FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Virtuální historický průvodce - navigační modul
Student:	Ivo Strejc
Vedoucí:	Ing. Jiří Chludil
Studijní program:	Informatika
Studijní obor:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce letního semestru 2018/19

Pokyny pro vypracování

Týmový projekt Virtuální historický průvodce má za cíl uživatelům přívětivě vizualizovat historii architektury českých měst. Cílem práce je navrhnout algoritmy pro hledání optimální cesty uživatele v urbanistické infrastruktuře podle zadaných parametrů.

1. Analyzujte vhodné algoritmy pro hledání optimální cesty a navrhněte možné úpravy nejlépe vyhovujícího algoritmu.
2. Analyzujte a navrhněte způsob sbírání a zpracování dat o průchodnosti tras a navrhněte vhodný způsob ukládání.
3. Definujte funkční a nefunkční požadavky, případy užití pro integraci nalezeného algoritmu do jádra systému.
4. Navrhněte komunikační API s klienty (implementace bude v jiné BP).
5. Implementujte navigační modul splňující navržené komunikačnímu rozhraní.
6. Modul podrobte vhodným testům (paměťovým a zátěžovým).

Seznam odborné literatury

Dodá vedoucí práce.

doc. Ing. Jan Janoušek, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 12. února 2018

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Virtuální historický průvodce - navigační modul

Ivo Strejc

Katedra teoretické informatiky
Vedoucí práce: Ing. Jiří Chludil

16. května 2019

Poděkování

Chěl bych poděkovat Ing. Jiřímu Chludilovi za vedení této práce a za jeho trpělivost při konzultacích a RNDr. Jiřině Scholtzové, Ph.D., za dodatečné konzultace. Dále bych rád poděkoval mojí rodině a blízkým přátelům za podporu, kterou mi projevili během vyprácovávní této práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 16. května 2019 .....

Odkaz na tuto práci

Strejc, Ivo. *Virtuální historický průvodce - navigační modul.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Abstrakt

Tato práce se zaměřuje na vytvoření navigačního modulu pro týmový projekt *Virtuální historický průvodce*. Náplní práce modulu je řešení *Problém obchodního cestujícího* na dynamicky ohodnocených grafech. Práce se zabývá analýzou již existujících navigací, řešením uvedeného problému konstrukce tras, predikcí zaplněnosti bodu zájmu a sbírání dat pro tyto predikce. Závěrem je popsána implementace modulu a jeho následné testování.

Klíčová slova Teorie grafů, Hledání cest, Problém obchodního cestujícího, Navigační modul, REST API - Representational state transfer Application programming interface

Abstract

This thesis focuses on creating navigation module for group project *Virtuální historický průvodce*. Module will be tasked with solving *Traveling Salesman Problem* on dynamicly weigted graphs. Part of this thesis is analysis of already existing navigations, describing ways of solving mentioned Traveling Salesman Problem on dynamicly weighted graphs and designing delay predictions and data collecting for these predictions. Final part is dedicated to description of implementation and testing of this module.

Keywords Graph theory, Path finding, Traveling salesman problem, Navigation module, REST API - Representational state transfer Application programming interface

Obsah

Úvod **1**

1 Teorie **3**
1.1 Použité grafové pojmy 3
1.2 Problém obchodního cestujícího 5
1.3 Algoritmy pro hledání nejkratších cest 9
1.4 Problém hledání trasy z pohledu teorie grafu 10

2 Analýza **13**
2.1 Možnosti tvoření tras 13
2.2 Existující navigační systémy 15
2.3 Predikční model 22
2.4 Původní návrh zapojení do projektu 23
2.5 Funkční/nefunkční požadavky 24

3 Návrh **25**
3.1 Zapojení do projektu 25
3.2 Návrh architektury modulu 26
3.3 REST API 27
3.4 Sběr dat 27
3.5 Předzpracování grafu 28
3.6 Návrh vytváření tras 29
3.7 Návrh výpočetního části 30
3.8 Návrh predikčního modelu 30

4 Realizace **33**
4.1 Zapojení do projektu 33
4.2 Použité technologie 33
4.3 Dokumentace 35
4.4 API 35

4.5 Výpočetní část 36
4.6 Zprovoznění projektu 37

5 Testování **39**
5.1 Testovací data 39
5.2 Porovnávané algoritmy 39
5.3 Časové testování 40
5.4 Přesnost navrhovaného algoritmu 40
5.5 Paměťové testování 41

Závěr **45**
Budoucí práce 45

Literatura **47**

A Seznam použitých zkratek **51**

B Obsah přiloženého CD **53**

Seznam obrázků

1.1 Příklad TSP 5
1.2 Ukázka 2-aproximačního algoritmu 7
1.3 Příklad 2-opt kroku 8
1.4 Ukázka prohledávání A* 9

2.1 Google Play žebříček 15
2.2 Google Maps - Populární časy a zaplněnost 17
2.3 Google Maps - Navigace 17
2.4 Mapy.cz - plánování výletu po okolí 18
2.5 Aplikace Waze 19
2.6 Původní návrh architektury týmového projektu 23

3.1 Napojení na jádro 25
3.2 Architektura modulu 26
3.4 Ukázka ortodromy 29
3.5 Příklad otočení orientace při 2-opt kroku 29

4.1 Diagram napojení do projektu 34

5.1 Graf závislosti času na počtu vrcholů 41

Seznam tabulek

3.1 Tabulka vybraných koeficientů a časových konstant pro predikci . 31

5.1 Naměřené časy 42
5.2 Průměrné odchylky řešení pro různé počty vrcholů 43
5.3 Naměřená maximální požadovaná paměť 43

Úvod

Týmový projekt *Virtuální historický průvodce* se zaměřuje na prohlídky historických center měst s využitím technologie augmentované reality. Pro dobrý zážitek je důležité, aby byla co nejlepší viditelnost na památky, tedy především, aby nebylo jejich okolí přeplněné. Proto vznikla potřeba navigačního modulu, který by rozhodoval, v jakém pořadí má turista památky navštívit tak, aby se optimalizoval čas a zaplněnost okolí památky. A právě tento modul budu navrhovat a implementovat v této práci.
Podobný problém je již popsán jako *Problém obchodního cestujícího*, anglicky *Traveling salesman problem*, a jedná se o celkem známý problém v informatice a matematice, neboť patří mezi jedny z časově nejnáročnějších a ten, kdo by nalezl efektivní řešení by pomohl vyřešit jednu z velkých otázek matematiky, což by bylo náležitě finančně ohodnoceno.

Cíle

Cílem práce je navrhnout modul, který bude tento problém řešit pomocí vytvořeného algoritmu založeného na analýze. Součástí modulu bude matematický model na predikci zaplněnosti pomocí dříve nasbíraných a aktuálních dat pro výpočet optimální trasy. Důraz je kladen na návrh způsobu ukládání a zpracování dat o zaplněnosti v čase a následný způsob predikce a konstrukce výsledných tras.

Rozbor zadání

1. Analyzujte vhodné algoritmy pro hledání optimální cesty a navrhněte možné úpravy nejlépe vyhovujícího algoritmu.
Nejprve provedu analýzu a srovnání vhodných algoritmů pro hledání nejkratších cest a tras. Následně vyberu nejlépe vyhovující algoritmy či upravím existující algoritmy pro potřeby práce.

2. Analyzujte a navrhněte způsob sbírání a zpracování dat o průchodnosti tras a navrhněte vhodný způsob ukládání.
Při analýze predikčního modelu se také zaměřím na data, které bude potřeba sbírat a následně navrhnu způsob sbírání, zpracování a ukládání těchto dat.

3. Definujte funkční a nefunkční požadavky, případy užití pro integraci nalezeného algoritmu do jádra systému.
Po analýze definuji funkční a nefunkční požadavky. Následně nastíním integraci modulu do celého systému.

4. Navrhněte komunikační API s klienty (implementace bude v jiné BP).
V rámci práce bude navrženo komunikační API sloužící ke komunikaci se zbytkem projektu.

5. Implementujte navigační modul splňující navržené komunikačnímu rozhraní.
Na základě navržených požadavků a komunikačního rozhraní modul implementuji.

6. Modul podrobte vhodným testům (paměťovým a zátěžovým).
Po zvolení vhodné metodiky testování otestuji časovou a paměťovou náročnost vybraných implementovaných algoritmů a následně celý modul otestuji na jeho celkovou náročnost.

Kapitola 1

Teorie

1.1 Použité grafové pojmy

Nejprve si vymezím použité pojmy z teorie grafů. Většina je převzatá z předmětů BI-AG1[1] a BI-AG2[2].

- **Neorientovaný graf** je uspořádá dvojice (V, E), kde V je neprázdná konečná množina vrcholů a E množnina hran.
- **Hrana** je dvouprvková podmnožina V, značíme $\{u, v\}$; $u, v \in V$.
- **Orientovaná hrana** je uspořádaná dvojice vrcholů, značíme (u, v); $u, v \in V$.
- **Orientovaný graf** je uspořádaná dvojice (V, E), kde E je množina orientovaných hran[1].
- Nechť $n \geq 1$, poté graf $(V, \binom{V}{2})$, kde $|V| = n$ nazveme **úplným grafem**, značíme K_n.
- Nechť $m \geq 0$, poté graf $(\{0, ..., m\}, \{\{i, i+1\} \mid i \in \{0, ..., m-1\}\})$ nazveme **cestu** z vrcholu 0 do vrcholu m délky m.
- **Stupněm vrcholu** v v grafu G označíme počet hran grafu G obsahující vrchol v.
- Graf H je **podgrafem** grafu G, když $V(H) \subseteq V(G)$ a $E(H) \subseteq E(G)$.
- **Souvislým grafem** nazveme graf, ve kterém existuje cesta mezi každými dvěmi vrcholy.
- Nechť $n \geq 3$, poté graf $(\{1, ..., n\}, \{\{i, i+1\} \mid i \in \{1, ..., n-1\}\} \cup \{\{1, n-1\}\})$ nazveme **kružnicí** s délkou n.
- **Hamiltonovská kružnice** v grafu je kružnice obsahující všechny vrcholy grafu.
- **Sled** v orientovaném grafu G je posloupnost $(v_0, e_1, v_1, e_2, ..., e_n, v_n)$, pro který platí $v_i \in V(G)$ pro všechna $i \in \{0, ..., n\}$ a $e_i = (v_{i-1}, v_i) \in E(G)$ pro všechna $i \in \{1, ..., n\}$.

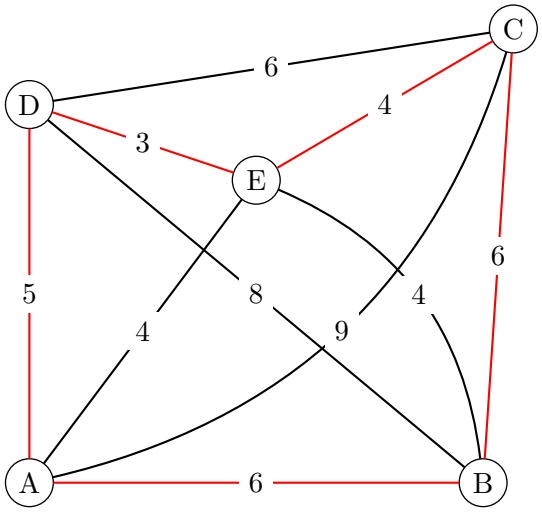
[1] V opět značí neprázdnou konečnou množinu vrcholů

- **Tah** je sled v grafu, ve kterém se neopakují hrany. Pokud navíc končí ve stejném vrcholu, ve kterém začíná, označníme tento tah za uzavřený.
- **Eulerovský tah** v grafu $G(V, E)$je tah, který obsahuje všechny vrcholy V a hrany E.
- **Eulerovský graf** je graf, ve kterém existuje uzavřený eulerovský tah.
- **Váhová funkce** je funkce $\omega : E \to \mathbb{R}$.
- **Ohodnocený (ne)orientovaný graf** je uspořádaná trojice (V, E, ω), kde V je množina vrcholů, E množina (ne)orientovaných hran a ω je váhová funkce s oborem hodnot V.
- **Ohodnocení grafu** je $\sum_{e \in E} \omega(e)$.
- **Nejkratší cesta** z vrcholu u do vrcholu v v grafu G je cesta s nejnižším ohodnocením.
- Mějme souvislý graf G, poté graf K označíme jako **kostrou grafu** G, pokud je K podgrafem G, $V(K) = V(G)$ a K neobsahuje kružnici[2].
- **Minimální kostra** grafu G je kostra s nejnižším ohodnocením.
- **Multigraf** je graf, ve kterém může existovat více hran mezi dvojicí vrcholů.

[2]Neexistuje žádný podgraf, který je kružnicí.

1.2 Problém obchodního cestujícího

Problém obchodního cestujícího, dále jen **TSP**, je problém nalezení nejkratší hamiltonovské kružnice v úplném neorientovaném ohodnoceném grafu. Jak název samotný napovídá, problém má kořeny i v běžném životě. Příkladem uvedeného problému je obchodní cestující, který chce za nejkratší možnou dobu navštívit všechny domy na jeho seznamu a opět skončit na místě, ze kterého začínal.



