



MATEMATICKO-FYZIKÁLNÍ FAKULTA Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Jan Fürst

Vizualizace dostupnosti veřejnou dopravou

Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: Mgr. Martin Pilát, Ph.D.

Studijní program: Informatika

Studijní obor: Programování a softwarové systémy

Praha 2022

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného titulu.

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 má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V dne
Podpis autora

Rád bych poděkoval panu Mgr. Martinu Pilátovi, Ph.D. za cenné rady při vedení mé práce. Dále bych rád poděkoval členům své rodiny za podporu během psaní bakalářské práce.

Název práce: Vizualizace dostupnosti veřejnou dopravou

Autor: Jan Fürst

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: Mgr. Martin Pilát, Ph.D., Katedra teoretické informatiky a matematické logiky

Abstrakt: Mnoho lidí jezdí denně veřejnou dopravou a mnohým z nich by pomohlo, kdyby bydleli na místě ze kterého se dopraví rychleji na jimi často navštěvovaná místa. Problém však je, jak takové místo k bydlení najít. To řešíme v naší webové aplikaci, ve které zobecňujeme klasické vyhledávání cest v jízdních rádech a umožňujeme tak vyhodnocovat dostupnost z uživatelem zadaných míst na všechna ostatní místa. Pro snadné vyhledávání vizualizujeme, na všech dostupných místech, vypočtenou dostupnost v interaktivní mapě. Naše webová aplikace pracuje s interní knihovnou, která zpřístupňuje funkcionality potřebnou pro vyhodnocení dostupnosti. Tuto knihovnu lze využít samostatně a řešit tak jiné problémy, které vyžadují vyhodnocení dostupnosti veřejnou dopravou.

Klíčová slova: GTFS, RAPTOR, Vizualizace dostupnosti

Title: Travel Times Visualization Based on Public Transport Timetable

Author: Jan Fürst

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Martin Pilát, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: People often use public transport to travel to places they visit on regular basis. For these people, it would be very useful if they could live in a place, from which they could get faster to all the places they visit. The problem is how to find such a place. This problem is what we are trying to solve in our web application. By generalizing the standard public transport search we make it possible to calculate the travel times from user-defined places to all other reachable places. We visualize calculated travel times in an interactive map to simplify the search for the suitable place with the lowest travel times. Our web application uses our internal library which contains the functionality required for travel time calculations. This library could be used independently on the web application to solve other problems that require evaluation of travel times by public transport.

Keywords: GTFS, RAPTOR, Travel Times Visualization

Obsah

Úvod	3
1 Cíle práce	4
1.1 Terminologie	4
1.2 Požadavky na aplikaci	4
1.3 Problémy k řešení	5
2 Data jízdních řádů	7
2.1 Zdroje dat	7
2.2 GTFS	7
3 Hledání cest	9
3.1 RAPTOR	9
3.1.1 Data	9
3.1.2 Popis algoritmu	9
3.1.3 Zrychlení algoritmu	11
4 Úpravy algoritmu RAPTOR	12
4.1 Námi provedené úpravy	12
4.2 Vyhledávání oběma směry	13
5 Zobecnění Vyhledávání	14
5.1 Zobecnění vstupní zastávky na místo	14
5.2 Zobecnění cílové zastávky na místo	15
5.3 Zadávání více vstupních míst	15
5.4 Přesnější výsledky výpočtem dostupnosti v intervalu	16
5.5 Efektivní hledání zastávek z okolí	16
5.6 Alternativní řešení	17
6 Vizualizace dostupnosti	18
6.1 Výpočet dostupnosti pro body v rastru	18
6.2 Statické vizualizace	18
6.3 Interaktivní vizualizace	20
6.4 Možné vylepšení	22
7 Implementace	23
7.1 Modul GTFSData	24
7.1.1 Třída RawData	24
7.1.2 Třída Timetable	24
7.1.3 Možné vylepšení	28
7.2 Modul RaptorAlgo	29
7.2.1 Existující implementace	29
7.2.2 Třída Raptor	30
7.2.3 Třída StopData	31
7.2.4 Třída Journey	31
7.2.5 Možné optimalizace	32

7.3	Modul StopAccessibility	33
7.3.1	Třída NearestStops	33
7.3.2	Třída StopAccessibilityFinder	33
7.3.3	Možná zlepšení	34
7.4	Modul Web	35
7.4.1	Části webové aplikace	35
7.4.2	Backend	36
7.4.3	Frontend	38
7.5	Modul Config	40
8	Výsledky	41
8.1	Chicago	41
8.2	Německo	42
8.2.1	Problémy s velkými daty	42
Závěr		44
Seznam použité literatury		45
Seznam obrázků		46
A	Přílohy	47
A.1	Dokumentace	47
A.1.1	Instalace	47
A.1.2	Konfigurační soubor	47

Úvod

Každý by chtěl bydlet v blízkosti míst, která často navštěvuje. V dnešní době se však většina těchto míst nachází ve větších městech kde jsou ceny bydlení vyšší. Ne každý má dostatek finančních prostředků k tomu, aby mohl bydlet v blízkosti navštěvovaných míst. Mnozí, zejména ti kteří cestují hromadou dopravou, pak tráví dennodenně dlouhé hodiny na cestách.

Lidé navíc často navštěvují různorodá místa, jako jsou třeba zaměstnání, škola, známí, lékaři či různé obchody. Ani pro člověka s dostatkem finančních prostředků tedy nemusí být snadné naleznout ideální lokalitu, odkud by byl schopen dostat se na často navštěvovaná místa v minimálním čase.

V současné době existuje mnoho způsobů, jak nalézt optimální cestu hromadnou dopravou mezi dvěma místy. Pokud vím, není k dispozici vyhledávač, který by tento koncept rozšiřoval a byl schopen nalézt optimální cesty mezi několika zadanými a všemi ostatními místy.

Na základě tohoto rozšířeného konceptu se v této práci pokusíme ohodnotit všechna místa podle dostupnosti veřejné dopravy ze zadaných míst v daných casech. Vypočtenou dostupnost vizualizujeme ve všech městech a tím umožníme snadnou orientaci ve výsledcích. Pro zpřístupnění aplikace běžným uživatelům vytvoříme webovou aplikaci. Výsledky aplikace budou pro uživatele dopočteny v přijatelném čase. Samotná aplikace bude dostatečně obecná, aby byla schopna pracovat s jízdními rády z různých lokalit.

V první kapitole detailně popisujeme cíle práce. Ve druhé kapitole popisujeme data jízdních rádů a jejich poskytovatele. Ve třetí kapitole popisujeme námi zvolený algoritmus pro hledání cest v jízdních rádech. Ve čtvrté kapitole popisujeme vlastní úpravy algoritmu zvoleného v předchozí kapitole. V páté kapitole zobecňujeme algoritmus z předchozí kapitoly a umožňujeme tak vyhledávat dostupnost z několika vstupních míst na všechna ostatní výstupní místa. V šesté kapitole popisujeme přístupy k vizualizaci dostupnosti. V sedmé kapitole popisujeme implementaci myšlenek, popsaných v předchozích kapitolách a popisujeme také implementaci webové aplikace. V osmé kapitole si ukážeme, jak naše aplikace funguje s jízdními rády z různých lokalitách. V příloze dokumentujeme, jak se má naše aplikace zprovoznit a jak ji lze nakonfigurovat.

1. Cíle práce

V této kapitole si detailně popíšeme cíle, kterých se budeme snažit dosáhnout. Zbylé kapitoly budou řešit některé z cílů uvedených zde.

1.1 Terminologie

- **Dostupnost** definujeme jako dobu, za jakou se ze vstupního místa dostaneme na cílové místo. S takto definovanou dostupností se nám, v porovnání s konkrétními časy, bude snáze pracovat ve statistických výpočtech.
- **Místo** je nějaká pozice v prostoru, která není příliš vzdálená od některé ze zastávek. V této práci zásadně odlišujeme význam místa a zastávky.
- **Zastávka** je místo, na kterém staví nějaký spoj.
- **Trip** překládáme jako **jízdu**. Jízda je posloupnost zastávek.
- **Route** překládáme jako **trasu**. Trasa je množina jízd jedoucích přes stejné zastávky.

V kapitole 2 však pracujeme s **GTFS trasou**, jejíž definice se liší od definice tras, kterou používáme v následujících kapitolách. GTFS trasy totiž mohou obsahovat jízdy, které jedou přes různé množiny zastávek.

- **Transfers** překládáme jako **přestupy**. Přestupovat můžeme mezi dvěma zastávkami a přestup trvá nějaký čas.
- **Spoj** používáme pro nějaký dopravní prostředek, který jede v konkrétní čas po zastávkách jízdy.
- **Journey** překládáme jako cestu. Cesta je posloupnost spojů a přestupů, kterými se doprovíme z počátečního místa na cílové.

1.2 Požadavky na aplikaci

Frontend by měl:

- Poskytovat rozhraní pro zadávání vstupu uživatelem.
Uživatel může zadávat více často navštěvovaných míst.
Zadané místo musí být v dostatečné blízkosti nějaké zastávky.
S místem lze asociovat důležitost, která určuje, jak moc je dostupnost z tohoto místa zohledněna ve vypočtených dostupnostech.
Uživatel může s jedním místem asociovat více časů, ve kterých chce z místa odjízdět.
Uživatel má mít možnost nezadávat čas přesně, ale nechat aplikaci vyhodnotit dostupnost pro nějaké okolí zadaného času.

- Poskytovat rozhraní pro zobrazování výstupu uživateli.
Na výstupu se uživateli vizualizuje vypočtená dostupnost.
Dostupnost má být vizualizovaná na všech místech.
Uživatel má být ve vizualizaci schopen snadno najít dostupnost pro konkrétní místa.
- Posílat požadavky na backend, který běží na serveru.

Backend by měl:

- Zpracovávat dotazy z frontendu v rozumném čase.
- Aktualizovat data jízdních řádů.
- Delegovat vyhodnocení dostupnosti na knihovnu.

Knihovna by měla:

- Být nezávislá na webové aplikaci.
- Dostatečně obecná, aby mohla pracovat s jízdními řády po celém světě.
- Načítat data jízdních řádů a přetransformovat je do formátu vhodného pro vyhledávání.
- Pro daný vstup vyhodnocovat dostupnost na všech místech.

1.3 Problémy k řešení

Data

Nejprve potřebujeme mít přístup k datům jízdních řádů. Bez samotných dat se nemůžeme pokoušet vyhodnocovat dostupnost.

Vyhodnocení dostupnosti

Pro vyhodnocení dostupnosti musíme nejprve nalézt cesty, které vedou mezi vstupními a výstupními místy. Tento problém si rozdělíme na následující podproblémy, které budeme postupně řešit.

- Vyhledání cesty mezi 2 zastávkami. To je klasický problém, který řeší běžné vyhledávače.
- Vyhledání cesty mezi 1 vstupní zastávkou a všemi ostatními zastávkami. Toto je důležité, neboť cílíme na vyhodnocení dostupnosti na všech místech. Vyhodnocení dostupnosti na všech zastávkách je tedy přirozený mezikrok.
- Zadávání více vstupních zastávek. Vstupní zastávky jsou ty, které uživatel zadává jako často navštěvovaná místa. Takových zastávek je, v porovnání se všemi zastávkami, velmi málo. Tento mezikrok potřebujeme, abychom uživatelům umožnili zadávat více často navštěvovaných míst.

- Zobecnění cílové zastávky na místo. V této fázi jsme již schopni vyhodnocovat dostupnost na všech zastávkách. Abychom byli schopni vyhodnocovat dostupnost na libovolných místech, potřebujeme zobecnit zastávky na místa.
- Zobecnění vstupní zastávky na místo. Toto zobecnění potřebujeme, neboť chceme našim uživatelům umožnit zadávat na vstupu libovolná místa, ne jen zastávky.
- Vyhledávání oběma směry. Předchozí podproblémy počítají se situací, kdy uživatelem zadaná místa jsou ta, ze kterých chce vyjízdět. Uživatele by však mohla zajímat situace, kdy na zadaná místa chce v daný čas přijízdět. Tuto možnost mají některé vyhledávače, které umožňují vyhledávat spoje dle času dojezdu.
- Vyhodnocování dostupnosti pro časový interval. Vyřešení tohoto podproblému by umožnilo uživatelům nezadávat vstupní čas přesně. Mohli bychom totiž vyhodnocovat dostupnost pro nějaké okolí zadaného času.

Vizualizace

Potřebujeme zajistit způsob vizualizace, který uživatelům umožní dohledat ideální místo podle dostupnosti.

Potřebujeme najít způsob, jak vizualizovat dostupnost pro všechna místa.

2. Data jízdních řádů

V této kapitole řešíme naši potřebu přístupu k datům, zmíněnou již v sekci 1.3.

Tato data jsou dostupná v různých datových formátech, v České republice například ve formátu JDF (Jednotný datový formát). Tento a jemu podobné proprietární formáty však sdílí stejnou nevýhodu. Aplikace pracující s nimi buď nejsou schopny pracovat s daty od jiných zprostředkovatelů, nebo musí vytvářet netriviální adaptéry. Takový přístup je zcela nepřípustný, zvláště existují-li alternativy.

Vhodnou alternativou je formát GTFS (GOOGLE, 2022). Tento formát je spravovaný společností Google a je ve světě v podstatě standardem pro popis jízdních řádů.

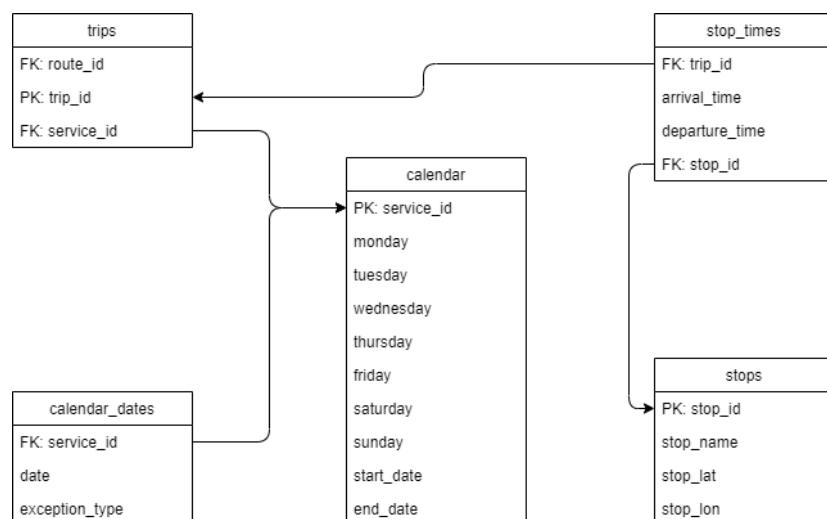
2.1 Zdroje dat

Za účely vývoje aplikace je vhodné pracovat s daty, která nám jsou co možná nejbližší, proto budeme pracovat s daty jízdních řádů v České republice. Taková data, ve vhodném formátu GTFS (GOOGLE, 2022), poskytuje společnost Pražské integrované dopravy PID (2022).

