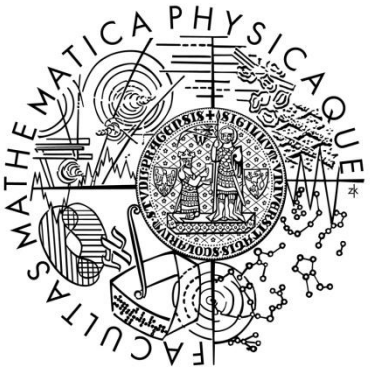
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Zbyněk Jiráček

Vyhledávač spojení s využitím pěších přechodů

Katedra aplikované matematiky

Vedoucí bakalářské práce: Mgr. Robert Babilon

Studijní program: Informatika

Studijní obor: Programování

Praha 2011

Děkuji Mgr. Robertu Babilonovi za vedení práce, jeho rady a doporučení po čas vývoje. Poděkování patří i mé přítelkyni Lence za její nadšení pro mou práci a také všem ostatním, kteří mi pomohli svou radou či podporou.

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

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

V Praze dne ..... Zbyněk Jiráček

Název práce: Vyhledávač spojení s využitím pěších přechodů

Autor: Zbyněk Jiráček

Katedra / Ústav: Katedra aplikované matematiky

Vedoucí bakalářské práce: Mgr. Robert Babilon

Abstrakt: Vyhledávání co nejrychlejší cesty mezi danými dvěma body je v dnešní uspěchané době poměrně užitečná věc. Vzhledem ke složitosti dopravních sítí ve větších městech je nějaký vyhledávač coby asistent téměř nezbytností. Typické programy tohoto zaměření příliš neřeší cesty mezi zastávkami a povolují pouze pěší přesuny mezi blízkými stanovišti za účelem přestupu. Mnoho lidí, nejvíce pak ti, kteří se v okolí příliš nevyznají, si však neuvědomí, že po městě se dá poměrně rychle přesouvat i pěšky. Cílem projektu je navrhnout a vytvořit program, který bude schopen člověka v rámci nějakého města dovést do cíle co nejefektivněji a to za využití hromadné dopravy a chůze, přičemž cílem může být nejen zastávka, ale i třeba adresa nebo libovolné jiné místo.

Klíčová slova: vyhledávání spojení, pěší přesuny, hromadná doprava, navigace

Title: Public transport route searching using moves afoot

Author: Zbyněk Jiráček

Department: Department of Applied Mathematics

Supervisor: Mgr. Robert Babilon

Abstract: Finding the fastest way between two given places is in recent hurried times quite useful thing. According to the complexity of traffic networks in larger cities a route searching engine becomes almost a necessity. Typical programs that focus on this problem usually do not pay attention to paths between individual stops and allow only movement in the scope of a single station. Many people, mostly those who do not know the area well enough, do not realize that in a city they can relatively quickly move by foot. Goal of this project is to design and implement a program which will be able to navigate user in the scope of a given city to a specified destination using means of public transport and walking. Instead of navigating only between stations, any place in map should be accepted.

Keywords: route searching, moves afoot, public transport, navigation

Obsah

1. Úvod

Vyhledávání dopravního spojení je problém, se kterým se každý často setkává. K hledání toho nejefektivnějšího nám často může pomoct nějaká aplikace. Pokud jde o dopravu automobilovou, existují různé zejména mapové systémy, které hledání efektivních cest poměrně dobře řeší, někdy jsou schopny započítat i aktuální dopravní situaci. V sítích hromadné přepravy je však problém o dost složitější, protože je třeba koordinovat svou trasu s pozicemi zastávek a s jízdními řády, které navíc nevystihují reálnou situaci přesně. Často existuje více podobně rychlých spojení, která používají různé prostředky a jsou různě spolehlivá. Typické vyhledávače ale vypisují pouze to nejrychlejší z nich. A také je tu jeden prostředek dopravy, který se často opomíjí – chůze. Ta sice není příliš rychlá, na druhou stranu má svůj speciální význam. Mimo město, kde doprava nebývá příliš hustá, může být přesun na vzdálenější zastávku mnohdy značně výhodnější; ve městě se pro změnu dopravní prostředky pohybují velmi pomalu a nepřímo, často se tedy k překonání některých úseků vyplatí i delší pěší přesun. V neposlední řadě je pěší doprava jednoznačně nespolehlivější.

1.1 Cíl projektu

Cílem práce je navrhnout a implementovat desktopovou aplikaci, která bude schopna vyhledávat nejrychlejší spojení ve městě, a to za použití jízdních řádů hromadné dopravy a mapových podkladů, pomocí kterých se budou hledat pěší trasy. Použití mapy by mělo poskytovat reálnější informace o době přestupu a hlavně přináší možnost přesouvat uživatele pěšky i na delší vzdálenosti než v rámci blízkých zastávek, stejně jako možnost nalezenou trasu popsat či vyobrazit na mapě. Jako počáteční a cílový bod by mělo být možno použít kromě zastávek i třeba adresu nebo GPS souřadnici v mapě. Použití desktopu jako platformy dává k dispozici větší výpočetní výkon (než například mobil nebo web). Důraz se bude klást na přesnost (optimálnost) spojení, rychlost jeho vyhledání není kritická. Plusem by pak jistě byla budoucí rozšiřitelnost programu novými zajímavými funkcemi a také možnost aplikaci prakticky použít.

1.2 Přehled kapitol dokumentu

Následující, druhá kapitola se věnuje problému vyhledávání spojení v sítích hromadné přepravy obecně (se zaměřením na městskou hromadnou dopravu) a podává přehled o existujících implementacích tohoto problému, diskutuje jejich výhody a nevýhody, které se bude tato práce snažit eliminovat.

Kapitola třetí pak na základě předchozích poznatků určí některé cíle, které se projekt bude snažit splnit, a následně seznámí čtenáře s koncepcí projektu, která z těchto cílů vychází. Pojednává o struktuře projektu a o použitých prostředcích a technologiích.

Ve čtvrté kapitole je popsána idea vyhledávacího algoritmu použitého v programu.

Následuje kapitola pátá věnovaná potenciálním uživatelům, která popisuje práci s ukázkovým programem samotným.

Programátorskou dokumentaci reprezentuje kapitola číslo šest. Obsahuje detailní popis jednotlivých částí projektu, vysvětluje způsob implementace a problémy, které při ní bylo třeba řešit.

Předposlední kapitola ještě před tou závěrečnou vypichuje některé nedostatky a problémy projektu a snaží se nastínit jejich možné řešení. Také stručně popisuje některé funkce, které by uživatelé od aplikace mohli uvítat, a případně navrhuje, jak by bylo možno je implementovat do tohoto projektu.

2. Vyhledávání spojení obecně

Jak jsem již nastínil v úvodu, tato práce se zaobírá pouze dopravou hromadnou. Tu lze rozlišit na městskou a dálkovou (meziměstskou), které mají odlišné charakteristiky (např. vzdálenosti zastávek, četnost spojů); tento projekt se soustředí na vyhledávání ve městech. Teoreticky bude možno program použít i pro dálkovou dopravu, nemusí však fungovat příliš efektivně.

Vyhledávání spojení v MHD samo o sobě nemusí být zas tak složitým problémem, i když je o něco náročnější, než prosté hledání nejkratší cesty v grafu. Úloha sama ale skýtá mnoho možností k vylepšování. Kromě nejrychlejšího spojení můžeme chtít znát třeba také spojení nejspolehlivější, nejméně fyzicky náročné, s co nejméně přestupy nebo ideálně nějaký kompromis. O možných užitečných vylepšeních píše kapitola 7.2. V práci jsem se ale soustředil zejména na propojení dopravní sítě s mapou a také na reálnost výsledných spojení.

Pěší přesuny mají ve městě poměrně značný význam. Síť zastávek je zde hustá, a tak mezi nimi mohou přesuny být dost rychlé. Linky mají sice malé intervaly, ale mezi některými i blízkými zastávkami může být spojení slabé, či dokonce žádné. Vysoká hustota dopravy způsobuje její výrazné zpomalení. Následující tabulka ukazuje rychlost průjezdu vybraných linek centrem Prahy.

Linka	Začátek	Konec	Km	Čas	Průměrná rychlost
Tram 17	Právnická fakulta	Palackého nám.	3,1	15 min	12,4 km/h
Tram 22	Náměstí Míru	Malostranská	4,4	21 min	12,6 km/h
Tram 9	Hlavní nádraží	Anděl	3,95	18 min	13,2 km/h
Bus 176	Švandovo divadlo	Karlovo nám.	1,55	7 min	13,3 km/h

Zdroj: Detaily spojení [3]

Ani mimo samotné centrum města není situace výrazně lepší. Ze *[4]* plyne, že v roce 2009 byla průměrná rychlost tramvají v Praze 18 km/h a autobusů 26 km/h, avšak včetně příměstských linek. Rychlejší dopravní prostředky, pokud jsou, se většinou ukrývají pod zemí a nemohou být příliš hustě rozmístěny, takže vyžaduje

jistou námahu se k nim vůbec dostat a na krátké vzdálenosti se jejich využití většinou moc nevyplatí.

Celá práce byla vyvíjena a testována pro vyhledávání v Praze. Ta, jako naše největší město, představuje takový „nejhorší případ“ pro vyhledávací algoritmy. Zároveň je to město, ve kterém se vyznám nejlépe, a tedy můžu dobře porovnávat výsledky práce s praxí. Samotný program by měl být dobře použitelný i pro jiná města, je pouze potřeba sehnat pro ně potřebná data.

2.1 Potřebné pojmy a zkratky

Před samotnou hlubší analýzou problému je třeba ještě definovat či vysvětlit několik pojmů a zkratek, které se ve zbytku dokumentu budou používat. V práci se vyskytují také pojmy z teorie grafů, které v této kapitole vysvětleny nejsou. Lze se o nich dočíst v *[1]* nebo *[2]*.

2.1.1 Zkratky

MHD – městská hromadná doprava

IDOS (Informační DOpravní Systém) – softwarové rozhraní pro vyhledávání dopravního spojení v hromadné dopravě (více viz 2.2.1)

OSM (OpenStreet Map) – projekt shromažďující geografická data pro tvorbu volně dostupných map (více viz kapitola 3.3)

GPS (Global Positioning Systém) – polohový systém, pomocí GPS souřadnic lze určit libovolné místo na zemi; tvoří ji dvojice (zeměpisná šířka, zeměpisná délka)

UTM (Universal Transverse Mercator) – polohový systém, na rozdíl od GPS pozici určuje trojice (zóna, x, y), předpokládá rozdělení planety do zón, více viz *[5]*

JRGPS – aplikace pro SmartPhone schopná hledat nejkratší spojení v Praze pomocí MHD a pěších přesunů mezi zastávkami (více viz 2.2.2) a *[6]*

PTN – Public Transport Navigator – označení programu spojeného s touto prací

2.1.2 Definice pojmů

Linka

Linku tvoří všechny spoje se stejnou identifikací (typicky číselnou), ty mají stejnou nebo velmi podobnou trasu. Každá *linka* se skládá z několika *směrů* (typicky dvou), ale jsou i výjimky, např. okružní *linky* mohou mít jen jeden; *linka*, která má více dostatečně odlišných variant může mít zase směrů více.

Směr

Směr je určen posloupností *zastávek* a zároveň sdružuje spoje, které těmito *zastávkami* v tomto pořadí projíždí (nemusí všemi, můžou některé vynechávat). Každý směr se skládá z *tras*.

Trasa

Trasa sdružuje spoje ze stejného *směru*, které navíc projíždí stejnou podmnožinu *zastávek* a zároveň mají stejné časové intervaly mezi jednotlivými *zastávkami*. Všechny spoje v jedné trase také jedou ve stejné dny. *Trasa* je tedy určena *směrem* a posloupností dvojic (*příjezd*, *odjezd*), kterých musí být stejně jako *zastávek* v daném *směru*. K tomu pak patří také dny, kdy jsou spoje této *trasy* vypravovány. *Příjezdy* a *odjezdy* jsou vyjádřeny jako počet minut od počáteční *zastávky*; nemusí být definované, pokud spoje dané trasy zastávku projíždí. *Trasa* obsahuje seznam odjezdů spojů z první (výchozí) zastávky.

Spoj

Spoj je jednoznačně určen *trasou* a časem odjezdu z výchozí zastávky. *Trasa* mu určuje přesné časy v konkrétních zastávkách (přičtením k odjezdu z výchozí zastávky), i dny, kdy je spoj vypraven.

Zastávka, skupina zastávek

Zastávka je určena přesnou polohou v mapě, svým typem (tramvajová/autobusová/…) a *skupinou zastávek*, do které náleží. Jedna *zastávka* tedy odpovídá jednomu konkrétnímu „sloupku“/nástupišti.

Skupinu zastávek tvoří všechny *zastávky* stejného jména, ať už jsou jakéhokoliv typu.

2.2 Existující implementace

2.2.1 Webové vyhledávače

Nejznámější verzí IDOSu je asi jeho webová podoba. Portál http://www.idos.cz někdy použila nejspíš většina lidí odkudkoliv z ČR; Pražané mohou znát také třeba verzi pro pražskou MHD na adrese http://idos.dpp.cz/idos, která je od první zmíněné odvozená, disponují tedy podobnými vlastnostmi. **Webový IDOS** dále bude zastupovat právě tyto zmíněné stránky. Existují i jiné webové verze vyhledávající spojení, zpravidla ale pouze předávají dotaz na výše uvedené servery.

Webový IDOS má hned několik výhod. Předně fakt, že jde o internetové stránky, umožňuje snadnou dostupnost ze kteréhokoliv počítače a dnes již i z většiny mobilních telefonů a podobných zařízení s připojením k internetu. Dále je oficiální, tedy obsahuje nejaktuálnější data, což je u vyhledávání spojení poměrně klíčová věc.

Pokud jde o dostupné funkce, jsou sice dostatečné pro průměrně náročného cestujícího, přesto však webový IDOS obsahuje dost místa pro potenciální zlepšení, i když některé funkce nelze podporovat kvůli jejich výpočetní náročnosti. Vzhledem k ní webový vyhledávač například neumí příliš dobře pracovat s pěšími přestupy. Umožňuje přestup typicky pouze mezi blízkými zastávkami, časy jsou pouze odhadnuté a mnohdy ne úplně přesné, navíc nezohledňující rychlost chůze. Zejména v případě větších měst je pak absence mapového podkladu také nevýhoda pro cestující, kteří se v okolí příliš nevyznají. Znají-li pouze cílovou adresu, musí si sami zjistit jméno zastávky, na kterou chtějí dorazit, a vyhledat si potřebnou cestu. Navíc cesta na zastávku a ze zastávky není do délky spojení započítána. To se projeví například v případě někoho, kdo má v okolí svého bydliště více různých zastávek, které jsou obsluhovány různými linkami. Spojení z více zastávek sice ve webovém IDOSu částečně vyhledávat lze, nicméně se počítá ve všech se stejným počátečním

časem, což zvýhodňuje pro cestujícího vzdálenější zastávky. V důsledku pak člověk často musí hledat spojení opakovaně a sám si určit to optimální.

Další nevýhodou je, že webový IDOS uvažuje pouze „dvouhodnotově“; pomocí předepsaného času na přestup určí, zda se stihne, či nestihne, a druhá alternativa už se pak neřeší. Kvůli tomu aplikace může zatajit spoje, které považuje za nestihatelné, nebo naopak nezobrazit některá rozumná spojení, protože bylo nalezeno jiné, rychlejší, které ale nemusí být příliš spolehlivé. Tato přehnaná „sebedůvěra“ také často nutí uživatele, aby si sám našel další spojení z přestupních bodů pro případ, že by to navrhované nevyšlo, či aby si sám zjistil, jestli neexistuje šance stihnout dřívější přípoj. Pokud nalezené spojení používá linky s dostatečně krátkými intervaly, ve výsledku dochází k menšímu zpoždění, příp. předstihu oproti nalezenému spojení, protože pěší přesuny mezi zastávkami mohou trvat o dost méně nebo více, než odhaduje vyhledávač, a výsledné spojení pak nevychází, jak bylo zamýšleno (buď se nestíhá, nebo by naopak existovalo lepší).

2.2.2 Projekt JRGPS

JRGPS je aplikace pro platformu PDA vytvořená skupinou studentů Matematicko-fyzikální fakulty jako softwarový projekt. Umožňuje vyhledávání spojení mezi libovolnými body ve městě, a to pomocí chůze a MHD. Součástí aplikace je i navigace po mapě.

Asi největší výhodou tohoto projektu je právě použitá platforma. Mobilní zařízení je totiž vhodné pro použití „v terénu“, kde se často vyhledání spojení hodí. Jelikož výpočet probíhá lokálně, není k němu třeba ani připojení k internetu. Použití mapových podkladů pro hledání pěších přesunů řeší velkou část nedostatků webového IDOSu. Vyobrazení výsledné trasy na mapě pak umožňuje i těm, co své okolí příliš neznají, dostat se snadno k cíli.

Aplikace je však vzhledem ke své platformě omezena výkonnostně. Podle dokumentace jsou pěší přesuny uvnitř trasy omezeny jen na určitou množinu, která se počítá předem (*„Hrany byly vytvořeny tak, že pro každý zastávkový ostrůvek byly*

nalezeny cesty k okolním zastávkovým ostrůvkům“ [7]). Ve většině případů nejde o závažný problém, protože uvnitř trasy se obvykle delší přesuny nevyskytují. V okrajových částech měst ale své využití najdou, jelikož tam doprava není tak hustá.

Aplikace podle *10.1.6 v [7]* příliš neřeší výběr spojení z množiny těch nejrychlejších a zajišťuje pouze to, že nalezené spojení je opravdu nejrychlejší. Ale zvláště jsou-li k dispozici i pěší přesuny, existuje stejně rychlých cest poměrně hodně. Touto problematikou se více zabývá kapitola 4.1.

Nedostatkem aplikace je i problém mimoúrovňových křížení. Jak se píše v *10.1.2 v [7]*, *„V případě křížících se polyline byl totiž do místa jejich průsečíku zřejmě automaticky doplněn nový vrchol…“*, což způsobí možnost přesunu např. mezi mostem a cestou vedoucí pod ním.

3. Návrh řešení a použité technologie

Předchozí kapitola rozebírala dvě implementace tohoto problému, tato bude mít za úkol popsat, jaké prostředky a technologie používá PTN. Důraz byl kladen zejména na vyřešení problémů, kterými zmíněné projekty trpí.

Než však lze učinit rozhodnutí o konkrétních prostředcích, technologiích a způsobech implementace, je třeba si stanovit, co se od výsledného programu očekává. To postupně probírají následující tři podkapitoly, na konci každé je v bodech shrnuto, co důležitého z ní vyplývá, zejména pak pro strukturu projektu.

3.1 Požadavky na výsledný program

Jak webový vyhledávač, tak projekt JRGPS jsou omezeny výpočetním výkonem. Na rozdíl od nich bude projekt PTN realizován jako desktopová aplikace, která bude mít dostatečný výkon využitelný k implementaci „důkladnějšího“ vyhledávacího algoritmu a případně i některých funkcí navíc. Nevýhodou je pak ale horší dostupnost výsledné aplikace pro uživatele, jelikož je třeba ji explicitně stáhnout. Kromě aktualizace by pak ale nemuselo být při samotném vyhledávání potřeba připojení k internetu, ačkoliv už by to dnes nebyla příliš omezující podmínka.

Aby však bylo možno v budoucnu vybranou platformu s co nejmenším úsilím změnit, jistě pomůže oddělení výpočetních částí do samostatné knihovny tak, že připravit verzi pro jinou platformu by nemělo být příliš náročné.

Jak již bylo zmíněno, aplikace bude zpracována pro pražskou MHD, nicméně knihovna by měla být dostatečně univerzální, aby fungovala nezávisle na oblasti a jízdních řádech, nad nimiž pracuje.

- Samostatná knihovna jako oddělená část projektu obsahující vyhledávací funkce; nezávislá na konkrétním městě.

- Desktopová aplikace poskytující uživatelské rozhraní využívající funkce knihovny – zde je třeba vybrat vhodnou knihovnu pro implementaci uživatelského rozhraní.

3.2 Vyhledávání spojení

V první řadě je potřeba určit způsob vyhledávání spojení. Přímočarou možností by bylo dotazovat se na spojení webového IDOSu. To ale nelze provést, protože při vyhledávání existuje mnoho různých cest a čekání na odpověď serveru by přineslo výrazný výkonnostní propad. Program si tedy bude muset hledat spojení vlastními prostředky. Pak je ale potřeba vyřešit otázku zdroje jízdních řádů.