Obrázek 1.1: Příklad problému obchodního cestujícího, červeně je označeno řešení

I přesto, že obchodních cestujících výrazně ubylo, tento problém je stále zkoumaný a nalezení rychlejšího způsobu řešení by bylo značným přínosem například logistickým službám či při plánování pohybů montážních robotů nad jednotlivými výrobky.[3]
TSP se řadí mezi NP-úplné problémy, tedy problémy, které řeší v polynomiálním čase nedeterministický Turingův stoj.

1.2.1 Další varianty TSP

Kromě základní verze existují i další varianty, jako například *Problém čínského listonoše*, popsaný např. v další bakalářské práci vzniklé na fakultě [4], TSP na asymetrickém grafu, jehož výstupem je orientovaná kružnice, či dynamické TSP, tedy TSP na dynamicky ohodnoceném grafu, který zpravidla bývá i asymetrický. Poslední zmíněné variantě se do značné hloubky zabývá článek [5]. Řešení navrhované v tomto článku je velice složité a opírá se o využití pokročilé statistiky a heuristických funkcí.

1.2.2 Způsoby řešení TSP

1.2.2.1 Naivní řešení

Naivní algoritmus je na implementaci nejjednodušší řešení, které zkouší všechny kružnice a zvolí takovou kružnici, která má nejmenší ohodnocení. Takovéto řešení má časovou složitost $\mathcal{O}(n!)$, což se stává velice rychle nepraktické s nárůstem počtu vrcholů. Paměťová složitost naivního algoritmu je $\mathcal{O}(n)$.

Prořezávání Na nezáporně ohodnocených grafech lze navíc při systematickém vytváření omezit počet kružnic, které budeme skutečně sestrojovat. Při postupném vytváření všech možností se můžeme podívat, zda cena dosavadně sestavené cesty není větší, než cena nejlepšího nalezeného řešení. Poté nemusíme vytvářet cesty, které začínají či dokonce obsahují současně sestavenou cestu, neboť ohodnocení je nezáporné a tedy výsledná permutace nemůže být lepší než nejlepší nalezené řešení. Tímto postupem zůstává horní hranice časového odhadu stále $\mathcal{O}(n!)$, neboť nelze zaručit, že při každé iteraci nedojde ke zlepšení.
Příklad takového způsobu vytváření kružnic je použití DFS pro vytvoření všech cest délky n a následné přidání hrany z posledně přidaného vrcholu do počátečního vrcholu.

1.2.2.2 Řešení pomocí dynamického programování

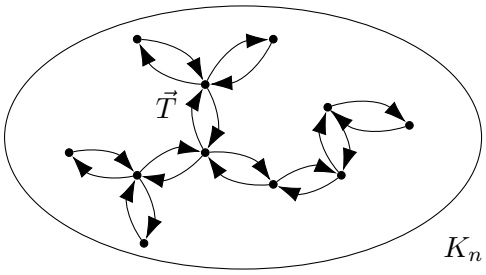
Tento algoritmus se nazývá Bellman–Held–Karpův algoritmus [6] a je ukázkovým příkladem dynamického programování. Principem je řešit problém na menší množině bodů, počínaje od triviálních problémů, a postupně skládat nejlepší řešení vždy pro množinu o jeden vrchol větší, až získáme celou množinu vrcholů, na které trasu hledáme. Výhodou je možnost ukládat výsledky jednotlivých podproblémů, neboť se často stává, že je třeba stejný podproblém vyřešit vícekrát.
Tímto principem se složitost snižuje na $\mathcal{O}(2^n n^2)$, což je stále exponenciální složitost, jedná se o značné zlepšení. Paměťová složitost tohoto algoritmu je dle [7] rovna $\mathcal{O}(\frac{2m!}{(m!)^2})$. Rovněž je zde možnost na nezáporně ohodnocených grafech využít prořezávání, což ale nesnižuje asymptoticky časovou složitost.

1.2.2.3 c-aproximační algoritmy

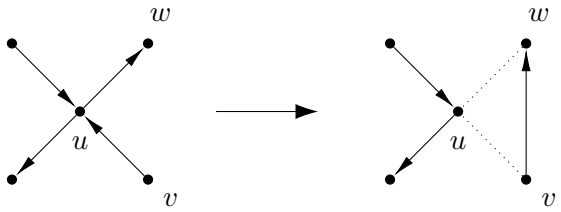
Kromě exaktních řešení existují i takzvané aproximační algoritmy, jejichž účelem je řešit NP-těžké problémy v polynomiálním čase s určitou nepřesností. Pro minimalizační problémy nazveme algoritmus A' c-aproximačním, pokud existuje konstanta $c > 1$, pro kterou platí, že výsledek A' je nejvýše c-násobkem optimálního řešení.
Představíme si zde známá 2-aproximační a $\frac{3}{2}$-aproximační řešení, která jsou

aplikovatelná na úplných grafech s nezáporným ohodnocením hran ω, pro které platí trojúhelníková nerovnost, tj. pro každé tři hrany ve tvaru (x, y), (y, z), (x, z) platí $\omega((x, y)) + \omega((y, z)) \geq \omega((x, z))$.[3]

2-aproximační řešení Prvním krokem algoritmu je nalezení minimální kostry grafu, při symetrickém zorientováním grafu se zachováním vah hran. Z minimální kostry vznikne Eulerovský sled.



(a) Příklad Eulerovského sledu vzniklého z minimální kostry



(b) Zkracování sledu

Obrázek 1.2: Ukázka 2-aproximačního algoritmu[3]

Dokud existuje ve vzniklém sledu vrchol u s alespoň dvěma výstupními hranami, nahradíme hrany (u, v) a (u, w) hranou (v, w). Tímto přetvoříme sled do kružnice. Vzniklou kružnici vrátíme jakožto řešení.
Algoritmus doběhne v polynomiálním čase a výsledné řešení je maximálně 2krát větší, než optimální řešení[3]. Celková časová složitost se odvíjí především od algoritmu zvoleného k nalezení minimální kostry grafu, následné zkrácení sledu jsme schopni provést v $\mathcal{O}(|V|)$ a to tak, že vzniklý sled procházíme a kdykoliv nalezneme vrchol, který jsme již navštívili, přeskočíme jej. Na nalezení minimální kostry grafu se nabízí Jarníkův algoritmus, jehož časová složitost se odvíjí od implementace prioritní fronty. Implementace využívající striktní Fibonacciho haldu má celkovou časovou složitost $\mathcal{O}(|E| + |V| \log |V|)$, tedy $\mathcal{O}(|V|^2)$, neboť se jedná o úplný graf[3].

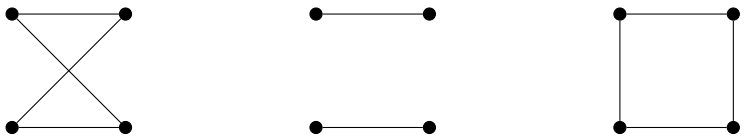
[3]V úplném grafu je $\binom{|V|}{2} = \frac{|V|*(|V|-1)}{2}$ hran, tedy asymptoticky $|V|^2$

$\frac{3}{2}$-aproximační řešení Christofidův algoritmus[8] je nejlepší známé c-aproximační řešení TSP s $c = \frac{3}{2}$, které stejně jako 2-aproximační algoritmus využívá minimální kostru, k dosažení lepších výsledků používá perfektní minimální párování na vrcholech, které mají v minimální kostře lichý stupeň. Následně aplikuje stejné zkracování sledu jako v předchozí variantě.

Algoritmus tedy krom společného problému hledání minimální kostry potřebuje řešit problém nalezení minimálního perfektního párování. Tento problém se dá řešit pomocí algoritmu *Blossom V* s časovou složitostí $\mathcal{O}(|V|^3|E|)$[9].

1.2.2.4 Další heuristické metody

Krom zmíněných c-aproximačních algoritmů existují i další heuristické algoritmy, kterými se zabývá článek [10]. Článek porovnává výsledky algoritmů a závěrem navrhuje využití Lin–Kernighan heuristiky, případně 2-opt jako rychlejší variantu. Oba algoritmy jsou zaměřeny na optimalizaci již sestavených kružnic. V každém kroku 2-opt algoritmus odebere z kružnice dvě nesousedící hrany, čímž se kružnice rozpadne na 2 cesty, které následně opět spojí použitím jiných hran, než které byli původně odebrány. Pokud je ohodnocení nově vzniklé kružnice menší než původní, ponecháme tyto změny. Tento postup opakujeme dokud existující zlepšující kroky.



Obrázek 1.3: Příklad 2-opt kroku

Uvedený algoritmus lze upravit na k-opt algoritmus, kde se pokaždé odebírá k hran. Lin–Kernighanův algoritmus se soustředí na volení správného k pro použití k-opt algoritmu v každé iteraci. Více do hloubky se zabývá následující práce[11], který zároveň navrhuje efektivní implementaci.

1.3 Algoritmy pro hledání nejkratších cest

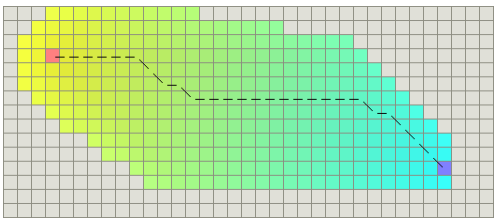
Hledání nejkratších cest v grafu je pro navigační modul kritické. Jelikož se jedná o graf sestavený nad mapou, všechna ohodnocení hran jsou kladná, tedy není třeba se zabývat algoritmy, které jsou určeny pro hledání cest nad grafy se záporně ohodnocenými hranami.

1.3.1 Dijkstrův algoritmus

Dijkstrův algoritmus najde v grafu s nezáporně ohodnocenými hranami vždy nejkratší cestu mezi dvěma vrcholy.
Základní myšlenka algoritmu je vždy expandovat do vrcholu s nejnižší vzdáleností od počátku a ještě neuzavřené sousedy přidat do prioritní fronty s hodnotou rovné součtu vzdálenosti původního vrcholu a vzdálenosti souseda od počátečního vrcholu. Pokud již daný soused ve frontě je, pouze se upraví jeho priorita.
Existuje i obousměrná varianta, kdy se expanduje zároveň jak z počátečního uzlu, tak i z uzlu cílového, a v momentě, kdy nastane průnik mezi vrcholy expandovanými z počátečního a koncového uzlu, nalezli jsme nejkratší cestu. Tento postup nalezne nejkratší cestu expandováním přibližně poloviního počtu vrcholů, což je sice značné zlepšení, nesnižuje to však asymptotickou složitost.
Pro implementaci algoritmus je důležitá implementace prioritní fronty, která může značně ovlivnit časovou náročnost. Při použití striktní Fibonacciho haldu je časová složitost $\mathcal{O}(|E|+|V|\log|V|)$, při použití haldy $\mathcal{O}(|E|+|V|^2)$. Paměťová složitost algoritmu je $\mathcal{O}(|V|)$.

1.3.2 Algoritmus A*

A* vychází z Dijkstrova algoritmu a kombinuje jej s heuristikou, pro kterou musí platit, že je přípustná,tedy že nepřeceňuje vrchol. Pokud je navíc heuristika monotónní, A* vždy najde nejkratší cestu. Díky správně zvolené heuristice může A* procházet méně vrcholů oproti Dijkstrův algoritmu.



Obrázek 1.4: Ukázka prohledávání A*
Ukázka prohledávání A* s euklidovskou vzdáleností jako heuristikou.[12]

Mezi nejzákladnější heuristiky patří vzdálenost vrcholu do cíle. Pro většinu grafů založených na mapách vede použití této heuristiky k nalezení nejkratší cesty. Další oblíbenou heuristikou je kombinace vzdálenosti vrcholu od cíle a vzdálenosti od přímky procházející počátkem a cílem.
Časová složitost tohoto algoritmu je závislá na heuristické funkci. Při zvolení nevhodné heuristické bude časová složitost stejná jako u Dijkstrova algoritmu. Paměťová složitost je $\mathcal{O}(|V|)$.

1.3.3 Floyd-Warshallův algoritmus

Tento algoritmus slouží především k hledání matice vzdálenosti mezi každou dvojicí vrcholů. Rozšířená verze algoritmu také vrací matici předchůdců, ze které můžeme samotné cesty zrekonstruovat. Tento algoritmus je tedy především vhodný, pokud chceme získat vzdálenost mezi všemi vrcholy. Samotný algoritmus využívá principy dynamického programování.
Časová složitost tohoto algoritmu je $\mathcal{O}(n^3)$ a paměťová složitost je $\mathcal{O}(n^2)$

1.4 Problém hledání trasy z pohledu teorie grafu

Hlavní problém celé hledání trasy v této práci by se dal popsat jako dynamické TSP na asymetrickém grafu.
Na začátek je důležité stanovit si, jak budeme hodnotit trasy. Klasické ohodnocení hran grafu $G(V, E)$ je funkce $\omega : E \to \mathbb{R}$, kdy při hledání cest v reálném světě můžeme omezit obor hodnot na $\mathbb{R}^+$. Nejčastěji se v reálném světě hodnotící funkce pro hranu (u, v) odpovídá vzdálenosti bodů u a v, či doba potřebná pro přepravu mezi těmito body. Náš problém se orientuje na optimalizaci času, tedy budeme předpokládat, že ohodnocení hrany odpovídá času potřebnému na uražení jejich fyzické vzdálenosti a je zároveň přímo úměrné vzdálenosti, tedy že cestujeme konstantní rychlostí a zanedbáváme další parametry například stoupání či náročnost terénu. Zároveň však musíme brát v potaz zdržení v samotném vrcholu (místu zájmu) v daném čase. Tento časový údaj bude predikován predikčním modelem. Zatím si tento model můžeme pro jednoduchost představit jako funkci

$$p : V \times C \to \mathbb{R}^+$$

. Jedná se tedy o funkci vrcholu a času[4], ve kterém nás zdržení zajímá. Výsledná cena trasy se počítá jako součet ohodnocení hran a ohodnocení vrcholů v příslušných časech, viz následující příklad pro trasu $G(\{u, v, w\}, \{e, f\})$, kde $e = (u, v), f = (v, w)$ v čase vyražení $c \in C$

$$\omega(e) + p(v, c + \omega(e)) + \omega(f) + p(w, c + \omega(e) + \omega(f))$$

[4]Časem nebereme pouze čas v rámci dne, ale takový údaj, kterým jsme schopni určit všechny běžné časové údaje např datum, den v týdnu atd. Příkladem takového údaje mohou být unixové timestampy.