Dalším zdrojem dat, nejen pro Českou republiku, je <https://transitfeeds.com/>. Vůbec největším zdrojem jednotných dat, se kterými budeme pracovat jsou GTFS for Germany, dostupné na https://gtfs.de/en/feeds/de_nv/.

2.2 GTFS

Formát GTFS je rozšířením formátu CSV (Comma-separated values). GTFS nevynucuje typ dat, nedá se tedy spoléhat na to, že např. **id** budou uložena jako čísla. Všechna data z formátu GTFS, relevantní pro náš problém, jsou zobrazena na obrázku 2.1.



Obrázek 2.1: Relevantní obsah souborů ve formátu GTFS.

Následuje krátký popis nejdůležitějších rysů každé z používaných složek.

- Soubor stops.txt obsahuje geografické souřadnice ve formátu WGS84 (World Geodetic System), ty použijeme k vizualizaci a ke generování přestupů. Formát GTFS sice obsahuje přestupy v transfers.txt, ty však obsahují jen oficiální přestupy, ne všechny možné.

Mezi zastávkami mohou existovat takové zastávky, na kterých nestaví žádný spoj. Takové zastávky budeme chtít odstranit.

- Soubor stop_times.txt obsahuje informace o příjezdu a odjezdu z jednotlivých zastávek. Dle specifikace formátu GTFS, může časový formát příjezdů a odjezdů přesahovat klasický 24 hodinový formát. Taková situace nastává u spojů, které jedou souvisle přes půlnoc.
- **Route_id** ze souboru trips.txt identifikuje GTFS trasu, pod kterou patří skupina jízd.

Route_id společně se všemi zastávkami, přes které trasa jede, použijeme k identifikaci pozměněného významu trasy, který používáme v následujících kapitolách.

- Soubor calendar.txt přiřazuje k jízdě dny, kdy je provozována. Každý řádek v kalendáři má navíc omezenou platnost. Podle GTFS best practices, popsaných na adrese <https://gtfs.org/schedule/best-practices/>, by měla být minimální platnost nejméně 7 dnů.
- V souboru calendar_dates.txt jsou explicitně vypsány výjimky, kdy je jízda provozována a podle kalendáře by být neměla, nebo naopak kdy jízda není provozována a podle kalendáře by být měla.

3. Hledání cest

V této kapitole řešíme problém hledání cesty mezi dvěma zastávkami, který jsme popsali již v sekci 1.3.

Pro vyhledávání cest v jízdních rádech potřebujeme nějaký algoritmus. Existující řešení spadají v zásadě do dvou kategorií.

1. Algoritmy řešící problém jako grafový. Hledání nejkratších cest v jízdních rádech však není snadná úloha, neboť cestovat hromadnou dopravou se dá jen ve chvíli, kdy jede nějaký spoj. Existující přístupy pak musí řešit problém čekání na zastávce. Důsledkem toho je obecně pomalejší hledání.
2. Algoritmy neřešící problém jako grafový. Jedním z hlavních zástupců této kategorie je algoritmus RAPTOR Delling a kol. (2015). Tento algoritmus nevyžaduje předzpracovaná data, každou trasu navštíví maximálně jednou a dá se počítat paralelně. Navíc umožňuje omezit počet přestupů během výpočtu.

3.1 RAPTOR

3.1.1 Data

RAPTOR bude pracovat s daty z formátu GTFS. Nejprve je však musíme převést do datové struktury, která bude podporovat operace vyžadované algoritmem RAPTOR.

Pro každou trasu si potřebujeme pamatovat zastávky, přes které trasa vede, uspořádané od první po poslední. Dále si chceme zapamatovat také jízdy, které jedou po trase, uspořádané od nejdřívější k nejpozdější.

Pro každou zastávku si uložíme všechny trasy, na kterých zastávka leží. Ke každé zastávce přiřazujeme **multilabel**, který pro každou iteraci obsahuje nejdřívější čas příjezdu.

Dále si potřebujeme uložit přestupy, které musí být tranzitivní. Ty však nejsou součástí GTFS dat a budeme si je tedy muset vygenerovat. Generování přestupů konkrétně popíšeme v sekci 7.1.2.

Budeme si chtít ukládat také seznam označených zastávek. Ten slouží k optimálnějšímu procházení tras.

3.1.2 Popis algoritmu

RAPTOR je založený na metodě dynamického programování, pracuje tedy v iteracích. Aktuální iteraci označujeme indexem k .

Vstupem je počáteční zastávka, čas výjezdu a cílová zastávka.

Výstupem je cesta s minimálním časem dojezdu skládající se z maximálně k jízd a $k - 1$ přestupů.

Při výpočtu každé iterace dodržujeme invariant, že na začátku k -té iterace jsou hodnoty **multilabelu** do indexu $k - 1$ správné.

Začínáme inicializací. Nejdřívější časy příjezdu v **multilabelu** každé zastávky nastavíme na nekonečno. Počáteční zastávku přidáme do seznamu označených

zastávek. **Multilabel** v nulté iteraci nastavíme na čas výjezdu ze vstupu. Aktuální iteraci nastavíme na 1.

Dále pro každou iteraci provádíme následující tři fáze.

1. V první fázi propagujeme, v rámci všech **multilabelů**, příjezdové časy z předchozí iterace do aktuální iterace. Tím nastavíme horní mez pro příjezd v této iteraci.

Dále projdeme označené zastávky a uložíme si množinu označených tras, které obsahují označenou zastávku. Pro každou označenou trasu si navíc pamatujeme zastávku skrze kterou byla trasa označena. Pokud označujeme již označenou trasu, zapamatujeme si tu zastávku, která leží blíže začátku trasy.

Vyprázdníme seznam označených zastávek, abychom v následující iteraci procházeli jen nově označené zastávky.

```
foreach stop:  
    stop.multilabel[k] = stop.multilabel[k-1]  
  
    foreach markedStop in markedStops:  
        foreach route in markedStop.Routes:  
            markedRoutes.addOrUpdate(route, markedStop)  
  
markedStops.clear()
```

2. Ve druhé fázi projdeme všechny označené trasy.

Postupně procházíme zastávky na trase, od první označené, dokud nenarazíme na zastávku pro kterou máme definovanou nejdřívější jízdu. Jednodušeji řečeno, hledáme jízdu na kterou můžeme přestoupit, neboli jízdu, ve které z aktuální zastávky odjízdíme poději, než na ni přijedeme v $k - 1$ iteraci.

Následně projdeme všechny zbylé zastávky a pokusíme se aktualizovat čas příjezdu na tyto zastávky s využitím právě nalezené nejdřívější jízdy.

Mohlo se nám však stát, že jsme v $k - 1$ iteraci přijeli na některou ze zastávek dříve, než kdy na ni přijedeme aktuálně nejdřívější jízdou. Z takové zastávky se musíme znova pokusit nalézt nejdřívější jízdu.

```
foreach (route, markedStop) in markedRoutes:  
    # find earliest trip  
    foreach stop on route (in order) beginning from markedStop:  
        foreach trip on route (in order):  
            if(stop.multilabel[k-1] < trip.departureTimeFrom(stop)):  
                currentStop = stop  
                currentTrip = trip  
                break  
  
                foreach stop on route (in order) beginning from currentStop:  
                    currentTrip = tryFindEarliestTripThan(currentTrip)  
                    stop.updateMultilabelByTrip(currentTrip) #marks updated
```

- Ve třetí fázi využijeme přestupy z nově označených zastávek. Každým přestupem se pokusíme aktualizovat čas příjezdu na zastávku, na kterou přestupujeme.

```
foreach markedStop:
    foreach transfer from markedStop:
        transfer.targetStop.updateMultilabelByTransfer()#marks updated
```

Aktualizací času myslíme zápis příjezdového času do **multilabelu** pro k -tou iteraci. Při úspěšné aktualizaci příjezdového času na zastávku ve druhé a třetí fází zastávku označujeme.

Algoritmus se zastaví v případě, že dosáhl omezení počtu iterací, nebo pokud v první fází neexistují žádné označené zastávky, tedy není co vylepšovat.

3.1.3 Zrychlení algoritmu

Optimalizace

RAPTOR popsaný v Delling a kol. (2015) navíc využívá optimalizace **lokální prořezávání** a **cílové prořezávání**.

- Optimalizace **lokálního prořezávání** zavádí navíc pro každou zastávku položku, jenž obsahuje dosavadní nejdřívější čas příjezdu. Díky této optimalizaci nemusíme v první fázi algoritmu propagovat příjezdové časy. K zápisu příjezdových časů do **multilabelu** tedy dochází jen ve chvíli, kdy časy aktualizujeme. Vypozorovali jsme, že tato optimalizace vede ke zhruba 5% zrychlení vyhledávání.
- Optimalizace **cílového prořezávání** spočívá v tom, že neoznačujeme zastávky s příjezdovým časem vyšším než je aktuální příjezdový čas na cílové zastávce.

Paralelizace

Algoritmus je možné zrychlit paralelizací viz Delling a kol. (2015, Sekce 3.3). Konkrétně se jedná o paralelizaci druhé fáze algoritmu, která je časově nejnáročnější.

4. Úpravy algoritmu RAPTOR

V této kapitole si popíšeme, jak jsme upravili algoritmus RAPTOR popsaný v kapitole 3.

Následující úpravy nám pomohou řešit problémy, související s vyhodnocováním dostupnosti, popsané v sekci 1.3.

Vyřešíme problém:

- Vyhledání cesty mezi 1 vstupní zastávkou a všemi ostatními zastávkami

Navrhнемe řešení problému:

- Vyhledávání oběma směry.

4.1 Námi provedené úpravy

Zrušení cílového prořezávání

Optimalizace cílového prořezávání, popsaná v sekci 3.1.3, je vhodná, pokud hledáme pouze dobu příjezdu na cílovou zastávku. Pokud ale tuto optimalizaci nevyužijeme a neurčíme dokonce ani cílovou zastávku, pak můžeme tento algoritmus využít k výpočtu příjezdového času na všechny dostupné zastávky.

Spouštění algoritmu z více počátečních zastávek

Pro zobecnění vstupní zastávky na místo budeme potřebovat vypočítat časy příjezdu ze zastávek v okolí zadaného místa na všechny cílové zastávky. Toto zobecnění detailně popisujeme v sekci 5.1.

Opakované spouštění algoritmu RAPTOR je velice neefektivní. Naštěstí se dá tento problém řešit mnohem jednodušeji úpravou algoritmu.

Místo jedné počáteční zastávky spustíme RAPTORa na několika počátečních zastávkách. Počátečními zastávkami myslíme zastávky v okolí zadaného místa. V inicializaci označíme všechny počáteční zastávky a každé zastávce nastavíme hodnotu **multilabelu** v nulté iteraci na součet vstupního času a času potřebnému k příchodu na zastávku ze zadaného místa.

Hledání cest v intervalu

Pro statisticky přesnější výpočty dostupnosti bychom chtěli spouštět RAPTORa opakovaně v určitém intervalu. Opakované spouštění RAPTORa však vede k násobnému zpomalení výpočtu. Řešením je upravená verze rRAPTOR detailně popsaná v Delling a kol. (2015, Sekce 4.2).

Nevýhodou tohoto řešení je, že v něm nemůžeme využít optimalizace lokálního prořezávání, popsaného v sekci 3.1.3. Použití položky s dosavadním nejdřívějším příjezdem by vedlo ke ztrátě významu hodnot **multilabelu** v jednotlivých iteracích, nemohli bychom tedy omezeným počtem iterací omezit počet přestupů. Navíc bychom tím porušili invariant.

V upravené verzi rRAPTOR spouštíme algoritmus postupně od posledního času v intervalu k prvnímu. Mezi jednotlivými spuštěními však zachováme hodnoty **multilabelu**. Navíc v první fázi algoritmu smíme povolit jen propagaci příjezdových časů, které jsou dřívější, než příjezdové časy nastavené v aktuální iteraci.

Co je velice důležité a v původním článku není popsáno, je vyhodnocování nejdřívějšího času příjezdu na zastávku. Jelikož se výpočty intervalu mohou lišit v počtu iterací, není jasné, ve které iteraci se uchovává celkově nejdřívější čas příjezdu. Při vyhodnocování tedy musíme projít příjezdové časy **multilabelu** pro všechny iterace a nejmenší z časů je námi hledaným ohodnocením.

Tato upravená verze vede ke zrychlení celkového výpočtu časů z intervalu. Míra zrychlení závisí na časových odstupech v intervalu a na hustotě jízd. Pro hrubou představu jsme se při výpočtech dvou časů v intervalu pěti minut dostali ze 2-násobného na zhruba 1,1-násobné zpomalení celkového času výpočtu.

4.2 Vyhledávání oběma směry

Chtěli bychom, aby vypočtená dostupnost na zastávkách byla symetrická. Tím myslíme to, že by měla být vypočtena nejen z času, za který se ze zadaného místa dostaneme na cílové místo, ale i z času, za který se z cílového místa dostaneme na zadáne místo.

Jízdní řády se zdají být poměrně symetrické. Mohlo by se tedy zdát, že opačný směr můžeme zanedbat. Není to však pravidlem.

Jednoduchá záměna počátečního a cílového místa není možná, neboť počátečních míst je mnohem méně než cílových míst. Následně by algoritmus RAPTOR musel počítat dostupnost z každého cílového místa na všechna ostatní místa a to není možné vypočítat v rozumném čase.

Tento problém bychom však mohli vyřešit inverzí chodu algoritmu RAPTOR.

„Inverze“ chodu algoritmu

RAPTOR umí počítat jen s časem výjezdu z počáteční zastávky. Pokud bychom chtěli počítat s časem dojezdu na cílovou zastávku, mohli bychom využít inverze chodu algoritmu.

Inverze chodu je zde použita ve smyslu obrácení významu času. Takový RAPTOR by cestoval proti směru jízd a hledal by navazující jízdy, které jedou dříve, než je zastávka navštívena.

5. Zobecnění Vyhledávání

V této kapitole vyřešíme zbylé problémy s vyhodnocováním dostupnosti ze sekce 1.3:

- Zobecnění vstupní zastávky na místo.
- Zobecnění cílové zastávky na místo.
- Zadávání více vstupních zastávek.
- Vyhodnocování dostupnosti pro časový interval.