Jízdní řády někdy bývají k dispozici ke stažení, některá města je ale veřejně neposkytují. S ohledem na to se volba vytvořit aplikaci nad pražskou MHD ukázala jako lehce nešťastná, protože Praha je zrovna jedno z měst, které volně ke stažení jízdní řády nedává[1]. Pro testovací účely aplikace by stačilo vytvořit náhodné jízdní řády, nakonec se ale povedlo získat potřebná data posíláním dotazů na stránky webového IDOSu, i když to s sebou nese dost problémů. Jasné je, že bude potřeba mezivrstva, která jízdní řády zpracuje do předepsaného formátu, aby to nemuselo být řešeno až ve vyhledávací knihovně. O konkrétním způsobu stahování JŘ píše kapitola 6.1.1.

Samotné jízdní řády však obsahují pouze jména *skupin zastávek*, ne ale konkrétní *zastávky* ani jejich souřadnice. Zastávky bývají zaneseny v OSM, ale poměrně nepřesně, navíc některé chybí, proto je potřeba využít jiný zdroj. Jako další možnost se ukázala databáze zastávek pražské MHD dostupná ke stažení z http://www.poi.cz. Její nevýhodou je neaktuálnost údajů, nicméně v síti zastávek se změny nedějí příliš často. Horší je nutnost řešit následné propojení s jízdními řády. Jádro problému spočívá v tom, že v jízdních řádech známe jména zastávek,

[1] Existují snadno stáhnutelná data (http://www.chaps.cz/ke-stazeni-aktualizace-idos-win.asp), ta jsou ale v neveřejném a na první pohled nečitelném formátu, navíc výslovně chráněna autorským zákonem, takže je nelze použít.

jenže jméno jednoznačně identifikuje *skupinu*, nikoliv konkrétní *zastávku*. Byl by tedy třeba algoritmus, který by se snažil toto propojení nějak odhadnout.

Potřebné informace o zastávkách lze též vyčíst z map webového IDOSu. Je však třeba souřadnice zastávek převést, jelikož od pohledu nejde o GPS souřadnice, navíc pozorováním jsem zjistil, že bývají občas nepřesné. Velkou výhodou je ale samovolné vyřešení propojování zastávek s jízdními řády, protože v rámci IDOSu mají *zastávky* jednoznačné identifikátory, které lze z každého spoje získat. Známe tedy rovnou konkrétní *zastávky*, po kterých linky jezdí.

Oba zmíněné zdroje, tedy poi.cz a webový IDOS, samy o sobě neposkytují dostatečně kvalitní informace, proto se nakonec seznam zastávek vytváří z jejich kombinace. Toto propojení je sice komplikované, ale přesnější souřadnice z poi.cz pak přináší jistou korekci. Důsledkem je pouze občasné prohození některých zastávek právě kvůli nepřesnosti jejich souřadnic stažených z webového IDOSu. Více o vytváření databáze zastávek poví kapitola 6.2.5.

- Samostatná část schopná získat (stáhnout) jízdní řády a informace o zastávkách.

3.3 Mapové podklady

Pro splnění zadání je samozřejmě také třeba zajistit si podklady pro mapovou část. Vlastně jediným testovaným zdrojem byly OpenStreetMaps (OSM) *[9]*, které se nakonec ukázaly jako celkem vhodné. Jejich struktura je jednoduchá, obsahují kromě silnic i chodníky, cestičky a adresy. Jsou k dispozici téměř pro celý svět, ač někde nekompletní; ve větších městech ale bývají dostatečně podrobné, což umožňuje jednoduchou adaptaci programu na jiná města než Prahu. Důležitou výhodou je také jejich volná dostupnost.

K popisu mapy však používají formát XML *[10]*, což vytváří dost velké objemy dat. I zde se tedy vyplatí data před použitím vyhledávací knihovnou nejprve nějak zpracovat. Blíže k formátu OSM a k jeho zpracování se dostane kapitola 6.2.5.

Je třeba také myslet na úzké propojení databáze zastávek a mapy, jelikož zastávky se musí do mapy nějakým způsobem zanést. Proto je možná lepší ponechat kompetenci zpracování jízdních řádů a zastávek stejné aplikaci, která zpracovává mapové podklady.

- Samostatná část schopná zpracovat mapové podklady, jízdní řády a zastávky do předepsaného formátu a poskytnout je vyhledávací knihovně.

3.4 Shrnutí

Zde si shrneme, jaké rozdělení projektu vyplývá z předchozích kapitol (3.1 – 3.3) a jaké technologie budou potřeba. Detaily k jednotlivým projektům jsou popsány v šesté kapitole rozebírající implementaci.

1. Stahování dat

Úkolem aplikace pro stahování dat je pouze stáhnout potřebné informace o jízdních řádech a zastávkách hromadné dopravy v Praze a uložit je v nějakém formátu na disk. Tato část projektu se k uživateli vůbec nedostane, nemusí být tedy platformně nezávislá, ani extrémně efektivní, bude však potřeba co nejjednodušeji stahovat soubory z internetu. K tomu lze využít například funkcí .NET Frameworku (resp. jazyka C#), který pro účely napsání této aplikace poslouží dostatečně.

2. Příprava dat

Kromě stažení jízdních řádů je třeba je také společně s mapovými a zastávkovými údaji zpracovat a co nejlépe připravit. Podobně jako aplikace pro stahování, ani příprava dat se nebude vyskytovat u koncového uživatele, platí pro ni tedy stejné podmínky. Je taktéž implementována v jazyce C# v .NET Frameworku, jenž se v tomto případě hodí k načítání vstupních souborů, které jsou ve formátu XML. O účelu a formátu výstupních souborů (tedy vstupních pro vyhledávání) pojednává příloha B.

3. Vyhledání spojení

Vyhledání spojení bude mít na práci samostatná statická knihovna. Ta už by měla být co možná nejrychlejší a pokud možno také nezávislá na platformě. K tomu je ideální použít jazyk C++.

4. Uživatelské rozhraní

Uživatelské rozhraní bude představovat desktopová aplikace používající vyhledávací knihovnu. Na implementaci rozhraní by se sice spíš hodilo také využít .NET Framework, ale komunikace s objektovou C++ knihovnou by byla obtížná. Na druhou stranu jazyk C++ není na psaní rozhraní příliš vhodný, bylo tedy potřeba najít nějakou externí knihovnu, ve které by to šlo jednodušeji. Nakonec za tímto účelem posloužila knihovna Qt *[11]*, která je pro C++.

Výsledkem také je, že části projektu, které se dostanou ke koncovému uživateli, budou tvořeny v jazyce C++, který je sám o sobě multiplatformní. Vzhledem k tomu, že se neplánuje využití žádných knihoven závislých na konkrétní platformě, měla by výsledná aplikace být kompilovatelná pro různé operační systémy. Nicméně implementace bude provedena pod systémem Windows, stejně tak i kompilace. Je tedy možné, že případné vytvoření verze pro jiný operační systém bude vyžadovat drobné úpravy vzhledem k malým odlišnostem kompilátorů.

4. Vyhledávání spojení

Jádrem celé práce je samotný algoritmus, který optimální spojení hledá. Tato kapitola se věnuje jeho návrhu, implementační detaily jsou pak doplněny v kapitole 6.3.2. Vyhledávání je založeno na Dijkstrově algoritmu *[2]*. Vrcholy grafu jsou tvořeny uzly mapy, hrany rozlišujeme dvojího druhu. *Pěší hrany* spojují uzly[2], mezi kterými vede cesta (silnice, pěšina, chodník). *Dopravní hrany* jsou určeny trasami spojů linek MHD, na rozdíl od pěších jsou orientované a spojují zastávky po trase každého spoje. Navíc nemají pevně dané ohodnocení, jelikož záleží na čase, kdy do vrcholu dorazíme. V grafu platí, že z každého vrcholu vede alespoň jedna *pěší hrana*. *Dopravní hrany* pak mohou vést pouze mezi vrcholy reprezentujícími zastávky. Vynechávají se v tomto případě všechny tranzitivní hrany, spojeny jsou tedy pouze zastávky bezprostředně po sobě následující na trase nějakého spoje.

Algoritmus na vstupu přijímá dva uzly (je důležité rozlišit startovní a cílový), počáteční čas a případné další volitelné parametry, jako například rychlost chůze. Jeho výsledkem je pak jedna nejrychlejší cesta mezi těmito body. V základní verzi se od Dijkstrova algoritmu principiálně neliší, jak ukazuje následující nástin *4-1*:

```
function FindPath(start, target, startTime, walkSpeed) :
1.  for each node:
2.      node.time := ∞ // každý vrchol zatím označen za nedosažený
3.      node.previous := NULL
4.  end for
5.  start.time := startTime
6.  var openset = { start } // množina otevřených vrcholů
7.  while openset is not empty :
8.      var current := extract node with min. node.time from openset
9.      if current = end then return current.time
10.     for each neighbour of current :
11.         var arrival := arrive time to neighbour
12.         openset <- neighbour // přidání mezi otevřené vrcholy
13.         if arrival < neighbour.time then
14.             neighbour.time := arrival
15.             neighbour.previous := current
16.         end if
17.     end for
18. end while
```

Alg. 4-1 – Pseudokód vyhledávacího algoritmu

[2] V grafech, na nichž Dijkstrův algoritmus funguje, se typicky používá termín *vrcholy*. V tomto případě však *vrcholy* jsou *uzly* OpenStreetMapy. V kontextu této práce považuji tedy oba zmíněné pojmy za synonyma.

Sousedy vrcholu na řádce 10 se zde myslí vrcholy dosažitelné pomocí libovolného z obou druhů hran. Liší se pak jen výpočet dočasné hodnoty času na řádku 11. U *pěších hran* je jednoduchý, vzdálenost lze určit z mapy a rychlost chůze je algoritmu zadána. V případě *dopravní hrany* je třeba znát spoje, které jsou jí reprezentovány (těch je typicky více než jeden), z nich se vybere ten první a zjistí se čas, ve kterém spoj projíždí následující zastávkou. Otevírá se tedy pouze vrchol následující zastávky na trase.

K implementaci algoritmu je potřeba nějaká struktura, která umí evidovat otevřené vrcholy a vracet ten s nejmenší dočasnou vzdáleností (v kódu 4-1 proměnná *openset*). Pro tento účel byla implementována jednoduchá halda, jejíž prvky udržují informace o každém navštíveném vrcholu. Pro efektivní běh algoritmu se také hodí umět rychle najít hrany vedoucí z daného vrcholu, tedy určit sousedy v mapě a zjistit seznam odjezdů ze zastávky. Na to bylo třeba pamatovat při implementaci příslušných tříd, konkrétně je to popsáno v kapitole 6.3.1 o datových třídách. Zjištění seznamu sousedů v mapě je časově konstantní operace, každý vrchol si tento seznam uchovává. Každá zastávka zná také navíc seznam odjezdů spojů, které jí projíždí, setříděný podle času odjezdu. Nalezení nejbližšího spoje podle aktuálního času pak tedy potřebuje čas O(log s) vzhledem k délce seznamu *s*.

4.1 Spolehlivost

Skutečný implementovaný algoritmus je nicméně o trochu složitější než uvedená verze. Dijkstrův algoritmus má ze své podstaty totiž několik vad, které nalezená spojení činí někdy až nepoužitelnými. Uvedená verze nijak nepenalizuje přestupy, nesnaží se jejich počet minimalizovat, takže ve výsledných spojeních se objevují hlavně chyba „předbíhání spoje“. Ta se projevuje následujícími způsoby:

1. Když je čas na čekání na zastávce dost dlouhý na to, aby cestující stihl spoj i při přesunu na další zastávku na trase, program tento přesun doporučí.
2. Pokud lze v některé zastávce vystoupit a dojít na některou z dalších zastávek stejného spoje, na které se dá na ten samý spoj vyčkat, program tento přesun doporučí (vzniká zejména u klikatých linek).

3. Příkladem: jsme na zastávce A a čekáme na spoj S do zastávky C přes zastávku B. Dříve však přijede spoj jiné linky, který jede pouze do zastávky B. Program doporučí do něj nastoupit a v zastávce B přestoupit do spoje S, na který jsme ale mohli počkat už na zastávce A a ušetřit si přestup i čas.

Uvedené důsledky nejsou vlastně chybami, nicméně nekorespondují se standardním chováním lidí, kteří by zbytečné přestupy nepodnikali. Nehledě na to, že nadcházet si spoj až dokud to jde, může být nebezpečné, pokud špatně odhadneme svou rychlost, nebo pokud daný spoj pojede dříve. Jako řešení se nabízely dvě metody. První je snažit se kromě doby cesty minimalizovat i nachozené metry, druhá pak snažit se místo toho minimalizovat počet přestupů. První metoda však selhává u „zbytečných přestupů“ (bod 3), naopak občas přidává nelogické přestupy za účelem ušetření si několika desítek metrů. Druhá metoda zase neřeší nadbíhání spojů (bod 1). Byl tedy potřeba nějaký kompromis – tím se právě stala **spolehlivost**.

Spolehlivost je hodnota z intervalu 0 – 1 (lze si ji představit jako procenta), která se počítá pro každý přestup a musí být nepřímo úměrná času, který je na přestup k dispozici (tedy rozdílu odjezdu spoje od příchodu k zastávce). Výsledná spolehlivost spojení je pak součin spolehlivostí všech přestupů. Tento model řeší všechny tři uvedené případy. Ad 1, přesun na vzdálenější zastávku by typicky snížil čas na přestup, proto takové spojení nebude doporučeno. Ad 2, zbytečný výstup a nástup by pouze snížil spolehlivost za stejného dojezdového času, spojení nebude doporučeno. Ad 3, opět by zbytečný přestup pouze způsobil snížení spolehlivosti. Tento model jsem vymyslel také proto, že v budoucnu by se spolehlivost mohla počítat složitějším způsobem, například i v závislosti na typu dopravního prostředku (např. metro je nejspolehlivější) nebo aktuálním čase (špička/noc).

Zbývá pouze vymyslet, jak to realizovat. Rozdíl je ten, že každý vrchol nyní může mít více dočasných časů pro různé spolehlivosti. Pokud se totiž do nějakého vrcholu dostaneme sice později, ale s větší spolehlivostí, nemůžeme tento záznam zahodit, protože se tato cesta může ukázat jako stejně rychlá a spolehlivější a měla

by tedy dostat přednost. Každý uzel tak obsahuje více záznamů (z položek *dist* a *previous* se stane jejich seznam), ty jsou seřazeny podle času a pouze první z nich je odkazován z haldy. Místo uzavírání vrcholu se pak uzavře pouze daný záznam a do haldy se zatřídí případný následující dočasný čas. Tato změna nicméně způsobí několikanásobné zvětšení grafu, protože štěpí vrcholy podle různých spolehlivostí, a při hledání na větší vzdálenosti pak může za cenu lepších výsledků vzrůstat čas běhu algoritmu až do řádu vteřin.

Uložení nalezeného spojení

Samozřejmě nalezení spojení by nemělo smysl, kdybychom výsledek hned zahodili, je tedy třeba zajistit, aby po skončení algoritmu bylo možno nalezenou trasu zrekonstruovat. To lze provést od konce, protože každý vrchol má přidruženu vlastnost *previous*, která ukazuje na jeho předchůdce na cestě. Nakonec se postupně dostaneme k vrcholu, který tuto hodnotu definovánu nemá, a právě ten je počátkem celé nalezené cesty. Nutno poznamenat, že vrcholem v tomto případě není uzel mapy, ale dvojice (uzel, spolehlivost).

5. Uživatelská dokumentace

Tato kapitola je věnována případným uživatelům programu. Vzhledem k jednoznačnému účelu programu není pochopení jeho ovládání příliš složité, nicméně ne vše je od pohledu zřejmé.

5.1 Instalace a spuštění

Program je distribuován v ZIP souboru (případně jiném komprimovaném formátu). Ten obsahuje následující soubory:

- data\
  - addresses.dat – databáze adres
  - nodes.dat – databáze uzlů mapy
  - stopnames.dat – databáze jmen zastávek
  - stops.dat – databáze zastávek
  - ways.data – databáze cest
- program\
  - PTN GUI.exe – spouštěcí soubor aplikace
  - Podpůrné knihovny potřebné pro spuštění programu

Pro instalaci programu je třeba rozbalit veškerý obsah do libovolné složky. Spouštěcí soubor aplikace obsahuje podsložka *program*.

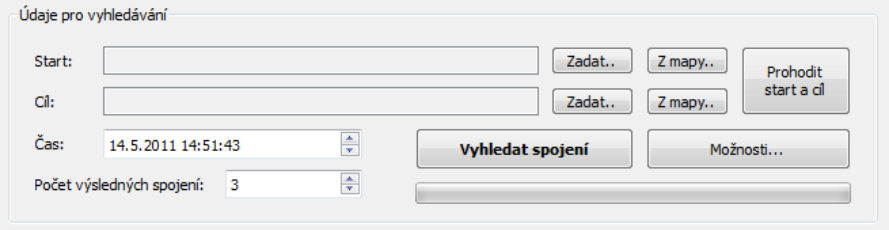
5.2 Hlavní okno

Většina práce v programu se odehrává pouze v jednom okně. V něm se zadávají parametry hledání spojení, provádí se samotné vyhledávání a zároveň pak zobrazují výsledky.

Hlavní okno aplikace je pomocí rámečků rozděleno do tří částí. V horní části se zadávají vstupní *údaje pro vyhledávání*. V rámečku pro *výsledky* se pak zobrazí seznam nalezených spojení. Vybrání konkrétního z nich ukáže detaily v rámečku *detail spojení*. Všechny tři části jsou podrobněji popsány níže.

5.2.1 Údaje pro vyhledávání

Části *údaje pro vyhledávání* je třeba věnovat pozornost nejdříve. Obrázek 5-1 ji ukazuje tak, jak vypadá po spuštění programu.



Obr. 5-1 – Rozhraní pro zadání vstupních údajů

Na první pohled je zřejmé, že políčka *Start* a *Cíl* je třeba vyplnit. K tomu je však třeba vědět, co vše vlastně může být v programu startovním nebo cílovým objektem. Všechny možnosti mají společné to, že musí jít o konkrétní místo na mapě, rozdíl je pouze ve způsobu, jakým bude místo určeno. Možnosti jsou:

a. zastávkou MHD
b. adresou
c. (GPS) souřadnicí v mapě

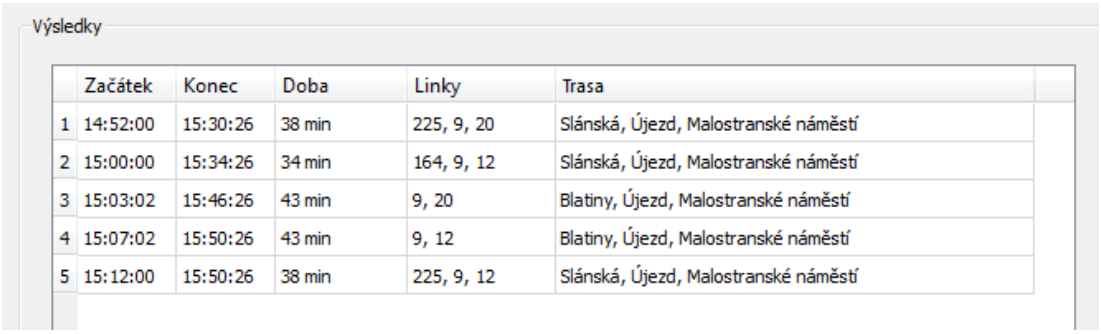
V prvních dvou případech se využije tlačítko *Zadat* u políčka *Start* nebo *Cíl*, které zobrazí příslušné okno pro zadání jména objektu. Jeho podoba a ovládání jsou popsány v kapitole 5.2. Zadat startovní nebo cílový bod pomocí souřadnic v mapě lze pomocí tlačítka *Z mapy*. To zobrazí speciální okno s mapovým rozhraním, ve kterém lze najít a vybrat libovolné místo. O mapovém rozhraní se detailněji píše v kapitole 5.3.