Pro jednodušší počítání můžeme vytvořit novou funkci hodnotící hrany v čase, která bude v sobě zahrnovat dobu zdržení v cílovém vrcholu

$$\omega' : E \times C \to \mathbb{R}^+$$

$$\omega'(e, c) = \omega(e) + p(v, c + \omega(e))$$

kde $e = (u, v); u, v \in V$ a čase $c \in C$. Nyní máme tedy jednu funkci, která nově má jako druhý parametr čas, ve kterém se hodnocení uvažuje. Počítání ohodnocení celé trasy však nyní není přímočaré, jako u klasického ohodnocení. Celkové ohodnocení cesty p a trasy t s počátečním časem c je možné spočítat pomocí následujících pseudokódu

function GetPathWeight(Path p, Time c, WeightFunction ω')
  $weight \leftarrow 0$
  for all *Edge* $e \in E(p)$ **do**
    $edgeWeight \leftarrow \omega'(e, c)$
    $weight \leftarrow weight + edgeWeight$
    $c \leftarrow c + edgeWeight$
  end for
  return $weight$
end function

function GetTourWeight(Tour t, Time c, WeightFunction ω')
  $weight \leftarrow$ GetPathWeight(t \ last(E(t)), c, ω')
  $e \leftarrow last(E(t))$
  $edgeWeight \leftarrow \omega(e)$
  $weight \leftarrow weight + edgeWeight$
  return $weight$
end function

Nyní můžeme problém přesněji dodefinovat jako nalezení orientované kružnice s minimálním ohodnocením $GetWeight$ v čase c na grafu s hodnotící funkcí ω'.

Kapitola 2

Analýza

2.1 Možnosti tvoření tras

V následující sekci se zaměříme na možnosti řešení TSP na dynamicky ohodnocených grafech.

2.1.1 Počítání pomocí naivního algoritmu

I dynamická varianta TSP se dá vyřešit pomocí naivního algoritmu. Po vytvořením všech permutací bodů zájmů stačí vybrat permutaci P s minimálním hodnotou $GetWeight((k_1, k_2, \ldots, k_n), c, \omega')$, kde $k_i \in P$; $i \in 1, 2, \ldots, n$. Opět by se zde dalo využít prořezávání, stejně jako klasické varianty TSP.
Složitost tohoto řešení se bude odvíjet od časové náročnosti predikční funkce p, pokud budeme však předpokládat konstantní složitost, získáváme $\mathcal{O}(n!)$.

2.1.2 Počítání pomocí dynamického programování

Další možným způsobem řešení je použití metody dynamického programování. Nyní se musí do zadání podproblému projevit i čas, ve kterém daný podproblém řešíme. Tímto se četnost opakování stejných podproblému výrazně zmenšuje. Tento problém lze eliminovat přidáním odchylky, se kterou budeme dva časy považovat za totožné.

2.1.3 Možnost využití aproximačních algoritmů

Základní podmínkou pro oba výše zmíněné c-aproximační algoritmy je platnost trojúhelníkové nerovnosti, jinak není zaručeno, že algoritmus bude c-aproximační. Pro náš problém je třeba tuto podmínku lehce upravit přidáním času na následující podmínku

$$\forall u, v, w \in V \ \wedge \forall c \in C : GetWeight((e), c, \omega') \leq GetWeight((f, g), c, \omega')$$

kde $e = (u, v), f = (u, w), g = (w, v); e, f, g \in E$.
Víme, že platí trojúhelníková nerovnost pro ohodnocovaní funkci ω, kterou si můžeme přepsat jako $\omega(e) + \varepsilon = \omega(f) + \omega(g)$, kde $\varepsilon \in \mathbb{R}^+$. Nyní po rozepsání nové podmínky a dosazením získáváme

$$\begin{aligned}
\omega'(e, c) \leq & \omega'(f, c) + \omega'(g, c + \omega'(f, c)) \\
\omega(e) + p(v, c + \omega(e)) \leq & \omega(f) + \omega(g) + p(w, c + \omega(f)) + p(v, c + \omega(g) + \omega(f) + \\
& + p(w, c + \omega(f))) \\
\omega(e) + p(v, c + \omega(e)) \leq & \omega(e) + \varepsilon + p(w, c + \omega(f)) + p(v, c + \omega(e) + \varepsilon + \\
& + p(w, c + \omega(f)))
\end{aligned}$$

Nejhorší případ, jaký může nastat, je, že $p(w, k)$ pro libovolné $k \in C$ se blíží nule. Pro jednoduchost tedy položíme $p(w, k) = 0$. Po upravení předchozí nerovnice dostáváme

$$\begin{aligned}
p(v, c + \omega(e)) &\leq \varepsilon + p(w, c + \omega(f)) + p(v, c + \omega(e) + \varepsilon + p(w, c + \omega(f))) \\
p(v, c + \omega(e)) &\leq \varepsilon + p(v, c + \omega(e) + \varepsilon) \\
\varepsilon &\geq p(v, c + \omega(e)) - p(v, c + \omega(e) + \varepsilon)
\end{aligned}$$

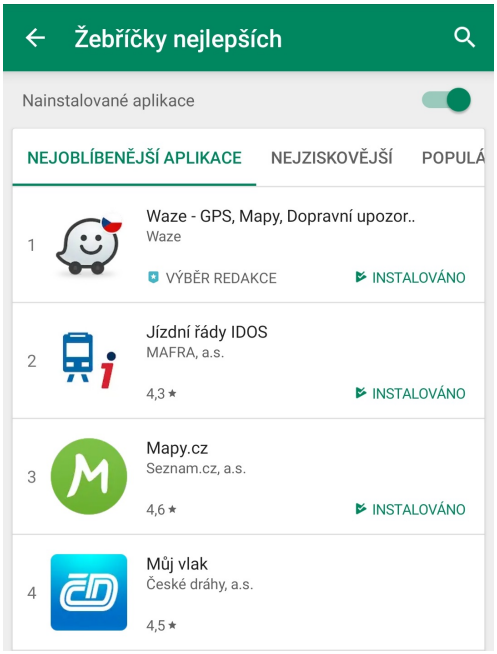
Pro zobecnění poslední nerovnice stačí $c + \omega(e)$ považovat za konstantu k.

$$p(v, k \ + \varepsilon) - p(v, k) \leq \varepsilon$$

Toto je výsledná podmínka pro predikční funkci. Tato podmínka by se také možné vyslovit jako: *Funkce predikce zdržení může nejvíce klesat lineárně.* Tato podmínka naopak nic neříká o tom, jak rychle může zdržení stoupat.
Všechny mnou popsané c-aproximační algoritmy také vyžadují konstrukci minimální kostry grafu. Je tedy třeba nelézt podobnou alternativu, díky kterou bych mohl využít a upravit stávající popsané algoritmy. Při použití optimálního větvení, což je orientovaná varianta minimální kostry, z počátečního vrcholu po dopočítání opačných hran nemáme zaručený žádný poměr vůči původnímu větvení, tedy ani nejsme schopni zaručit, že algoritmus bude c-aproximační pro nějaké c.
Další možností je využití různých heuristických algoritmů. Nabízí se vytvoření trasy, například pomocí již zmíněného optimálního větvení či hladovou heuristikou, a následné použití 2-opt algoritmu, který vzniklou trasu dále vylepší. Mezi další možnosti patří optimalizace mravenčí kolonií, kterou se dopodrobna zabývá článek [13].

2.2 Existující navigační systémy

Hlavním cílem této sekce je analyzovat již existující běžně používané navigační aplikace. Analýzu jsem rozdělil do dvou částí – jednu zaměřenou na analyzování vlastností a schopností, které využije běžný uživatel, a druhou analýzu z programátorského hlediska.
Rozhodl jsem se analyzovat 3 nejpoužívanější navigace na Android dle *Google Play Store* pro Česko. První dle počtu instalací je první *Google Maps* s počtem instalací přes 5 miliard[5], dále je *Waze* a třetí české *Mapy.cz*, viz obr. 2.1.



Obrázek 2.1: Screenshot Google Play žebříčku nejoblíbenějších aplikací pro ČR v kategorii „Mapy a navigace“ ke dni 10. 3. 2019
Pozn.: Google neuvádí Google Maps v žebříčku, nejspíše proto, že již bývá předinstalovaný na telefonech

[5] Celosvětově

2.2.1 Běžné užití

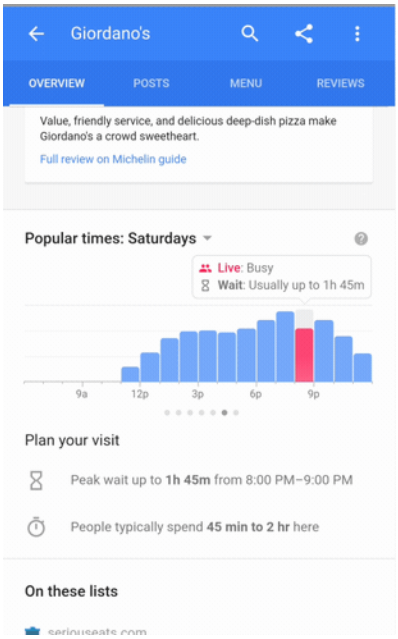
V této části se budu soustředit především na to, co daná aplikace umožňuje běžnému uživateli. Hlavní body analýzy jsou:

- Funguje aplikace pouze online nebo i offline?
- Je aplikace turn-by-turn navigace?
  Turn-by-turn navigace je taková navigace, která uživatele průběžně zadává pokyny podle aktuální polohy.
- Zohledňuje aplikace aktuální dopravní situací při hledání nejrychlejších cest?
- Jak aplikace zohledňuje historická data, především při nedostatku aktuálních informací o dopravní situaci?
- Nabízí aplikace během používání lepší trasu na základě nově vzniklých dopravních situací?
- Umožňuje aplikace z množiny míst, které chce uživatel navštívit nalézt nejkratší trasu?
- Umožňuje aplikace navrhnout opravu/změnu map či informací nabízených aplikací?
- Jaké další funkce aplikace nabízí.

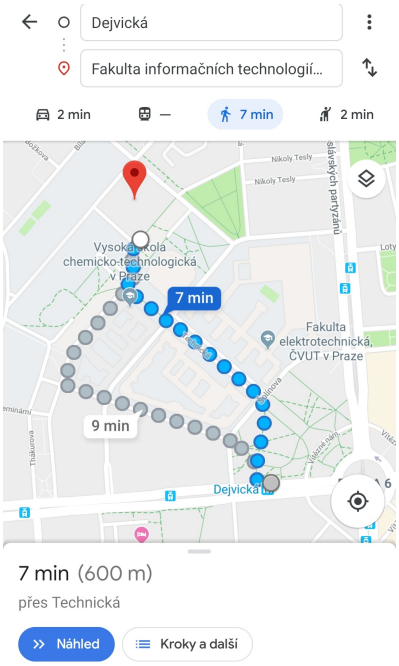
Některé tyto body, např. možnost navrhovaní oprav map, nejsou přímo spojené s mojí prací ale přesto byly po domluvě s vedoucím práce vybrány k analýze. Samotné aplikace jsem podrobil testování při přepravě v různých dopravných prostředcích po Praze.

2.2.1.1 Google Maps

Díky hojnému počtu uživatelů se dá předpokládat, že aplikace má celkem dobré informace o dopravních situacích, což se při analýze potvrdilo. Zároveň umí odhadnout pro jednotlivé hodiny, jak bude daná trasa zaplněná na základě historie. Během používaní navigace aplikace nabízí uživateli lepší trasy na základě nově vzniklých dopravních situacích. Uživatel si následně může vybrat, zda chce tuto novou trasu využít. Zároveň také nabízí odlišné trasy podobné délky a uživatel si může zvolit nabízenou alternativu. Veškeré informace o dopravní situaci se nevztahují na dopravu pěšky. Pokud je aplikace v offline režimu, nezohledňuje nikterak historické data, samotnou trasu však je schopna poskytnout, pokud si uživatel dopředu stáhl balíček s offline mapami.
Důležitou součástí aplikace je integrace s Google Places – aplikace poskytující informace o bodech zájmu, např. otevírací dobu nebo webové stránky. Krom



Obrázek 2.2: Google Maps - Populární časy a zaplněnost[14]



Obrázek 2.3: Google Maps - Navigace

toho pro některé body[6] jsou dostupné statistické údaje sbírané samotným Googlem o zaplněnosti a průměrné době strávené v daném místě pro jednotlivé otevírací hodiny, viz obr. 2.2. V případě, že aplikace detekuje, že uživatel byl v místě, o kterém má málo informací, požádá uživatele o vyplnění doplňujících informací, případně o recenzi místa. Krom těchto doplňujících informací navigace také nabízí navrhnout úpravy map, ať už se jedná o přidání uzavírek či jiné drobné úpravy.