5.1 Zobecnění vstupní zastávky na místo

Pro zobecnění vstupní zastávky na místo potřebujeme vypočítat dostupnost ze všech zastávek v okolí vstupního místa na všechny dostupné zastávky. Zastávky v okolí vstupního místa jsou ty, na které jsme schopni přijít pěšky a odjet z nich nějakým spojem.

Dostupnost ze vstupního místa na cílovou zastávku je rovna součtu doby strávené chůzí na zastávku v okolí, času, kdy čekáme na příjezd spoje, a doby jízdy z této zastávky na cílovou zastávku. Pro každou cílovou zastávku navíc bereme právě minimální dostupnost ze všech dostupností vypočtených ze zastávek v okolí vstupního místa.

Problém však je, že zastávek v okolí vstupního místa může být mnoho a následné spouštění algoritmu RAPTOR z každé této zastávky je velice výpočetně náročné.

Dále popíšeme naše přístupy k optimalizaci výpočtu.

Výběr k -nejbližších sousedů

Výběrem k -nejbližších sousedů snížíme počet sousedních zastávek a tím urychlíme výpočet.

Pro místa s velkou hustotou zastávek však mohou být výsledky velice nepřesné, neboť zanedbáváme velké množství zastávek, které sice nepatří mezi nejbližší, ale jsme schopni na ně dojít pěšky.

Navíc nevyužíváme vlastnosti, že přes blízko položené zastávky často jezdí stejně spoje.

Rozdělení zastávek do ekvivalenčních tříd

Definujeme relaci ekvivalence pro zastávky dle spojů, které přes tyto zastávky jezdí. Do stejné ekvivalenční třídy tedy umístíme všechny zastávky, přes které jezdí stejně spoje.

Dále z každé ekvivalenční třídy vybereme jednoho reprezentanta, např. takovou zastávku, která je nejblíže k zadanému místu.

Seznam všech reprezentantů tvoří zastávky v okolí, ze kterých nám stačí spouštět výpočet.

Toto řešení sice eliminuje některé zastávky, ukázalo se však, že výsledný počet zastávek je stále příliš vysoký.

(Minimální) pokrytí množiny

V tomto přístupu máme množinu zastávek z okolí zadané zastávky a chceme najít podmnožinu této množiny takovou, že sjednocení všech spojů jedoucích přes zastávky množin budou ekvivalentní.

Tento problém lze řešit výpočtem minimálního pokrytí množiny. To je sice NP-úplný problém, my si však vystačíme s aproximačním algoritmem, který tento problém řeší v polynomiálním čase.

Výsledky tohoto přístupu jsou lepší než ty, ke kterým vedly předchozí přístupy. Algoritmus RAPTOR je však stále spouštěn pro každou zastávku zvlášť, přestože trasy které algoritmem procházíme se značně překrývají.

RAPTOR spuštěn s více počátečními zastávkami

Úprava, popsaná v kapitole 4.1, nám umožňuje označit za počáteční zastávky všechny zastávky z okolí zadaného místa.

To znamená, že si vystačíme s jediným spuštěním algoritmu RAPTOR a bez znatelného zpomalení běhu algoritmu.

5.2 Zobecnění cílové zastávky na místo

Vyhledávání příjezdového času z jednoho zadанého místa na libovolné místo, které je v pěší vzdálenosti od dostupných zastávek, lze řešit minimálně následujícími dvěma způsoby.

Rozdelení mapy na segmenty

Mapu rozdělíme na segmenty. Pro každý segment určíme zastávku ze které approximujeme, v závislosti na rychlosti chůze, dojezdové časy do ostatních míst v segmentu.

Pokud bychom zvolili segmenty stejně velikosti, měli bychom buď příliš mnoho segmentů, nebo by výpočty pro lokality s velkou hustotou zastávek nebyly příliš přesné. Ideální by tedy bylo mít segmenty různě velké, v závislosti na hustotě zastávek.

Tento přístup se zdá být z pohledu implementace poměrně složitý, navíc approximace v rámci segmentu nemusí dávat dostatečně přesné výsledky.

Využití zastávek z okolí

Tento způsob je podobný přístupu, popsanému v úvodu sekce 5.1.

Podíváme se na dostupnosti na zastávkách v okolí cílového místa a přičteme k nim čas, za který se z těchto zastávek dostaneme pěšky do cílového místa. Minimem z vypočítaných dostupností pak ohodnotíme cílové místo.

5.3 Zadávání více vstupních míst

Dostupnost pro více míst umíme vypočítat spuštěním algoritmu RAPTOR pro každé ze zadaných míst. Dostupnosti vypočtené ze zadaných míst potřebujeme agregovat, abychom získali jednu dostupnost pro každé cílové místo.

Jako agregační funkci zvolíme vážený průměr, neboť očekáváme že naše uživatelé bude zajímat právě průměrná dostupnost. Navíc jim tak umožníme určovat důležitost vstupních míst.

Agregaci lze však korektně použít jen pro zastávky, neboť pro každé vstupní místo může existovat jiná zastávka z okolí cílového místa, přes kterou na cílové místo dojdeme pěšky.

Kdybychom nejdříve agregovali dostupnost zastávek a až poté počítali dostupnost na cílovém místě, nebyly by vypočtené výsledky korektní, neboť bychom na cílové místo docházeli jen z jedné ze zastávek v okolí.

Pro výpočet dostupnosti na cílovém místě tedy musíme nejprve vypočítat dostupnost z jednotlivých vstupních míst. Až poté můžeme dostupnost aggregovat.

5.4 Přesnější výsledky výpočtem dostupnosti v intervalu

Aktuální způsob výpočtu dostupnosti má tu nevýhodu, že nezohledňuje frekvenci jízd. V případě, že se zadaným časem trefíme přesně do odjezdu jízdy, která jede jen jednou za hodinu, můžeme dostat lepsí dostupnost než z jízdy která jezdí každých 10 minut, jenže na ni musím v daném čase 9 minut čekat.

Pro zohlednění frekvence jízd při výpočtu dostupnosti využijeme několika výpočtů dostupnosti v intervalech okolo zadaného času. Následným výpočtem průměru z dostupností se zbavíme nadhodnocení jízd s nízkou frekvencí odjezdů.

Správná volba délky intervalu je klíčová. Pokud bychom zvolili délku intervalu příliš krátkou, stal by se opakováný výpočet příliš náročný. V případě, že naopak zvolíme délku intervalu příliš dlouhou, může se stát že výsledky nebudou příliš užitečné.

Dále se potřebujeme vyhnout situaci, kdy je délka intervalu stejná jako perioda jízd.

Následující dvě řešení volby délky intervalu se jeví jako použitelná.

1. Zvolíme tak krátký interval, aby dokázal zastihnout většinu odjezdů, ale ne kratší. Jako vhodná délka takového intervalu se ukázala být jedna minuta, neboť takové časové rozlišení je používáno v jízdních rádech.
2. Zvolíme delší interval, například pětiminutový. V tomto intervalu určíme náhodnou minutu pro níž budeme dostupnost počítat. Toto řešení snižuje počet potřebných výpočtů a vyhýbá se situaci, kdy je délka intervalu stejná jako perioda jízd.

Při implementaci budeme chtít využít optimalizace algoritmu RAPTOR pro vyhledávání v intervalu, viz sekce 4.1. Vzhledem ke zrychlení, které nám tato optimalizace přináší, není problém spouštět RAPTORa opakovaně pro intervaly délky jedna minuta.

5.5 Efektivní hledání zastávek z okolí

K řešení předchozích problémů často používáme zastávky z okolí jiné zastávky. Triviální řešení, které porovnává vzdálenosti mezi všemi dvojicemi zastávek, je

příliš pomalé.

Tento problém se obecně nazývá **Range searching** a existuje mnoho přístupů, které ho řeší. My jsme zvažovali následující přístupy:

1. **Quad-tree**. Poměrně dobré řešení.
2. **Spatial hashing**. Horší než **quad-tree** pro velké množství objektů, pro detailní porovnání viz Barker (2022).
3. **Binary space partitioning**. Vhodné spíše pro různorodé tvary objektů, viz <https://stackoverflow.com/a/26517609/17686273>. Pro hledání sousedních zastávek se spíše nehodí.
4. **R-tree**. Lepší než quad-tree, zejména pro hledání sousedů v daném okolí, pro detailní porovnání viz Kothuri a kol. (2002).

Nejlepším řešením našeho problému se ukázal být **R-tree**. Abychom si ušetřili práci, chtěli bychom najít již hotovou implementaci této datové struktury. Pro jazyk C# se nabízí knihovna **Rbush**. Tato knihovna implementuje datovou strukturu **R-tree**, společně s dalšími optimalizacemi.

Složitost nalezení sousedních zastávek pro danou zastávku byla, za použití triviálního řešení, $O(n)$. Po optimalizaci se v průměrném případě dostaváme na složitost $O(\log n)$.

5.6 Alternativní řešení

Alternativou k předchozím krokům by mohlo být takzvané unrestricted walking, popsané v článku Phan a Viennot (2019).

Jedná se o rozšíření algoritmu RAPTOR o neomezené přecházení mezi zastávkami. Algoritmus může umožnit nalezení tras rychlejších o zhruba deset procent za cenu zhruba šestinásobného zpomalení výpočtu a předvýpočtu přechodů pomocí techniky hub labeling, trvajícího několik hodin.

Původní řešení sice počítá jen s přestupy do 100 metrů, mohlo by se však podařit prodloužit tuto vzdálenost na jednotky kilometrů a tím umožnit vyhledávání z libovolného místa na všechna dostupná místa.

6. Vizualizace dostupnosti

V této kapitole řešíme problémy s vizualizací ze sekce 1.3. Abychom uživatelům usnadnili orientaci ve výsledcích a umožnili jim snadné vyhledávání nejdostupnějších míst, vizualizujeme dostupnost na mapě. Vizualizace dělíme do dvou druhů, statické a interaktivní.

6.1 Výpočet dostupnosti pro body v rastru

Vizualizace dostupnosti na zastávkách je poměrně snadná. Problém nastane, chceme-li vizualizovat místa. Všech míst může být totiž nekonečně mnoho.

Potřebujeme tedy zvolit nějaké rozlišení, pro které budeme místa vizualizovat. Místa v daném rozlišení rozmísťujeme pravidelně do mřížky. Takto uspořádaná místa nazveme rastrem.

Již pro rozlišení 100×100 , tedy $10\,000$ míst trvá vyhodnocení dostupnosti v místech řádově desítky sekund. Tento čas částečně zkrátíme využitím optimalizace popsané v sekci 5.5. Pro praktické použití je však tento výpočet příliš pomalý.

Nejnáročnejší část výpočtu je hledání sousedních zastávek konkrétního bodu, které potřebujeme pro vyhodnocení dostupnosti.

Při pevně daném rozlišení se však tyto body, ani jejich sousední zastávky nemění. Toho můžeme využít tak, že si sousední zastávky bodů vyhledáme předem. Dosáhneme tak výrazného zrychlení a dostupnost pro stejných $10\,000$ bodů jsme schopni vypočítat za méně než jednu sekundu.

6.2 Statické vizualizace

Statické vizualizace můžeme využít pro zobrazení výsledků během vývoje aplikace. V porovnání s interaktivními vizualizacemi je jejich implementace poměrně snadná. Nejsou však příliš vhodné pro uživatele, neboť neumožňují snadné vyhledání nejdostupnějších míst.

Implementace

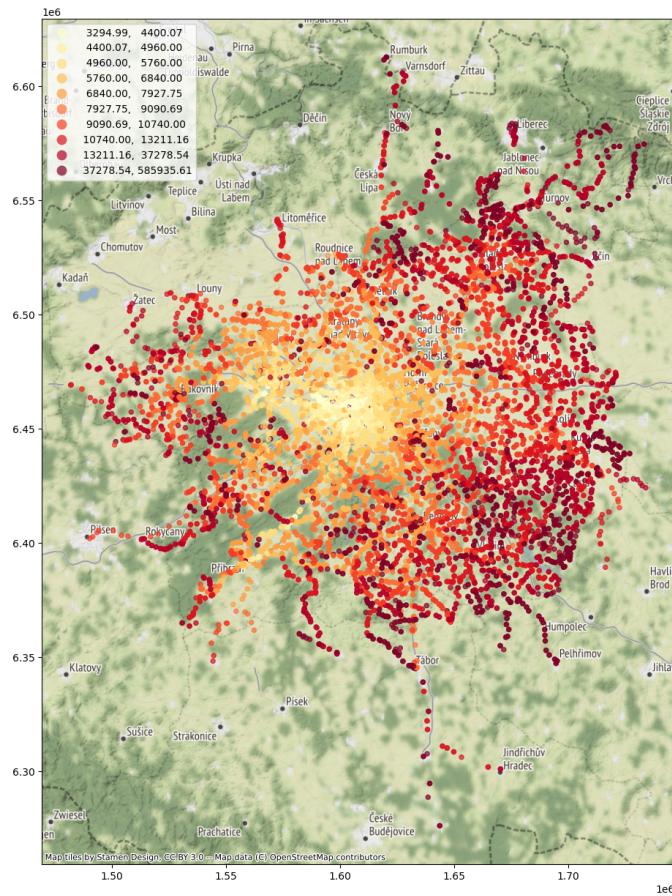
K implementaci statických vizualizací jsme využili jazyk Python, neboť zpřístupňuje knihovny, které velice usnadňují práci s daty a jejich následnou vizualizaci.

Konkrétně jsme využili následující knihovny.

- Pandas — pro čtení dat ve formátu CSV.
- Shapely — pro reprezentaci bodu pomocí souřadnic.
- GeoPandas — pro práci s body.
- Contextily — pro přidání podkladové mapy.
- Matplotlib — pro vykreslení vizualizace.

Vizualizace zastávek

Dostupnost na zastávkách vizualizujeme pomocí barevných bodů, viz obrázek 6.1. Pro souřadnice každé zastávky vykreslíme do mapy bod. Barva bodu je závislá na dostupnosti vypočtené pro zastávku pomocí algoritmu RAPTOR.