Dalšími údaji potřebným k vyhledání spojení jsou *počáteční čas* a *počet výsledných spojení*. Program vyhledává spojení z daného bodu v daném čase, po jeho nalezení opraví čas odjezdu/odchodu ze startovního místa podle prvního dopravního prostředku na cestě tak, aby jeho odjezd akorát navazoval na příchod na zastávku. Další zobrazené spojení pak bude takové, které by bylo nejrychlejší možné za předpokladu, že to předchozí právě ujelo.

Po zadání všech potřebných údajů dá kliknutí na tlačítko *Vyhledat spojení* programu pokyn, aby tuto akci vykonal. Při spojeních na dlouhé vzdálenosti (hodina a více) může vyhledávání chvíli trvat, program však vypisuje výsledky postupně. Vyhledávání je možno přerušit kliknutím na stejné tlačítko (v tu chvíli *Přerušit vyhledávání*). Je při tom třeba vzít na vědomí, že i zrušení vyhledávání může chvíli trvat, protože běh algoritmu není možné ukončit v libovolný čas.

5.2.2 Výsledky vyhledávání

Větší část okna zabírají výsledky vyhledávání, které se objeví po vyhledání spojení podle zadaných parametrů. Výsledky tvoří zobrazení stručných informací o každém z nalezených spojení, jak ilustruje obrázek 5-2 (každý řádek reprezentuje jedno spojení).



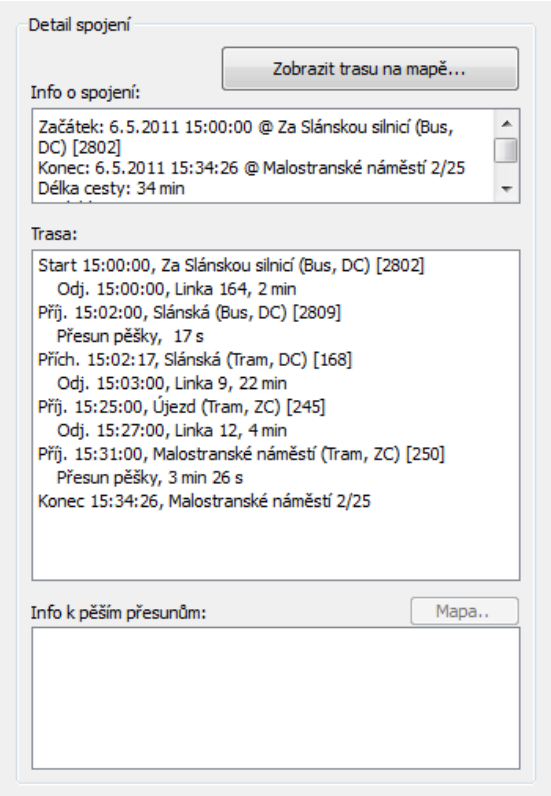
Obr. 5-2 – stručný seznam nalezených spojení

Spojení jsou seřazena chronologicky, *začátek* je čas, kdy je třeba být ve startovním bodě (respektive z něho vyrážet), *konec* je pak čas předpokládaného příchodu/příjezdu do bodu cílového. *Linky* jsou uvedeny v pořadí, v jakém jsou ve spojení použity, to samé platí i pro významné body na trase, což jsou zjednodušeně zastávky, ve kterých probíhají přestupy.

Toto samozřejmě nejsou veškeré informace, které o spojení lze z programu získat, vybráním některého z nich kliknutím myší na příslušný řádek lze zobrazit detaily spojení.

5.2.3 Detail spojení

Detail spojení je v okně umístěn vpravo dole. Do vybrání některého z výsledků vyhledávání jsou políčka prázdná. Jinak ale zobrazují informace o konkrétním spojení, jak ukazuje obrázek 5-3.



Obr. 5-3 – Detail spojení

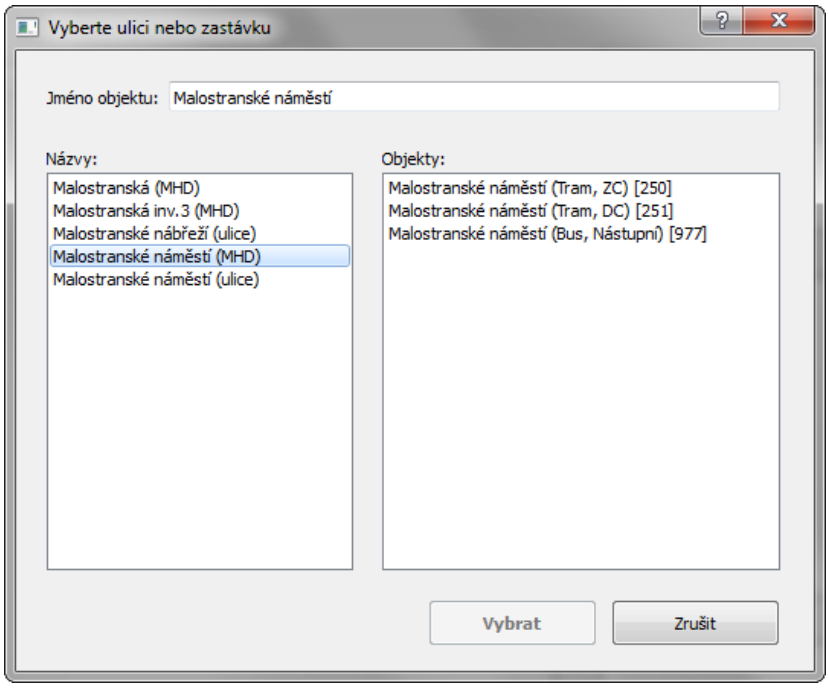
Info o spojení v horní části obsahuje informace o spojení jako takovém, většina údajů je vidět už přímo z *výsledků vyhledávání*. Navíc obsahuje informaci o součtu délek pěších přesunů a odhad spolehlivosti nalezeného spojení. Spolehlivost je závislá na počtu přestupů, přesněji na dobách, které jsou na přestupech k dispozici jako rezerva. Je udávána v procentech, nicméně nelze ji chápat jako přesné číslo vyjadřující nějakou pravděpodobnost, jde jen o odhad programu. Tlačítko *Zobrazit trasu na mapě*, které je zpřístupněno při vybrání spojení, zobrazí trasu v okně s mapovým rozhraním (o tom bude kapitola 5.3).

Pod *informacemi* je vypsána *trasa vybraného spojení*. Seznam obsahuje odjezdy z jednotlivých přestupních bodů a mezi nimi je vždy mírně odsazený řádek

informující o způsobu dopravy mezi sousedními body. U pěších přesunů je vybráním daného řádku možné získat ještě detailnější informace. V políčku *Info k pěším přesunům* se zobrazí slovní popis trasy. Kliknutím na tlačítko *Mapa* lze zobrazit na mapě pouze vybranou část trasy.

5.3 Okno pro zadání adresy nebo zastávky

Při zadávání výchozího nebo cílového bodu vyhledávání bude asi nejčastější použití existujícího objektu. K tomu slouží okno, které je ukázáno na obrázku 5-4. Jak se s ním pracuje, popisují následující odstavce.



Obr. 5-4 – Rozhraní pro vybrání adresy nebo zastávky

Základní zadávání jména objektu

Psaním do kolonky *jméno objektu* se v *seznamu názvů* (vlevo) objevují jména zastávek a ulic, která obsahují vložený text jako podřetězec (na obrázku 5-4 vznikl *seznam názvů* zadáním slova „malostranské“). Nezáleží při tom na velikosti písmen ani na diakritice. Po vybrání určité položky ze *seznamu názvů* se vybraný název použije, vyplní se jím tedy políčko *jméno objektu* a v *seznamu objektů* (vpravo) se zobrazí konkrétní objekty (situace na obrázku 5-4). V případě jména zastávky se zobrazí zastávky daného jména, v případě ulice jsou to všechny adresy v ulici.

Zastávky jsou vypsány ve tvaru „*Jméno zastávky* (*Typ*, *Směr*) [*ID*]“. *Typ* může být buď „Tram“ pro tramvajové ostrůvky, „Bus“ pro autobusová nástupiště, nebo „Metro“ pro stanici metra. *Směr* určuje, zda jde o zastávku směrem do centra („DC“), nebo centra („ZC“), či dokonce „Nástupní“ nebo „Výstupní“. Označení ZC/DC je však pouze orientační a zvláště v samotném centru ani není příliš nápomocné, jelikož se tam směr definuje dost těžko. Nakonec *ID* je vnitřní identifikátor, slouží pouze k odlišení jednotlivých zastávek (protože např. u přestupních uzlů je zastávek do centra i z centra více). Adresy jsou vždy ve formátu „*Ulice čp./čo.*“, kde *čp.* reprezentuje číslo popisné (unikátní v rámci katastrálního území) a *čo.* zastupuje číslo orientační (unikátní v rámci ulice). Seznam je řazen podle druhé z hodnot. Po vybrání konkrétního objektu (zastávky nebo adresy) se pak zpřístupní tlačítko *Vybrat*, kterým se zvolí vybraný objekt a okno se zavře.

Zjednodušení při vybírání objektů

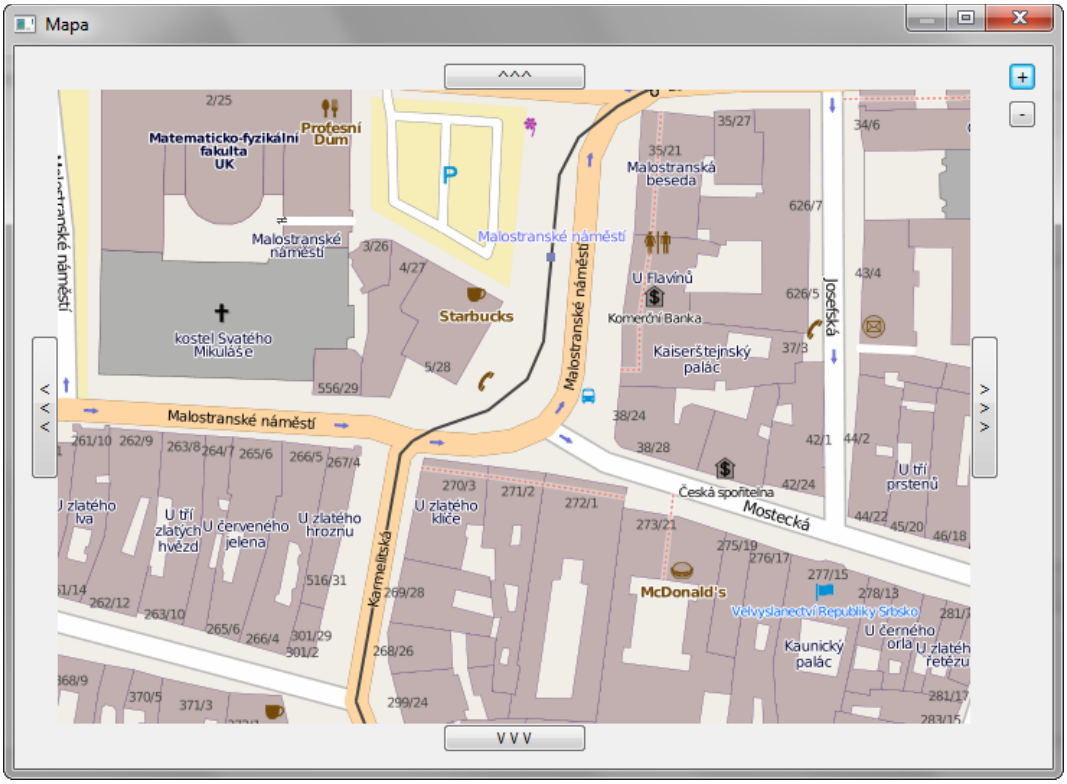
Protože zadávání startovních a cílových bodů je v programu poměrně častá akce, existuje i několik způsobů zjednodušení zmíněného postupu.

V případě, že znáte celé jméno daného objektu, lze jej napsat do políčka *jméno objektu* a pokud je zadání přesné (až na velikost písmen a diakritiku), zobrazí se rovnou objekty daného názvu v *seznamu objektů*. Není tedy třeba mezikrok, ve kterém se vybírá *název objektu* ze seznamu. V případě ukázkového příkladu tedy stačilo napsat do vyhledávacího políčka „malostranské náměstí“ a rovnou by se ukázaly zastávky a adresy tohoto jména v *seznamu objektů*.

V případě adres je navíc možné zapsat za jméno ulice i číslo. Například v případě zadání textu „malostranské náměstí 2“ by se v seznamu objektů zobrazily dvě adresy: „Malostranské náměstí 271/2“ a „Malostranské náměstí 2/25“, protože ze zadání není jasné, zda je zamýšleno číslo popisné či orientační. Lze však zadat i obě čísla (oddělená lomítkem), pak je adresa jednoznačná a v *seznamu objektů* se zobrazí jediný výsledek. Pokud danému zadání odpovídá pouze jediná položka, automaticky je vybrána a dojde ke zpřístupnění potvrzovacího tlačítka *Vybrat*. Místo klikání na něj také stačí stisk klávesy Enter.

5.4 Mapové rozhraní

Mapové rozhraní je nedílnou součástí programu. Má dva hlavní účely: lze přes něj zadávat výchozí nebo cílový bod hledání a zobrazuje trasy nalezených spojení. Jde o samostatné okno, které tvoří pouze mapa a ovládací tlačítka, jak je vidět na obrázku 5-5. Čtyři tlačítka jsou na posouvání směru a dvě na změnu přiblížení. Nicméně posouvání mapy je možné provést i za pomoci myši stisknutím levého tlačítka a posouváním do stran. K přibližování a oddalování pak dobře poslouží kolečko myši; přibližuje se vždy k aktuální pozici kurzoru na mapě. Okno lze také libovolně rozšiřovat a zužovat podle potřeby.



Obr. 5-5 – Okno mapového rozhraní

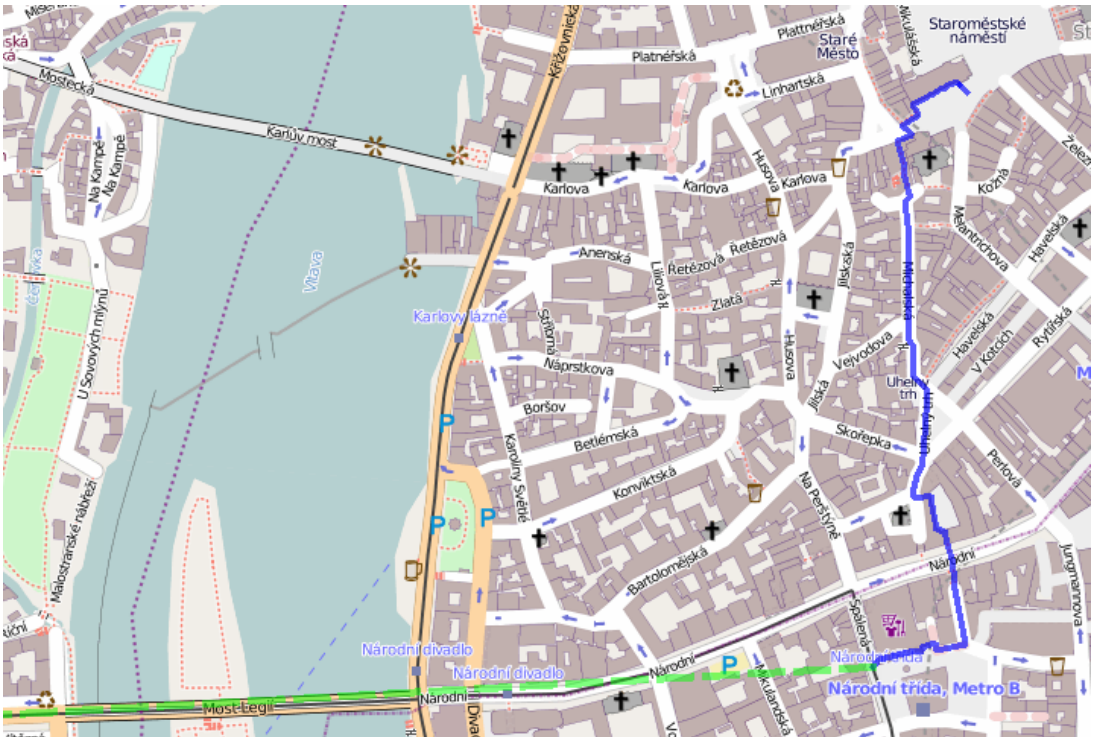
Samotné vyhledávání spojení nepotřebuje aktivní připojení k internetu. Pro využití mapy však je internetové připojení potřeba. Distribuce map společně s programem by totiž byla neúnosná, proto se její části stahují z internetu až při potřebě zobrazit danou oblast. Důsledkem je zobrazování mapy „po kouskách“. Program si jednou stažené části map ukládá, čímž šetří čas i vytížení internetového připojení při opětovném zobrazování stejných oblastí.

Použití mapového rozhraní při zadávání souřadnic

Jednou z možností, jak zadat výchozí nebo cílový bod vyhledávání, je souřadnicí vybranou z mapy. V tomto režimu, navíc oproti standardnímu ovládání, dvojklikem na nějaké místo v mapě vyberete bod (respektive jeho souřadnice), který se nachází pod kurzorem.

Použití mapového rozhraní při zobrazování trasy

Druhou, hlavní funkcí je zobrazování trasy nalezeného spojení v mapě. Přes mapu je v tomto režimu znázorněna trasa, například jak ukazuje obrázek 5-6.



Obr. 5-6 – Vyobrazení trasy v mapovém rozhraní

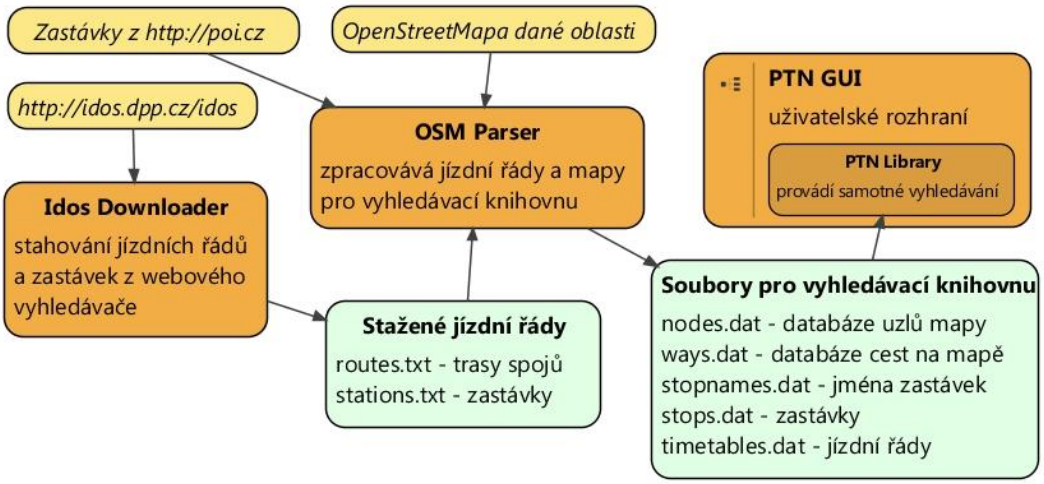
Jak je vidět, různé druhy přesunů jsou v mapě barevně odlišeny. Modře jsou znázorněny pěší cesty a zelená přerušovaná čára nastiňuje spojení dopravní. Zatímco u spojení pěšího je cesta znázorněna přesně, dopravní spojení jsou zobrazena vždy přímým spojením počáteční a koncové zastávky.

5.5 Nastavení

Poslední dosud nezmíněnou částí programu je nastavení. To se dá měnit kliknutím na tlačítko *Možnosti* v hlavním okně vedle tlačítka pro vyhledání spojení. Je možno nastavit *rychlost chůze*, *maximální dobu čekání na zastávce* a *minimální čas na přestup*. Změny v nastavení se ukládají, program si tedy pamatuje uživatelovy preference i při opakovaném spuštění.

6. Implementace

Implementace je rozdělena do čtyř oddělených částí, jak již bylo nastíněno v kapitole 3.4. Na následujících stránkách jsou rozebírány jejich funkce, architektura a detaily implementace. K vývoji bylo použito Visual Studio 2010, v jehož terminologii se jednotlivé části nazývají *projekty*; stejně je budu označovat i v této práci. Každému projektu pak odpovídá samostatná aplikace. Obrázek 6-1 připomíná rozdělení včetně vstupu a výstupu jednotlivých aplikací. Oranžově jsou obarveny jednotlivé programy, žlutě zdroje dat a zelené jsou vstupní a výstupní soubory.