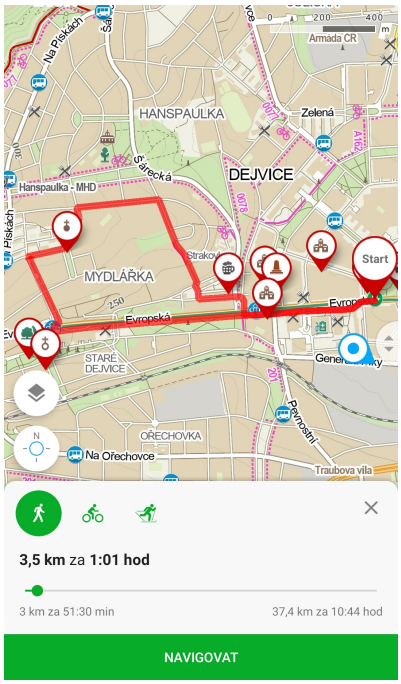
Aplikace neumožňuje zvolit si množinu bodů, ze které chce uživatel sestrojit nejkratší trasu. Aplikace vždy vyžaduje dané pořadí bodů, na kterých následně hledá nejkratší cesty.

Pro někoho velmi zajímavou, pro někoho nejspíše znepokojující, vlastností je, že při povolení získávání polohy si Google Maps zaznamenává, kde se uživatel pohybuje a na konci měsíce uživateli pošle email se souhrnem, který obsahuje místa, která uživatel navštívil, kolik nacestoval v dopravních prostředcích a kolik ušel pěšky. Tyto informace si může uživatel také zobrazit kdykoliv v položce „Vaše časová osa“.

[6] především obchody

2.2.1.2 Mapy.cz

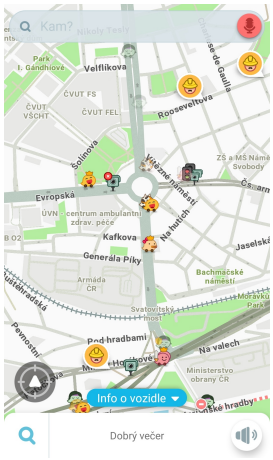
Tato česká aplikace od Seznamu patří mezi hojně využívané aplikace českými uživateli. Aplikace se zaměřuje převážně na Česko a mapuje jej mnohem detailněji než ostatní analyzované aplikace, obzvláště propracovanější jsou turistické trasy a cyklotrasy.
Součástí aplikace je real-time navigace, která byla dlouhou dobu v beta režimu, je však evidentní, že aplikace má primárně sloužit k naplánování trasy.
Aplikace nebere v potaz žádné aktuální dopravní situace, dokonce ani dopravní situace v minulosti, tím pádem nenabízí vždy nejrychlejší cesty a také během používání nenabízí žádné lepší cesty na základě aktuální situace.
Při vytváření tras je pořadí bodů stejné, v jakém byly zadány. Aplikace tedy nenabízí sestrojení nejkratší trasy z množiny bodů.
Po stažení balíčků s offline mapami je možné aplikaci využívat i bez připojení k internetu.
Pro návrhy úprav map slouží nástroj pro hlášení chyb, ve kterém stačí vyplnit návrh na úpravu a případně přiložit fotku či vyznačit část mapy.
Velice zajímavou funkcí je takzvaný „Výlet po okolí“, který vám podle zvoleného časového limitu sestaví trasu po zajímavých místech ve vašem okolí. Tato funkce je přístupná i v offline režimu.



Obrázek 2.4: Mapy.cz - plánování výletu po okolí

2.2.1.3 Waze

Navigace Waze, nyní patřící pod Google, se soustředí především přepravu osobním automobilem, naopak pro chodce je velice neoptimální, v mapách chybí chodníky a pěší zóny, a nepodporuje speciální pruhy pro kola, autobusy a taxy. Hlavní předností je možnost sdílení aktuálních dopravních situací, např. autonehody, v reálném čase uživateli. Podle toho pak Waze reaguje a může nasměrovat ostatní na lepší trasy. Také se snaží integrovat sociální aspekty a částečně i herní elementy, jako například zvyšování úrovně vaší postavy spojené s vaším účtem[15].
Aplikace obsahuje turn-by-turn navigaci, která je doprovázena mapou aktuálních situací, uzavírek a bouraček na silnici. Na základě těchto aktuálních dopravních situací pak aplikace hledá nejrychlejší trasy, případně upravuje současnou trasu. Součástí je i funkce nalezení nejlepší doby na odjezd pro zvolený čas dojezdu na základě historických dat a predikce z nich. Při používání navigace se doporučuje párkrát projet navrhovanou trasu, i když nemusí být optimální, aby aplikace získala data, a následně mohlo dojít ke korekci navrhované trasy[16].
Bez připojení k internetu aplikace nabízí navigaci bez predikce či dalších sociálních funkcionalit. Při testování offline režimu se mi načetla mapa obsahující převážnou část České republiky, nepodařilo se mi však zjistit, zda lze spravovat tyto stažené mapy a zda lze stáhnout větší část mapy.
Vytváření tras s více zastávkami je v této aplikaci celkem obtížné, uživatel nejprve musí zvolit cílovou destinaci a až poté může postupně přidávat zastávky. Aplikace uživatelovi při přidávání zastávek zároveň navrhuje restaurace či benzínové pumpy po cestě, vlastní zastávku však lze přidat pouze jednu. Pokud chce následně uživatel tento seznam zastávek měnit, musí celou trasu zrušit.



Obrázek 2.5: Aplikace Waze

Uživatel je aplikací pomocí sociálních aspektů vybízen přidávat návrhy na změny a informovat o dopravních situací. Toto je uživateli umožněno díky jednoduchému systému hlášení.

2.2.2 Programátorský backend

Při analýze jsem se zaměřil především na tyto informace:

- Jaké algoritmy daná aplikace používá?
- Jaké technologie daná aplikace používá?
- Umožňuje daná aplikace nalézt optimální trasu na množině bodů?
- Obsahuje dané aplikace predikci zpozdění?
- Umí daná aplikace reagovat na aktuální události a případně přesměrovat uživatele?
- Jaké další informace dané řešení shromažďuje a jak bych je mohl využít?
- Vystavuje daná aplikace surové informace?

Většinu informací jsem získával ze stránek samotných aplikací určené pro vývojáře, v některých případech jsem také čerpal z diskuzních fór, na kterých často odpovídají samotní vývojáři.

2.2.2.1 Google Maps

Google Maps nejsou open-source a jejich způsob hledání není veřejnosti známý, mohu tedy jen odhadovat, jaký algoritmus využívají. Dle mého odhadu pravděpodobně využívají algoritmus A* s optimalizovanou heuristickou funkcí. Krom toho také odhaduji, že z důvodů optimalizace mají trasy na některých částech grafů předpočítané.
Google nabízí velké množství webových API, co se týče map a navigace nad nimi nabízí *Google Map Directions API* a *Google Maps Places API*.
Directions API slouží k nalezení cest mezi dvěma zadanými místy, včetně možnosti přidávat zastávky. Odpověď obsahuje délku cesty a časový odhad včetně různých možností cest s podrobnými detaily. V parametrech požadavku lze nastavit, zda chceme brát v potaz aktuální dopravní situaci a zda se má použít pesimistický nebo optimistický model na základě historických dat či zda se má pokusit o co nejpřesnější odhad kombinací historických dat s aktuální dopravní situací. Ve vývojářské dokumentaci [17] je uvedeno, že čím blíže je doba začátku trasy, tím více při této predikci upřednostňují aktuální dopravní situaci. Tyto predikce jsou však dostupné pouze pro automobilovou dopravu. Zajímavostí je, že toto API také nabízí možnost optimalizovat pořadí zastávek,

tedy řešit TSP. Ve vývojářské dokumentaci [17] je uvedeno, že hlavním optimalizovaným parametrem je doba trasy, je však možné, že se budou v potaz brány další parametry, jako například počet odbočení.
Places API nabízí přístup do databáze bodů zájmů, pro tuto práci zajímavá tím, že skrze toho API je možné získávat detaily o místech zájmu[18], především otevírací dobu. API nenabízí možnost získávání dat o době zdržení a zaplněnosti v daných místech zájmu, i když tyto informace lze v aplikaci zobrazit. Existuje knihovna třetí strany, která tyto informace je schopna získat pomocí webové verze aplikace, tyto informace jsou však dostupné především pro obchody, nikoliv pro památky, na které se týmový projekt zaměřuje.
Google Maps do určité části odhaduje zpoždění na základě historických dat a také zohledňuje aktuální dopravní situaci, dá se tedy předpokládat, že má matematický model pro predikce zpoždění na cestě. K přesnosti jim pomáhá především velká uživatelská základnu, která jim dodává dostatek statistických údajů pro vytvoření predikce. Google však tyto informace o aktuálních událostech samostatně neposkytuje a zároveň neumožňuje dotazovat se na samostatnou predikci.

2.2.2.2 Mapy.cz

Samotný algoritmus hledání cest není opět veřejně známý. Můj odhad je, že využívají algoritmus A* s heuristikou pro hledání nejkratších cest. Pro tuto práci asi nejzajímavější je způsob pro hledání „Výletu po okolí“, který také nejspíše nebude zcela triviální.
Jelikož aplikace při plánování tras nezohledňuje aktuální ani historické dopravní události, lze předpokládat, že nemá žádný model predikce, nebo jej teprve připravují, případně testují.
Mapy.cz nabízejí vývojářům API na jejich služby. Přestože Mapy.cz obsahují nejvíce relevantních informací o památkách a bodech zájmů, na které se týmový projekt soustředí, Mapy.cz API je pro náš účel nepoužitelné, neboť neposkytuje žádná surová data[19]. API je spíše zaměřené pro zakomponování Mapy.cz pro zobrazování míst či tras na mapě.

2.2.2.3 Waze

Waze svoje algoritmy používané k hledání cest nezpřístupnilo veřejnosti, odhaduji tedy, že používají A* s heuristickou funkcí, která zároveň prioritizuje cesty, které například dlouho nikdo neprojel či jsou nově vzniklé.
Waze žádné API nenabízí, existuje pouze Waze SDK, které slouží k vytvoření aplikací integrující Waze. Existuje také způsob nahrávání dopravních informací partnery přes webový kanál v rámci „Connected Citizens Program“, kde jsou všechny informace uložené ve formátu XML nebo JSON a je zde možné ukládat informace o uzavírkách, dopravních nehodách a dokonce o pohybu sněžných pluhů či popelářských vozů[20].

Waze s největší pravděpodobností používá predikční model na základě historických dat pro doporučování času odjezdu, ale při real-time navigaci využívá především aktuální informace, na které také dokáže reagovat a upozornit uživatele na novou rychlejší trasu.

2.2.3 ČVUT Navigátor

Zvlášť jsem se rozhodl analyzovat ČVUT navigátor, projekt který vznikl na Fakultě informačních technologií za účelem usnadnit studentům navigaci po prostorách školy a kampusu.
Pro ČVUT navigátor vznikl modul [21] podobný této práci, ten se však zaměřoval primárně na hledání cest z bodu A do bodu B v rámci jedné vícepatrové budovy. Součástí modulu také byl sběr dat, která jsou následně v určenou dobu zapracována do grafu. Po zpracování grafu se přepočítá matice vzdáleností pomocí Floyd-Washallova algoritmu. Tento postup shledávám, přestože funkčním, nevhodným, neboť nereaguje na aktuální situace, navíc prohledávání pomocí Floyd-Washallova algoritmu je pomalé. Obor autora však byl Softwarové inženýrství, tedy návrh algoritmu nebyl hlavní náplní práce.
Důležitou součástí k analyzování jsou také mapové podklady pro celý ČVUT Navigátor. Na jejich vytváření, upravování a ukládání se zaměřuje další práce [22]. Výstupem práce je velmi přehledný a jednoduše použitelný systém, který není závislý na zbytku ČVUT Navigátora, tedy v budoucnosti je možné tuto práci v týmovém projektu využít. Celkem jednoduše se zde dají vytvářet podklady pro budovy na základě plánů a Google Map pro správné umístění. Systém je sice orientovaný na vytváření podkladů pro budovy s možností více pater, pro náš projekt stačí vytvořit jedno patrovou budou reprezentující celou turistickou oblast, pro kterou bude mapa určena.