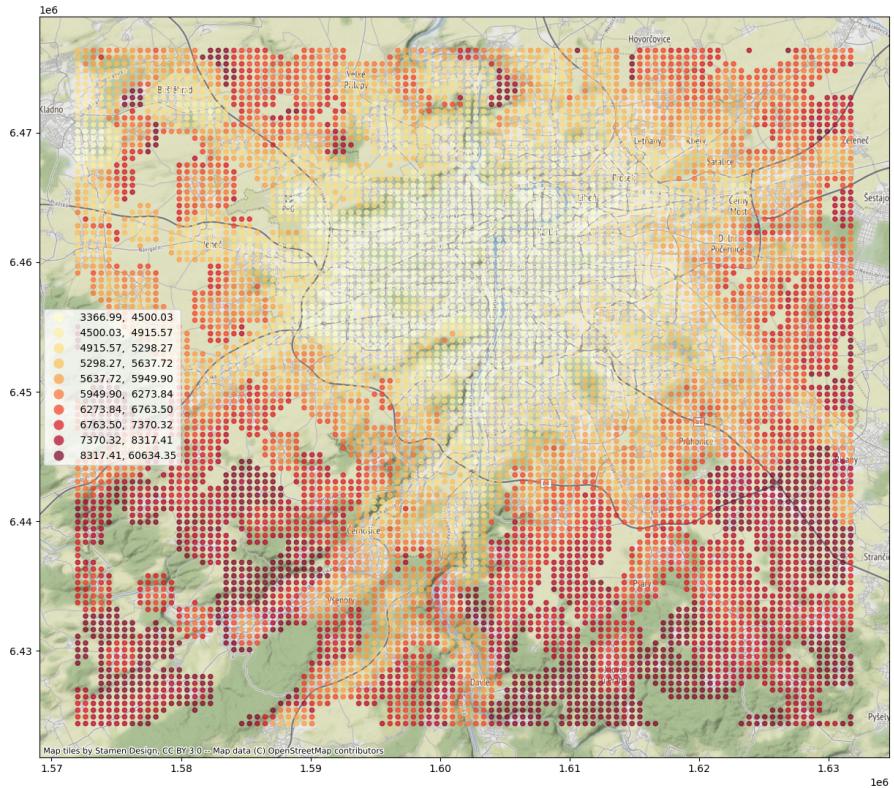
Obrázek 6.1: Vizualizace dostupnosti na zastávkách z Malostranského náměstí, Kladna a Příbrami. Hodnoty v legendě vyjadřují dostupnost v sekundách.

Vizualizace míst

Nejprve potřebujeme vybrat místa, která budeme vizualizovat. Místa budeme volit v rastru. Tuto metodu jsme popsali i s optimalizací v sekci 6.1.

Pro ohodnocení míst použijeme metodu popsanou v sekci 5.2.

Samotná vizualizace probíhá stejně jako vizualizace zastávek, viz obrázek 6.2.



Obrázek 6.2: Vizualizace dostupnosti v rastru 100 x 100 z Malostranského náměstí, Kladna a Příbrami. Hodnoty v legendě vyjadřují dostupnost v sekundách.

6.3 Interaktivní vizualizace

Dostupnost můžeme vizualizovat v interaktivní mapě. Tím umožníme uživatelům nejen zobrazovat výsledky na mapě, ale také pohybovat se po mapě a zjišťovat tak dostupnost na konkrétních místech.

Vizualizaci zpřístupníme uživatelům skrze webovou aplikaci. Z toho důvodu jsme pro implementaci vizualizace zvolili jazyk JavaScript.

Výběr knihovny

Při implementaci této vizualizace budeme chtít využít existující knihovnu, abychom si ušetřili práci. Knihoven pro práci s interaktivní mapou existuje mnoho. Zvážíme následující možnosti.

1. Google map API. Google sice umožňuje využívat API po dobu prvního půl roku zdarma, obecně se ale jedná o placenou službu.

2. Mapquest. Placená knihovna, prvních 15 000 transakcí by mělo být zdarma.
3. OpenLayers. Open-source knihovna, zcela zdarma.

Preferovanou knihovnou je OpenLayers, neboť je jako jediná zdarma. Nenutí nás tedy vynucovat poplatky na uživatelích aplikace. Navíc nevynucuje žádné komerční závislosti pro případné další vývojáře naší aplikace.

OpenLayers

Knihovna OpenLayers používá jako výchozí projekci WGS84 (World Geodetic System). Stejná projekce je použita i pro geografická data popsaná ve formátu GTFS. To nám ušetří práci s případnými konverzemi projekcí.

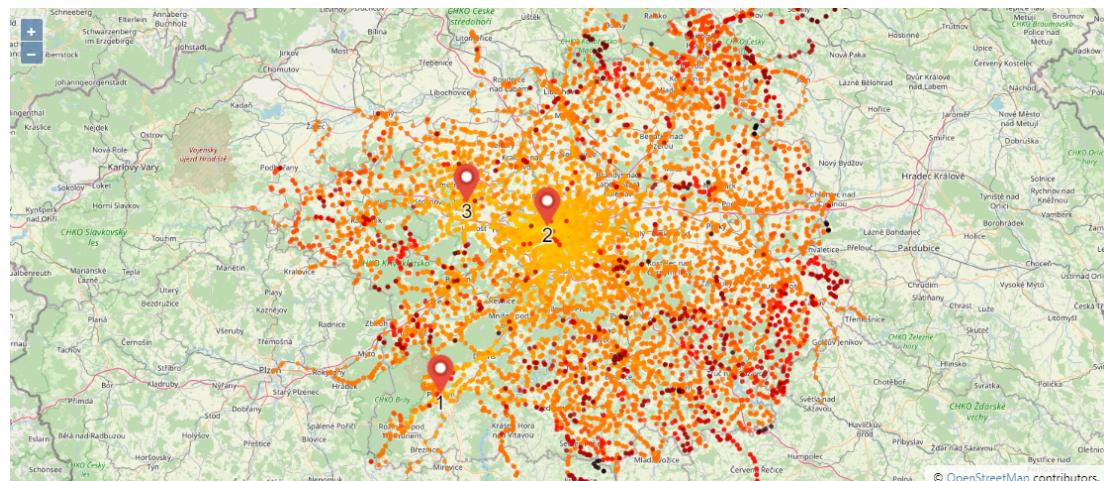
Umožňuje kreslení na podklad map z OSM (OpenStreetMap). Jedná se o volně dostupné mapy, pro nás tedy ideální volba.

Pracuje s vizualizačními vrstvami, není tedy problém přidat vrstvu obsahující vizualizované body a vrstvu obsahující uživatelem zadaná vstupní místa pro výpočet dostupnosti.

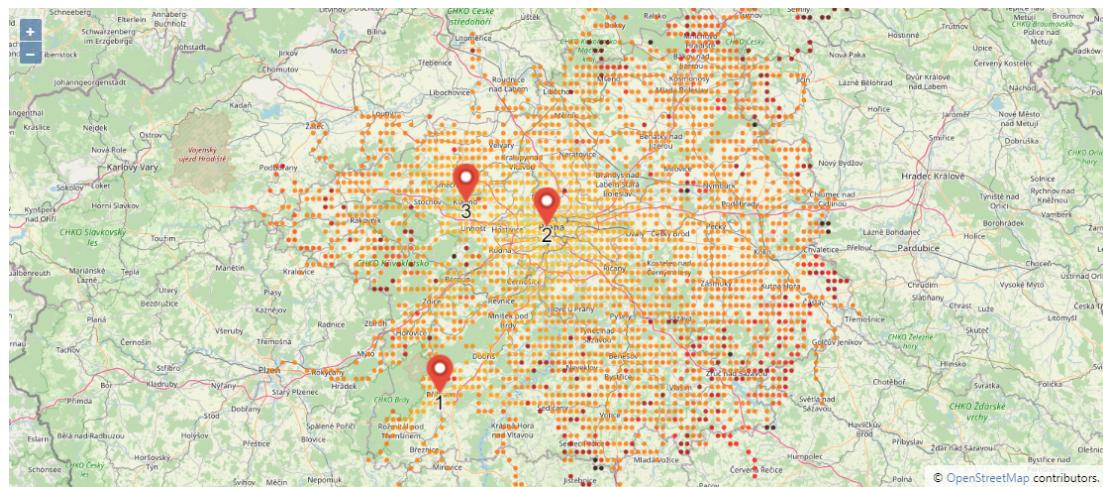
Umí pracovat s WebGL (Web Graphics Library), které umožňuje rychlejší vykreslování velkého množství bodů.

Výsledná vizualizace

Následující obrázky zobrazují interaktivní vizualizaci dostupností na zastávkách, viz obrázek 6.3 a vizualizaci dostupností na bodech v rastru, viz obrázek 6.4.



Obrázek 6.3: Interaktivní vizualizace dostupnosti na zastávkách z Malostranského náměstí, Kladna a Příbrami.



Obrázek 6.4: Interaktivní vizualizace dostupnosti v rastru 100 x 100 z Malostranského náměstí, Kladna a Příbrami.

6.4 Možné vylepšení

Spíše než vizualizace bodů by pro uživatele mohla být zajímavější možnost vizualizace pomocí jednolitého barevného filtru, který by překrýval mapu.

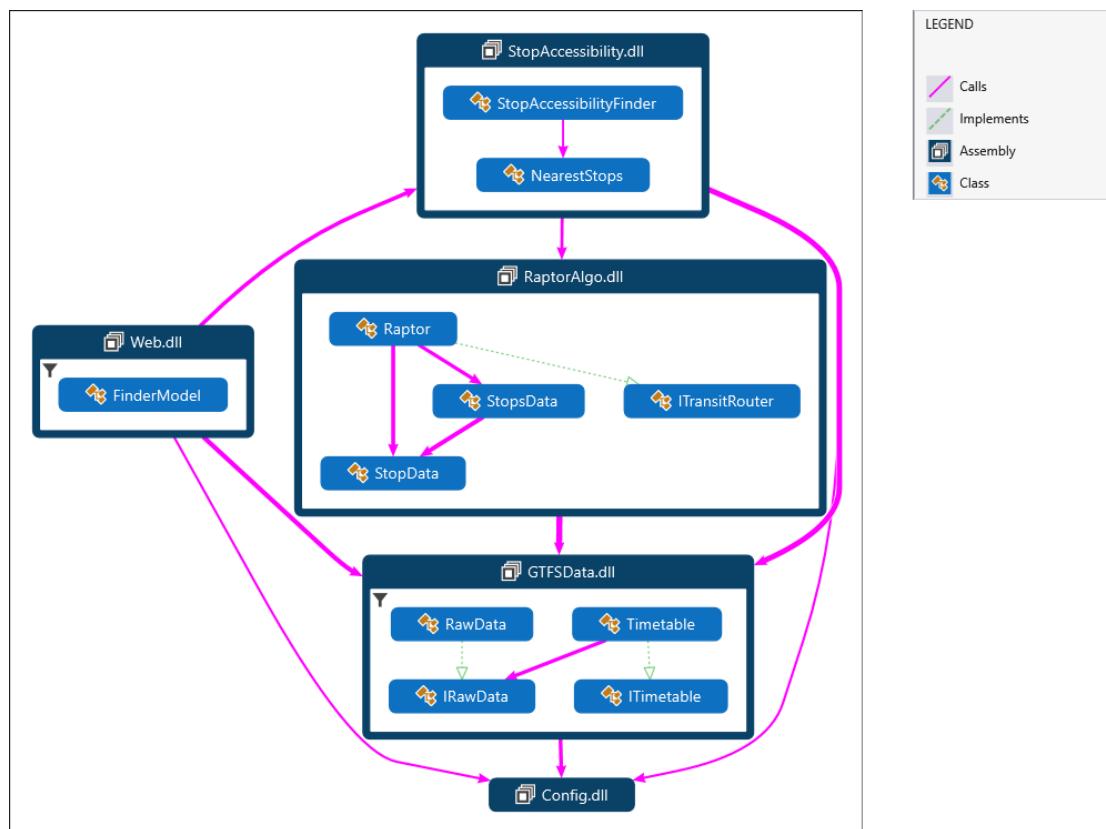
Základem takového filtru by byla dostupnost v bodech. Dostupnost na místech mezi body bychom mohli interpolovat pomocí dostupností na okolních bodech.

7. Implementace

Projekt jsme psali primárně s použitím jazyku C# pro verzi .NET Core 5. Součástí projektu je webová aplikace využívající framework ASP.NET, v této části se objevují kromě jazyku C# také jazyky Javascript, HTML a CSS.

Náš projekt je koncipovaný jako knihovna a webová aplikace. Knihovna zpracovává data jízdních řádů a vypočítává dostupnost. Webová aplikace používá knihovní část a umožňuje uživatelům zadávat vstupy, vizualizovat vypočtenou dostupnost a procházet vizualizace v interaktivní mapě.

Knihovní část aplikace je rozdělena na moduly, které vzájemně nezávisí na implementačních detailech a tím je umožněna jejich snadná výměna. Tyto moduly nazýváme GTFSData, RaptorAlgo, StopAccessibility a Config. Závislosti mezi moduly si můžeme prohlédnout na obrázku 7.1.



Obrázek 7.1: Moduly aplikace a jejich vzájemné propojení.

Knihovní část projektu testujeme pomocí Unit Testů, psaných nástrojem XUnit. Testy pokrývají všechna základní použití knihovny, doplněné o testy založené na chybách nalezených při vývoji.

7.1 Modul GTFSData

V tomto modulu načítáme data jízdních řádů a následně je zpracováváme. Součástí modulu jsou také pomocné funkce, které usnadňují práci s projekcí WGS84 a se zpracováním času.

Hlavní třídy modulu jsou:

- **RawData**, která načítá data ze souborů formátu GTFS.
- **Timetable**, která načítá data do datových struktur vhodných pro vyhledávání.

7.1.1 Třída RawData

V této třídě implementujeme myšlenky, popsané v kapitole 2. Třída obsahuje specifikaci dat, která budeme načítat do paměti ze souborů formátu GTFS. Formát GTFS jsme popsali již v sekci 2.2.

Pro snazší parsování GTFS dat ve formě CSV používáme knihovnu **CsvHelper**. Tato knihovna, mimo jiné, určuje formát ve kterém popisujeme specifikaci parsovaných dat.

Při parsování **arrival_time** a **departure_time** ze souboru **stop_times** převádíme časy na **TimeSpan**, který nám umožňuje ukládat kromě času i případné dny. Tímto způsobem ukládání dat se vyhneme problému s časy, které přesahují klasický 24 hodinový formát.

Datum parsujeme do struktury **DateTimeOffset**. UTC posun určujeme v pomocné třídě **TimeConverter**. Naše implementace určuje UTC posun dle hodnoty **UTCOffset** z konfiguračního souboru.

Možnost rozšíření o jiný formát dat

Implementaci třídy **RawData** můžeme nahradit, za podmínky dodržení specifikovaného rozhraní **IRawData**. To nám umožní pracovat s daty jízdních řádu v jiném formátu než je formát GTFS. Například bychom mohli pracovat s daty ve formátu JDF, popsaném v kapitole 2.