Obr. 6-1 – Diagram projektů a dat

6.1 Stahování dat

Protože jsem si jako ukázkové město pro práci vybral Prahu a rozhodl se také pro ni ukázkovou aplikaci vytvořit, bylo potřeba získat pražské jízdní řády. Ty nikde volně ke stažení nejsou, takže v rámci použitelnosti výsledného programu bylo jedinou možností stáhnout potřebná data z webového IDOSu hlavního města *[3]*. To má na starosti projekt ***IdosDownloader***, který kromě stahování provádí také filtrování dat od zbytečných HTML tagů, což šetří diskový prostor a usnadňuje další zpracování, jelikož stažených stránek je mnoho.

6.1.1 Princip stahování

Je jasné, že webový IDOS sám obsahuje veškeré potřebné informace (tedy kompletní jízdní řády všech linek), otázkou však je, jak je získat kompletní a zda je to vůbec možné. K tomu nakonec dobře poslouží stránka „Dráha spoje“ (obr. 6-2), která v nalezeném spojení zobrazuje celou trasu daného spoje včetně časů v jednotlivých zastávkách. Vhodným dotazem si pak jde webovému vyhledávači říct o libovolný spoj. Sloučením všech spojů pak může vzniknout kompletní jízdní řád.

Obr. 6-2 – Dráha spoje [3]

Na obrázku 6-2 je také vidět, že u každé zastávky se nachází odkaz na mapu, která je součástí webového IDOSu. Ta opravdu zobrazuje pozici zastávky, z čehož vyplývá, že systém rozlišuje konkrétní *zastávky* (nejen jejich jména). Z tohoto odkazu lze vyčíst vnitřní identifikátor zastávky a cílová stránka umí na základě onoho identifikátoru zobrazit bod na mapě. V kódu stránky je navíc souřadnice zastávky přímo napsána, není to však souřadnice GPS, nýbrž UTM *[5]*; jejich převod už ale není v kompetenci projektu *IdosDownloader*, ten má za úkol soubory pouze stáhnout a vybrat z nich důležité údaje.

6.1.2 Filtrování stažených dat

Při stahování stránek s jednotlivými spoji vznikne mnoho souborů. Jen samotných spojů je kolem 70 000, zastávek pak kolem 3 000. Každý spoj i zastávku reprezentuje jedna stránka, tedy jeden soubor HTML. Jde o relativně malé soubory (typicky 10 – 20 kB), nicméně dohromady zaberou kolem 1 GB, přičemž většinu obsahu tvoří zbytečně „ukecaný“ HTML kód. Navíc se s takovým množstvím souborů špatně pracuje. Filtrování (nebo také parsování) stažených dat vytvoří ze skupiny

HTML souborů jeden textový soubor, ve kterém už jsou pouze potřebné údaje. Filtrování je různé pro jízdní řády a pro zastávky, ve výsledku vzniknou dva soubory, jeden se všemi spoji a jeden se zastávkami.

6.1.3 Použití programu

Stahování stránek z internetu i jejich filtrování jsou oboje značně nespolehlivé operace. Server může dočasně přestat odpovídat nebo vrátit jinou stránku, než se předem očekává (např. při chybě nebo při změnách na serveru), což může způsobit chybu při následném zpracování stránky programem. Z tohoto důvodu je projekt rozdělen do dvou částí. Nejprve se stáhnou všechny potřebné stránky a uloží se tak, jak jsou, což vytváří jakýsi záchytný bod. Potom se teprve jednotlivé stránky zpracují a vznikne textový soubor obsahující potřebné informace. Kdyby při zpracování došlo k chybě, nebo ještě hůře žádná chyba nahlášena nebyla, ale stránky byly zpracovány špatně, nemusíme je znovu stahovat. Odpovídají tomu i přípustné parametry programu. Ty musí být vždy dva, u každého jsou k dispozici dvě možnosti:

IdosDownloader.exe {download|parse} {routes|stations}

První parametr rozhoduje, zda se budou data stahovat, nebo se mají zpracovat stránky již stažené. Druhý určuje, zda se budou stahovat/zpracovávat jízdní řády (resp. trasy – *routes*) nebo zastávky (*stations*). Stahování probíhá pomalu, aby nevytěžovalo server, pro získání kompletních dat je vhodné provádět tuto operaci přes noc. Při stahování tras se zároveň vytváří seznam použitých zastávek, který se pak použije při stahování zastávek, je tedy vhodné provádět stahování jízdních řádů nejdříve. Zároveň je logické, že zpracování může proběhnout až po stažení. Z toho vyplývá i téměř jednoznačné pořadí při použití programu pro získání kompletních jízdních řádů (pouze kroky 3 a 4 je možno prohodit).

1. *IdosDownloader.exe download routes*[3]
2. *IdosDownloader.exe download stations*

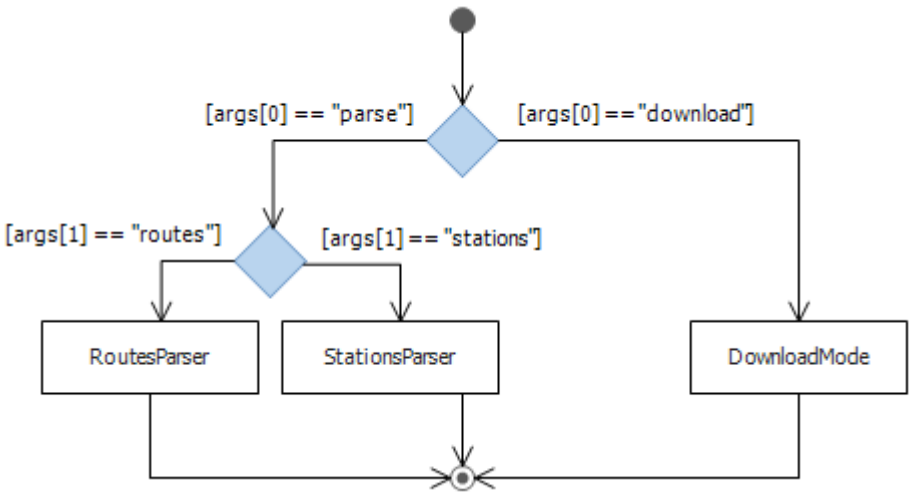
[3] Při stahování jízdních řádů je třeba zadat počáteční a koncový index spoje. Za *start index* se zvolí 0, *end index* lze zjistit např. z http://idos.dpp.cz/IDOS/TTInfo.aspx?tt=pid odečtením jedničky od *počtu spojů* pod *Pražskou integrovanou dopravou*.

3. *IdosDownloader.exe parse routes*
4. *IdosDownloader.exe parse stations*

Výsledkem jsou dva soubory: routes.txt a stations.txt reprezentující spoje a zastávky. Jako meziprodukt vzniknou také složky Routes a Stations obsahující stažené stránky. Ty se doporučuje ponechat pro případ dodatečného nalezení chyby v datech, nejsou nicméně dále potřeba.

6.1.4 Architektura

Obrázek na diagramu 6-3 představuje třídy programu a ukazuje, které z nich se předá řízení v závislosti na argumentech programu. Zbytek kapitoly pak sdělí detailnější informace o jednotlivých třídách.



Obr. 6-3 – Základní třídy programu IdosDownloader

Třída DownloadMode

Je-li přes příkazovou řádku stanoveno, že program má stahovat spoje nebo zastávky, předává se řízení třídě *DownloadMode*. Ta má za úkol zjistit od uživatele případné další údaje, spustit stahování, informovat uživatele o průběhu a umožnit mu akci přerušit. Jako parametr obdrží *zdroj*, což mohou být buď spoje (*routes*) nebo zastávky (*stations*). Podle toho určí obecný tvar webové stránky, která se bude stahovat, což může být jedna z následujících konstant:

```
public const string RoutesWebAddress =
"http://idos.dpp.cz/IDOS/Route.aspx?ttInd=1&i={0}&tt=pid";
public const string StationsWebAddress =
"http://idos.dpp.cz/IDOS/Map/MapView.aspx?a=SHOWLOC&station={0}&ttInd=1
&tt=pid";
```

Vyznačená *{0}* se pak pro každý spoj/zastávku přepisuje jeho/jejím indexem. Tuto adresu, společně se seznamem indexů, pak obdrží třída *PageDownloader*, která pro každý index ze seznamu jím nahradí onu *{0}* a stáhne stránku, kterou reprezentuje výsledný odkaz. To probíhá asynchronně a třídě *DownloadMode* při tom zpět posílá události při stažení každého souboru i při dokončení stahování všech souborů, aby bylo možno informovat uživatele o aktuálním stavu.

Třída RoutesParser

Třída *RoutesParser* má pouze jedinou statickou metodu a také jediný úkol: zpracovat stažené soubory s drahami spojů tak, aby neobsahovaly nepotřebné údaje, a sloučit je všechny do jednoho souboru. Vstupem je tedy až několik desítek tisíc zdrojových kódů stránek podobných té z obrázku 6-2. Výstup tvoří jeden soubor *routes.txt*, ve kterém jsou všechny spoje pod sebou a to v následujícím tvaru:

```
{Tram|Bus|Metro|NTram|NBus|...} <číslo linky>
<jméno zastávky> IDNUM <ID zastávky> <čas příjezdu> <čas odjezdu>
...
(další zastávky na trase)
...
<dny, ve které je spoj vypraven>
(prázdná řádka)
(další spoj...)
```

Transformace probíhá pouze na základě odstraňování přebytečných částí stránek, jako jsou HTML tagy, nadpisy a hypertextové odkazy.

Třída StationsParser

StationsParser funguje principielně podobně jako *RoutesParser*. Vstupem jsou HTML soubory s javascriptovým kódem zobrazujícím mapu a na ní pozici dané zastávky. Výstupem je jeden soubor *stations.txt*, kde každou zastávku zastupuje jediný řádek v následujícím tvaru:

```
<jméno zastávky> IDNUM <ID zastávky> <x-souřadnice> <y-souřadnice>
```

Opět se pouze vytahují důležité údaje a nijak se neřeší jejich význam. Dešifrovat význam souřadnic má za úkol až následující projekt.

6.2 Příprava dat

Pro uživatele neviditelnou, avšak velice důležitou součástí práce je i projekt ***OSMReader***. Opět jde pouze o pomocnou aplikaci, ač tentokrát o dost komplikovanější, než byl *IdosDownloader*.

Účelem tohoto projektu je připravit veškerá data pro samotnou vyhledávací knihovnu. Výstupem je několik datových souborů, o nichž bohatě pojednává příloha B. Vstupním datům se pak věnuje celá následující kapitola.

6.2.1 Vstup

1. OpenStreet mapa

Základem je mapa oblasti, kterou se chystáme dopravně obsluhovat, reprezentovaná jedním OSM souborem *[10]*. Od toho je také odvozen název projektu. *OSMReader* danou část mapy zpracuje, odstraní z ní nepotřebné části, jako jsou budovy, řeky, apod., které nejsou k vyhledávání potřeba, jelikož hledání bude probíhat výhradně po cestách. Zároveň pro uložení použije úspornější formát, než je XML, kterým jsou popsány OpenStreet mapy.

2. Jízdní řády

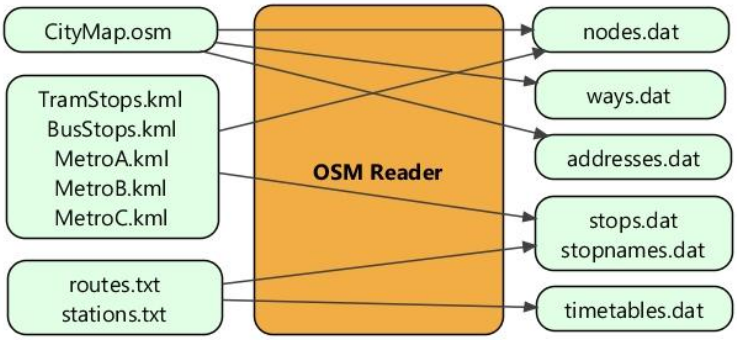
Samozřejmě je potřeba zpracovat i soubory, které jsme získali díky projektu *IdosDownloader*. Program očekává soubory routes.txt a stations.txt ve formátu popsaném v 6.1.4. Zde se stává název projektu *OSMReader* poněkud zavádějícím, nicméně zpracování map bylo zamýšleno jako původní a hlavní úkol programu, proto jsem tento název ponechal.

3. Zastávky z poi.cz

Z důvodů zmíněných v kapitole 3.2 se používá i databáze zastávek získaná z poi.cz. Do budoucna je možné závislost na těchto souborech zrušit, hodí se ale ke korekci a také v případě změny webových stránek a nemožnosti dalšího získávání zastávek. Program přijímá soubory vyexportované z poi.cz ve formátu KML *[12]*.

Shrnutí

Obrázek 6-4 obsahuje všechny soubory, které jsou potřeba na vstupu, společně se soubory, které se objevují na výstupu. Naznačen je vliv vstupních souborů na výstupní.



Obr. 6-4 – Vstup a výstup programu OSMReader

6.2.2 Použití programu

Se vstupem a výstupem jsou pevně spjaty argumenty programu:

OSMReader.exe <osm> <trams> <buses> <metro_prefix> <timetables> <output>

Zde *<osm>* zastupuje cestu k souboru s mapou (v příkladu „CityMap.osm“), *<trams>* cestu k souboru s tramvajovými zastávkami („TramStops.kml“), *<buses>* analogicky pro autobusové a *<metro_prefix>* soubory se zastávkami linek metra bez konkrétního písmene a přípony (u obr. 6-4 by to bylo „Metro“); *<timetables>* je třeba nahradit složkou obsahující soubory routes.txt a stations.txt a *<output>* je složka, do které se mají uložit výsledné datové soubory.

Program po spuštění sám začne zpracovávat vstup, je třeba mu dát čas v řádu desítek vteřin až minut na provedení operace. Pokud vše probíhá, jak by mělo, budou se pouze vypisovat prováděné akce.

6.2.3 Rozdělení projektu

Hlavním úkolem programu *OSMReader* je parsování souborů (znázorněno na obr. 6-4). Tento úkol plní tři skupiny tříd:

1. *Parsery*, které obstarávají načtení vstupu, včetně interpretace dat.
2. *Seznamy*, které mají za úkol udržovat kolekci nějakých prvků a organizovat jejich uložení
3. *Datové třídy* tvořící prvky *seznamů*

Mimo tuto hierarchii leží kromě pomocných tříd také skupina zajišťující tzv. *reporting*. Tím se budeme zabývat nejdříve.

6.2.4 Reporting

Během parsování se zpracovává velké množství údajů. V těch ale můžou být chyby a není v lidských silách je předem odhalit a připravit se na všechny možné situace. OpenStreet mapy mají sice definovaný formát, nicméně ten neurčuje množství informací, které musí být o každém objektu známé. Navíc tyto mapy jsou tvořeny „pouze“ lidmi, tudíž mohou obsahovat chyby nebo nestandardní záznamy. Z toho plyne, že kromě struktury se nelze na nic spoléhat. Například rozhodně není pravidlem, že každý dům má číslo popisné i číslo orientační a že pokud je má, jsou to opravdu čísla. U databáze zastávek získávané z poi.cz je situace podobná, ne-li horší. Názvy zastávek totiž obsahují různé informace navíc (např. identifikaci, směr, přestup na metro), jejichž problém spočívá zejména v nejednotnosti formátu. Občas chybí některý údaj nebo jen obyčejná závorka. Asi nejsložitější na celém projektu *OSMReader* bylo, aby dokázal být co nejvíce benevolentní vzhledem k chybám ve vstupu, dokázal chybná data opravit, je-li to možné, a informoval o všem uživatele. Výsledkem je, kromě jiného obslužného kódu navíc, skupina tříd nazývaná

Reporting. Jak název napovídá, jejich účelem je poskytnout informace o chybách a nestandardních situacích během parsování. Protože těchto „chyb“ se typicky objeví velké množství, nemá smysl je při běhu vypisovat. Kromě výstupních datových souborů tedy program generuje do složky, ve které je umístěn, soubory ve tvaru *report.*.log*. Jediné chyby, které se vypisují i do konzole jsou ty, které program nedokáže sám rozumně opravit a které mají typicky za důsledek nekonzistenci dat. K těm dochází spíše kvůli vnitřní chybě samotného parseru a vyžadují zvýšenou pozornost.

Každý z *log* souborů obsahuje seznam nějakých objektů s případnými dalšími informacemi. V hlavičce je pak napsána informace, čím se vlastně vypsané objekty provinily. Například soubor *report.stopstoofarfromways.txt* obsahuje „zastávky, které jsou příliš vzdálené od nejbližší cesty“. Je tvořen seznamem zastávek, včetně jejich vzdálenosti od nejbližší cesty. Zvláštní význam pak mají soubory *report.warnings.log*, který obsahuje již zmíněné kritičtější chyby, a *report.infos.log*, který obsahuje informace o dalších nestandardních situacích, které ale nejsou kritické.

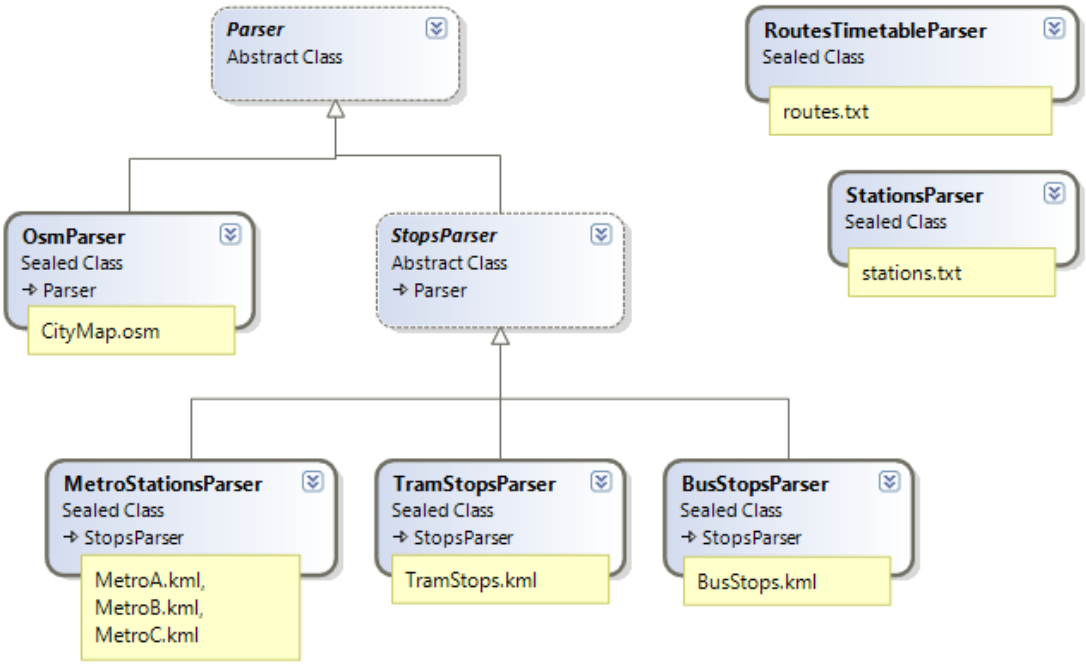
Implementace reportingu

Základem implementace *reportingu* je generická třída *Report<T>*, jejíž instance odpovídá jednomu výslednému souboru. Typový parametr *T* je typem objektu, který daný soubor obsahuje, přítomen je tedy seznam objektů typu *T*. Dále pak třída obsahuje identifikaci, která bude součástí jména souboru a stručný popis chyby. Před ukončením programu je instruována k uložení obsahu na disk. Vytvoří tedy soubor ve tvaru *reporting.<identifikace>.log*, do hlavičky vloží informační řetězec a seznam objektů. K tomu se využívá metoda *ToString()*, kterou v .NETu musí mít každý objekt. Statická třída *Reporting* pak obsahuje pro každou typičtější chybu jednu instanci třídy *Report<T>*.

6.2.5 Parsery

Parsery jsou zejména třídy odvozené od abstraktní třídy *Parser*, která sdružuje třídy načítající data z XML. K nim patří ještě dvě třídy pro načítání souborů routes.txt

a stations.txt, které v XML formátu nejsou. Každá instance nějaké takové třídy má na starosti parsování jednoho souboru. Tuto skupinu ukazuje obrázek 6-5. U jednotlivých instanciovatelných tříd jsou zároveň uvedeny soubory (z obr. 6-4), které má daná třída zpracovávat.