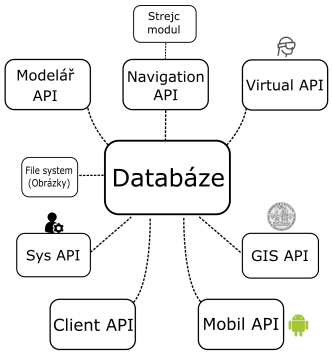
2.3 Predikční model

Cílem predikčního modelu bude ohodnocovat vrchol grafu v čase na základě dříve nasbíraných dat. Toto ohodnocení bude reprezentovat předpokládané zdržení a budeme předpokládat, že je přímo úměrné zaplněnosti daného vrcholu.
Pro takovýto predikční model by se dala vycvičit umělá neuronová síť, vycvičení takové sítě by však bylo na samostatnou práci a proto se zaměřím na vytvoření jednoduššího matematického modelu.
K této predikci využijeme nasbíraná data, která se vhodně rozdělíme do časových intervalů a předzpracujeme pro urychlení celého výpočtu. Zvolení samotného základního časového intervalu, po kterém budeme data zpracovávat zásadně ovlivní kolik informací se bude ukládat. Pokud zvolíme krátký interval, skladovaná data budou velmi podobná, naopak při dlouhém intervalu ztrácíme

přesnost predikce. Zároveň je třeba si určit, jak dlouho budeme informace skladovat, a jaký formát použít. Problémem ukládání dat potřebných pro přepravu se zabývá také následující práce[23], je však zaměřená na ukládání informací o spojích veřejné dopravy.
Za závěr bude také třeba nasbíraná data také ošetřit od tzv. *outlierů*, což jsou extrémní případy, které se odlišují od ostatních vzorků, neboť by mohly zásadně ovlivnit budoucí predikce. Příkladem takového outlieru je člověk, který se například rozhodně v určitý moment zůstat na místě a občerstvit se, probíhající trasu by však stále nechal aktivní. Tím může zásadně ovlivnit průměrný čas strávený na daném místě při běžné prohlídce.
Samotný predikční model následně pro vrchol a čas spočítá určitou predikci. Jelikož data budeme zpracovávat po určitém intervalu, predikce se do dalšího zpracování měnit nebude.

2.4 Původní návrh zapojení do projektu

První návrhy celé architektury, podle kterých vznikalo i zadání této práce, jsou ke dni odevzdávání této práce již přes rok a půl staré a s postupným vývojem se i tento návrh měnil. Dle původního návrhu měl navigační modul



Obrázek 2.6: Původní návrh architektury týmového projektu *Virtuální historický průvodce*

Autor: Daniel Vančura

komunikovat skrze API, které bude poskytovat jádro. Toto se v průběhu však ukázalo jako nevhodný návrh, neboť se má jádro naopak dotazovat modulu na zjištění trasy.
Součástí modulu tedy bude vlastní API, které bude umožňovat jádru dotazovat se na trasy. Jelikož výpočty při větším počtu zastávek na trase mohou být časově náročné a navrhovaná trasa se s postupem času může měnit, je třeba využít takové rozhraní, které dovoluje vytvořit takovýto dotaz a následně se

na něj nezávisle dotazovat či měnit a mazat. *REST API*, které se používá i v dalších částech projektu, přesně tyto požadavky splňuje.

2.5 Funkční/nefunkční požadavky

2.5.1 Funkční požadavky

Funkční požadavky definují, které funkce bude software plnit.

F1 – Výpočet trasy podle zadaných parametrů
Podle zadaných parametrů vypočítat optimální trasu. Mezi parametry patří počáteční bod, množina bodů, které chce uživatel navštívit a zda chce uživatel skončit opět v počátečním bodu.

F2 – Nabízení lepších tras v průběhu trasy
Umožnit nabízení nově vzniklých lepších tras (např. při aktualizaci zaplnění památky na trase).

F3 – Automatické přepočítávání podle potřeb uživatele
Dle aktuálních dat a nových požadavků uživatele se budou trasy v hodnou chvíli přepočítávat.

F4 – Nabízet body zájmu, které jsou blízko trasy
Při objevení bodů zájmů, které jsou poblíž naplánované trasy, nabídnout uživateli přidání těchto bodů do trasy. Uživatel následně zvolí podmnožinu nabízených bodů a modul musí upravit trasu.

2.5.2 Nefunkční požadavky

Nefunkční požadavky definují technické požadavky softwaru.

N1 – Komunikace pomocí REST API
Modul bude komunikovat s jádrem aplikace pomocí REST API.

N2 – Modul poběží jako služba
Modul bude realizován jako služba na serveru, která může nepřetržitě běžet na pozadí.

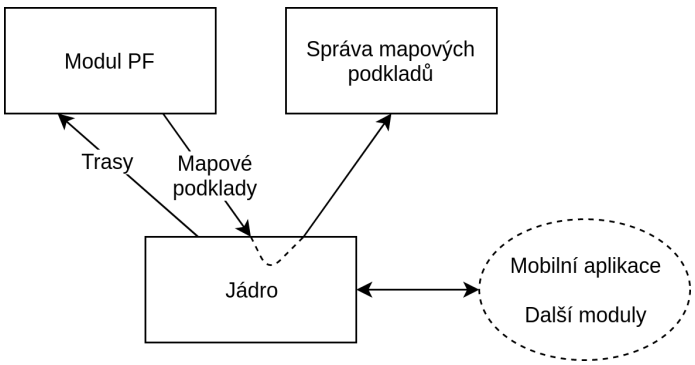
N3 – Výpočty budou bezstavové
Mezi jednotlivými požadavky se nebude udržovat žádný stav, všechny případná data se budou ukládat do databáze. Nikdy by však předchozí výpočty různých tras neměli ovlivnit následující výpočty.

Kapitola 3

Návrh

3.1 Zapojení do projektu

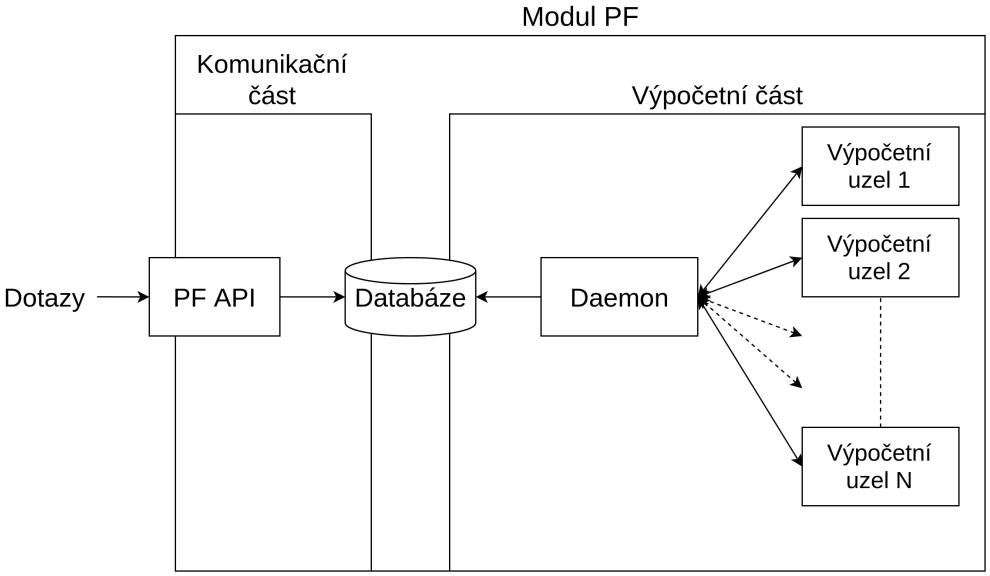
Celý projekt Virtuální průvodce je rozdělen do jednotlivých částí, které spojuje „Jádro“, se kterým všechny moduly komunikují. I můj modul bude komunikovat pouze s jádrem. Rozhraním této komunikace, které se vybralo jako standart celého projektu, je RESP API. Proto i můj modul bude podléhat tomuto standartu.
Hlavní komponentou celého projektu je jádro, které bude sloužit jako řídící prvek celé aplikace a ostatní funkce budou obsluhovat moduly, které budou komunikovat pouze s jádrem. Díky tomuto návrhu nemusí jednotlivé moduly o ostatních modulech vědět a tedy odpadá složitá konfigurace. Hlavní náplní práce modulu z pohledu jádra bude obsluhovat dotazy na trasy. Jádro tedy pošle dotaz na REST API mého modulu a bude se následně dotazovat na výsledek. Naopak můj modul se bude jádra dotazovat na potřebné mapové podklady. Jádro samotné nebude mapové podklady spravovat, tuto práci přenechá na modul k tomuto určenému, a bude fungovat jako spojovací prvek mezi mým modulem a příslušným modulem.



Obrázek 3.1: Napojení na jádro

3.2 Návrh architektury modulu

Celý modul rozděluji na dvě logické části - **komunikační** a **výpočetní** část. Vše bude spojovat databáze, do které budou mít obě části přístup a budou jí využívat k vzájemné výměnně dat.



Obrázek 3.2: Architektura modulu

Komunikační část bude obstarávat obsluhování dotazů, tzn. validaci vstupu, ukládání nových požadavků na trasu do databáze a odpovídat na samotné dotazy a bude implementována jako REST API. Zároveň bude pro udržení čistoty databáze promazávat dotazy, na které se již nikdo dlouho nezeptal a zapomněl, případně nemohl například z důvodu ztracení spojení, poslat požadavek na smazání.

Výpočetní část bude složena z potenciálně několika výpočetních uzlů a daemona, který bude průběžně nové dotazy na trasu uložené v databázi rozdělovat jednotlivým uzlům a po dokončení výpočtu opět uloží výsledky do databáze. Daemon může taky během doby, kdy jsou některé uzly nevyužity, rozhodnou o přepočítání z některých stále aktivních tras. Pokud by došlo k náhlému přívalu nových dotazů, daemon může také uzlům, které přepočítávají, přikázat přerušení daných zadání a začít počítat nové dotazy. Toto všechno umožní nejen objevení lepší trasy na základě aktuálních dat, ale zároveň lepší využití výpočetní síly s minimálním zdržováním výpočtů nových dotazů.

3.3 REST API

Jelikož můj modul bude pouze nabízet možnost dotazování na trasy, stačí, aby REST API mělo pouze jeden příslušný end-point - **/tours** - s CRUD (Create, Read, Update, Delete) operacemi nad jednotlivými dotazy na trasy. Jako formát předávaných dat jsem se rozhodl použít JSON, který je běžný pro REST komunikaci.
POST `/tours` bude sloužit k vytvoření dotazu na trasu. V těle dotazu bude množina bodů na mapě, které mají na trase být. Odpovědí je přidělené ID sloužící k přistupování dokumentu odpovídající danému dotazu. Zároveň hlavička odpovědi obsahuje položku `Location`, ve které je relativní cesta k danému dokumentu.
GET `/tours/{id}` slouží k přistupování dokumentu dotazu. V těle odpovědi je stav a případně vypočítaná trasa.
DELETE `/tours/{id}` maže samotný dokument.
PUT `/tours/{id}` slouží k aktualizování dokumentu. Bude se jednat především o změny zadání či aktualizaci současné polohy uživatele.
Podrobnější popis samotného API ve formátu Swagger je součástí elektronické přílohy.

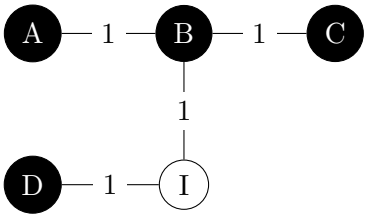
3.4 Sběr dat

Sbírání dat bude prováděno přes REST API, kdy bude uživatelská aplikace v pravidelných intervalech zasílat `PUT` požadavky s aktuální pozicí uživatele. Tyto změny budou zaznamenávány a při detekování odchodu z bodu zájmu se spočítá celková doba strávená na daném místě. Každých 10 minut se všechny tyto naměřené časy pro každý vrchol vyčistí od outlierů a následně se spočítá průměr. Tyto hodnoty se budou ukládat do databáze. Celkově se budou ukládat zprůměrované hodnoty v 10 minutových intervalech po dobu jednoho roku. Teoretická maximální paměť potřebovaná na uložení naměřených hodnot na vrchol při použití 32-bitového čísla je $6 * 24 * 365 * 32$bitů, což se rovná přibližně 206KiB. Pokud však pro daný vrchol nebudou v daném rozsahu žádné naměřené hodnoty nebude se do databáze nic zapisovat a velikost záznamů tím bude daleko menší, například už jenom proto, že neočekáváme využití v nočních hodinách.
Během nočních hodin, kdy modul nebude využívaný, se zaintegrují naměřené hodnoty z posledního dne do celkového průměru pro daný vrchol, který se bude využívat v případě, že nejsou pro některé časové úseky dostupná data.
Trasy se identifikují přiděleným ID, o uživateli můj model nic neví. Není tedy možnost spojit uživatele s nasbíranými daty či s dotazy na trasu, proto se není třeba zabývat ochranou osobních údajů.
Na závěr je potřeba zvolit vhodný způsob detekování outlierů. Můžeme například zvolit práh, při kterém řekneme, že jsou data nevalidní. Tento postup by však

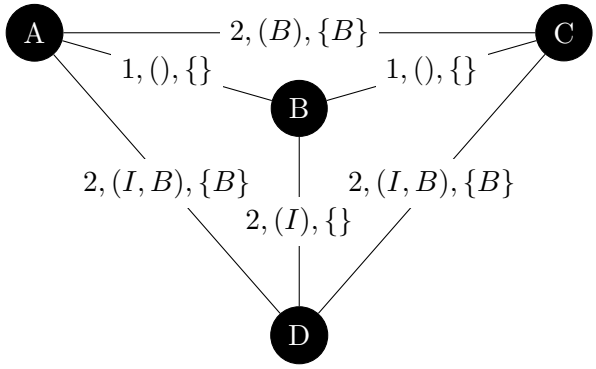
vyžadoval pečlivě vybírat tyto meze pro každý bod zájmu. Přesto je jedná o velice hrubý filter, který neodstraní příliš mnoho extrémů. Další možností je využít *Local outlier faktor*, který pomocí vzdáleností od k-nejbližších sousedů odhadnout, zda se jedná o extrém[24], který následně můžeme odstranit. Zde je tedy důležité zvolit správnou hodnotu k tak, aby nedocházelo ke zkreslování. Při nízkém k detekujeme obecně méně outlierů a při vysokém k bude výpočet pomalý. Dost také záleží na velikosti nasbíraných dat. Proto navrhuji ze začátku $k = 3$, je však pravděpodobné, že bude potřeba tuto hodnotu upravit na základě reálných dat Pokud se naměří méně než k vzorků, všechny zahrneme do průměru.