7.1.2 Třída Timetable

Tento třídou reprezentujeme datové struktury uchovávající data jízdních řádů ve formě vhodné pro vyhledávání. Potřebná data jsme popsali již v sekci 3.1.1. Součástí této třídy jsou také metody, pomocí nichž stavíme zmíněné datové struktury.

Třída **Timetable** závisí na datech jízdních řádu, které ji poskytneme skrze rozhraní popsané výše v sekci 7.1.1. Využívá návrhový vzor **Singleton**, který zajišťuje, že v aplikaci bude existovat jen jediná instance této třídy. To je velice užitečné, neboť v aplikaci pracujeme jen s jedněmi daty jízdních řádů a ty nechceme duplikovat.

Popis datových struktur

- Struktura **Transfer**

Tento strukturou popisujeme přestupy mezi zastávkami. Obsahuje čas přestupu a zdrojovou a cílovou zastávku.

- Struktura **Stop**

Strukturou **Stop** reprezentujeme zastávky z jízdních řádů. Obsahuje kolekci přestupů mezi zastávkami a seznam tras, na kterých se zastávky nachází.

Pro optimálnější hledání sousedních zastávek využíváme optimalizace popsané v sekci 5.5. Důsledkem této optimalizace je **Stop** potomkem **ISpatialData**.

- Struktura **StopTime**

Tato struktura obsahuje informace o příjezdu a odjezdu z odpovídající zastávky.

- Struktura **TripWithDate**

Pomocí struktury **TripWithDate** reprezentujeme jízdu jedoucí v konkrétním datu. Každá jízda obsahuje kolekci **StopTime**.

Jelikož každá jízda může jet v několika dnech, tedy i několika datech, vedl by tento přístup k redundantnímu ukládání kolekcí **StopTime**.

Abychom tomu předešli, vytvořili jsme pomocnou strukturu **Trip**, která obsahuje informace sdílené mezi stejnými jízdami, včetně kolekce **StopTime**. Samotná struktura **TripWithDate** udržuje referenci na Trip a přidává k ní datum.

- Struktura **Route**

Tento strukturou reprezentujeme trasu ve významu popsaném v kapitole 3. Obsahuje množinu jízd, jedoucích po stejných zastávkách a zastávky samotné.

Jak jsme již zmínili v sekci 3.1.1, je nutné, abychom obsažené jízdy udržovali uspořádané. K tomu slouží metoda **SortTrips**.

Fáze stavby datových struktur

- Inicializace

Při inicializaci nastavujeme počáteční datum a dobu platnosti dat z jízdních řádů. Doba platnosti je vždy omezená a dá se nastavit skrze konfigurační soubor.

Po přidání zastávek dále inicializujeme **StopPositionLimits** a stavíme **stopsInRTree**.

Do vlastnosti **StopPositionLimits** ukládáme souřadnice obdélníku, který ohraničuje všechny zastávky. Toho využijeme později při vizualizaci.

Strukturu **stopsInRTree** stavíme pro optimálnější vyhledávání, viz sekce 5.5.

- Přidání zastávek — metoda **AddStops**

V této fázi jen ukládáme zastávky z dat jízdních řádů.

- Přidání tras — metoda **AddRoutes**

Během přidávání tras přidáváme také jízdy, které patří pod jednotlivé trasy.

Jízdy vytváříme spojením dat z různých souborů formátu GTFS pomocí klíčů. Konkrétně spojujeme **Calendar**, **CalendatDate** a **Trips** přes klíč **service_id** a výsledek dále spojujeme se **StopTimes** přes klíč **trip_id**. Vytvořené jízdy přidáváme do tras dle jednoznačného identifikátoru.

Každá vytvořená jízda obsahuje **route_id**. To samotné však k jednoznačné identifikaci tras nestacha, neboť definice GTFS trasy se liší od námi definované tras. Jak jsme již zmíňovali v kapitole 1.1, GTFS trasa může obsahovat jízdy, které jezdí přes různé zastávky.

Abychom přiřadili jízdu pod správnou trasu, vygenerujeme si jednoznačný identifikátor trasy pomocí metody **GetRouteKey**, která jej generuje dle **route_id** a **id** každé ze zastávek na trase.

Tato metoda pro každou trasu navíc vytváří **RouteStops**. Tedy seznam všech zastávek, přes které daná trasa vede.

- Přidání StopRoutes

Metoda **AddStopRoutes** přidává pro každou zastávku seznam všech tras, které přes zastávku jezdí. Toto přidání nelze provést už při volání metody **AddStops**, neboť v tu chvíli ještě nemáme uložené trasy.

- Odebrání nepoužitých zastávek

Ukázalo se, že přinejmenším v datech společnosti PID existují zastávky, přes které nevedou žádné trasy. Takové zastávky chceme odstranit, neboť zbytečně zpomalují výpočet. Samotné odstranění obstarává metoda **RemoveUnusedStops**.

- Generování přestupů

Ke generování přestupů slouží metoda **GenerateTransfers**. Vzdálenost přestupů jsme omezili konstantou **MAX_TRANSFER_DISTANCE** a rychlosť přestupu určujeme konstantou **WALKING_SPEED**. Obě konstanty jsou nastavitelné v konfiguračním souboru.

Pro hledání blízkých zastávek, mezi kterými definujeme přestupy, využíváme optimalizaci, popsanou v sekci 5.5. Bez využití optimalizace jsme nuceni porovnávat všechny dvojice zastávek a to je velice neefektivní. Metoda **GenerateTransfers** je sice volaná maximálně jednou za běh programu, generování přestupů je však ze všech fází výpočetně nejnáročnější a zvláště pro velké datasety je tato optimalizace nepostradatelná.

Vzdálenost mezi zastávkami, pro zjednodušení, určujeme jako vzdálenost vzdušnou čarou. Možné zlepšení navrhujeme v sekci 7.1.3.

Pro správné fungování vyhledávače je nutné, aby byly přestupy tranzitivní. Toho jsme docílili tak, že se na zastávky a přestupy mezi nimi díváme jako na vrcholy a hrany grafu. V takovém grafu následně hledáme, pomocí DFS (Depth-first search), komponenty silné souvislosti. Každá taková komponenta tvoří množinu zastávek, mezi kterými existují tranzitivní přestupy.

Ukládání dat

Stavba datových struktur, zejména generování přestupů, zabírá netriviální dobu při spouštění programu. Abychom nemuseli struktury opakovaně stavět při každém startu programu, bylo by vhodné si je někam uložit.

1. Databáze

Jednou z možností datového úložiště je databáze. Při vyhledávání spojů však potřebujeme pracovat se všemi daty, dotazovat se vždy databáze by bylo příliš nepraktické. Navíc se dá předpokládat, že data jízdních řádů se vejdou do paměti.

Databázi bychom mohli používat jen pro načtení dat při startu aplikace. Pak ale plně nevyužíváme výhody poskytované databází, mimo jiné proto, že data chceme pouze číst, ne do nich zapisovat. Samotná databáze nám tedy do aplikace zavádí zbytečnou složitost.

2. Serializace

Vhodnou alternativou k databázi je serializace do souboru. Datové struktury postavíme jen jednou, při aktualizaci dat jízdních řádů a serializujeme je. Následující spuštění aplikace deserializuje datové struktury a tím urychlí spuštění aplikace.

V naší aplikaci jsme pro ukládání dat použili serializaci do souboru. Data serializujeme do formátu JSON (JavaScript Object Notation). Jako serializační knihovnu jsme použili knihovnu **JSON.NET**, která oproti standardní serializaci poskytuje mnoho výhod. Dokáže například serializovat data s cyklickými referencemi.

Samotná deserializace ze souboru však není příliš rychlá. Konkrétně, stavba **StopRoutes** je časově méně náročná, než jejich deserializace. Stavba **RouteStops** by mohla být také rychlejší než deserializace, museli bychom ji však nejprve oddělit od metody **AddStopRoutes**.

Při deserializaci velkých datasetů může docházet k překročení maximální hloubky zanoření dat, serializovaných ve formátu JSON. Tu je však možné nastavit v deserializačním konstruktoru.

Práce s kalendářem

Naše prvotní implementace pracovala s časem jako s počtem sekund ve dni, stejně jako je to popsáno v článku (Delling a kol., 2015). To se však ukázalo jako velice nepraktické. Nejen, že je reprezentace času jako čísla nevhodná. Tato reprezentace navíc neumožňovala rozlišení dnů ve kterých spoje jezdí, ani práci se spoji jedoucími přes půlnoc.

Rozhodli jsme se tedy čas reprezentovat ve strukturách k tomu určených. Těmi jsou v jazyku C# **DateTimeOffset** a **TimeSpan**. Tato reprezentace umožnila rozlišení jednotlivých dnů a dokonce i práci s výjimkami v jízdních řádech. Vyřešila také spoje jedoucí přes půlnoc, neboť k takovému spoji lze jednoduše přičíst den.

Tento přístup však může mít i nevýhody. Jako hlavní nevýhodu můžeme chápout složitější zadávání vstupu uživatelem v případě, že uživatel pracuje přímo s touto reprezentací, což se v naší webové aplikaci děje.

Tato nevýhoda by se dala vyřešit abstrakcí uživatele od naší implementace. Problém ale je, jak by tato abstrakce měla vypadat. Pokud bychom nechali uživatele pracovat jen s jednotlivými dny, není jasné, zdali tyto dny mají reprezentovat běžný den, či sváteční den. Navíc bychom, například pro prázdniny, museli pracovat se staršími daty, neboť jízdní řády v takovém období bývají značně pozměněné.

Cenou za abstrakci by tedy byla práce s nepřesnými či neaktuálními daty jízdních řádů. To však nechceme. Necháme tedy uživatele pracovat s konkrétními daty.

7.1.3 Možné vylepšení

Lepší generování přestupů

V této chvíli, během generování přestupů, měříme vzdálenost mezi zastávkami vzdušnou čarou. Tato vzdálenost, se však může velice odlišovat od vzdálenosti pěší chůzí. Příkladem mohou být například zastávky, které jsou sice poměrně blízko, jenže mezi nimi teče řeka. V takovém případě se vzdálenost vzdušnou čarou může od pěší vzdálenosti lišit dramaticky.

V ideálním případě bychom k měření pěší vzdálenosti využili existující navigaci. Přímo pro mapy OSM bychom mohli použít některou z navigací zmíněných na stránce https://wiki.openstreetmap.org/wiki/Routing/offline_routers.

Optimálnější uložení dat

Datové struktury ukládající data ve třídě **timetable**, jsou psané v objektově orientovaném stylu, neboť tento styl je dobře čitelný a preferovaný v prostředí jazyku C#. Nevýhodou je však špatná datová lokalita, což vede k více cache missům a zpomaluje tak vyhledávání algoritmu RAPTOR.

Možným řešením je reprezentovat data v datových strukturách přesně tak, jak jsou popsány v článku Delling a kol. (2015, Appendix A).

7.2 Modul RaptorAlgo

Tento modul implementuje algoritmus RAPTOR, viz kapitola 3, který používáme k vyhledávání nejrychlejší cesty ze zadané zastávky, v daný čas a den, na ostatní zastávky. Implementace algoritmu nám navíc umožňuje zadávat více vstupních zastávek, čímž implementujeme zobecnění zastávky na místo, popsané v sekci 5.1.

7.2.1 Existující implementace

Před samotnou implementací tohoto modulu jsme prozkoumali alternativní možnosti, které by nám mohli práci usnadnit. Tyto možnosti lze zařadit do dvou kategorií. Knihovny, umožňující vyhledávání v jízdních rádech a již existující implementace algoritmu RAPTOR.

Knihovny

Mezi námi zkoumané knihovny patří **OpenTripPlanner**¹ a **Itinero**².

OpenTripPlanner je knihovna psaná v jazyku Java, což je jazyk, který spíše nepreferujeme pro tvorbu této aplikace.

Itinero je knihovna pro jazyk C#.

Tyto knihovny však neřeší přímo náš problém. Abychom knihovny využili, museli bychom buď přepsat jejich část, nebo adaptovat naše řešení na možnosti knihoven.

Zmíněné knihovny řeší široké spektrum problémů a jsou tedy velice komplexní. Z toho důvodu jsou jakékoli jejich úpravy značně složité. Nová implementace algoritmu RAPTOR se zdá být snazší, než přepisování části existující knihovny.

Adaptace našeho problému na možnosti knihoven se nezdá být vůbec snadná.

V případě, že bychom přece jen byli schopni adaptaci zprovoznit, by toto řešení stále nebylo ideální. Naše poměrně malá knihovna by totiž závisela na mnohem větší knihovně, jejíž funkcionality bychom využívali jen ve velmi malé míře.

Existující implementace algoritmu RAPTOR

Existující implementace by nám kromě ušetření času mohly poskytnout také optimalizované řešení, což by zaručilo rychlý běh algoritmu. Mezi nejrychlejší implementace algoritmu by patřily ty, napsané v jazyku C/C++. Takové implementace bychom z naší knihovny, psané v jazyku C#, mohli volat pomocí technologie **P/Invoke**.

1. Implementace³ psaná v jazyku C++. Tato implementace s sebou bohužel nenese dokumentaci. Samotný kód se zdá být dosti nestrukturovaný, možná i z optimalizačních důvodů. To je však důvod, proč se provedení námi navrhovaných změn algoritmu zdá být v této implementaci příliš pracné.

¹Dostupná na githubu, viz <https://github.com/opentripplanner/OpenTripPlanner>

²Více informací lze dohledat na stránce <http://www.itinero.tech/>

³Implementace je dostupná na githubu, viz <https://github.com/lviennot/hl-csa-raptor>

2. Další dostupná implementace⁴, psaná v jazyku C++, bohužel také neobsahuje dokumentaci. Zdá se však, že je tato implementace trochu lépe strukturovaná. Při spuštění jsme však narazili na problém, že implementace předpokládá typované hodnoty v GTFS datech. Formát GTFS však žádné typy nevynucuje. Například pro **trip_id** knihovna předpokládá numerický typ, zatímco data společnosti PID jsou pod položkou **trip_id** uloženy v řetězovém typu. Z tohoto důvodu se nám ani nepodařilo spustit implementaci na našich datech.
3. Poslední zkoumaná implementace⁵ je psaná v jazyku Java. Tato implementace je dobře dokumentovaná a její kód je přehledný. Tuto knihovnu jsme nevyužili, neboť naší aplikaci chceme psát v jazyku C#. Posloužila však jako inspirace pro naší implementaci a usnadnila nám tak práci.

Ze zmíněných implementací není žádná zcela vyhovující. Z toho důvodu jsme se rozhodli implementovat si algoritmus RAPTOR sami.

