běhu programu byla dokonce i horší, což nechápu - tak jsem se vrátil k unordered_mapě... Popřípadě existuje v STL něco rychlejšího, co plní roli hashovací tabulky? Jako klíč mám povětšinou nějaký pointer, který je v tabulce obsažen pouze jednou...

Profiler (ještě nefinální stav): DateTime jako v .NET 178 ms per journey, time_t 171 ms per journey, boost::date_time 242 ms per journey Rozhodl jsem se zatím využívat time_t, ačkoliv se mi to nelíbí, protože pro to musím mít statické metody a programuji "objektově"jako v C. Možná se vrátím k té první možnosti, kde využívám time_t enkapsulovaný do objektů (asi určitě se k tomu vrátím). Pravděpodobně využiji i boost, konkrétně třídu ptime z namespace posix_time. Bude to nahrazovat strukturu tm z C. Teda ptime bude sloužit k interakci s uživatelem, zadávání vstupu a zobrazování výstupu, protože funkce localtime a mktime se mi nelíbí, viz. problémy s timezones.

Po mikro optimalizacích vylepšeno na 140 ms per journey (testování na 10.000 vyhledávání mezi náhodnými dvěma zastávkami a finální čas - čas inicializace je vydělen 10.000).

Po dalších úpravách (minimum 64 bitových dělení) nyní 120 ms per journey, ale to ještě určitě půjde dolů. Nutno podotknout, že jelikož se jedná o náhodné stanice, velmi často se vyhledávají exotická spojení typu ze zastávky 30 kilometrů severozápadně od Prahy do zastávky 30 kilometrů jihovýchodně od Prahy. Zde je hodně přestupů a jelikož je dopravní síť

pro Prahu velmi hustá, musí se projít úplně všechno. Když se vyhledávají normální spojení v rámci Prahy, tak třeba prvních 1000 spojení z mého domova do školy to najde za 48 sekund, což je 48 milisekund per journey. Ještě s tím nejsem ani zdaleka hotov, jde to zparalelizovat (to si počkám na přednášky o vláknech), čímž se pak dostanu na nějakých 20 ms (a v případě exotických spojení minimálně na 60 ms), což je můj cíl. Program by měl jít v pohodě použít například na oblast celé střední Evropy bez ztráty výkonu, protože se nepředpokládá, že "na venkově"by měla být stejně hustá síť jako například v Praze.

6) Byl by případný čas kolem 50 ms per journey špatný, nebo je to v pohodě? Bojím se jen toho, že když budou chodit na sever stovky požadavků za minutu, tak se může brzy zahltit, když nebude moc výkonný. Zas na druhou stranu to ale celé zparalelizuji, tak nevím no... Koukal jsem na nějakou diplomku na ČVUT (strana 86 https://dspace.cvut.cz/bitstream/handle/10467/70 DP-2017-Chalupa-Jakub-chaluja7-diploma-thesis-2017.pdf?sequence=-1isAllowed=y) podobnou té mé bakalářce a tam vyhledání spojení mezi Dejvickou a Karlovo náměstím (obyčejná trasa 1-2 přestupy) trvalo 600 ms per journey (nicméně i s odezvou sítě, takže řekněme polovinu času a navíc pracuje s databází, já s pamětí, ale to je asi z hlediska uživatele jedno a na dnešní poměry velikosti RAM je to jedno i správci serveru) - u mě v současné chvíli (ještě není hotovo...) se 500 takových spojení našlo za 4 sekundy, což je 8 ms per journey... Řeším to tak moc jen proto, že v tom článku o algoritmu psali, že vyhledání libovolného spojení v rámci Londýna trvá 8 ms, mně to v rámci Prahy může momentálně trvat až +/- 10x déle, no... Nicméně oni to v tom článku mají zoptimalizované, zparalelizované a ani tam nepíší, na jakém procesoru to počítají, takže tak...

Teď o víkend 18.3.2018 mám v plánu vydat první alfa verzi s miniaturním uživatelským rozhraním, ovládání z konzole v rámci C++ procesu.

Pozitivní zpráva na závěr: čas inicializace od minulé konzultace zlepšen o 600%, z 13 sekund na 2 sekundy :)

Pozn. na githubu mám i jakýsi XML projekt, to je jen vypisování dat do XML formátu, což jsem měl za úkol na předmět Technologie XML

Reference

[1] GTFS Format Reference
`https://developers.google.com/transit/gtfs/reference/`

[2] RAPTOR: Round-Based Public Transit Routing Algorithm
`https://www.microsoft.com/en-us/research/wp-content/uploads/2012/01/raptor_alenex.pdf`