3.5 Předzpracování grafu

Pro jednodušší a rychlejší počítání výsledných tras je vhodné vytvořit si úplný graf všech bodů zájmů, kde hrana reprezentuje nejkratší cestu. Je však nutné, aby z tohoto grafu bylo možné danou cestu zrekonstruovat. Takový graf bude generovaný z mapy, kterou bude nabízet modul pro správu mapových podkladů, nalezením všech nejkratších cest mezi body zájmu, vytvořením nového grafu $G(POI, \binom{POI}{2})$, kde POI je množina vrcholů označených jako body zájmu s ohodnocením hran odpovídají ohodnocení nejkratší cesty mezi danými vrcholy. Dále se bude uchovávat samotná nejkratší cesta (stačí pouze jako list navštívených bodů). Pro každou cestu se také zapíšou další body zájmu, které na nejkratší cestě náleží, aby mohly být nabídnuty jako možné zastávky v případě, že tudy trasa povede a zatím nebyly tyto zastávky naplánovány.



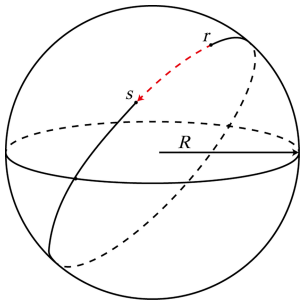
(a) Příklad mapy, černě jsou označené body zájmu



(b) Výsledný úplný graf graf

Tento graf je třeba vytvořit pokaždé, když se mapy změní. Vhledem k tomu, že se jedná o pěší mapu a body zájmu jsou památky, které často nevznikají nebo nezanikají, bude se jednat pouze o občasné změny. Toto přepočítávání může probíhat například v noci, kdy se neočekávají žádné požadavky.
Pro vytvoření prvotního grafu či přepočítání při velkých změnách je vhodné využít Floyd-Warshallův algoritmus, který najde nejkratší cestu mezi všemi

vrcholy. V případě přidání nového bodu zájmu navrhuji použít A* s heuristickou funkcí vzdálenosti od cílového bodu pro nalezení nejkratších cest k ostatním bodům zájmu, neboť již není třeba přepočítávat ostatní cesty, u kterých jediná potřebná úprava může být přidání nového bodu zájmu do množiny navrhovaných bodů.

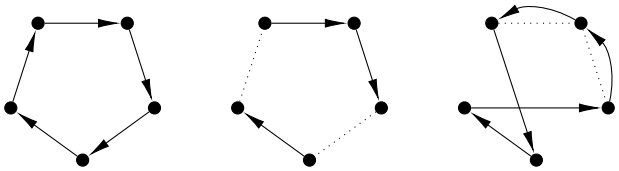


Obrázek 3.4: Ukázka ortodromy[25]

Samotné ohodnocování hran se bude počítat jako velikosti ortodromy mezi vrcholy dělená průměrnou rychlostí chůze, což je přibližně 5 km/h. Díky uvedenému ohodnocování zaručíme platnost trojúhelníkové nerovnosti, která při používání Euklidovské vzdálenosti nemusí pro všechny souřadnice platit, a i když samotné trasy nejspíše nebudou dostatečně daleko od sebe, aby se tyto problémy projevili, budou výpočty přesnější. Podrobný popis a algoritmus pro počítání velikosti ortodromy můžeme nalézt např. zde [26].

3.6 Návrh vytváření tras

Pro samotné vytváření tras nad předzpracovaným úplným grafem navrhuji použití vytváření trasy pomocí tzv. hladové heuristiky, která vždy vybírá nejbližší vrchol. Toto řešení je samo o sobě rychlé, není však dostatečně přesné. Navíc navrhuji použití modifikovaného 2-opt algoritmu, který případně takto vytvořenou trasu vylepší.
Samotné modifikace 2-opt algoritmu spočívají v upravení pro orientované kružnice. Hlavním rozdílem oproti původnímu algoritmu je nutnost otočení orientace některé ze dvou vzniklých částí.



Obrázek 3.5: Příklad otočení orientace při 2-opt kroku

Jelikož se změnou orientace zásadně mění ohodnocení a tedy i čas, ve kterém se napojuje zbytek trasy, cena cesty se bude muset často, i když některé části lze předpočítat, a budeme tedy předpokládat, že každý krok zabere $\mathcal{O}(n)$ času pro spočítání výsledné ceny trasy. Jelikož zkoušíme všechny nesousedící dvojice, tedy $\mathcal{O}(n^2)$ dvojic, algoritmus má asymptotickou složitost $\mathcal{O}(n^3)$

3.7 Návrh výpočetního části

Jak jsem již zmiňovat, výpočetní část bude rozdělena na výpočetní uzly a démona. Pro jednoduchost zatím budeme předpokládat, že budou tyto dvě části sdílenou paměť, aby komunikacemi mezi těmito částmi byla jednodušší. Do budoucna by bylo možné i toto rozšířit například pomocí open-source knihovny `Open MPI`, která je určená k usnadnění komunikace procesů bez sdílené paměti.
Výpočetní uzly budou vyzvedávat z fronty zadání, které mají vypracovat. Po dokončení výpočtu zapíší výsledek a čekají, dokud není ve frontě další zadání. O svém stavu vždy informují hlavní vlákno, daemona, aby mělo přehled, jaké je vytížení.
Daemon bude vždy v pravidelných intervalech kontrolovat, zda není nový dotaz v databázi či zda není hotové řešení na nějaký dotaz. V případě nového dotazu vytvoří zadání a přidá jej do fronty pro výpočetní uzly. Naopak pokud je nový výsledek pro dotaz, zapíše jej do databáze. Jestliže jsou stále aktivní nějaké dotazy, ke kterým jsou již dostupné aktuální dopravní informace, a není velké vytížení uzlů, může zadat přepočítat vybraný dotaz.

3.8 Návrh predikčního modelu

Jak jsem již stanovil v analýze, predikční model bude přiřazovat nezáporná ohodnocení vrcholům v určitém čase, na základě historických dat, do kterých se počítají i poslední zpracované hodnoty. Rozhodl jsem se, že predikce se bude počítat jako $p(v, c) = \sum_{i=0}^{n} \alpha_i h_i(v, c)$, kde α je vektor kladných koeficientů pro který platí $\sum_{i=0}^{n} \alpha_i = 1$. Funkce h_i vrací pro daný vrchol zdržení zjištěné v čase $c - konstanta$, která je určená dané funkci. Příkladem jednou z těchto funkcí může být zpoždění v tomto vrcholu před 24 hodinami. Je také důležité rozhodnou, co se stane, pokud v dané době není naměřena žádná hodnota. V takovém případě se dosadí průměrná hodnota pro daný vrchol.
Jelikož jsem navrhl tvoření tras pomocí algoritmu, který nevyžaduje platnost trojúhelníkové nerovnosti, není třeba v tomto bodě řešit, zda predikční model splňuje podmínku stanovenou v části 2.1.3. Pokud by se v budoucnosti algoritmus měnil na takový, který vyžaduje tuto nerovnost, musel by se přizpůsobit a model.
V tabulce 3.1 jsou uvedené experimentálně zvolené časové posuny funkcí h_i

i	Časový posun	Koeficient α
1	10 minut	0.3
2	hodina	0.2
3	den	0.2
4	týden	0.15
5	měsíc	0.1
6	rok	0.05

Tabulka 3.1: Tabulka vybraných koeficientů a časových konstant pro predikci

a jejich koeficienty.

Takto zvolené koeficienty budou upřednostňovat aktuální informace (posledních 10 minut) a stejnou denní dobu, ale zároveň je ponechán prostor pro trendy opakující se každý měsíc či rok. Pro lepší přesnost bude třeba nasbírat reálná data v provozu a následně upravit zvolené koeficienty, případně přidat či odebrat další funkce.

Kapitola 4

Realizace

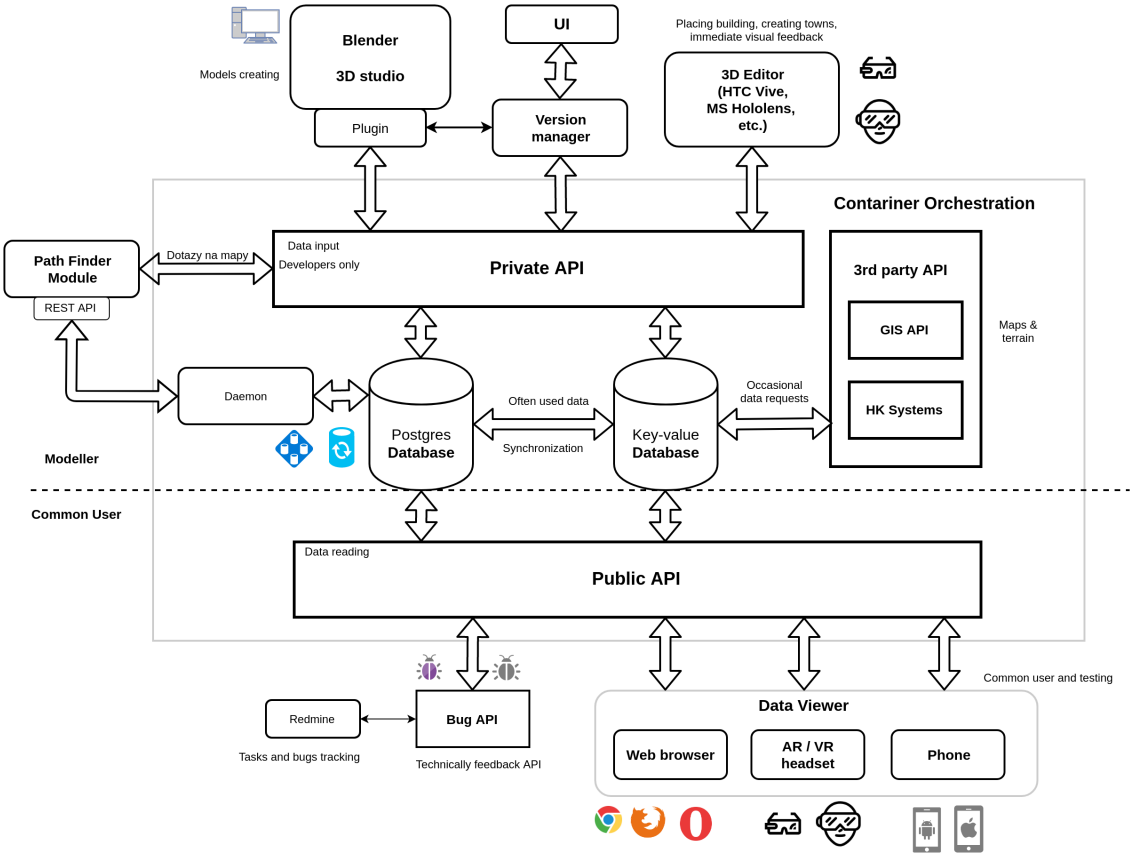
4.1 Zapojení do projektu

Jelikož je projekt stále ve vývoji, některé moduly, jako například modul pro správu mapových podkladů, nejsou hotové. Proto je implementace nekompletní, neboť není stanovené, jaké rozhraní budou jednotlivé části nabízet. Součástí mojí implementace je API, které původně nemělo být součástí a mělo se implementovat v rámci jádra. Jak jsem však již v analýze vysvětlil, jednalo se o starý návrh, který se prokázal jako nevhodný.
Na obrázku 4.1 je diagram reprezentující stav projektu po napojení navigačního modulu.