Obr. 6-5 – Parsery (skupina tříd)

Třída Parser

Abstraktní třída *Parser* obsahuje jméno XML souboru a instanci třídy *XmlDocument*, která obsahuje načtený vstupní soubor. Abstraktní metoda *Parse* má pak za úkol nějakým způsobem vstupní soubor zpracovat. Odvozené *parsery* tedy musí „pouze“ implementovat tuto metodu.

Třída OsmParser

Třída *OsmParser* zpracovává vstupní OSM soubor. Načtené informace předává *seznamům adres, uzlů, zastávek* a *cest* (o *seznamech* detailněji v kapitole 6.2.7).

Účelem parsování OSM souboru je zejména zmenšení objemu dat a odstranění nepotřebných prvků z mapy. Vstupní soubor s pražskou mapou má totiž kolem 250 MB, zatímco výstupní datové soubory, které jsou výsledkem zpracování mapy (tedy

nodes.dat, addresses.dat a ways.dat), dohromady nezaberou ani 5 MB. Třída (resp. její metoda *Parse*) prochází vstupní OSM soubor[4], přičemž na každý element „node“ reprezentující uzel volá metodu *AddNode*, která rozpozná typ uzlu a buď jej zahodí, nebo přidá do *seznamu uzlů* nebo *adres*. Na elementy „way“ reprezentující cestu se volá metoda *AddWay*, která zjistí, o jaký typ cesty jde a případně ji zařadí do *seznamu cest*.

Třída StopsParser

Abstraktní třída *StopsParser* je vytvořena z toho důvodu, že všechny soubory se zastávkami (tedy soubory *.kml) jsou stejného typu, sdružuje tedy společné charakteristiky, zejména načítání, a částečně i interpretaci, nicméně každý typ zastávky vyžaduje trochu jiné zacházení, které pak dořeší odvozené třídy.

Informace o formátu KML lze získat např. z *[12]*, pro nás je důležité však pouze to, že jde opět o XML a každá zastávka je reprezentována elementem *placemark*, který obsahuje podelementy *name, description* a *point*. Vzhledem k této jednotné struktuře může třída *StopsParser* implementovat metodu *Parse*, kterou pak potomci této třídy už implementovat nemusí. Metoda *Parse* volá na každý podelement elementu *placemark* příslušnou funkci (*ParseStopName, ParseStopDescription, ParseStopPoint*). Ty jsou všechny virtuální, první zmíněná dokonce abstraktní, potomci je tedy mohou implementovat po svém. Druhou abstraktní metodou je *CreateStopNode*, která se volá ve výchozí implementaci funkce *ParseStopPoint* a slouží k vytvoření uzlu v mapě, který bude danou zastávku reprezentovat. To je různé podle jejího typu (tramvajové se tvoří na tramvajových tratích, autobusové na silnicích). Abstraktnost funkce *ParseStopName* plyne z toho, že u každého typu zastávek jsou názvy zapsány trochu jinak. Po zpracování všech tří podelementů elementu *placemark* je výsledný objekt přidán do *seznamu zastávek*, pokud však

[4] V práci počítám s tím, že se čtenář s formátem OSM seznámí např. z *[10]*, nicméně k porozumění by mělo stačit vědět, že mapu tvoří XML soubor, pod jehož kořenovým elementem jsou uzly (element *node*) a cesty (element *way*), přičemž cesty se skládají z uzlů. Zároveň každý element může obsahovat různé *tagy* ve tvaru <tag k=“klíč“ v=“hodnota“ />, které o něm sdělují dodatečné informace.

některá z načítajících funkcí nevrátí chybovou hodnotu indikující, že zastávka nemá být přidána.

Třída TramStopsParser

Třída *TramStopsParser* je odvozená od třídy *StopsParser*, která většinu práce udělá za ní. Implementuje pouze ony dvě abstraktní metody. Metoda *CreateStopNode* nechá *seznamem cest* vytvořit uzel na tramvajové trati nejbližší uvedeným souřadnicím zastávky. Metoda *ParseStopName* si musí poradit s názvy zastávek, které obsahují i jejich identifikátory[5]. Obecný formát se dá těžko popsat, jelikož není předepsán, příklady jmen mohou být třeba „TRAM-Čechův most-68/3 (zc)“ nebo také „TRAM-Hlubočepy-občasná nástupní-147/1 (dc)“.

Třída BusStopsParser

Situace u třídy *BusStopsParser* je obdobná jako u tramvajové verze. Navíc je ale třeba vyrovnat se s tím, že názvy zastávek nejsou tak důsledně rozdělovány pomlčkami (viz např. „BUS-Nové Butovice (metro B) výstupní 602/1“). Autobusy mají na rozdíl od tramvají v elementu *description* uvedenu adresu nebo jen ulici, ve které zastávka leží, proto třída *BusStopsParser* implementuje i metodu *ParseStopDescription*. Vytěžené jméno ulice se použije při implementování metody *CreateStopNode*, která se primárně pokouší o vytvoření uzlu na dané ulici. Posledním rozdílem je, že seznam obsahuje i zastávky mimo území Prahy, které lze rozeznat od pražských pomocí *description*. Ty se do výsledného *seznamu zastávek* nepřidávají, čehož se docílí tím, že metoda *ParseStopDescription* vrátí chybovou hodnotu *false*.

Třída MetroStationsParser

U metra je situace o něco odlišnější. Souřadnice uvedené v KML souborech totiž reprezentují vstupy do metra, ne samotné zastávky. Zde se využívá toho, že

[5] Identifikátory zastávek jsou tvořeny dvěma čísly, která nazývám *ID* a *SubID*, bývají zapsány ve tvaru *ID*/*SubID*. Hodnota *ID* přibližně odpovídá skupinám zastávek, *SubID* je pak identifikace v rámci skupiny, nicméně neplatí to vždy. Stejnými identifikátory bývají označeny zastávkové sloupky. Při stahování jízdních řádů z *[13]* lze též tyto identifikátory zahlédnout. Přestože jsem to měl původně v plánu, tyto identifikátory se nakonec v programu vlastně k ničemu nepoužívají.

alespoň stanice metra jsou v OpenStreet mapách zaneseny všechny, jsou tedy použitelné jako zdroj. U některých stanic se dokonce objevují pokusy propojit ji s okolím podzemními cestami, které mapy také podporují (lze to pozorovat např. u stanic Dejvická a Hradčanská). U většiny stanic toto však zatím není, proto její propojení se vstupem na povrchu program provádí provizorním přímým spojením bodů podzemními schody.

Třída *MetroStationsParser* tedy ve skutečnosti zastávky nepřidává, metoda *ParseStopPoint* načte souřadnice a následně vrátí *false*, tedy chybovou hodnotu, která způsobí vyřazení dané zastávky. Místo toho se však v metodě *CreateStopNode* nechá na nejbližší cestě/ulici vytvořit uzel, který se propojí novou cestou se skutečnou stanicí metra.

Třída StationsParser

Zbylé dvě třídy fungují trochu jinak než ty předchozí. Nejsou totiž potomky abstraktní třídy *Parser*, jelikož jejich zdrojem není XML, ale textový dokument. Třída *StationsParser* také není přímo propojena s nějakým *seznamem*, do kterého by rovnou ukládala data. Zastávky jsou totiž zpracovávány třídami odvozenými od *StopsParser*, zatímco tato třída má význam trochu jiný.

Třída *StationsParser* má na základě údajů v souboru stations.txt vytvořit propojení mezi identifikátory zastávek používané ve webovém IDOSu[6], odkud jsou data stahována, a zastávkami načtenými předchozím zmíněnými třídami. V konstruktoru obdrží *seznam zastávek* a vstupní soubor a zavolání metody *Parse* vrátí *Dictionary<int, Stop[]>* neboli překlad identifikátoru používaného v jízdních řádech IDOSu na odpovídající načtené zastávky. Teoreticky by stačilo propojení s jednou zastávkou (*Stop* místo *Stop[]*), nicméně párování nemusí být přesné a kvůli chybám v souřadnicích uvedených v IDOSu také často není; tedy když se později zjistí, že navrhovaná zastávka je například jiného typu než spoj (např. autobus nemůže stavět na stanici metra), zkusí se použít další v seznamu.

[6] Zde je podstatné připomenout, že jako zdroj databáze zastávek se primárně používají KML soubory stažené z poi.cz, které musí být v tuto chvíli již načtené. Webový IDOS nicméně používá své identifikátory zastávek, je tedy třeba zjistit, který identifikátor odpovídá které načtené zastávce.

Již bylo nastíněno, že souřadnice přečtené z map webového IDOSu nejsou souřadnicemi GPS, nýbrž UTM *[5]*, a že *IdosDownloader* je při stahování nepřevádí, pouze předává dál. K přepočtu mezi GPS a UTM souřadnicemi lze použít např. *[14]*. Kód této stránky je v javascriptu, upravená verze běží i v *OSM Parseru* jako funkce *MapFunctions.ConvertUtmToWsg84*.

Třída RoutesTimetableParser

Druhým *parserem*, který není odvozen od třídy *Parser*, je třída *RoutesTimetableParser*. Jejím vstupem předávaným v konstruktoru je cesta k souboru routes.txt, ze kterého se jízdní řády sestavují, a mapování zastávek vytvořené třídou *StationsParser*. Výstupem metody *Parse* je *seznam linek*, což je de facto kompletní jízdní řád.

Úkol této třídy je rozdělen do dvou fází. V té první se načtou všechny spoje a roztřídí se podle linek. Každý spoj je v této fázi uložen v samostatném *směru*. V druhé fázi se pak *směry* sloučí do typicky dvou a z původních *směrů* se stanou *trasy* směrů nových.

Načtení dat není algoritmicky příliš zajímavé, snad s výjimkou načtení posledního řádku, na kterém se nachází krátký text informující o dnech, ve které je spoj vypravován. Snahou programu je importovat všechny spoje, takže jízdní řád je správný i nějaký čas po zpracování, jelikož v databázi webového IDOSu jsou obsaženy i spoje, které zatím nejezdí, ale v blízké době budou. Zmíněný řádek nemá jasně definovaný obecný tvar, jde o text typu „jede v pracovní dny“ nebo „jede od 15.5. do 30.6. v (1) – (4); nejede 1.6.“. Možných tvarů tohoto řetězce je mnoho, na tento úkol byla tedy vyčleněna zvláštní třída *RouteDaysLineParser*. Ta na základě vstupu určí rozsah dat a dny v týdnu, kdy je spoj vypravován. Funguje tak, aby i při nestandardních situacích alespoň něco vrátila. Ve výsledku je schopna správně interpretovat různé vstupy, existují ale příklady, na které nestačí. Parsování vstupu nicméně není hlavní účel programu, ani není nijak zajímavé, třída byla tedy odladěna pouze do takového stavu, ve kterém drtivou většinu spojů načte správně.

Zajímavým problémem je pak sloučení spojů. Trasy všech spojů jedné linky budou jistě velice podobné, velká část z nich dokonce stejná. Určitě také bude existovat ke každému směru nějaká posloupnost zastávek, která zastřešuje všechny *trasy* daného směru. Předně je však třeba stanovit, jaké dva spoje mají ještě dost podobné *trasy* na to, aby byly sloučeny v rámci jednoho *směru*, a jaké už ne. Pro příklad mějme linku, jejíž spoje jezdí mezi zastávkami A a Z přes zastávky B, C a D. Některé spoje však zastávky C a D vynechávají a některé zase po cestě zastavují navíc v zastávce D. Očekávali bychom tedy dva *směry* - (A,B,C,D,E,Z) a (Z,E,D,C,B,A). Pokud bychom však slučovali nedostatečně, mohly by třeba (A,B,E,Z) a (A,B,C,D,Z) být pochopeny jako různé směry. Na druhou stranu se to nesmí se slučováním spojů přehnat. Teoreticky lze totiž všechny spoje sloučit do jediného směru (A,B,C,D,E,Z,E,D,C,B,A), což ale v tomto případě není žádoucí. Které spoje tedy sloučit a které už ne, není jasně definováno.

Slučování má na starosti statická třída *LineRoutesJoiner*. Ta (respektive její metoda *JoinRoutesToSingleLine*) přijme jako vstup množinu *směrů*[7], přičemž každý z nich zatím reprezentuje jeden spoj. Výstupem je též množina *směrů*, avšak tentokrát už jich je málo (typicky 2), každý obsahuje své *trasy*. Třída nejprve sjednotí směry obsahující stejné *trasy*, tedy ty, které obsahují identické posloupnosti zastávek a časové intervaly mezi nimi a také reprezentují spoje jedoucí ve stejné dny. To počet vstupních *směrů* značně redukuje. Potom se třída pokusí po dvojicích *směry* slučovat. To dělá sekvenčním průchodem zastávek obou směrů, přičemž sestavuje společný, který by dokázal reprezentovat vstupní směry jako své *trasy*. Zde je třeba určit limit, kdy výsledek operace použít a kdy ne, jelikož z předchozího odstavce plyne, že sloučit by šly i opačné, dokonce i naprosto disjunktní směry. Sloučení se tedy uznává za platné, pokud se použila alespoň polovina zastávek z kratšího ze vstupních *směrů*. Pokud se rozhodne ve prospěch slučování, je každý ze směrů vložen do *směru* výsledného jako *trasa*. Běh algoritmu znázorňuje následující pseudokód:

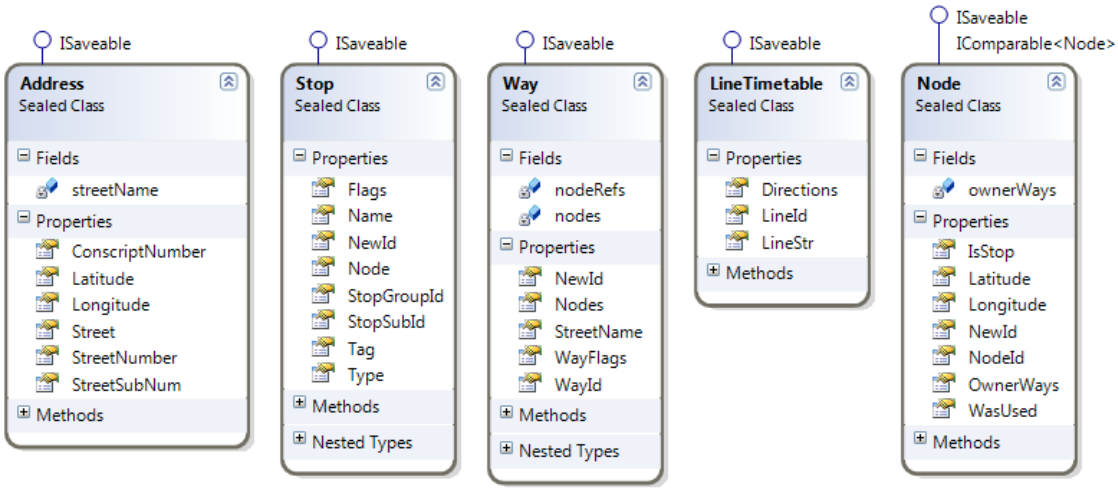
[7] Je důležité nenechat se zmást pojmy. Metoda sice přijímá *směry* (jako datový typ), nicméně ty v tomto případě vlastně reprezentují *spoje*. Účelem metody je právě redukce počtu různých *směrů*, které reprezentují stejné nebo podobné *trasy*.

```
function combinePaths(stopList1, stopList2):
1. // předpokládá se, že stopList1 není delší než stopList2
2. commonStopList := {}, j := 0
3. // i prochází první seznam, j druhý
4. for i = 1 to stopList1.count
5.    // první výskyt zastávky ve zbytku druhé trasy
6.    k := min { l | l > j && stopList2[l] = stopList1[i] }
7.    if k not defined:
8.       commonStopList <- stopList1[i]
9.       i := i + 1
10.   else:
11.      while j <= k:
12.         commonStopList <- stopList2[j]
13.         j := j + 1
14.      end while
15.   end if
16. end for
17.
18. // doplnění zbylých zastávek druhé trasy
19. while j <= stopList2.count:
20.   commonStopList <- stopList2[j]
21.   j := j + 1
22. end while
23.
24. if commonStopList.count < stopList2.count + stopList1.count / 2:
25.   use commonStopList
26. else
27.   use stopList1 and stopList2 // sjednocení tras je příliš dlouhé
```

6.2.6 Datové třídy

Datové třídy jsou třídy uchovávající informace o objektech. *Parsery* jejich instance během načítání vstupu průběžně vytváří a přidávají do příslušných *seznamů* (viz 6.2.7). *Datové třídy* mají společnou schopnost uložit se do souboru. Implementují rozhraní *ISaveable*, které jim předepisuje metodu *SaveTo*, kde obdrží jako parametr odkaz na *BinaryStream*, do kterého mají uložit svůj obsah. Uložení *seznamu* objektů pak lze lehce zrealizovat zavoláním této metody na každý objekt ze *seznamu*.

Na obrázku 6-6 jsou důležitější *datové třídy* včetně svých vlastností.



Obr. 6-6 – Datové třídy

Třída *Address* reprezentuje jednu adresu tvořenou ulicí, popisným a orientačním číslem a souřadnicemi na mapě. Zastávku zastupuje instance třídy *Stop*, u ní určujeme ID, uzel v mapě, kterým je reprezentována, ID skupiny, do které patří, a typ (Tram/Bus/Metro). Cestu reprezentuje třída *Way*, ta má ID původní – přiřazené z OSM, a nové – přidělené programem, a samozřejmě seznam uzlů. Třída *Node* slouží k evidenci uzlu v mapě, u něj jsou nejdůležitější souřadnice, evidovány jsou i cesty, na kterých leží a podobně jako u cest původní a nový identifikátor.

Jízdní řády

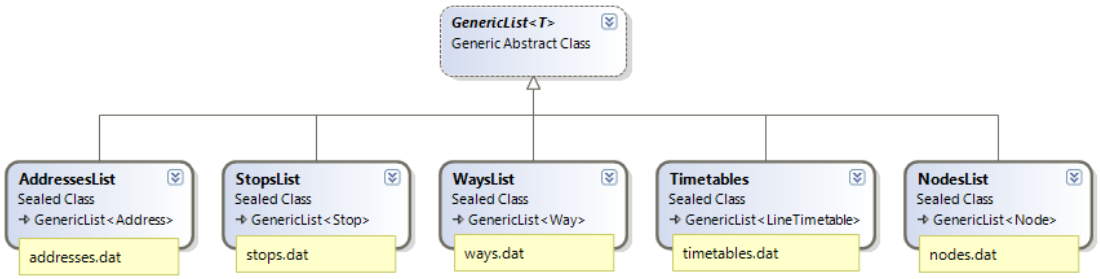
Jízdní řády tvoří seznam jízdních řádů *linek*, které reprezentuje třída *LineTimetable*. Ta obsahuje identifikaci linky a seznam *směrů*, které zastupuje třída *LineDirectionTimetable* skládající se ze seznamu zastávek a všech *tras* daného směru. *Trase* odpovídá instance třídy *LinePath*. Ta obsahuje seznam časů odjezdů z počáteční zastávky a dobu dojezdu do každé zastávky ve *směru*. Konečně je pak třeba reprezentovat dny, ve kterých je spoj vypraven. Celý tento systém je odvozen z toho, jak jsou dané pojmy definovány v kapitole 2.1.2.

6.2.7 Seznamy

Seznamy jsou též důležitou částí projektu. Jejich hlavním úkolem je udržovat seznam nějakých objektů a poskytovat funkce pro jejich organizaci a uložení do souboru. *Seznamy* jsou třídy odvozené od abstraktní třídy *GenericList<T>*, kde

typový parametr *T* udává typ objektu, se kterým seznam pracuje, přičemž je doplněna podmínka *where T : ISaveable*, což omezuje typy na *datové třídy*.

Zatímco *parsery* (respektive jejich instance) odpovídaly vstupním souborům, každá instance nějakého *seznamu* odpovídá jednomu výstupnímu souboru[8]. Obrázek 6-7 zobrazuje třídy z této kategorie, lze si všimnout i konkrétního odvození od *GenericList* a tříd, které jednotlivé *seznamy* používají jako své prvky. Je vidět, že přestože třída *GenericList* je generická, její konkrétní potomci nikoliv. V poznámkách u každé třídy jsou uvedeny soubory, do kterých ukládají svůj obsah.