7.2.2 Třída Raptor

Tento třídou reprezentujeme vyhledávač cest. Velká část implementace jen realizuje myšlenky popsané v kapitole 3.

Názvosloví pro práci s časem

V této třídě používáme často následující názvosloví pro práci s různými časy.

- **Walk time** — doba trvání pěší cesty z daného místa na zastávku.
- **Travel time** — doba strávená na cestě jedoucím spojem.
- **Departure time** — čas výjezdu spoje ze zastávky.
- **Arrival time** — čas příjezdu spoje na zastávku.
- **Start time** — čas, od kterého začínáme vyhledávání.
- **Accessibility** — dostupnost, neboli doba která uplynula mezi nejdřívějším **arrival time** a **start time**, viz sekce 1.1.

Zajímavosti lišící se od implementace z článku

Vytvářením jízd pro každé datum kdy spoj jede, viz **TripWithDate** popsaný v sekci 7.1.2, vzniká velký počet jízd. Tento nárůst počtu jízd může zpomalit běh metody **GetEarliestTripFromStop**, která vyhledává nejdřívější jízdu, na kterou můžeme z dané zastávky nastoupit. Jednoduchou optimalizací je použití binárního vyhledávání pro nalezení hledané jízdy.

Metoda **Init** v naší implementaci navíc označuje i sousedy vstupní zastávky. Bez označení sousedů by algoritmus nenašel přestupy ze vstupní zastávky na

⁴Implementace je dostupná na githubu, viz <https://github.com/ducminh-phan/RAPTOR>

⁵Implementace je dostupná na gitlabu, viz <https://gitlab.fel.cvut.cz/kasnezde/raptor>

zastávky v jejím okolí. Důvodem je to, že v první fázi algoritmu použijeme označenou vstupní zastávku k označení tras a zrušíme existující označení zastávek. Ve třetí fázi, kde aplikujeme přestupy z označených zastávek, už není vstupní zastávka označena, tudíž přestupy z ní nejsou nikdy využity.

Třída navíc obsahuje metodu **GetTravelTimeByStops**, která zpřístupňuje dostupnosti na zastávkách, které jsme vypočítali během vyhledávání. Tato metoda slouží jako propojení tohoto modulu s modulem **StopAccessibility**, popsaným v sekci 7.3.

Možné zlepšení

Tento modul nepředpokládá běžné využívání hodnoty **TotalRounds**, která omezuje počet přestupů v nalezených cestách. Pro ohodnocení dostupnosti na všech dosažitelných místech sice tohoto omezení nevyužíváme, ale některé uživatele knihovny by vedle dostupnosti mohli zajímat i počet přestupů. Tato změna by vyžadovala úpravu API tohoto modulu.

7.2.3 Třída StopData

Tuto třídou reprezentujeme vypočtená data asociovaná se zastávkou. Jedná se zejména o **multilabel arrivalTimes**, který pro každé kolo obsahuje nejdřívější příjezdové časy. Dále třída obsahuje **transfer**, **previousStop**, **trip**, která slouží k vypisování cesty, viz sekce 7.2.4.

Multilabel arrivalTimes rozšiřujeme líně, tedy až v případě nutnosti. Příjezdové časy **multilabelu** pro neexistující a nová kola vrací konstantu **UNREACHABLE**, která reprezentuje hodnotu nekonečno, jak jsme popsali již v inicializaci algoritmu RAPTOR v sekci 3.1.2.

Všechna vypočtená **StopData**, pro dosažitelné zastávky, uchováváme ve třídě **StopsData**, jenž slouží jako wrapper nad dictionary.

7.2.4 Třída Journey

Journey jsme v počáteční fázi vývoje používali k vypisování cest mezi zadáným vstupním a výstupním místem, kterou RAPTOR nalezl.

S touto třídou je spojena logika tříd **Raptor** a **StopData**. Když ve třídě **Raptor** dojde během vyhledávání k aktualizaci **arrival_time**, voláme na instanci **StopData**, asociované s danou zastávkou, metodu **ArriveByTransfer** či **ArriveByTrip**. V těchto metodách dochází k uložení úseku cesty vedoucí na danou zastávku.

Využití

Tuto funkcionality v naší aplikaci již nevyužíváme. Třídu **Journey** jsme však zachovali, neboť ji hojně využíváme při testování třídy **Raptor**.

Pokud by však uživatele naší knihovny zajímalá cesta, kterou se dostanou ze zadaných míst na některé konkrétní místo. Můžeme cestu poměrně snadno vyhledat, využitím právě této třídy.

7.2.5 Možné optimalizace

Naše implementace algoritmu RAPTOR je napsaná v jazyku C#. Pro optimálnější běh algoritmu by však bylo vhodnější zvolit jazyk C++. Implementaci psanou v jazyku C++ bychom mohli z naší aplikace volat pomocí technologie P/Invoke, jak už jsme zmiňovali v sekci 7.2.1.

Algoritmus lze také optimalizovat využitím paralelizace, viz kapitola 3.1.3.

7.3 Modul StopAccessibility

V tomto modulu zobecňujeme vyhledávání implementované ve třídě **Raptor** podle metod popsaných v kapitole 5. Konkrétně umožňujeme vyhledávání dostupnosti na libovolném výstupním místě, viz sekce 5.2, vyhledávání z více vstupních míst, viz sekce 5.3 a výpočet dostupnosti v intervalu, viz sekce 5.4.

Uvnitř tohoto modulu stále pracujeme s časovým názvoslovím, které jsme popsali již v sekci 7.2.2.

7.3.1 Třída NearestStops

Třída **NearestStops** obsahuje pomocnou metodu **GetStopsWithWalkTime**. Touto metodou nalezneme sousední zastávky do maximální vzdálenosti, dané konstantou **DISTANCE_LIMIT**, od zadaných souřadnic. Konstantu **DISTANCE_LIMIT** načítáme z konfiguračního souboru. Pro každou z nalezených sousedních zastávek navíc počítáme čas který potřebujeme, abychom na zastávku došli pěší chůzí ze zadaných souřadnic. Pro efektivní vyhledávání sousedních zastávek využíváme optimalizaci zmíněnou v sekci 5.5.

7.3.2 Třída StopAccessibilityFinder

Základními interními metodami této třídy jsou metody **GetAvgTravelTimeByStop** a **CalcAccessibility**.

Metoda GetAvgTravelTimeByStop

Za pomoci této metody agregujeme dostupnosti z více zdrojů. V případě, že jsou specifikovány váhy, agregujeme dostupnosti váženým průměrem. V případě že váhy nejsou specifikovány, použijeme standardní průměr.

Uvnitř metody nejprve sčítáme dostupnosti na jednotlivých zastávkách a odstraňujeme zastávky, které nejsou dostupné z některého ze vstupních míst. Následně počítáme průměr z dostupností.

Metoda CalcAccessibility

V metodě **CalcAccessibility** postupně procházíme vstupní zastávky. Pro každou vstupní zastávku nalezneme, s pomocí metody **GetStopsWithWalkTime**, sousední zastávky, které dále využíváme k výpočtu dostupností naší upravenou verzí algoritmu RAPTOR.

Každá vstupní zastávka může mít přidruženo větší množství počátečních dat s časy. Dostupnosti pro každé datum a čas počítáme zvlášť a následně je průměrujeme metodou **GetAvgTravelTimeByStop**.

V případě, že je metoda **CalcAccessibility** volaná s parametry pro interval, počítáme navíc dostupnost pro každý čas v intervalu. Tento postup jsme popsali již v sekci 5.4. Výsledky z intervalu se opět průměrují metodou **GetAvgTravelTimeByStop**.

Zobecnění vyhledávání na více vstupních míst

Toto zobecnění jsme popsali v sekci 5.3 a zajišťují jej metody **GetAvgAccessByStop** a **GetStatisticalAvgAccessByStop**. Obě metody využívají metodu **CalcAccessibility** pro ohodnocení jednotlivých zastávek a výsledné dostupnosti průměrují metodou **GetAvgTravelTimeByStop**.

Zobecnění vyhledávání na libovolné výstupní místo

Toto zobecnění jsme popsali již v sekci 5.2. Zobecnění je implementováno metodami **GetAccessForCoords** a **GetAccessForCoordNbors**.

Metoda **GetAccessForCoords** pro zadané souřadnice nejprve nalezne sousední zastávky a čas potřebný pro jejich dosažení chůzí pomocí metody **GetStopsWithWalkTime** a následně volá metodu **GetAccessForCoordNbors**.

Metoda **GetAccessForCoordNbors** je také součástí API, neboť pro vyhodnocení dostupnosti na velkém množství míst pro nás opakované hledání sousedních zastávek může známenat výrazné zpomalení. Pro velké množství míst si můžeme sousední zastávky předpočítat a následně využít tuto metodu. Využití této metody jsme popsali již v sekci 6.1.

V samotné metodě **GetAccessForCoordNbors** nejprve vypočteme dostupnosti pomocí metody **CalcAccessibility**. Následně procházíme všechny vstupní zastávky a pro každou z nich nalezneme zastávku sousedící se zadanými souřadnicemi, skrize kterou se na zadané souřadnice dostaneme nejrychleji. Dostupnost takto nalezené zastávky nám vyjadřuje dostupnost ze vstupní zastávky na zadané souřadnice. Ohodnocením místa na zadaných souřadnicích je vážený průměr dostupností ze všech vstupních zastávek.

7.3.3 Možná zlepšení

Lepší výpočet vzdálenosti

Aktuálně počítáme vzdálenost jako vzdálenost vzdušnou čarou. To je dosti nepřesné. Na stejný problém jsme narazili při generování přestupů a navrhované řešení jsme popsali v sekci 7.1.3.

Redundance u přidružení více dat s časy ke vstupní zastávce

Každá vstupní zastávka může mít přidružených více počátečních dat s časy. V tuto chvíli počítáme dostupnost pro každý z časů zvlášť. To však může být velice neefektivní.

V případě, že by uživatel zadal stejné časy pro po sobě jdoucí všední dny, například pondělí a úterý, může se stát že se pro tyto dny jízdní řády vůbec neliší. Počítáme tedy dvakrát to samé.

Takových případů může být více. Prozatím nenavrhujeme žádné řešení, jen poukazujeme na možnou redundanci.

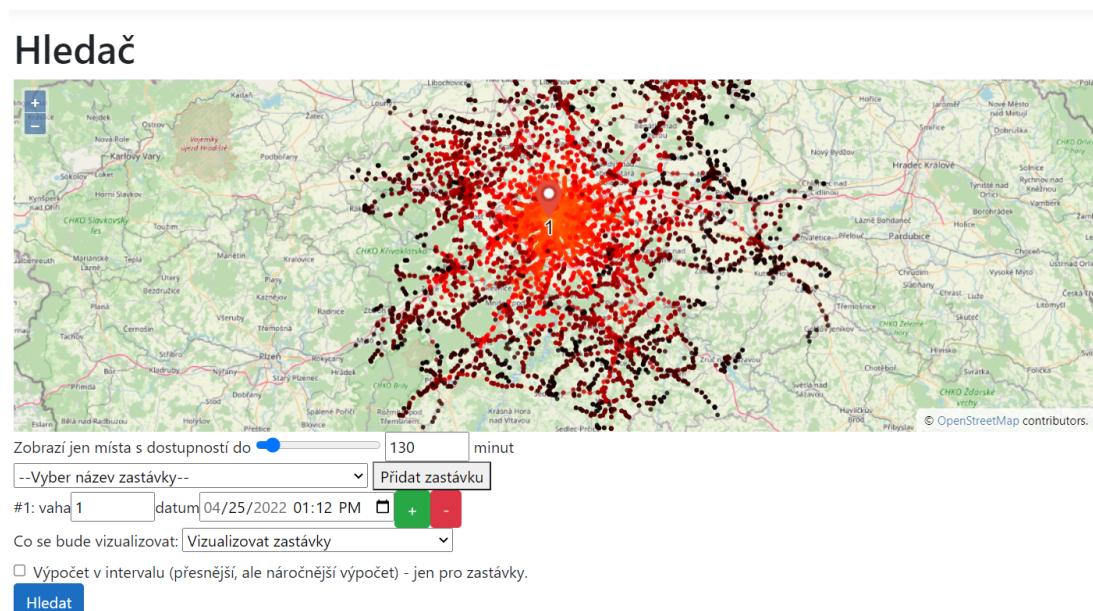
7.4 Modul Web

V tomto modulu implementujeme webovou aplikaci, která zpřístupňuje naši knihovnu, pro výpočet dostupnosti, běžným uživatelům. Součástí tohoto modulu je také implementace interaktivní vizualizace, popsané dříve v sekci 6.3.

7.4.1 Části webové aplikace

Před samotným popisem modulu si popíšeme části webové aplikace, které budeme implementovat.

Hlavní stránkou webové aplikace je stránka **Finder**, skrze níž mohou uživatelé vyhledávat dostupnost, viz obrázek 7.2.



Obrázek 7.2: Stránka Finder.

Značky

Přidání značky reprezentující vstupní místo je možné dvěma způsoby. Kliknutím do mapy, nebo výběrem zastávky ze seznamu všech zastávek.

Vybrané vstupní místo lze odstranit kliknutím na značku asociovanou k tomuto místu.

Značky lze po mapě přesouvat myší.

Formulář se vstupními místy

Formulář se vstupními místy generujeme dynamicky. Pro každé nové vstupní místo přidáváme položku.

S každým vstupním místem asociovujeme váhu a skupinu dat s časy. Váhu používáme k výpočtu váženého průměru. Data s časy určují dobu, kdy ze zastávky chceme vyjíždět. Data s časy lze přidávat tlačítkem + a odebírat tlačítkem -.

Vizualizovat můžeme zastávky, body nebo obojí.