4.2 Použité technologie

Díky tomu, že modul bude běžet jako samostatná aplikace a bude komunikovat pomocí API, nejsem nikterak svázán zbytkem týmového projektu a mohl jsem se rozhodnout, jaké technologie budou použity.
Vzhledem k povaze modulu je vhodné využít kompilovaný jazyk, neboť jsou zpravidla rychlejší než jazyky interpretované. V úvahu připadaly jazyky C a C++, se kterými mám již zkušenost z předchozího studia. Následně je tedy třeba odhadnout případnou složitost implementace RESP API v obou jazycích. Pro jazyk C se jedná o celkem nízkoúrovňové programování pomocí socketů a POSIX threadů[7], naopak pro C++ existuje hotová open-source knihovna C++ *REST SDK*, dříve známá jako projekt „Casablanca“, od společnosti Microsoft poskytující moderní asynchronní rozhraní ve standard C++11.
Další potřebou je možnost komunikace s databází. Jako hlavní databázi pro ukládání dat jsem vybral dokumentovou NoSQL databázi *MongoDB*, mezi

[7]Existují i určité knihovny od uživatelů Githubu, zde však se nemůžeme spoléhat na správnost a bezpečnost knihovny



Obrázek 4.1: Diagram napojení do projektu
Autor: Daniel Vančura

jehož hlavní výhodu patří dobrá škálovatelnost a jednoduchá replikace databáze. Pro komunikaci s databází existuje knihovna pro *libmongoc* C i *libmongocxx* pro C++. Distribuce Linuxu, která se dosavadně používají pro testování a postupné nasazování zbytku projektu, Debian 9 není připravený balíček s C++ verzí knihovny, proto jsem se rozhodl zůstat u verze pro C, neboť by v budoucnosti bylo potenciálně mnohem složitější nasadit modul na nový server.

Na závěr bylo třeba vyřešit konfiguraci modulu, jako například přístupové informace do databáze či klíče. Kritériem je, aby se dal modul jednoduše konfigurovat pomocí dobře lidsky čitelného konfiguračního souboru. Při hledání možných hotových řešení jsem narazil na open-source knihovny *libconfig* a *Config4Cpp*. *Libconfig* je celkem jednoduchá knihovna pro C/C++ a používá vlastní formát pro konfigurační soubor velice podobný formátu JSONu. Knihovna je také minimalistická, tudíž je vhodná pro systémy omezené velikostí paměti[27]. *Config4Cpp* patří do rodiny parserů konfiguračních souborů *Config4**, která zatím zastřešuje ještě parser pro Javu. Postupem času by se měla rodina těchto

parserů rozšířit[28], tedy hlavní výhodou je podobnost rozhraní pro různé jazyky a tedy použití v jiném jazyce neobnáší učení se celé nové knihovny. Nakonec jsem tedy zvolil minimalističtější a jednoduše dostupnou[8] knihovnu *libconfig*.

4.3 Dokumentace

Zdrojové kódy jsou dokumentované pomocí komentářů ve stylu *Doxygen*, které se následně stejnojmenným programem vygenerovat ve formátu HTML stránek či PDF. Součástí přílohy je vygenerovaná dokumentace ve formátu HTML.

```
/**
* Abstract class for Dynamic TSP solver
*/
class DynamicTSPSolver {
        public:
        /**
        * Solves dynamic TSP
        * @param nodes Nodes we want to visit on the tour
        * @param startNode Starting node
        * @param endInStart Whether to finish back in start
        * @param time Starting time
        * @param graph Dynamically weighted FULL graph
        * @return
        */
        virtual RouteInfo solveTSP(const std::vector<Node>& nodes,
                const Node& startNode, bool endInStart,
                const std::chrono::system_clock::time_point& time,
                DynamicGraph& graph) = 0;
}
```

Listing 4.1: Ukázka dokumentace pomocí komentářů

4.4 API

Jak již bylo zmíněno, API jsem se rozhodl realizovat pomocí knihovny *REST SDK*, která byla pro tento účel vytvořena. Knihovna dovoluje vytvořit tzv. `listener`, který obsluhuje danou URL cestu a pro jednotlivé HTTP metody můžete přiřadit funkci, která následně asynchronně požadavek zpracuje. Při zpracovávání validních požadavků se přistupuje do databáze, knihovna pro práci s Mongo databází však není „thread safe“, neboli pokud k některému z objektů přistupuje více vláken, není zaručené, že nedojde k poškození paměti. Proto je třeba, aby se při každém zpracování požadavku vytvořil nový MongoDB client.
Poslat dotaz na API z terminálu lze například pomocí příkazu 4.2 či 4.3.

[8] Pomocí balíčkového systému *apt* na linuxových distribucích založených na *Debianu*

```
curl -d '{"places":[{"POI_id": 1}, {"POI_id": 2}, {"POI_id": 3}],
        "end_in_start": true}' -H "Content-Type: application/json"
        -X POST http://localhost:8080/tours
```

Listing 4.2: Příklad curl příkazu pro POST na REST API

```
curl -H "Content-Type: application/json"
        -X GET http://localhost:8080/tours/507f1f77bcf86cd799439011
```

Listing 4.3: Příklad curl příkazu pro GET na REST API

4.4.1 Konfigurační soubor

API má svůj konfigurační soubor implicitně nazvaný `api.conf` a je očekávaný v aktuálním pracovním adresáři. Pro specifikaci vlastní cesty či jména, stačí jako argument spustitelnému souboru zadat cestu k požadovanému konfiguračnímu souboru.
Samotná konfigurace je rozdělena do dvou částí – konfigurace přístupových údajů do databáze a nastavení adresy, na které má API naslouchat.

```
Api : {
        host = "http://localhost"
        port = 8080;
};

MongoDb : {
        host = "localhost";
        port = 27017;
        database = "Path_finder";
};
```

Listing 4.4: Příklad konfiguračního souboru api.conf

4.5 Výpočetní část

Při programování této části jsem využil objektově orientované programování, hlavně díky možnostem dědění a polymorfismu k vytvoření abstraktních tříd představujících různé grafy, implementace algoritmů či komunikační protokoly. Následně je jednoduché zaměnit jednotlivé implementace splňující zadané rozhraní bez nutnosti upravování kódu, který tyto třídy volá.
Implementoval jsem A*, Dijkstrův a Floyd-Washallův algoritmus pro hledání nejkratších cest, naivní a 2-aproximační algoritmus pro TSP včetně řešení využívající dynamické programování. Pro dynamické TSP byly implementovány naivní algoritmus s prořezáváním a algoritmus využívající hladovou heuristiku. Taktéž je implementován zlepšující 2-opt algoritmus pro orientované kružnice. Navíc jsem algoritmům pro řešení dynamického TSP přidal možnost neukončovat trasu v počátečním bodu.

Pro vytváření vláken ve výpočetní části jsem se rozhodl použít `std::thread`, které jsou součástí C++11. Hlavní vlákno vytvoří potřebné implementované thread-safe fronty na zadání a výsledky v paměti a spustí výpočetní vlákna, která si následně z fronty berou zadání.

4.5.1 Konfigurační soubor

Konfigurační soubor výpočetní části má implicitní název `pathfinder.conf` a program očekává, že tento soubor je umístěn v aktuálním pracovním adresáři. Pro specifikaci jiného konfiguračního souboru, stačí předat cestu k souboru jako argument při spuštění.
Souboru je rozdělen do sekce pro konfiguraci počtu výpočetních vláken, sekce obsahující přístupové údaje do databáze společné s API. Připravena je také část určená pro konfiguraci přístupu k mapovým podkladům.

```
PathFinder : {
        threadCount = 4;
}

Maps : {
        host = "remotehost";
        port = 8080;
}

MongoDb : {
        host = "localhost";
        port = 27017;
        database = "Path_finder";
};
```

Listing 4.5: Příklad konfiguračního souboru pathfinder.conf

4.6 Zprovoznění projektu

Pro zprovoznění projektu je nejprve třeba jednotlivé části zkompilovat. Součástí zdrojových souborů se soubor `CMakeList.txt`, který slouží ve spojení s programem *cmake* k vyřešení všech závislostí a jednoduché správě nastavení jednotlivých spustitelných souborů. Ke zkompilování je třeba mít nainstalovanou knihovnu *CPP REST SDK* a všechny její závislosti, viz [29]. Po spuštění příkazu `cmake` je k dispozici `Makefile`, který se již obsluhuje pomocí příkazu `make`.
Po zkompilování spustitelných souborů *api* a *path_finder*, které stačí spustit, případně přiřadit do vašeho správce služeb ve vybrané distribuci.

Kapitola 5

Testování

Hlavním cílem testování bylo ověřit časovou a paměťovou složitost implementovaných algoritmů. Zaměřil jsem se hlavně na testování algoritmů pro hledání tras, neboť toto hledání bude hlavní náplní modulu.
Celé testování probíhalo na počítači s procesorem Intel(R) Core(TM) i7-6700K CPU @ 4.00GHz s 8 vlákny, 16GB RAM a operačním systémem Debian 9.

5.1 Testovací data

V současnosti nejsou připravené mapové podklady ani zvoleny body zájmů, které budou pro ve výsledné aplikaci použity. Proto jsem si vytvořil vlastní testovací množinu. Do této množiny jsem zařadil největší města počtem obyvatel České republiky a jejich souřadnice. Celkem se mi podařilo získat 88 měst a jejich GPS souřadnice [9] a přidal jsem jako startovní bod místo mého pobytu. Následně jsem si vytvořil úplný graf nad těmito body[10] a uložil je do lokální Mongo databáze, aby se při spouštění nemuselo vše znovu počítat. Pro testování následně náhodně vybereme požadované množství bodů[11], na kterých budeme trasu hledat.
Predikci jsem vzhledem k absenci dat pro testovací potřeby nahradil náhodnými hodnotami v normální rozdělení se střední hodnotou 3 minuty a rozptylem 1 minutou[12]. Toto bude simulovat fluktuaci v zaplněnosti daných vrcholů.

5.2 Porovnávané algoritmy

Hlavním testovaným algoritmem je navrhovaný postup tvoření tras, tedy pomocí hladové heuristiky vytvořit trasu a následně využít 2-opt algoritmu tuto

[9] i když teoreticky by stačilo pouze vygenerovat náhodné souřadnice, chtěl jsem, aby se data podobala reálným
[10] celkem tedy $\binom{89}{2}$ vzdáleností
[11] bez opakování
[12] Tyto hodnoty byly zvoleny experimentálně

trasu vylepšit. Pro porovnání jsem testoval naivní řešené, které zároveň slouží k určení odchylky předchozího algoritmu vůči optimálnímu řešení. K tomuto porovnání přidávám i trasu vytvořenou hladovým algoritmem, netestuji však tento algoritmus na časovou a paměťovou složitost.

5.3 Časové testování

Pro testování časové náročnosti využívám systémového času pomocí následujícího kódu, který vypíše na standardní výstup dobu výpočtu v milisekundách. Zvolil jsem tuto metodu naproti využití terminálového příkazu `time`, neboť bych měřil dobu běhu celého programu namísto zkoumaných částí kódů.

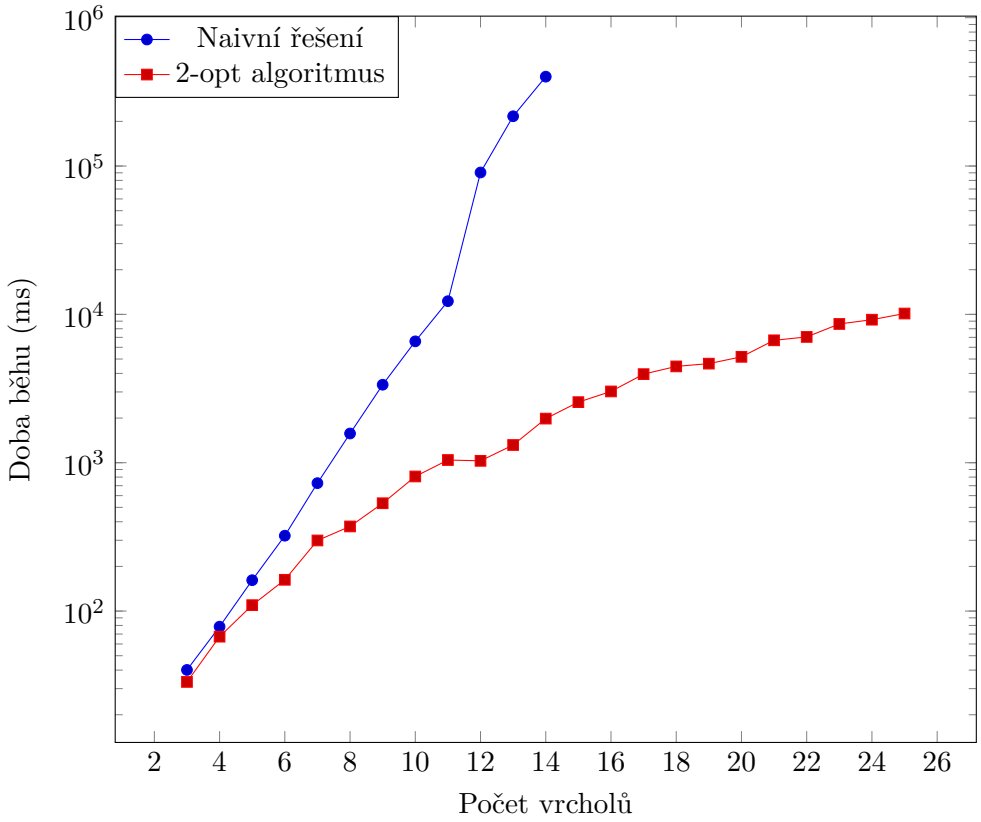
```
using namespace std;
using namespace std::chrono;

auto startT = system_clock::now();
/*
* execute code
*/
auto endT = system_clock::now();
long diff = duration_cast<milliseconds>(endT-startT).count();
cout<<diff<<endl;
```

Tímto jsem obalil samotné volání funkce, která řeší tvoření tras na dynamicky ohodnoceném grafu.
Každý test jsem spouštěl 10krát a následně zprůměroval výsledné časy, abych snížil možnou odchylku způsobenou zatížením procesoru operačním systémem. Měřil jsem časy pro různé počty vrcholů, počínaje 3 a konče 25. Maximální počet vrcholů, pro který jsem měřil naivní algoritmus, je 14, neboť už zde trval jeden z 10 požadovaných výpočtů necelých 7 minut.
Naivní řešení jsem testoval Jak můžeme vidět na grafu 5.1 časová složitost navrženého algoritmu s využitím 2-opt optimalizace dosahuje daleko lepších časových výsledků oproti naivnímu řešení. Sestavení trasy na 25 vrcholech je navrhovaný algoritmus schopný v podobném čase, jako to zvládá naivní algoritmus pro 10 vrcholů. Časy jsou detailně rozepsány v tabulce 5.1.