Obr. 6-7 – Seznamy (skupina tříd)

Třída GenericList<T>

Abstraktní třída *GenericList* obsahuje seznam *List<T>* pro uchovávání objektů a definuje několik virtuálních metod souvisejících s ukládáním souborů, z nichž nejdůležitější je metoda *Save*. Ta ve výchozí implementaci v této třídě jednoduše uloží hlavičku souboru s počtem záznamů a zavolá *SaveTo* na každý prvek seznamu, který drží. Toto chování lze však změnit vlastní implementací této metody nebo vlastní implementací metody *ItemsToSave*, jejíž účel je vrátit množinu objektů k uložení; ta standardně vrací celý seznam.

Třída AddressesList

Třída *AddressesList* drží seznam adres (tedy objektů typu *Address*). Na rozdíl od svého předka pouze v implementaci *ItemsToSave* určuje, že adresy, které mají

[8] Obráceně to neplatí, soubor stopnames.dat je totiž dodatečně generován třídou *StopsList*, nemá tedy přiřazen vlastní seznam.

popisné i orientační číslo rovno nule, nebudou uloženy, protože nenesou žádnou zajímavou informaci.

Třída StopsList

Třída *StopsList* uchovávající zastávky jako instance třídy *Stop* navíc zajišťuje evidenci skupin zastávek a jejich jmen. Uchovává si tedy asociativní pole (implementováno pomocí *Dictionary*) indexované jménem *zastávky* (resp. s*kupiny zastávek*), ze kterého lze jednoduše získat seznam zastávek patřících do této skupiny (tedy zastávek stejného jména). Při ukládání pak také navíc ještě vytváří soubor stopnames.dat, do kterého seznam skupin uloží. Každá zastávka (*Stop*) pak obsahuje pouze identifikátor skupiny, do které patří, a z ní si zjistí své jméno.

Třída WaysList

Třída *WaysList* spravuje seznam cest na mapě. Před uložením do souboru je třeba zavolat metodu *GenerateNewIds*, která přidělí cestám nové identifikátory a zároveň určí cesty, které nemá smysl ukládat, což jsou ty, které se skládají z méně než dvou uzlů. To ale tentokrát není vše. V kapitole o *parserech* (6.2.5) jsem několikrát zmínil potřebu vytvoření nových či případné propojování uzlů (např. při vytváření vstupu do metra je třeba jej propojit se stanicí metra). Tyto akce zajišťuje právě *WaysList* následujícími metodami. Metoda *CreateNodeInTramWay* vytvoří uzel na tramvajové trati a propojí ho s nějakou blízkou cestou. Primárně se snaží o propojení s chodníkem nebo jinou komunikací dobře použitelnou pro chodce (ne tedy např. silnicí I. třídy). Podobně funguje i metoda *CreateNodeInStreet*, která poskytuje podobnou službu autobusům a vytváří uzel na vhodné silnici. Opět se snaží o propojení s blízkým chodníkem.

Třída NodesList

Třída *NodesList* reprezentující *seznam uzlů* ničím výrazně nepřekvapí. Podobně jako *WaysList* umí navíc generovat svým prvkům nové identifikátory a vyřadit při tom nepotřebné uzly, což jsou ty, které neleží na žádné cestě. Z toho důvodu je třeba, aby se nejdřív zavolala metoda *GenerateNewIds* na *seznamu cest*, aby se vyřadily nepotřebné cesty a následně bylo možno odstranit více nepotřebných uzlů.

6.3 Knihovna

Snahou oddělit výpočetní část od uživatelského rozhraní i zpracování dat vznikla samostatná knihovna, která má na starosti vyhledávání spojení. Má o dost zjednodušenou práci tím, že díky *OSMReaderu* může počítat s jednotným formátem vstupu (ten je popsán v příloze B). Obsahuje 3 důležité části:

- *Datové třídy* (podobně jako v *OSMReaderu*)
- *Vyhledávání spojení*
- *Rozhraní knihovny*

Knihovna je implementována jako statická, kompilací tedy vznikne soubor *LIB*, který se musí přidat k parametrům linkeru v každém projektu, ve kterém má být knihovna použita. Projekt je psán objektově, i když nebylo stoprocentně využito dědičnosti, zejména z optimalizačních důvodů.

6.3.1 Datové třídy

Datové třídy zastávají ve vyhledávací knihovně podobný účel jako v *OSMReaderu*. Liší se hlavně způsobem načtení dat. Zejména z optimalizačních důvodů třídy nemají společného předka a nenačítají se ze vstupních souborů samy, nicméně načtení dat provádí samo jádro knihovny, což je statická třída *PTNLibrary*, zde konkrétně funkce *LoadData*. Většina datových tříd obsahuje záznam „*friend class PTNLibrary*“, který povoluje jmenované třídě přistupovat k jejím privátním položkám.

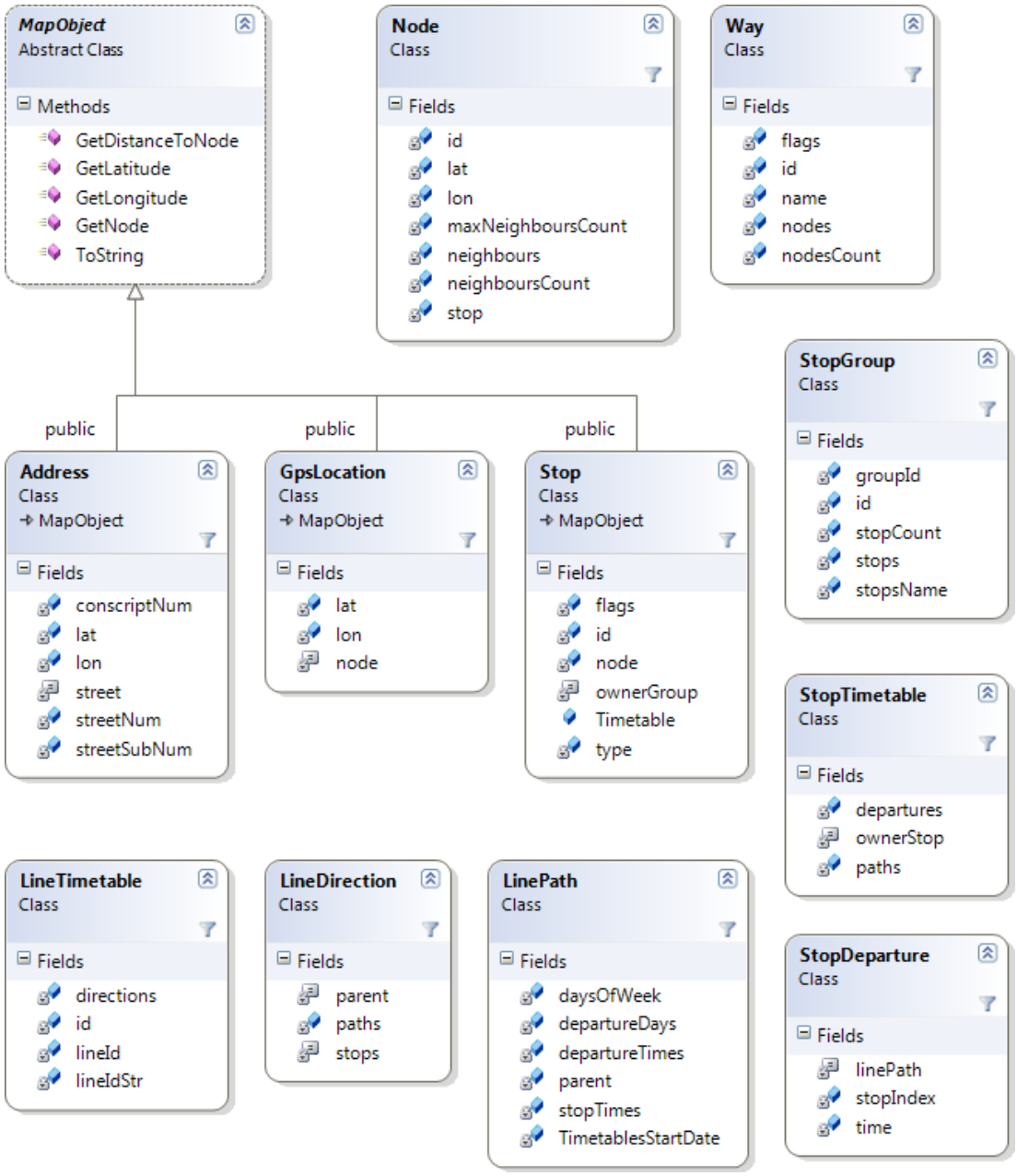
Třídy *Address, Node, Stop, Way, LineTimetable, LineDirection* a *LinePath* víceméně odpovídají stejnojmenným (nebo podobně se jmenujícím) C# třídám z projektu *OSMReader*. Třída *StopGroup* reprezentuje skupinu zastávek – obsahuje jejich společný identifikátor a jméno. *GpsLocation* reprezentuje souřadnici na mapě; *MapObject* je společný předek pro všechny třídy, které zastupují nějaké místo na mapě (tedy zastávku (*Stop*), GPS souřadnici (*GpsLocation*) a adresu (*Address*)), což se opírá o návrh, ve kterém bylo stanoveno, že právě tyto tři třídy objektů mohou

být používány jako počáteční a cílové body pro vyhledávání spojení. Třídy odvozené od *MapObject* musí implementovat metodu *GetNode*, kterou vrací uzel v mapě, který je nejlépe reprezentuje.

Další *datové třídy* souvisí s jízdními řády. Navíc je zde *StopTimetable* reprezentující jízdní řád zastávky. Ke každé zastávce existuje jedna instance, ta obsahuje seznam všech odjezdů popsaných prvky třídy *StopDeparture*. Ta obsahuje ukazatel na *trasu*, index určující kolikátou v pořadí tato zastávka na trase je a čas odjezdu. Ke specifikování času odjezdu slouží třída *DepartureTime*, což je vlastně jen takové rozšíření integeru. Jejím základem je hodnota určující čas ve formátu počtu minut od půlnoci. Zajímavé na této třídě je, že může nabývat i větších hodnot, než by člověk očekával, totiž hodnoty 1440, což je počet minut v jednom dni. Důvodem k tomu jsou jízdní řády okolo půlnoci. Zjednodušeně řečeno, pokud nějaký spoj odjíždí ve „24:15“ a jezdí každý pátek, je to stejné jako by jel v „0:15“ každou sobotu. Díky tomuto systému spoje vyrážející před půlnocí nemusí být roztrženy na dva, kde by jedna část jela v pátek a druhá v sobotu.

Zbývá vysvětlit význam dvou speciálnějších tříd. První z nich je třída *UnixDateTime*. Její účel je víc než jednoduchý, uchovává datum a/nebo čas ve formátu počtu sekund od 1.1.1970. Ve skutečnosti šlo použít např. datový typ time_t, nicméně ten je typicky 64bitový a není objektový. Hledat vlastní knihovnu jen pro takto jednoduchý účel mi zas přišlo zbytečné. Druhou třídou se speciálnějším významem je třída *array<T>*. Ta, jak se časem ukázalo, nebyla pojmenována úplně nejchytřeji, jelikož existuje stejnojmenná STL třída, která má ale podobnou funkcionalitu. Používá se všude tam, kde se předem zná počet položek seznamu.

Zmíněných tříd je možná trochu více, než je člověk v jednu chvíli schopen vstřebat, proto obrázek 6-8 tyto třídy včetně jejich nejdůležitějších vlastností rekapituluje. Jde typicky o třídy jednoduché, obsahující zejména data a přístupové funkce k nim, často jsou immutable.



Obr. 6-8 – Datové třídy knihovny PTN (přehled)

Na obrázku jsou zobrazeny pouze datové položky, nikoliv metody, jelikož takový obrázek už by byl zbytečně velký a nepřehledný, navíc většina metod jsou pouze přístupové funkce k těmto datovým položkám. Jsou však i výjimky, které popisují následující odstavce. Většinou lze význam jednotlivých datových položek poznat z názvu, případně bývají popsány přímo v kódu.

Implementace MapObject::GetNode()

Bylo stanoveno, že třídy *Stop, Address* a *GpsLocation* musejí umět vrátit uzel, který je v mapě reprezentuje. Situace je jednoduchá u zastávek, každá totiž má v mapě svůj uzel, který si drží a ten také vrátí. V případě adres by však bylo velice neefektivní vytvářet uzel pro každou existující adresu, protože nad uzly mapy se provádí vyhledávání spojení a přidání adres by přineslo spoustu uzlů a cest, z nichž uživatel využije za celou dobu jen minimum. Proto adresy v současném stavu vrací nejbližší uzel na své ulici. Ještě výraznější je tento případ u GPS souřadnic, kde vůbec nelze očekávat, jaké si uživatel vymyslí. Vrací se uzel nejbližší k daným souřadnicím. Třídy odvozené od *MapObject* by také měly implementovat metodu *GetDistanceToNode*, kterou vrátí vzdálenost k uzlu, který vrací *GetNode*.

Zjišťování, zda je spoj vypraven v daný den

Třída *LinePath* odpovídající *trase* sdružuje spoje, které jsou mimo jiné vypravovány ve stejné dny. Proto také musí umět rozhodnout, zda jsou její spoje v konkrétní den vypravovány. Dny, ve které spoje jedou, jsou popsány položkami *daysOfWeek* a *departureDays*. První zmíněná položka je kombinace flagů *MONDAY – SUNDAY* typu *DaysOfWeek*. *DepartureDays* jsou 64bitový integer, kde *i*-tý bit indikuje, zda je spoj vypraven *i*-tý den od pořízení jízdních řádů, což je statická hodnota *LinePath::TimetablesStartDate*. Následující funkce vrací logickou hodnotu indikující, zda je spoj v daný den *date* vypraven: (>> reprezentuje bitový posun)

```
function IsSetOutOn(date) =
  (departureDays >> min(63, date – TimetablesStartDate)) > 0 &&
  (daysOfWeek AND date.dayOfWeek) > 0;
```

Získávání odjezdů ze zastávky

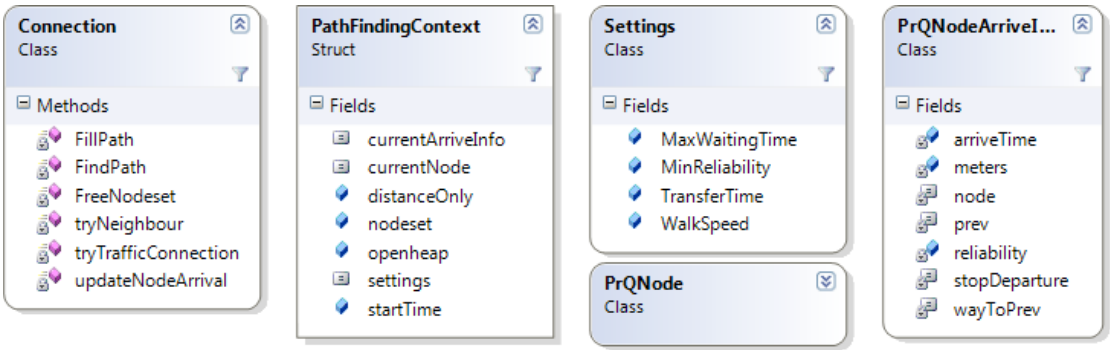
Při vyhledávání spojení je potřeba rychle vyhledat nejbližší odjezdy z dané zastávky vzhledem k zadanému času. K tomu účelu byly zřízeny třídy *StopTimetable* a *StopDeparture*. První jmenovaná obsahuje pro každou zastávku setříděný seznam odjezdů, a to podle skutečného času (např. „0:10“ < „24:15“ < „0:20“). K přístupu do tohoto seznamu slouží dvě metody. První, *GetDepartureIndex(time)* vrací první spoj, který ze zastávky odjíždí po daném čase, přičemž ohled se bere i na aktuální datum,

takže se počítají pouze spoje vypravené v daný den. Funkce najde první spoj po daném čase půlením intervalu a následně použije druhou, též veřejnou funkci *GetNextDepartureIndex(start, time)*, která vrátí první vypravovaný spoj od pozice start. Toho docílí sekvenčním průchodem od indexu *start* a přeskakováním spojů, které v daný den vypraveny nejsou. S pomocí těchto dvou funkcí pak lze jednoduše získat seznam odjezdů v daném časovém intervalu.

6.3.2 Vyhledávání spojení

Vyhledávání spojení je hlavní náplň aplikace – alespoň co do tématu. Během návrhu a vývoje se však objevilo tolik problémů spojených zejména se správnou interpretací dat, že rozsahově je vyhledávací algoritmus spíše menší částí veškerého kódu. Je to také způsobeno objektovým návrhem, který mnoho funkcí, které vyhledávací algoritmus používá, „roztřídil“ do objektů, k nimž se vztahují (např. třída *Node* disponuje funkcí, která jednoduše zjistí sousedy, třída *StopDeparture* zase umí jednoduše vrátit odjezdy od daného času).

Většina algoritmu je implementována v rámci třídy *Connection*, která na základě dvou bodů zadaných třídou *MapObject*, počátečního času a nastavení umí spojení vyhledat a výsledek uchovat. Ještě před samotnou implementací ukáže obrázek 6-9 třídy použité při vyhledávání.



Obr. 6-9 – Třídy pro vyhledávání (přehled)

U třídy *Connection* lze vidět funkce, které se podílí na samotném vyhledávání; struktura *PathFindingContext* sdružuje kontext, např. právě uzavíraný vrchol (*currentNode*) a informace o příchodu do něj (*currentArriveInfo*), množinu

otevřených a uzavřených vrcholů s přímým přístupem podle *ID* uzlu (*nodeset*), haldu spravující ukazatele na otevřené vrcholy (*openheap*), či nastavení jako ukazatel na instanci třídy *Settings*. Nastavení zatím program podporuje málo, jde zvolit maximální čas čekání na zastávce v minutách, minimální požadovanou spolehlivost (interval 0 – 1), minimální čas na přestup a zejména pak rychlost chůze při pěších přesunech. Konkrétní hodnoty se získávají od uživatele. Třída *PrQNode* obsahuje informace o jednom navštíveném vrcholu, konkrétně seznam dočasných hodnot (příchodů) do něj (instance třídy *PrQNodeArriveInfo*), kterých může být více než jen jeden kvůli různým spolehlivostem (viz kapitola 4.1). Pro každý příchod je třeba vědět hlavně čas a ukazatel na *PrQNodeArriveInfo* předchozího uzlu, aby pak bylo možno trasu zrekonstruovat.