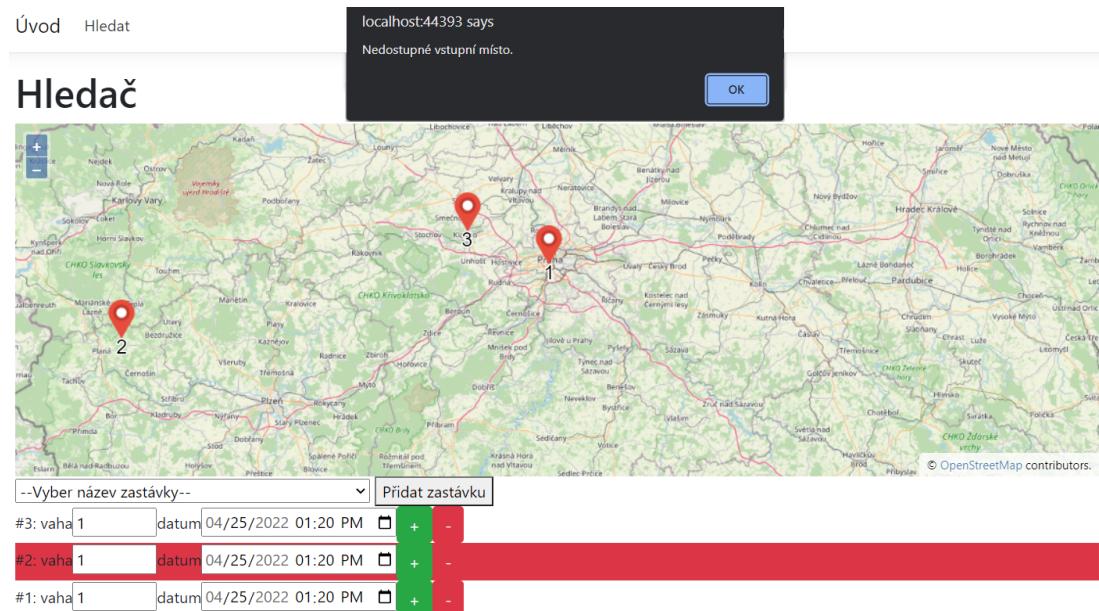
Při vizualizaci zastávek vizualizujeme všechny zastávky, které dostupné ze vstupních míst.

Při vizualizaci bodů vytváříme čtvercový rastr o rozlišení daném konstantou **VisualisedRasterPointsResolution**, viz sekce 6.1. Mezi vizualizovanými body jsou jen ty, které jsou v maximální vzdálenosti, dané konstantou **NearestStopsDistanceInMeters**, od nejbližší dostupné zastávky. Konstanty pochází z konfiguračního souboru.

Uživatel si může zažádat o výpočet v intervalu, viz metoda **CalcAccessibility** v sekci 7.3.2. Takový výpočet je náročnější než klasický výpočet, jeho výsledky jsou však přesnější.

Formulář lze odeslat ke zpracování kliknutím na tlačítko **Hledat**. Po vypočtení dostupnosti dojde k vizualizaci bodů na mapě.

Pokud byla některá značka na místě vzdáleném o více než **NearestStopsDistanceInMeters** od nejbližší zastávky, dojde k vypsání chyby a označení vstupního místa způsobujícího chybu, viz obrázek 7.3.



Obrázek 7.3: Hlášení chyby při zadání nedostupného místa.

Výsledná dostupnost

Dostupnost na vizualizovaných místech barevně rozlišujeme. Tmavě jsou označena méně dostupná místa a světle jsou označena dostupnější místa.

Pro snazší hledání dostupných míst jsme zavedli posuvník, kterým lze omezit maximální vizualizovanou dostupnost. Barvy na vizualizovaných místech překreslujeme v závislosti na aktuálně nastavené maximální dostupnosti.

Nejdostupnější místa vypisujeme v tabulce pod formulářem.

7.4.2 Backend

Náš backend je postavený nad webovým frameworkem ASP.NET Core. Framework ASP.NET jsme zvolili, neboť naše knihovní část je napsaná v jazyku C# a ASP.NET je velice populárním frameworkem pro tento programovací jazyk.

Volba modelu

Framework ASP.NET umožňuje vytvářet webové aplikace ve dvou modelech, **ASP.NET MVC** a **Razor Pages**.

- **ASP.NET MVC** je postavený na návrhovém vzoru Model–view–controller, který odděluje UI, data a logiku. Použití tohoto návrhového vzoru sice zvyšuje flexibilitu, cenou je však vyšší složitost. Důsledkem těchto vlastností je tento model vhodný spíše pro větší aplikace.
- Model **Razor Pages** je jednodušší, odděluje pouze UI a logiku.

My jsme zvolili model **Razor Pages**, neboť naše aplikace není příliš rozsáhlá a MVC by nám do aplikace zanášelo zbytečnou složitost.

Zpřístupnění knihovny jako Dependency Injection služby

Abychom mohli využívat námi implementovanou knihovnu pro výpočet dostupnosti, musíme ji nejprve zpřístupnit pro webovou aplikaci.

Toho docílíme načtením potřebných modulů jako služby. Načítání služby obstarává metoda **ConfigureServices** uvnitř třídy **Startup**. Každá služba má přiřazenou životnost, my používáme typy životnosti **singleton** a **scoped**.

- **Singleton** — existuje jedna instance pro všechny webové dotazy.
- **Scoped** — existuje nová instance pro každý webový dotaz.

Načtené služby jsou nám následně zpřístupněny za pomocí **Dependency Injection** skrze konstruktor.

Stránka Finder

Finder je hlavní stránka naší webové aplikace.

Metodu **OnPost** využíváme ke zpracování vstupu, zadанého uživatelem do formuláře.

Jelikož počet vstupních zástávek zadaných uživatelem není předem známý, potřebujeme data získávat z dynamicky rozšiřitelného formuláře. To bohužel znemožňuje použití **bindování**, které ASP.NET poskytuje pro uložení uživatelského vstupu přímo do atributů. Místo toho použijeme pro přenos dat formát **JSON** (JavaScript object notation).

Pro vizualizaci bodů potřebujeme znát sousedy těchto bodů. Tyto sousedy si můžeme vyhledat jednou a pracovat s nimi v dalších výpočtech, viz sekce 6.1. Toto je implementováno ve třídě **PointsWithNeighbors**.

Třída DataUpdater

Třída **DataUpdater** zajišťuje aktualizaci dat jízdních řádů.

Metoda **GetTimetable** poskytuje přístup k instanci třídy **Timetable**, popsané v sekci 7.1.2.

K aktualizaci dat dojde v případě, že je nastavený **ShouldUpdate** v konfiguračním souboru nebo pokud chybí serializace dat jízdních řádů.

Během aktualizace stahujeme zazipovaná data z adresy dané konstantou **GTFSSourceURI** a následně je extrahuje do složky na cestě dané konstantou **PathToGTFSFolder**. Konstanty pochází z konfiguračního souboru.

Možné zlepšení

Pokud bychom chtěli kromě webové aplikace vytvářet také mobilní či desktopovou aplikaci, mohli bychom využít ASP.NET Web API, které umožňuje vytváření **REST API**. Tento přístup jsme nevyužili, neboť bychom vedle samotného Web API museli vytvářet ještě samotnou webovou aplikaci a to by do naší aplikace zanášelo další složitost.

7.4.3 Frontend

Jako programovací jazyk pro frontend jsme zvolili populární **JavaScript**. Interaktivní mapu vytváříme s pomocí knihovny **OpenLayers**. Ke stylování HTML prvků používáme knihovnu **Bootstrap 4**.

Všechn JavaScriptový kód pro stránku **Finder** se nachází v souboru **map.js**. Vedle pomocných funkcí a konstant jsou v souboru obsaženy dvě hlavní třídy, **Form** a **OLMap**.

Třída Form

Tato třída obsahuje metody pro práci s formulářem, do kterého uživatel zadává vstupní místa.

Formulář je dynamický, díky čemuž můžeme za běhu přidávat vstupní místa, případně data s časy asociované se vstupními místy. To zajišťují metody **addField** a **removeField**.

Data z formuláře odesíláme na server ve formátu **JSON**. Zpracování dat a jejich **serializaci** obstarává metoda **submit**. Odpověď ze serveru je také zpracovaná touto metodou. V případě, že uživatel zadal nedosažitelné vstupní místo, volá se metoda **unreachableTargetHandler**. Pokud vše proběhlo v pořádku, vrátí se nám ze serveru dostupnost pro všechny zastávky či místa, s tou dálé pracujeme v metodě **handleResponse**.

Framework ASP.NET obsahuje ochranu proti CSRF (Cross-Site Request Forgery) útokům pomocí tokenu **RequestVerificationToken** vloženého do formuláře jako hidden field. Abychom umožnili posílání dat z JavaScriptu na server, musíme s nimi posílat také tento token.

Třída OLMap

V této třídě vytváříme interaktivní mapu pomocí knihovny **OpenLayers**.

V konstruktoru skládáme mapu z vrstev. **TileLayer** obsahuje mapy z OSM. **MarkerLayer** slouží pro zobrazení značek, které uživatel umísťuje do mapy. **VisualizationLayer** slouží k zobrazení bodů s barvou v závislosti na jejich dostupnosti.

V konstruktoru dále nastavujeme funkcionality potřebnou pro přesouvání značek umístěných v mapě. Logiku pro přesouvání značek zajišťují metody **updateFormOnMarkerMovement** a **changeCursorOnMarkerMovement**.

Třída dále obsahuje metodu **addOrRemoveMarker**, která reaguje na kliknutí do mapy a umožňuje tak přidávání a odebírání značek.

Možné vylepšení

Přestože naše knihovna dokáže vypočítat dostupnost pro zcela libovolné místo, naše webová aplikace to uživatelům neumožňuje. Webová aplikace umožňuje jen vyhodnocení dostupnosti v předem daných bodech a na zastávkách.

Pro uživatele by mohlo být zajímavé řešení, které by zobrazovalo dostupnost pro místo na které ukazuje kurzor myši.

Pokud bychom poslali dotaz pro výpočet dostupnosti při každém pohybu myši, nejspíše by došlo snadno k zahlcení serveru. Ideální by bylo provádět tento výpočet u klienta. Problém je však v tom, že kód počítající dostupnost je psaný v C# na straně serveru.

Možné řešení je přepsat kód do JavaScriptu, tím ale kód duplikujeme a tomu se chceme spíše vyhnout.

Lepším řešením by bylo využít framework **Blazor**, který umožňuje spouštění C# kódu v klientském prohlížeči skrze technologii **WebAssembly**.

7.5 Modul Config

Tento modul zpřístupňuje data potřebná ke konfiguraci naší aplikace.

Konfigurační soubor

Konfigurační soubor **appsettings.json** obsahuje konfigurační data, která lze ručně modifikovat.

Soubor je psaný ve formátu JSON. Formát JSON jsme zvolili, neboť je poměrně populární, dobře čitelný a stručný, srovnáme-li ho například s formátem XML (Extensible Markup Language).

Načtení dat konfiguračního souboru

Načítání dat zajišťuje třída **AppConfig**. V této třídě jsme pomocí statického konstruktoru zajistili, že k načtení konfiguračních dat dojde při prvním přístupu k jejich hodnotám.

Třída **AppPath** nám zajišťuje, že můžeme se souborovými cestami pracovat relativně vůči adresáři projektu.

V původním návrhu jsme soubor **appsettings.json** nechávali kopírovat do výstupního adresáře a následně jsme pracovali s kopii tohoto souboru. To však vedlo ke vzniku kopie pro každý modul a dokonce docházelo k chybám, kdy nedošlo k překopírování souboru **appsettings.json** do některého z modulů. Rozhodli jsme se tedy pracovat s tímto souborem přímo.

Přístup k načteným datům

Pro přístup k datům jsme vytvořili datovou třídu **AppSettings**. Položky této třídy zrcadlí strukturu souboru **appsettings.json**. Abychom zajistili typovaný přístup k datům, využíváme **options pattern**⁶.

Práci nám usnadňuje rozšíření **Microsoft.Extensions.Configuration**.

Instanci **AppSettings** vytváříme ve třídě **AppConfig**, ta ji také zpřístupňuje skrze položku **appSettings**.

⁶Detailní popis lze najít na adrese <https://docs.microsoft.com/en-us/dotnet/core/extensions/options>

8. Výsledky

V předchozích kapitolách jsme již ukázali, jak vypadají vizualizace dostupnosti. Pracovali jsme však jen s jízdními řády společnosti PID. Jak jsme však zmiňovali v úvodu, naše aplikace má za cíl pracovat s jízdními řády z různých lokalit.

V této kapitole si ukážeme, jak funguje naše aplikace pro data z různých lokalit. Dále zmíníme, jak dlouho trvají výpočty s jedním vstupním místem na procesoru **i5-6300U**. Pro intervalové výpočty volíme třiceti minutové okolí vstupního času s intervalom o délce jedné minuty.

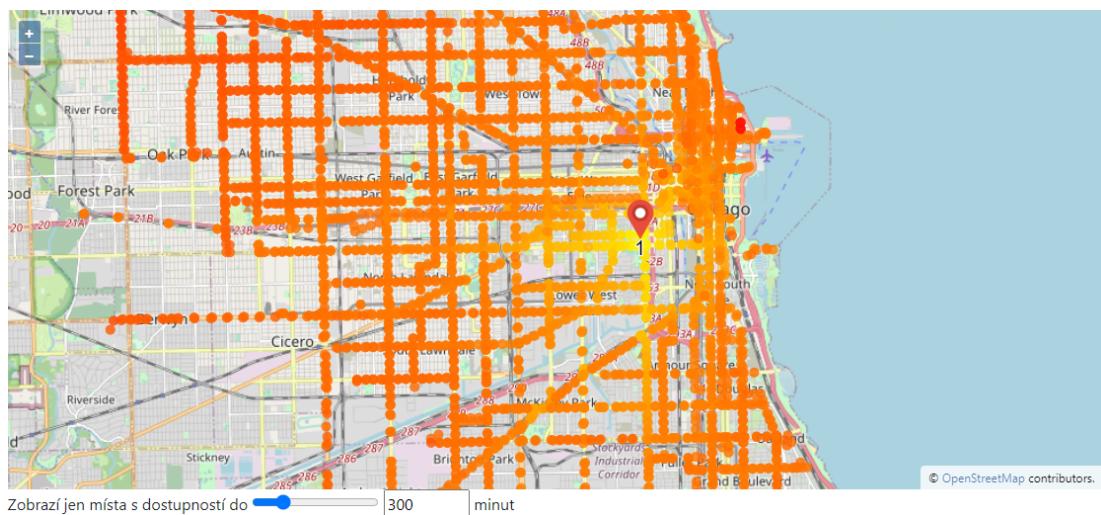
8.1 Chicago

GTFS data pro město Chicago jsou dostupná na stránce transitfeed¹.

Data jsou svou velikostí srovnatelná s daty poskytovanými společnosti PID. Data společnosti PID obsahují zhruba 16 000 zastávek a 83 000 jízd. Data města Chicago obsahují 11 000 zastávek a 94 000 jízd.