5.4 Přesnost navrhovaného algoritmu

Důležitým měřením je reálná přesnost navrhovaného algoritmu. V tabulce 5.2 jsou uvedeny průměrné odchylky od hodnoty řešení naivního algoritmu.
Zpětně si uvědomuji, že možná bylo vhodnější zvolit lepší parametry náhodného rozložení generující zdržení v bodě, neboť v poměru s délkami hran v některých případech nevytváří dostatečné zdržení. Přesto je vidět, že navrhovaný 2-opt algoritmus značně zlepšuje trasu vytvořenou hladovým algoritmem a jeho odchylka je velice malá.



Obrázek 5.1: Graf zobrazující závislost doby běhu programu na počtu vrcholů. *Pozn.: osa Y je logaritmického měřítka*

5.5 Paměťové testování

Zde se vyskytla zásadní otázka – jakou metodu měření použít. Lze použít program *valgring* a nástroj *massif*, který dělá pravidelně obrazy paměti, avšak podle autorů drasticky zpomalí běh programu, obvzláštní pokud ještě k tomu sledujeme velikost zásobníku [30]. Zde je vhodné použít na výstupní soubor *ms_print*, který jej zpracuje do čitelnější podoby a přidává na začátek výpisu graf vývoje využití paměti v čase, a však pouze pomocí ASCII symbolů. Pro interaktivní zobrazení se dá však použít grafický program *massif-visualizer*, který produkuje daleko detailnější grafy. Další variantou je *time*, který kromě měření času programu také měří maximální velikost obsazené paměti. Oproti programu *massif* nezpomaluje nijak výrazně celý program, tedy by celé testování probíhalo rychleji, ale nezískáme detailní graf vývoje.
Testoval jsem potřebovanou paměť při počítání trasy na různém počtu vrcholů. V tabulce 5.3 jsou uvedené naměřené hodnoty. Vzhledem k potřebné paměti pro 18 vrcholů lze s klidem tvrdit, že operační paměť nebude při provozu problémem.

Počet vrcholů	Naivní algoritmus Průměrná doba běhu[ms]	2-opt algoritmus Průměrná doba běhu[ms]
3	40	33
4	78	67
5	161	109
6	322	162
7	728	299
8	1573	371
9	3355	533
10	6579	807
11	12271	1042
12	90457	1029
13	216577	1316
14	400330	1983
15	-	2560
16	-	3023
17	-	3954
18	-	4459
19	-	4655
20	-	5174
21	-	6686
22	-	7048
23	-	8616
24	-	9207
25	-	10131

Tabulka 5.1: Průměrné naměřené časy v ms pro různé počty vrcholů

Počet vrcholů	Hladový algoritmus Průměrná odchylka [%]	2-opt algoritmus Průměrná odchylka[%]
3	0.7	0.0
4	2.7	0.1
5	7.3	0.1
6	4.4	0.7
7	9.2	0.6
8	10	1.7
9	12.9	0.2
10	6.8	0.8
11	9.5	0.3
12	10.4	1.8
13	17.4	1
14	9	0.8

Tabulka 5.2: Průměrné odchylky řešení pro různé počty vrcholů

Počet vrcholů	Potřebná paměť
3	267.8 KiB
4	274.3 KiB
5	285.0 KiB
6	304.4 KiB
7	315.1 KiB
8	351.9 KiB
9	382.0 KiB
10	382.7 KiB
11	467.9 KiB
12	431.9 KiB
13	457.1 KiB
14	853.4 KiB
15	705.2 KiB
16	631.8 KiB
17	1.2 MiB
18	1.5 MiB

Tabulka 5.3: Naměřená maximální požadovaná paměť

Závěr

Cílem práce bylo analyzovat grafové algoritmy pro hledání tras, navržení způsobu hledání tras, sběr dat a predikce zdržení v bodu zájmu a implementovat navigační modul využívající navržené prvky.
Veškeré body body splněny, modul však ještě není plně nasaditelný, neboť nejsou připravené další systémy, na kterým modul závisí. Hlavní důraz byl tedy kladen a analytickou a návrhovou část, které obsahují rozbor současných způsobů řešení hledání nejkratších cest a řešení *Problému obchodního cestujícího*. Také se mi v této práci podařilo navrhnou algoritmus na sestavování tras, který je dostatečně přesný a rychlý. Navržen byl také predikční model, způsob směru dat a zapojení do celého projektu. Oproti zadání bylo implementováno REST API, které bylo na základě analýzy architektury projektu nutné zahrnout do modulu. Závěrem je modul otestován na paměťovou a časovou složitost.

Budoucí práce

Je zde stále velký potenciál pro rozvoj modulu. V momentě, kdy se nasadí modul pro správu mapových podkladů, bude potřeba implementaci rozšířit o komunikátor s tímto modulem.
Dále jako hlavní možnost vylepšení vidím nahrazení predikčního modelu neuronovou sítí, která by mohla brát při predikci v potaz daleko více věcí, jako například počasí. Toto podle mého odhadu bude možné nejdříve, až celý projekt bude nasazen do reálného provozu, neboť pro vytrénování neuronové sítě je potřeba velké množství dat.
Jako další návrh padlo zohledňovat otevírací doby daných památek, se kterými se zatím nepočítá. Další novou funkcionalitou by mohlo být vymezení si určité doby na občerstvení či požadavek na pravidelné zastávky u toalet. Toto by však vyžadovalo značně upravit či zcela nahradit současný algoritmus hledání cest.
Pro více výpočetní síly by se výpočetní uzly mohly rozdělit mezi více proce-

sorů, což by vyžadovalo využití knihoven zajišťující meziprocesovou komunikaci, jako například `Open MPI`. Součástí tohoto rozšíření by také mohlo být možnost dynamicky výpočetní uzly přidávat či odebírat.

Literatura

[1] ČVUT v Praze, F. i. t.: Předmět BI-AG1. [cit. 2019-05-05]. Dostupné z: https://courses.fit.cvut.cz/BI-AG1{[online]}

[2] ČVUT v Praze, F. i. t.: Předmět BI-AG2. [cit. 2019-05-05]. Dostupné z: https://courses.fit.cvut.cz/BI-AG2{[online]}

[3] Ondřej Suchý, T. V.: BI-AG2, Přednáška č.11, Hamiltonovské grafy, problém obchodní cestující, aproximační algoritmy [online]. 2017, [cit. 2019-05-02]. Dostupné z: https://edux.fit.cvut.cz/courses/BI-AG2/_media/lectures/bi-ag2-p-vse.pdf

[4] Urban, P.: *Algoritmy pro (NP-) těžké problémy*. Bakalářská práce, ČVUT v Praze, Fakulta informačních technologií, 2015.

[5] Cheong, T.; White, C. C.: Dynamic Traveling Salesman Problem: Value of Real-Time Traffic Information. *IEEE Transactions on Intelligent Transportation Systems*, ročník 13, č. 2, June 2012: s. 619–630, ISSN 1524-9050, doi:10.1109/TITS.2011.2174050.

[6] Held, M.; Karp, R. M.: A Dynamic Programming Approach to Sequencing Problems. In *Proceedings of the 1961 16th ACM National Meeting*, ACM '61, New York, NY, USA: ACM, 1961, s. 71.201–71.204, doi:10.1145/800029.808532. Dostupné z: http://doi.acm.org/10.1145/800029.808532

[7] Bellman, R.: Dynamic Programming Treatment of the Travelling Salesman Problem. *J. ACM*, ročník 9, č. 1, Leden 1962: s. 61–63, ISSN 0004-5411, doi:10.1145/321105.321111. Dostupné z: http://doi.acm.org/10.1145/321105.321111

[8] Christofides, N.: Worst-Case Analysis of a New Heuristic for the Traveling Salesman Problem. *Carnegie Mellon University*, 02 1976: str. 10.

[9] Kolmogorov, V.: Blossom V: a new implementation of a minimum cost perfect matching algorithm. *Mathematical Programming Computation*, ročník 1, č. 1, Jul 2009: s. 43–67, ISSN 1867-2957, doi:10.1007/s12532-009-0002-8. Dostupné z: https://doi.org/10.1007/s12532-009-0002-8

[10] Nilsson, C.: Heuristics for the Traveling Salesman Problem. 01 2003.

[11] Helsgaun, K.: An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic. *European Journal of Operational Research*, ročník 126, 2000: s. 106–130.

[12] Patel, A.: Heuristics [online]. [cit. 2019-05-02]. Dostupné z: http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html

[13] Eyckelhof, C. J.; Snoek, M.: Ant Systems for a Dynamic TSP. In *Ant Algorithms*, editace M. Dorigo; G. Di Caro; M. Sampels, Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, ISBN 978-3-540-45724-4, s. 88–99.

[14] Google: Popular times, wait times, and visit duration - Google My Business Help [online]. [cit. 2019-05-13]. Dostupné z: https://support.google.com/business/answer/6263531

[15] Google: About Waze [online]. [cit. 2018-05-05]. Dostupné z: https://support.google.com/waze/answer/6071177

[16] Google: How does Waze work? [online]. [cit. 2018-05-05]. Dostupné z: https://support.google.com/waze/answer/6078702

[17] Google: Directions API Overview [online]. [cit. 2018-04-18]. Dostupné z: https://developers.google.com/maps/documentation/directions/intro

[18] Google: Places API for Web Overview [online]. [cit. 2018-04-18]. Dostupné z: https://developers.google.com/places/web-service/intro

[19] Seznam: Fórum nápovědy - Mapy API v4.0 [online]. [cit. 2018-05-04]. Dostupné z: https://napoveda.seznam.cz/forum/threads/68301/1

[20] Google: Data Feed - Overview [online]. [cit. 2018-05-05]. Dostupné z: https://developers.google.com/waze/data-feed/overview

[21] Kucbel, D.: *ČVUT Navigátor - Navigační modul*. Bakalářská práce, ČVUT v Praze, Fakulta informačních technologií, 2015.

[22] Langer, J.: *ČVUT Navigátor - Správa mapových podkladů*. Diplomová práce, ČVUT v Praze, Fakulta informačních technologií, 2013.

[23] Kandaurov, A.: *Mobile Application for Route Search in Prague Public Transport*. Bakalářská práce, ČVUT v Praze, Fakulta informačních technologií, 2018.

[24] Breunig, M. M.; Kriegel, H.-P.; Ng, R. T.; aj.: LOF: Identifying Density-based Local Outliers. *SIGMOD Rec.*, ročník 29, č. 2, Květen 2000: s. 93–104, ISSN 0163-5808, doi:10.1145/335191.335388. Dostupné z: `http://doi.acm.org/10.1145/335191.335388`

[25] Hu, X.; Zhao, S.; Shi, F.; aj.: Circuity analyses of HSR network and high-speed train paths in China. *PLOS ONE*, ročník 12, 09 2017: str. e0176005, doi:10.1371/journal.pone.0176005.

[26] Veness, C.: Calculate distance, bearing and more between Latitude/Longitude points [online]. [cit. 2019-03-20]. Dostupné z: `https://www.movable-type.co.uk/scripts/latlong.html`

[27] Lindner, M.: libconfig, C/C++ library for processing configuration files [online]. [cit. 2018-05-05]. Dostupné z: `https://hyperrealm.github.io/libconfig/`

[28] McHale, C.: Config4* [online]. [cit. 2018-05-05]. Dostupné z: `http://www.config4star.org/`

[29] Google: How to build for Linux [online]. Dostupné z: `https://github.com/Microsoft/cpprestsdk/wiki/How-to-build-for-Linux`

[30] Valgrind: Massif: a heap profiler [online]. [cit. 2018-04-30]. Dostupné z: `http://valgrind.org/docs/manual/ms-manual.html`

Příloha A

Seznam použitých zkratek

ČVUT České vysoké učení technické

POSIX Portable Operating System Interface

REST Representational state transfer

API Application programming interface

TSP Traveling salesman problem

GUI Graphical user interface

JSON JavaScript Object Notation

GPS Global Positioning System

SQL Structured Query Language

NoSQL Non SQL

XML Extensible Markup Language

HTML HyperText Markup Language

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

RAM Random-access memory

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `impl` ..... zdrojové kódy implementace
- `doc` ..... dokumentace vygenerovaná ze zdrojových kódů
  - `api.swagger` ..... specifikace REST API ve formátu Swagger
- `text` ..... text práce
  - `Strejc_Ivo_BP.pdf` ..... text práce ve formátu PDF
  - `Strejc_Ivo_BP.tex` ..... text práce ve formátu LaTeX