Vyhledávací algoritmus - základ

Spojení vyhledává metoda *FindPath*, která obsahuje jádro algoritmu. Naznačuje ji následující pseudokód.

```
function FindPath(start, end, startTime, settings) :
1.  var cxt : PathFindingContext (startTime, settings,
2.       nodeset := { start } )
3.  while cxt.nodeset is not empty :
4.      cxt.currentNode := cxt.openheap.ExtractMin()
5.      if cxt.currentNode = end then return // cesta nalezena
6.      for each neighbourPair in cxt.currentNode.neighbourPairs:
7.          tryNeighbour(neighbourPair.LeftNeighbour)
8.          tryNeighbour(neighbourPair.RightNeighbour)
9.      end for
10.     var stop := cxt.currentNode.GetRepresentedStop()
11.     if stop != NULL then
12.         tryTrafficConnection(stop.GetTimetable())
13. end while
```

V uvedeném kódu se na řádce 6 používají *páry sousedů* daného uzlu. Každý uzel má pro každou cestu, na které leží, jeden *pár sousedů*, což jsou dva uzly, které na dané cestě přímo sousedí s daným uzlem. Funkce *Node::GetRepresentedStop* použitá na řádku 10 vrací pro uzly, které reprezentují na mapě nějakou zastávku, ukazatel na ni, jinak *NULL*. Konečně funkce *Stop::GetTimetable* vrací zastávkový jízdní řád, tedy ukazatel na *StopTimetable*. Zbývají dvě členské funkce třídy *Connection*. První, *tryNeighbour* pouze spočítá čas potřebný na přesun do daného

bodu a zavolá metodu *updateNodeArrival(targetNode, arriveTime)*, popsanou níže. Funkce *tryTrafficConnection* už je o něco složitější, následuje její nástin:

```
function tryTrafficConnection(stopTimetable) :
1. var startTime := cxt.currentArriveInfo.getArriveTime()
2. var departure := stopTimetable.GetDeparture(startTime)
3. while departure != NULL :
4.    var departureTime := departure.GetDepartureTime() // odjezd
5.    var arriveTime := departureTime + departure.GetTimeToNextStop()
6.    if departureTime – startTime > settings.MaxWaitingTime then break
7.    updateNodeArrival(departure.GetNextStop(), arriveTime)
8.    departure := stopTimetable.GetNextDeparture(departure, startTime)
9. end while
```

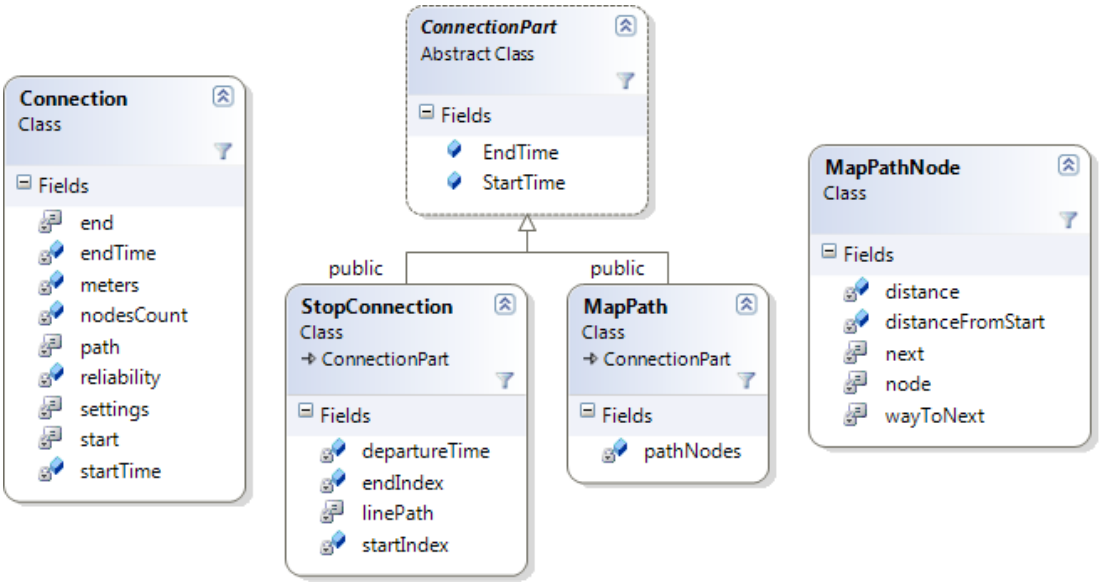
Na začátku se zjistí čas, ve který algoritmus na danou zastávku dorazí a následně se prochází jednotlivé odjezdy z této zastávky. Za zmínku stojí zdůraznit, že mezi otevřené vrcholy se přidává pouze následující zastávka na trase. Zbývá objasnit metodu *updateNodeArrival*:

```
function updateNodeArrival(targetNode, arriveTime) :
1. var prqNode := cxt.nodeset[targetNode.GetId()]
2. if prqNode := NULL then
3.    // první návštěva uzlu
4.    prqNode := new PrQNode(from := cxt.currentNode, arriveTime)
5. else :
6.    // vyhodnotí, zda je tento čas lepší než zatím nejlepší dočasný
7.    prqNode.addArriveInfo(from := cxt.currentNode, arriveTime)
8. end if
```

Ve skutečnosti se k otevřeným vrcholům uchovává více informací, viz třída *PrQNodeArriveInfo.*

Uložení nalezené cesty

Nalezené spojení ukládá sama třída *Connection* do své vnitřní proměnné *vector<const ConnectionPart*> path* jako cestu uloženou po souvislých částech. Obrázek 6-10 ukazuje důležité třídy pro uložení informací o spojení, včetně jejich datových položek.



Obr. 6-10 – Třídy pro reprezentaci nalezeného spojení (přehled)

Uložení spojení – třída ConnectionPart a její potomci

Spojení je uloženo jako posloupnost jeho částí, které zastupuje třída *ConnectionPart*. Je to abstraktní třída, jejíž dva potomci *MapPath* a *StopConnection* reprezentují spojení pěší, respektive spojení pomocí MHD, přičemž platí:

1. Pěší spojení reprezentované třídou *MapPath* je vždy nejdelší možné. Důsledkem je, že v seznamu nikdy nemohou být dvě instance *MapPath* za sebou (jinak by ani jedna z nich nereprezentovala nejdelší možné spojení).
2. Položka typu *StopConnection* reprezentuje jedno souvislé spojení (nejdelší možné bez přestupu).
3. V seznamu částí spojení obecně koncový bod *i*-té části je počátečním bodem (*i*+1). části.

Abstraktní třída *ConnectionPart* poskytuje svým dvěma potomkům položky pro čas začátku a čas konce daného úseku, dále pak metody pro zjištění počátečního a koncového uzlu. Třída *MapPath* reprezentuje pěší trasu seznamem uzlů s dodatečnými informacemi (pro každý uzel se uchovává vzdálenost od prvního uzlu, ukazatel na uzel následující a na cestu k němu; viz třída *MapPathNode*). Třída *StopConnection* obsahuje pouze ukazatel na konkrétní *trasu* spoje (*LinePath*) a indexy nástupní a výstupní zastávky, společně s časem odjezdu z nástupní zastávky.

Hlavní roli zde hraje metoda *Connection::FillPath*, která nejprve nechá vyhledat spojení a pak, protože si každý vrchol pamatuje, odkud a jak jsme se do něj dostali, cestu od konce zrekonstruuje.

6.3.3 Rozhraní knihovny

Programátorské rozhraní knihovny je poměrně jednoduché. Tvoří ho zejména statická třída *PTNLibrary*. Většinu objektů použitých v projektu nelze bez její pomoci ani instanciovat, jelikož mají privátní konstruktory. Jediný, kdo pak smí tyto objekty vytvářet, je právě *PTNLibrary*, která je třídami referencována jako *friend class*, což znamená, že může přistupovat k jejich privátním položkám. Jediný způsob, jak se k objektům dostat, je tedy použít tuto statickou třídu, která buď vrací konstantní ukazatele na objekty, které vlastní (např. adresy, zastávky) nebo je vytváří až při jejich potřebě (např. spojení). V druhém případě je pak třeba knihovnu požádat i o smazání tohoto objektu.

Přehled globálních funkcí knihovny

Důležitými funkcemi knihovny jsou *LoadData* a *Unload*, první načte potřebná data ze souborů, druhá je smaže z paměti a tuto paměť uvolní. K přístupu k nastavení slouží funkce *GetSettings*, která výjimečně nevrací konstantní ukazatel, aby aplikace mohla nastavení měnit. K reprezentaci GPS souřadnic slouží třída *GpsLocation*. Tu lze vytvořit pouze pomocí přetížené metody *CreateGpsLocation*. Konkrétní adresy lze získávat pomocí přetížené metody *FindAddresses*, která umí vyhledávat podle jména i popisného a orientačního čísla. Na hledání zastávek podle jména slouží *FindStop*. Zobecnění nad posledními dvěma jmenovanými přináší metoda *FindObjects*, která vrací seznam složený jak ze zastávek, tak adres odpovídajících zadání. Jistě stěžejní funkcí je pak metoda *FindConnections*, která vrátí požadovaný počet spojení mezi zadanými body od zadaného času. Pro objekty *GpsLocation* a *Connection*, které třída neschraňuje (protože na rozdíl od adres a zastávek nejsou předem známé), jsou pak zřízeny metody *FreeConnections* a *FreeGpsLocation*.

6.4 Uživatelské rozhraní

Účelem uživatelského rozhraní je rozumným způsobem využít implementované knihovny k vytvoření prakticky použitelné aplikace. Výsledkem je projekt *PTN GUI*, což je aplikace napsaná za použití rozhraní Qt *[11]*, které poskytuje jak prezentační prvky (okna, tlačítka, …), tak i některé další užitečné funkce.

Projekt není příliš složitý, jelikož veškerá logika související s vyhledáváním je umístěna ve vyhledávací knihovně. Rozhraní je tvořeno čtyřmi okny. Nejdůležitějším, avšak ne příliš programátorsky zajímavým, je *hlavní okno*, reprezentované třídou *PTNGUI*. To je poměrně bohatě osázeno ovládacími prvky, jejichž obsluze se také věnuje většina kódu. Třída používá knihovní funkci *FindConnections*, kterou hledá spojení, což je kvůli časové náročnosti této operace prováděno v paralelním vlákně. K určení bodů využívá služeb dialogů pro zadání jména a mapového okna. *Dialog pro zadání jména objektu* obsluhuje třída *SelectWin*. Ta sama také nemá příliš práce, protože funkce na vyhledávání objektů podle jména, případně podle části jména, implementuje knihovna jako *FindObjects* a *FindObjectNames*. Vlastní okno je i pro *nastavení*, to je řízeno třídou *SettingsWin*. Ta mění nastavení vyhledávání v závislosti na vstupu uživatele, čehož docílí funkcí knihovny *GetSettings*. Jedinou implementačně zajímavou částí je tedy zbývající *mapové rozhraní* a dialog, který jej představuje. Ten je ovládán třídou *MapWin*.

6.4.1 Mapové rozhraní

Součástí zadání práce je i vytvoření mapového rozhraní, ve kterém se nalezené trasy budou zobrazovat. Pro něj je třeba vyřešit jednu základní otázku – odkud vzít mapy, na které se bude spojení kreslit. Samozřejmě nejlepší volbou je použití OpenStreetMaps, protože v nich se spojení hledá a zabrání se tak nekonzistenci použitého mapového podkladu a nalezené cesty (jinými slovy, trasa nepovede mimo existující cesty). Ovšem během *parsování* z kompletních mapových údajů zůstane pouze seznam uzlů a cesty je spojující, což na rekonstrukci mapy nestačí. Navíc, i kdybychom se rozhodli k programu přiložit celý OSM soubor, vykreslit podle něj reálnou mapu není jednoduché. Z toho plyne jasný požadavek, že je potřeba najít

prostředek, který mapu vyrenderuje a aplikace by pak výsledek použila. To umí server *tile.openstreetmap.org [17]*. Podle vstupních údajů vrátí 256 × 256 pixelů velkou část mapy. Stahování větších oblastí pomocí dávkových dotazů na server je sice výslovně zakázáno, nicméně distribuovat program s možná až stovkami MB map je v dnešní době internetu stejně spíše zbytečné. Je tedy jasné, že mapy si bude program z internetu stahovat sám. Kvůli šetření provozu po síti a výkonu serveru je žádoucí jednou stažené části mapy už, alespoň na nějaký čas, nestahovat znovu, zvláště počítá-li se s tím, že uživatel se bude často pohybovat ve stejných oblastech.

Fungování serveru tiles.openstreetmap.org

Dotaz na server je vždy ve tvaru: *tiles.openstreetmap.org/zoom/x/y.png*, kde *zoom* je přiblížení (0 nejmenší, 18 největší) a *x* a *y* udávají souřadnice dílku (*tile)* mapy. *Dílek* je vždy čtverec o rozměrech 256 × 256 pixelů a v rámci jednoho přiblížení jsou všechny dílky disjunktní (sousední na sebe navazují). Vychází se při tom z toho, že dotaz na /0/0/0.png vrací mapu celého světa ve čtverci 256 × 256 pixelů. *Zoom* 1 pak rozděluje tento čtverec na 4 části, kde /1/0/0.png je levá horní a /1/1/1.png pravá spodní čtvrtina mapy. Takto se pak pokračuje dále, každé další přiblížení násobí počet částí na obou osách dvakrát, celkově tedy čtyřikrát. Obecně se *dílek* na souřadnicích (x, y) při přiblížení o 1 úroveň rozpadne na (x*2, y*2), (x*2+1, y*2), (x*2, y*2+1), (x*2+1, y*2+1). Platí tedy, že při přiblížení *z* je svět rozdělen na 2^{2z} *dílků* a pravý spodní má souřadnice (2^z-1, 2^z-1).

Pro účely mapového rozhraní potřebujeme zjistit při daném přiblížení souřadnice dílku, ve kterém je obsažen bod daný souřadnicemi GPS. Ty lze podle *[15]* spočítat takto:

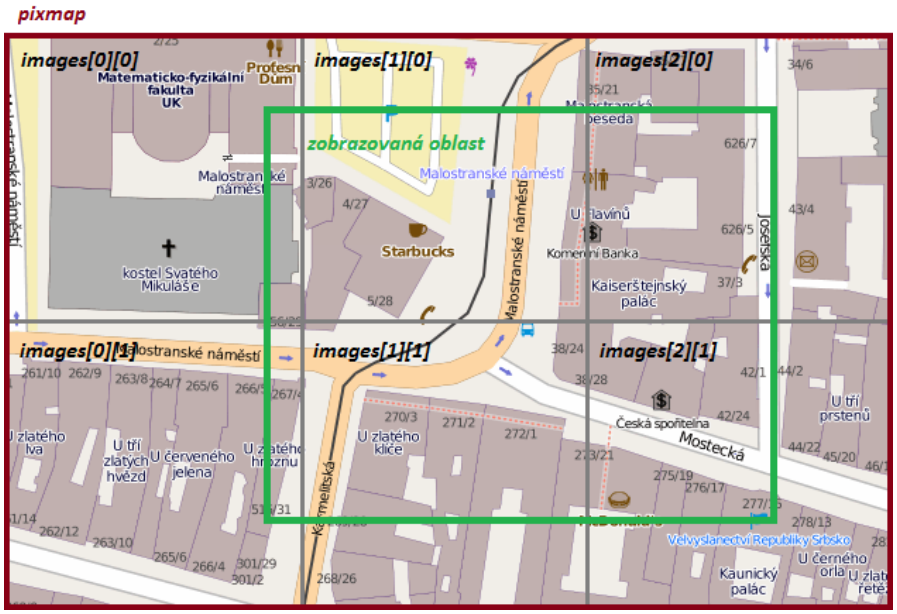
```
n = 2 ^ zoom
xtile = ((lon_deg + 180) / 360) * n
ytile = (1 - (log(tan(lat_rad) + sec(lat_rad)) / π)) / 2 * n
```

Stahování map

Po získání správných souřadnic dílku mapy a sestavení výsledné adresy je třeba položit tento dotaz serveru a stáhnout výsledný soubor (případně jej jen načíst z disku, byl-li už stažen dříve). K tomu slouží třída *OsmTilesDownloader*, která je založená na ukázce z *[16]*. Přijímá dotazy na části identifikované třídou *OsmTileId*, což není nic více, než ona trojice (*zoom, x, y*). Jakmile má danou část mapy k dispozici (což může být hned, je-li již stažena), informuje o tom událostí *tileReady*. Stahování probíhá asynchronně pomocí prostředků Qt. Tento systém dobře zastřešuje získávání map, stačí jen vypočítat, jaké dílky jsou potřeba, a třídu o ně požádat, není třeba se starat o to, kde je získá.

Vykreslování map

Překreslení mapy je poměrně častá událost. Nastává zejména při změně velikosti okna, posunu mapy nebo změně měřítka. Třída si v proměnné *tilesRect* uchovává ohraničení zobrazené části; hodnoty jsou z intervalu 0 – 1, kde (0, 0) je levý horní roh a (1, 1) pravý spodní. Zároveň položka *zoom* uchovává aktuální přiblížení, *images* je dvourozměrné pole načtených aktuálně použitých dílků mapy a v proměnné *pixmap* je uložena vykreslená část mapy složená z obrázků v *images*. *Pixmap* obsahuje o něco větší část mapy, než se skutečně vykreslí v okně, protože okraje jsou „zaokrouhleny“ tak, aby obrázek obsahoval vždy celé dílky mapy. Odstavec rekapituluje obrázek 6-11.

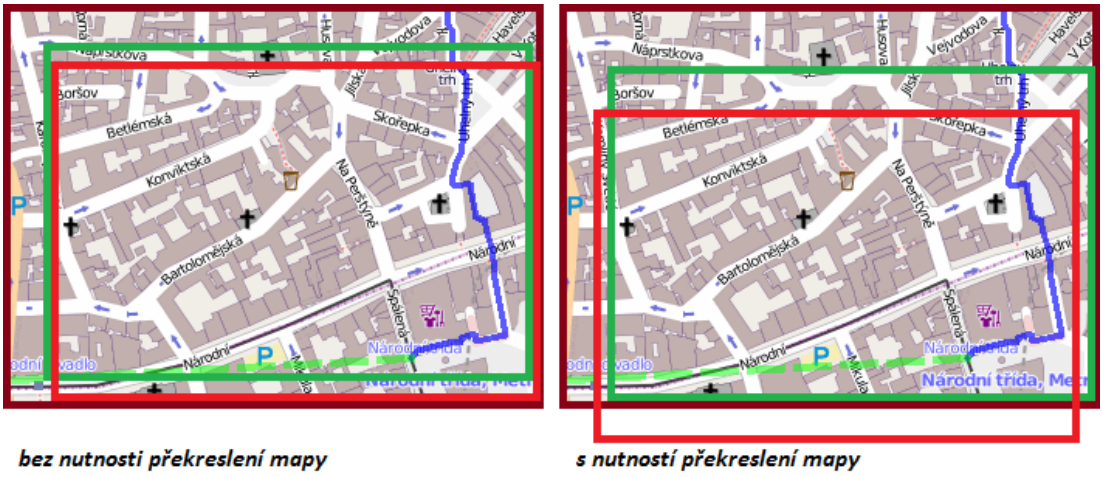


Obr. 6-11 – Uložení mapy ve třídě MapWin

Při kompletním překreslení, které provádí metoda *RefreshMap(newRect, newZoom)*, se určí, které dílky mapy budou potřeba, a dotaz na ně je směrován na třídu *OsmTilesDownloader*. Potom se při získání každého dílku načte stažený soubor na příslušné místo do *images* a metoda *CreatePixmap* překreslí obrázek v *pixmap*, čímž se dílek do mapy začlení. Zároveň se zobrazovaná část obsahu proměnné *pixmap* vykreslí přímo do okna. Vykreslení do okna se provádí zavoláním metody *repaint*, kterou implementuje Qt, a ta následně zavolá metodu *paintEvent*.

Kompletní překreslení je nicméně poměrně náročnou operací a rozhodně není potřeba při posunu mapy o každý pixel, provádí se tedy vždy pouze potřebná část celého procesu. Předně metoda *RefreshMap* ví, které dílky mapy byly použity při předchozím vykreslení a pokud některé z nich může použít znovu, nežádá o ně, čímž se ušetří opětovné načítání souboru z disku. Pokud se zjistí, že množina zobrazených dílků mapy je stejná, metoda *RefreshMap* se ani vůbec nevolá a stejně tak není potřeba ani metoda *CreatePixmap*. Pouze se lehce upraví oblast, která se z *pixmap* vykreslí do okna. O něco názorněji by to mělo být vidět z obrázku 6-12, který zobrazuje příklad situace, kdy není třeba mapu v paměti překreslovat, i situace, kde to potřeba je (musí se zavolat i funkce *RefreshMap* a *CreatePixmap*). Hnědý

obdélník je obsah *pixmap*, zelený obdélník ukazuje předchozí zobrazenou oblast a červený novou zobrazovanou oblast mapy.



Obr. 6-12 – Změna zobrazované oblasti mapy

Vykreslování tras

Při zobrazování trasy potřebujeme dané spojení umět do mapy nakreslit. Třídě *MapWin* je pak třeba předat ukazatel na spojení zastoupené třídou *Connection*. Z něho lze získat kompletní informace o spojení včetně seznamu všech bodů a jejich souřadnic na trase. U zastávkového spojení však neznáme přesnou trasu (zastávky ano, ale ne přesné cesty), proto je vykresleno jako přímé spojení nástupní a výstupní zastávky. Vykreslení trasy do mapy má na starost metoda *CreatePixmap*, takže trasa se nevykresluje znovu, když je to zbytečné.

7. Možnosti do budoucna

Ještě před samotným závěrem práce tato kapitola zhodnotí, co programu nakonec chybí nebo nefunguje, jak bylo zamýšleno, a nastíní možná řešení těchto problémů společně s případnými dalšími vylepšeními, která by šla do projektu zakomponovat.

7.1 Současné problémy programu

Jedním z cílů práce bylo vytvořit program, který by mohl být použitelný i v praxi. To se nakonec z většiny podařilo, díky použití skutečné mapy a skutečných jízdních řádů. Nicméně ve skutečnosti by bylo třeba dořešit některé z následujících problémů, aby byl program důvěryhodnější.