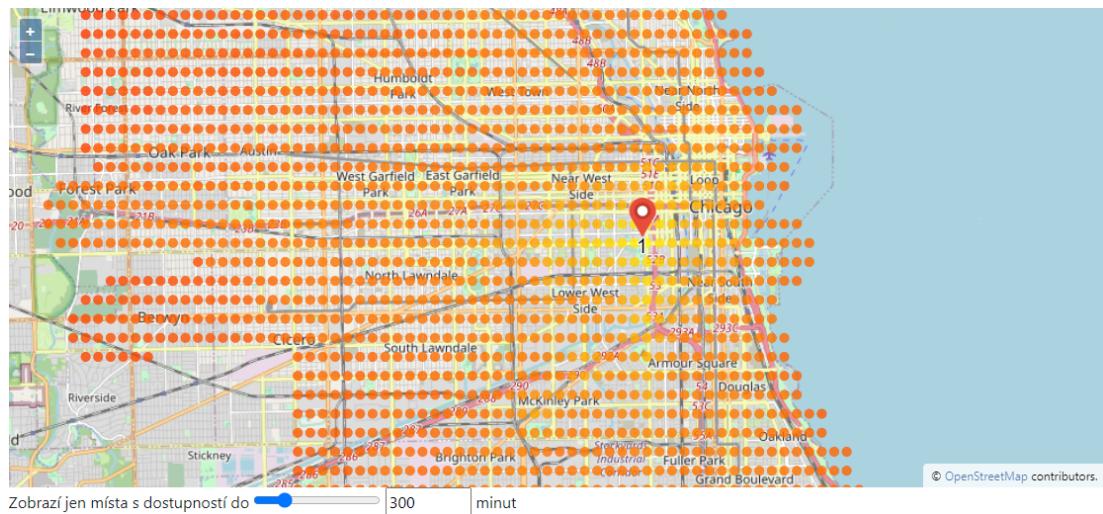
Průměrně trvá:

- Vyhodnocení dostupnosti na zastávkách: 210 ms.
- Vyhodnocení dostupnosti v intervalu na zastávkách: 2 560 ms.
- Výpočet sousedů bodů v rozlišení 100 x 100: 6 550 ms.
- Vyhodnocení dostupnosti pro body v rozlišení 100 x 100: 215 ms



Obrázek 8.1: Vizualizace dostupnosti do 300 minut na zastávkách v Chicagu.

¹Konkrétně na <https://transitfeeds.com/p/chicago-transit-authority/165>



Obrázek 8.2: Vizualizace dostupnosti do 300 minut na bodech v rastru pro Chicago.

8.2 Německo

GTFS data pro celé Německo jsou dostupná na stránce https://gtfs.de/en/feeds/de_nv/.

Jedná se o největší jednotný dataset, který se nám podařilo nalézt. Ve srovnání s daty společnosti PID obsahují tato data zhruba 27x více zastávek a 16x více jízd.

Průměrně trvá:

- Vyhodnocení dostupnosti na zastávkách: 17.5 s.
- Vyhodnocení dostupnosti v intervalu na zastávkách: 100 s.

8.2.1 Problémy s velkými daty

Použití takto objemných dat však není jednoduché.

Zpracování a serializace dat nám, za použití optimalizace popsané v sekci 7.1.3, trvala zhruba 2 hodiny.

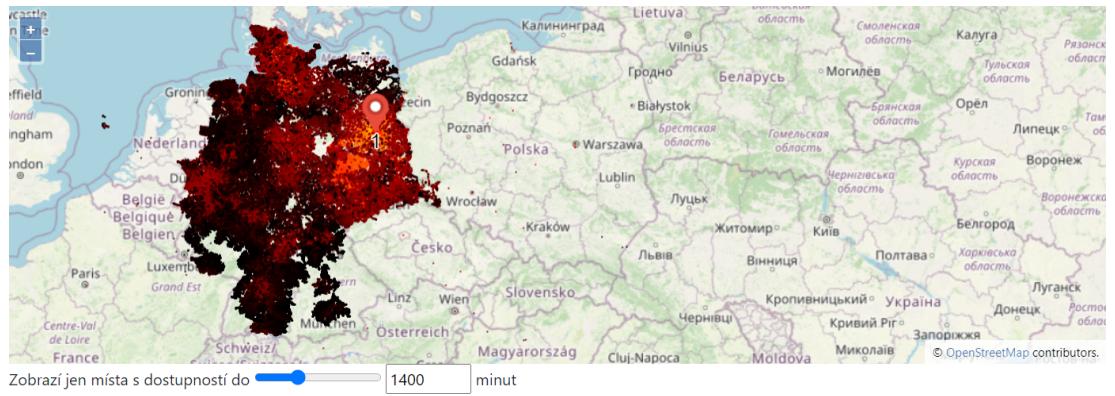
Při deserializaci jsme museli navýšit maximální hloubku zanoření při čtení formátu JSON a s tím i související velikost zásobníku.

Přetečení zásobníku jsme museli řešit i na frontendu, konkrétně v JavaScriptové funkci **Math.min**.

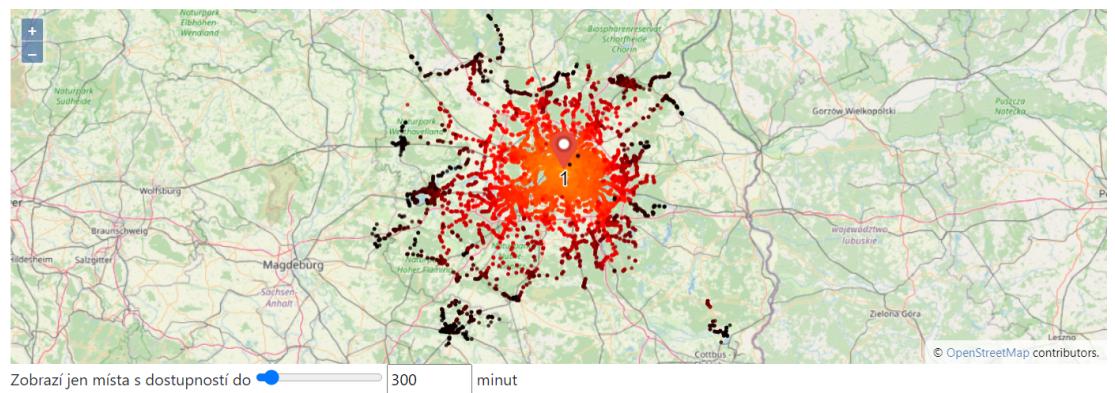
Nakonec se nám podařilo vizualizovat zastávky, viz 8.3 a 8.4. Pro takto velká data však naše aplikace není stavěná.

Vyhledávání je příliš pomalé pro praktické využití.

Nejsme schopni rozumně vizualizovat body v rastru, neboť území Německa je poměrně velké a body v rozlišení 100 x 100 jsou vzájemně příliš vzdáleny. Větší rozlišení by vyžadovalo déle trvající předvýpočet sousedů či optimalizaci hledání sousedů.



Obrázek 8.3: Vizualizace dostupnosti do 1400 minut na zastávkách v Německu.



Obrázek 8.4: Vizualizaci dostupnosti do 300 minut v okolí Berlína.

Závěr

V této práci jsme chtěli ohodnotit všechna místa podle dostupnosti veřejnou dopravou. To se nám v podstatě podařilo. Jediným možným nedostatkem je to, že počítáme dostupnost z míst zadaných uživatelem na všechna ostatní místa, ale neřešíme již dostupnost v opačném směru. Možné řešení tohoto problému jsme však navrhovali v sekci 4.2. Dostupnost jsme chtěli vizualizovat na všech místech. Nám se podařilo vizualizovat dostupnost pro body v daném rozlišení. Pro větší území potřebujeme počítat dostupnost v bodech ve větším rozlišení. Avšak předvýpočet spojený s vyhodnocováním bodů může být pro velké rozlišení neprakticky dlouhý. Jak jsme plánovali, výsledná aplikace je schopna pracovat s jízdními řády z různých lokalit. Při práci s velkými daty jízdních řádů, jako poskytuje například Německo, však začínáme mít výkonné problémy.

Výsledkem této práce je knihovna a webová aplikace. Knihovnu lze využívat nezávisle na webové aplikaci a nejspíše by se dala použít i pro řešení jiných problémů, které by vyžadovaly ohodnocení dostupnosti veřejnou dopravou. Webová aplikace pracuje s interaktivní mapou. Společně zpřístupňují naši knihovnu běžným uživatelům a usnadňují orientaci ve výsledných dostupnostech.

Vedle optimalizací a nápadů na zlepšení, zmíněných v předchozích kapitolách, bychom v budoucnu mohli rozšířit funkcionality naší aplikace. Naše aplikace by ve spojení s cenovou mapou² mohla vypočítávat poměr dostupnosti a ceny pozemku. Pokud bychom měli přístup k právě pronajímaným či prodávaným nemovitostem, mohli bychom je seřadit dle dostupnosti nebo bychom mohli opět určovat poměr dostupnosti a ceny. Aplikaci bychom mohli rozširovat i jiným směrem. Některé lidi by vedle dostupnosti veřejnou dopravou mohlo zajímat, v jaké vzdálenosti od daného místa jsou nemocnice, školy, obchody a další často navštěvované budovy. Takovéto budovy bychom mohli vyhledat například pomocí dotazů v jazyce SPARQL (SPARQL Protocol and RDF Query Language) na stránce wikidat³.

²Pro Prahu je cenová mapa dostupná na adrese <https://app.iprpraha.cz/apl/app/cenova-mapa/>

³Wikidata dotazy lze psát na adrese <https://query.wikidata.org/>

Seznam použité literatury

- BARKER, B. (2022). Quadtree spatial hash - brandonbarker.me [online]. URL https://brandonbarker.me/downloads/quadtree_spatialhash.pdf. [cit. 2022-05-11].
- DELLING, D., PAJOR, T. a WERNECK, R. F. (2015). Round-based public transit routing. *Transportation Science*, **49**(3), 591–604.
- GOOGLE (2022). Gtfs static overview [online]. URL <https://developers.google.com/transit/gtfs>. [cit. 2022-05-11].
- KOTHURI, R. K. V., RAVADA, S. a ABUGOV, D. (2002). Quadtree and r-tree indexes in oracle spatial: a comparison using gis data. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 546–557. ACM.
- PHAN, D.-M. a VIENNOT, L. (2019). Fast public transit routing with unrestricted walking through hub labeling. In *International Symposium on Experimental Algorithms*, pages 237–247. Springer.
- PID (2022). Otevřená data pid [online]. URL <https://pid.cz/o-systemu/opendata/>. [cit. 2022-05-11].

Seznam obrázků

2.1	Relevantní obsah souborů ve formátu GTFS.	7
6.1	Vizualizace dostupnosti na zastávkách z Malostranského náměstí, Kladna a Příbrami. Hodnoty v legendě vyjadřují dostupnost v sekundách.	19
6.2	Vizualizace dostupnosti v rastru 100 x 100 z Malostranského náměstí, Kladna a Příbrami. Hodnoty v legendě vyjadřují dostupnost v sekundách.	20
6.3	Interaktivní vizualizace dostupnosti na zastávkách z Malostranského náměstí, Kladna a Příbrami.	21
6.4	Interaktivní vizualizace dostupnosti v rastru 100 x 100 z Malostranského náměstí, Kladna a Příbrami.	22
7.1	Moduly aplikace a jejich vzájemné propojení.	23
7.2	Stránka Finder.	35
7.3	Hlášení chyby při zadání nedostupného místa.	36
8.1	Vizualizace dostupnosti do 300 minut na zastávkách v Chicagu. .	41
8.2	Vizualizace dostupnosti do 300 minut na bodech v rastru pro Chicago.	42
8.3	Vizualizace dostupnosti do 1400 minut na zastávkách v Německu.	43
8.4	Vizualizaci dostupnosti do 300 minut v okolí Berlínu.	43

A. Přílohy

A.1 Dokumentace

Uživatelská dokumentace

Uživatelská dokumentace je součástí webové stránky. Konkrétně je popsána na úvodní stránce. Pro jednodušší přístup přikládáme uživatelskou dokumentaci do elektronické přílohy této práce.

Programátorská dokumentace

Programátorská dokumentace je obecně popsána v kapitole 7. Dokumentace kódu je vytvořena pomocí XML komentářů. Samotný kód přikládáme v elektronické příloze.

A.1.1 Instalace

Prerekvizity

Ke spuštění či překladu aplikace potřebujeme platformu .NET 5.0, která je k dispozici na stránkách Microsoftu¹.

Instalace platformy .NET 5.0 nám zpřístupní příkaz **dotnet**, pomocí kterého aplikaci spustíme. Alternativně můžeme ke spuštění využít nástroje poskytované prostředím **Visual Studio**.

Spuštění

Aplikaci přeložíme zavoláním příkazu **dotnet build** z adresáře obsahujícího aplikaci.

Příkaz **dotnet run --project src/Web/Web.csproj** použijeme ke spuštění naší webové aplikace.

Pro ověření funkčnosti aplikace můžeme spustit **Unit Testy** pomocí příkazu **dotnet test**.

A.1.2 Konfigurační soubor

GTFS data

Konstanta **ShouldUpdate** určuje, zdali chceme při spuštění aplikace data aktualizovat, viz popis třídy **DataUpdater** v sekci 7.4.2. Při aktualizaci dojde ke stažení dat a k jejich deserializaci. Pokud data neaktualizujeme, serializujeme dřívě uložená data.

URL pro stažení dat specifikujeme konstantou **GTFSSourceURI**. Aktuálně podporujeme jen data archivovaná ve formátu zip.

Data stahujeme do složky určené konstantou **PathToGTFSFolder** a cestu k serializaci určuje konstanta **GTFSSerializationPath**.

¹Ke stažení na stránce <https://dotnet.microsoft.com/en-us/download/dotnet/5.0>

Platnost dat závisí na konkrétním zprostředkovateli a dá se nastavit skrze konstantu **ValidityInDays**.

UTC offset pro časy jízdních řádů, které uchováváme na serveru, lze nastavit konstantou **UTCOffset**.

Přestupy

Generování přestupů popisujeme v sekci 7.1.2.

Konstantu **MaxTransferDistanceInMeters** používáme pro omezení vzdálenosti během generování přestupů.

Konstantu **WalkingSpeedInMetersPerSec** určuje průměrnou rychlosť chůze. Používáme ji k odhadu času potřebného pro přestup.

Sousední zastávky

Sousední zastávky detailně popisujeme v sekci 7.3.1.

Konstantu **WalkingSpeedInMetersPerSec** používáme také pro odhad času potřebného k příchodu z nějakého místa na zastávky v okolí.

Konstanta **NearestStopsDistanceInMeters** omezuje vzdálenost, ve které hledáme z daného místa sousední zastávky.

Vizualizace bodů

Konstanta **VisualisedRasterPointsResolution** určuje rozlišení, pro které vizualizujeme dostupnost v bodech, viz sekce 6.1.