Neaktuální jízdní řády

Velkou nevýhodou zejména oproti webovému IDOSu je nutnost aktualizovat data. Stahování jízdních řádů je časově náročná operace, není vhodné ji opakovat příliš často kvůli zbytečnému vytěžování serveru. Program načítá všechny spoje, i ty, které v době pořízení ještě nejezdí, je tedy připraven na pár týdnů do budoucna. Jediným problémem můžou být různé náhlé změny, které se ale naštěstí dějí spíše výjimečně.

Mapové zdroje

OpenStreet mapy používané jako zdroj map jsou vytvářeny obyčejnými lidmi, přesto jsou však poměrně spolehlivé a jejich úroveň se dále zvyšuje. Problém tkví spíše v práci s daty. Obyčejné vyhledávání nejkratší cesty totiž nereflektuje požadavky průměrného člověka, například aby nemusel chodit po frekventovanějších silnicích a aby obecně vyhledávání dávalo přednost chodníkům. V tomto ohledu by se PTN ještě mohl zlepšit, zatím totiž pouze zakazuje pohyb chodců po dálnicích a tratích. Například silnice I. třídy sice také nejsou pro chodce příliš vhodné, zakázat je ale nelze, jelikož často nastává situace, že existuje cesta pro pěší podél této silnice, avšak tato cesta není v mapě znázorněna. Zakázáním silnice

pak program najde zbytečně dlouhou trasu, nebo se do daného místa vůbec nebude možno dostat. Prozatím je tedy třeba navrhované cesty brát trochu s rezervou.

Metro

Metro je zatím řešeno dost provizorně. Principiálně je stanice metra vždy jedna a v mapě k ní vedou podzemní cesty. Respektive, takto to zatím vypadá pouze na některých stanicích, u jiných je to pouze naznačeno a většina je od zbytku mapy oddělena. Nicméně pokud by někdy v budoucnu byly OpenStreet mapy v tomto ohledu dodělány, přesuny v metru by se staly velice přesnými, jelikož by odpovídaly realitě. Prozatím, aby bylo možno se dostat i do stanic, ke kterým v mapách zatím cesta nevede, jsou mezi každým vstupem do metra a stanicí nataženy virtuální schody, což se na některých zastávkách i podobá skutečnosti, často však v reálu přestup trvá déle, než program slibuje.

7.2 Nové funkce

Znalost kompletních jízdních řádů, kterou nyní program disponuje, se dá velice dobře zužitkovat. Nad aplikací v současném stavu by nebylo nijak problémové vytvořit například rozhraní pro zobrazování jízdních řádů, jak linkových, tak zastávkových. Implementačně náročné by nebylo ani vytváření souhrnných jízdních řádů, které by zobrazovaly jízdní řád více linek v jejich společném úseku. Vyhledávání by mohlo být interaktivnější, užitečnou funkcí by se mohlo stát dodatečné vyhledávání z přestupních bodů, které by uživateli rychle našlo náhradní spoje, kdyby přestupu nevěřil. Zajímavé by mohlo být také „obecné“ vyhledávání bez zadání data a času (nebo s omezením na nějaké intervaly), kde by se místo konkrétních spojů porovnávaly spíše různé trasy, kterými se lze ze startu do cíle dostat. Toho by se mohlo dát docílit opakovaným vyhledáváním v různé dobře zvolené časy. Vylepšit by se dalo i počítání *spolehlivosti* spojení. Ta se zatím počítá jednoduše, pouze na základě časů na přestupy, nicméně na spolehlivost může mít vliv mnoho dalších faktorů jako např. denní doba, typ a interval spoje.

7.3 Jiná města

Při vývoji jsem si samozřejmě dával pozor na to, aby byl program co nejméně závislý na použitém městě. Samotná knihovna vlastně ani neví, v jakém městě hledá, jelikož používá seznamy uzlů, cest, zastávek a adres, které jí připraví *OSMReader*. Ten už na daném městě závislý je, pouze však kvůli použitému zdroji jízdních řádů a zastávek. Pro jiné město by bylo třeba implementovat nové třídy na překlad formátu jeho dat. Pokud jde o mapu, program přijme libovolnou oblast, neměla by pouze být příliš velká. Teoreticky lze tedy *OSMReader* upravit pro libovolné město nebo obecně oblast, ke které lze získat jízdní řády i se zastávkami. Pouze pokud by tato oblast byla příliš velká (násobně větší než Praha), algoritmus by už mohl být neefektivní.

7.4 Jiné než desktopová verze

Prioritou projektu bylo poskytovat co nejlepší a nejpřesnější výsledky, což je vykoupeno potřebným výkonem. Proto je aplikace vytvořena jako desktopová. Pro její implementaci do mobilního nebo webového prostředí by bylo potřeba knihovnu upravit, aby poskytovala i odlehčenou verzi algoritmu, který by nebyl tak důkladný, ale mohl být použit v méně výkonném prostředí. Bylo by také třeba vytvořit rozhraní pro příslušnou platformu.

7.5 Zvýšení efektivity vyhledávání

Rychlost vyhledávacího algoritmu se nakonec přes původní obavy ukázala jako dostatečná. Dokonce bylo možno jej zkomplikovat sekundárním kritériem spolehlivosti (viz kapitola 4.1), které algoritmus značně zpomalí, jelikož každým vrcholem se prochází vícekrát. Takto modifikovaný algoritmus už ale zejména na vzdálenějších bodech může vyhledávat trochu déle. Přibližnou závislost doby vyhledávání na vzdálenosti ukazuje tabulka 7-1. Je třeba vzít na vědomí, že konkrétní časy záleží na výkonu počítače a také na konkrétních bodech (vyhledávání v centru je o něco pomalejší, protože síť vrcholů je tam hustší), mírně se liší i v závislosti na době vyhledávání, mimo špičku bývá rychlejší.

Start hledání	Cíl hledání	Délka cesty	Doba vyhledávání
Letiště Ruzyně	Můstek – A	32 min	30 ms
Letiště Ruzyně	Nádraží Strašnice	50 min	235 ms
Letiště Ruzyně	Nádraží Hostivař	1 hod, 2 min	650 ms
Letiště Ruzyně	Uhříněves	1 hod, 22 min	1,8 s

Tab. 7-1 – Srovnání doby běhu algoritmu vzhledem ke vzdálenosti bodů

Pro zvýšení efektivity vyhledávání byly provedeny všechny očekávatelné kroky. Hledání vrcholu s nejmenší dočasnou vzdáleností probíhá vzhledem k použití haldy v čase O(log n). Procházení sousedů uzlu nevyžaduje žádné úsilí navíc, protože si uzly své sousedy pamatují. Pokud jde o odjezdy linek ze zastávek, ty jsou uloženy v setříděném seznamu pro každou zastávku, vyhledání konkrétního podle času odjezdu trvá O(log s), kde *s* je počet spojů projíždějících zastávkou. Existují však další možnosti, jakými by šel běh algoritmu zrychlit.

7.5.1 Odstranění vrcholů stupně dva

Aby cesty a silnice v mapách odpovídaly skutečnému rozložení, jsou kromě křižovatek vloženy uzly i do cest samotných. Zvláště pak v zatáčkách je přidáno mnoho uzlů, ze kterých ale vedou jen dvě cesty – dál a zpět. Takové uzly jsou v grafu vlastně vrcholy stupně dva a ty se dají poměrně jednoduše eliminovat při sloučení hran na obou stranách do jedné ohodnocené součtem vzdáleností původních hran. Toto ale aplikace nedělá, hlavním důvodem je vykreslení výsledné cesty, které by bez těchto pomocných uzlů bylo značně nepřesné. Nicméně cesta vyhledaná v grafu s eliminovanými vrcholy stupně dva by šla zrekonstruovat i v grafu původním. Při vyhledávání by pak bylo potřeba nejprve najít cesty k okolním křižovatkovým uzlům a pak hledat spojení na úrovni křižovatek.

Z měření vyplývá, že dokonce 64% uzlů v mapě Prahy je stupně dva. Jejich eliminací by vznikl graf o třetinové velikosti vzhledem k původnímu, což by mohlo přinést citelné zrychlení, i když asymptotická složitost algoritmu zůstane stejná.

7.5.2 Předpočítaní pěších cest

Jedním z důvodů, proč nebyla optimalizace uváděná v předchozí kapitole realizována, je uvažování této, silnější, avšak paměťově o mnoho náročnější varianty. Princip je jednoduchý, spočítat si časy pěších přesunů pro každou dvojici zastávek. To ale přináší řadu komplikací a algoritmus se ukázal jako poměrně rychlý, od této metody jsem tedy upustil.

Hlavním problémem je asi prostor. V Praze je kolem 3100 zastávek, což dává necelých 5 milionů dvojic. Alternativou by mohlo být počítat vzdálenosti jen mezi celými skupinami zastávek s tím, že v rámci nich jsou přesuny krátké a počítaly by se pomocí map. Skupin zastávek je méně – kolem 1200, což dává kolem 720 tisíc dvojic. Další problémy jsou stejné jako u předchozí metody optimalizace. Navíc by takovouto úpravou vznikl víceméně úplný graf zastávek, což by způsobilo, že i při hledání mezi blízkými místy by se prošel celý (minimálně by se každý vrchol otevřel). Další komplikace by pak přišla například při preferencích výpočtu rychlosti cest. Nebylo by už možno vylepšení, ve kterém by uživatel zvolil rychlost chůze pro různé typy cest – typicky na schodech by volil rychlost nižší. Jedinou, avšak podstatnou výhodou by byla získaná rychlost. Graf, který měl při posledním měření téměř 150 tisíc vrcholů, by se rázem zredukoval na 3100 vrcholů.

8. Závěr

Hlavním cílem práce bylo vytvořit takový vyhledávač spojení, který započítává libovolně dlouhé pěší přesuny a ty pak zobrazí na mapě. To výsledná aplikace splňuje. Podařilo se získat potřebné podklady pro mapy i jízdní řády a implementovat nad daty z nich získanými samotný vyhledávací algoritmus. Mým plánem bylo též prozkoumat a případně implementovat nějaké zajímavé a unikátní funkce, které problém vyhledávání spojení přináší. V tomto bodě nakonec lehce zaostává implementační část. Přestože o nápady není nouze, implementovány byly nakonec pouze základy spolehlivosti spojení. Přednost totiž nakonec dostala snaha o vytvoření aplikace použitelné v praxi, což přineslo nutnost mnoha úprav a dalo za vznik třeba celému projektu *IdosDownloader*.

Hlavním přínosem je schopnost vyhledávat spojení podle mapy bez omezení možných přestupů podle jejich délky. Mezi další by se mohla řadit implementace systému schopného sestavit databázi potřebných objektů z různých veřejně dostupných zdrojů. Do jisté míry je přínosem i praktičnost použití těchto dat a programu celkově, i když v této oblasti ještě existuje potenciál pro zlepšení. Právě však díky vyladění pro praktické využití má nyní mnohem větší smysl budoucí implementace některých nových funkcí zmíněných v kapitole 7.2. Ty by pak mohly být dalšími přínosy programu, zároveň by činily tento projekt ještě o něco unikátnějším v dané oblasti.

Reference

[1] J. Nešetřil: Teorie grafů, SNTL Praha, 1979

[2] L. Kučera: Kombinatorické algoritmy, SNTL Praha, 1989

[3] Dopravní podnik hlavního města Prahy – Hledání spojení: http://idos.dpp.cz/idos

[4] Ročenka dopravy, Praha, 2009: http://portal-beta.tsk-praha.cz/rocenka/udi-rocenka-2009-cz.pdf

[5] Universal Trasverse Mercator (UTM) system: http://en.wikipedia.org/wiki/Universal_Transverse_Mercator_coordinate_system

[6] Stránky projektu JRGPS: http://www.ksi.mff.cuni.cz/~zemlicka/projects/team/JRGPS/index.cs.html

[7] Programátorská dokumentace projektu JRGPS (V. Martínek, osobní komunikace, prosinec 2010)

[8] POI.cz: http://www.poi.cz

[9] OpenStreetMaps: http://www.openstreetmap.org

[10] OSM file format: http://wiki.openstreetmap.org/wiki/.osm

[11] Qt framework: http://qt.nokia.com/products/qt-sdk

[12] Google code – KML tutorial: http://code.google.com/intl/cs-CZ/apis/kml/documentation/kml_tut.html

[13] ROPID: http://www.ropid.cz

[14] UTM coordinate converter: http://home.hiwaay.net/~taylorc/toolbox/geography/geoutm.html

[15] OpenStreetMap Wiki – Slippy map tilenames: http://wiki.openstreetmap.org/wiki/Slippy_map_tilenames

[16] QNetworkAccessManager class: http://www.java2s.com/Code/Cpp/Qt/DownloadfromURL.htm

[17] Geofabrik Download-Bereich: http://download.geofabrik.de/osm/europe/

[18] Osmosis: http://wiki.openstreetmap.org/wiki/Osmosis

Příloha A – Obsah přiloženého CD-ROM

Součástí práce je i přiložené CD-ROM, které obsahuje zkompilované programy i jejich zdrojové kódy a potřebné datové soubory v následující adresářové struktuře:

- data\ - složka se vstupními datovými soubory všeho druhu
  - OSMReader\ - vstupní data pro *OSMReader*, soubory KML a OSM
  - IdosDownloader\ - výstupní data *IdosDownloaderu – stations a routes*
- doc\ - obrázky a diagramy použité v této práci
- osmosis\ - program pro práci s OSM soubory *[18]*
- src-gui\ - zdrojové kódy C++ projektu *PTN GUI* (uživatelské rozhraní)
- src-idosdownload\ - zdrojové kódy C# projektu *IdosDownloader*
- src-lib\ - zdrojové kódy C++ knihovny *PTNLibrary*
- src-osmreader\ - zdrojové kódy C# projektu *OSMReader*
- stable\ - složka se ZIP souborem s funkčním programem
- PTN.sln - Visual Studio 2010 solution obsahující všechny zmíněné projekty
- readme.txt - soubor s dalšími informacemi

Spustitelný uživatelský program je umístěn v ZIP souboru ve složce stable, instalaci popisuje kapitola 5.1. Jelikož bylo k vývoji použito Visual Studio 2010, odpovídá tomu rozložení souborů ve složkách s jednotlivými projekty (tj. ty začínající „src-“); každá tato složka obsahuje příslušný projektový soubor (pro C++ .vcxproj, pro C# .csproj). Spustitelné soubory každého projektu pak lze nalézt na následujícím umístění: u C# projektů je to složka src-<*projekt*>\bin\Release, u C++ projektů src-<*projekt*>\Release.

Příloha B – Formát datových souborů

V práci se používá mnoho různých formátů souborů. Mapy jsou ve formátu OSM *[10]*, zastávky z poi.cz zastupují soubory KML *[12]*. Jízdní řády se stahují jako HTML stránky a transformují se do TXT souborů. Vstupem vyhledávací knihovny jsou však soubory DAT, které mají speciální formát, jenž popíšu v této kapitole. Převod všech vstupních dat do tohoto formátu má na starosti projekt *OSMReader*, důvodem k němu je snaha o nezávislost knihovny na konkrétním formátu vstupních dat a zejména v případě OpenStreetMap také odstranění nedůležitých částí a zefektivnění načítání dat. Datových souborů je šest. Zbytek kapitoly se jim věnuje jednotlivě

1. nodes.dat

Soubor nodes.dat obsahuje uzly mapy, které jsou potřeba pro vyhledávání. Vynechávají se tedy nedostupné uzly, které neleží na žádné cestě. Adresy, v OSM reprezentovány také jako uzly, jsou vyčleněny do vlastního souboru.

První 4 bajty obsahují *počet* položek v souboru, ve zbytku je uloženo právě tolik uzlů, každý na 9 bajtech. Každý záznam obsahuje *zeměpisnou šířku* a *délku* (*float*, 4B) a *počet cest*, na kterých uzel leží (*byte*, 1B).

2. ways.dat

Soubor ways.dat nad uzly v nodes.dat definuje cesty, které najde v mapě. Vynechány jsou cesty s méně než dvěma uzly. Soubor opět začíná 4B *počtem* záznamů a dále obsahuje jednotlivé cesty. Délka jednoho záznamu je proměnná podle počtu uzlů, které na cestě leží. Záznam o cestě se skládá z následujících položek:

- *Jméno* (nulou ukončený řetězec) – jméno mají pouze ulice, pokud daná cesta nemá jméno, přítomna je pouze ukončovací nula.
- *Příznaky (flags)* (2B) – možné hodnoty definuje výčtový typ *WayFlags*.
- *Počet uzlů cesty* (ushort, 2B)

- *Identifikátory uzlů* (každý 4B) – identifikátorem uzlu je jeho pořadí od začátku souboru v nodes.dat; indexuje se od jedničky, nula v programu zastupuje neplatný uzel.

3. stops.dat

Soubor stops.dat obsahuje seznam zastávek. Připomenu, že zastávkou je konkrétní nástupní ostrůvek. Na začátku souboru jsou opět 4B pro *počet* prvků. Následuje seznam zastávek, pro každou položka o 6B:

- *Identifikátor uzlu*, který zastávku reprezentuje v mapě (int, 4B).
- *Typ* (1B) – hodnota z výčtového typu *StopType*.
- *Příznaky (flags)* (1B) – kombinace hodnot z výčtového typu *StopFlags*.

4. stopnames.dat

Soubor stopnames.dat se skupinami zastávek začíná opět 4B pro *počet* prvků, následuje 4B údaj *maxId*, který se v současné verzi už však nepoužívá. Původně reprezentoval největší ID skupiny, které bylo větší než jejich počet, protože některé identifikátory nebyly použity. Současná verze ale identifikátory přečíslovává, aby tvořily souvislou oblast. Každý záznam se skládá z následujících položek:

- *Jméno* (nulou ukončený řetězec) – platí, že všechny zastávky ze skupiny mají stejné jméno, proto je definováno zde a ne u jednotlivých zastávek.
- *ID skupiny* (ushort, 2B) – původní identifikátor skupiny, který se však již nepoužívá
- *Počet zastávek ve skupině* (ushort, 2B)
- *Identifikátory zastávek* (každý 4B) – reprezentuje zastávku ze souboru stops.dat, identifikátor je jejím pořadím od začátku souboru, indexuje se tentokrát od nuly.

5. adresses.dat

Adresy jsou uloženy v souboru addresses.dat. Ten na prvních 4B obsahuje počet 21B záznamů, které tvoří zbytek souboru obsahující následující položky:

- *Souřadnice* (2 × float, 8B) – zeměpisná šířka a délka vchodu s touto adresou.
- *Identifikátor cesty* (int, 4B) – cestou adresy se rozumí ulice, do které patří. Identifikátor cesty je určen pořadím v souboru ways.dat, který je indexován od jedničky, nula reprezentuje neplatnou cestu.
- *Číslo orientační* (int, 4B)
- *Číslo popisné* (int, 4B)
- *Dodatečné písmenko č. o.* (char, 1B) – využije se například u č. o. „11b“

6. timetables.dat

Konečně zbývají jízdní řády. Formát jejich uložení je o něco komplikovanější, odpovídá však definicím pojmů ve 2.1.2 a reprezentaci popsané v 6.3.1. První 4B souboru obsahují datum, od kterého jízdní řády začínají (*TimetablesStartDate*). Následuje čtyřbajtový počet linek a záznamy o nich v následujícím tvaru:

- *Jméno linky* (nulou ukončený řetězec) – např. „9“, „B“ nebo „X9“.
- *ID linky* (int, 4B) – např. 9, 10002 nebo 809.
- *Počet směrů linky* (byte, 1B)
- *Směry linky* obsahující následující informace
    - *Počet zastávek směru* (ushort, 2B)
    - *Identifikátory zastávek* (každý 4B)
    - *Počet tras* (ushort, 2B)
    - *Jednotlivé trasy* tvořené následujícími položkami
        - *Dny vypravení* (64 bitů) – viz 6.3.1
        - *Dny v týdnu* (byte, 1B) – viz 6.3.1
        - *Časy příjezdů a odjezdů* ke každé zastávce trasy (každý 2 × 1B) – počet záznamů odpovídá *počtu zastávek směru.*
        - *Počet spojů dané trasy* (ushort, 2B)
        - *Časy odjezdů jednotlivých spojů* (každý 2B) uložené jako počet minut od půlnoci.