VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

# Implementace a optimalizace algoritmu pro alternativní plánování tras

# Implementation and Optimisation of the Algorithm for Alternative Route Planning

2018

Jan Faltýnek

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

# Zadání bakalářské práce

Student: **Jan Faltýnek**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Implementace a optimalizace algoritmu pro alternativní plánování tras
Implementation and Optimisation of the Algorithm for Alternative Route Planning

Jazyk vypracování: čeština

Zásady pro vypracování:

Student se v rámci bakalářské práce zaměří na problematiku optimalizaci směrovacího algoritmu doporučujícího sadu alternativních tras pro navigaci vozidel. V rámci optimalizace algortimu se bude zabývat možností využití paralelizace kódu a také efektivního využití paměti v případě vícenásobných dotazů s blízkými počátečními či koncovými body. Výslená implementace a optimalizace budou testovány nad reálnou dopravní sítí extrahovanou z Open Street Map.

Jednotlivé body práce jsou:
1. Prostudovat problematiku směrovacích algoritmů pro navigaci vozidel.
2. Implementace a optimalizace vybraného algoritmu.
3. Provedení experimentů nad reálnými daty.
4. Vyhodnocení experimentů.

Seznam doporučené odborné literatury:

[1] Andreas Paraskevopoulos, Christos Zaroliagis: Improved Alternative Route Planning, 13th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'13), pp. 108–122

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Jan Martinovič, Ph.D.**

Datum zadání: 01.09.2017

Datum odevzdání: 30.04.2018


doc. Ing. Jan Platoš, Ph.D.
*vedoucí katedry*

prof. Ing. Pavel Brandštetter, CSc.
*děkan fakulty*

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 23. dubna 2018 ..... Talbýnek .....

Rád bych na tomto místě poděkoval Ing. Janu Martinovičovi, Ph.D., který mi s prací pomáhal. Dále bych také rád poděkoval mým kolegům (Ing. Martin Golasowski, Ing. Vít Ptošek, Ing. Jiří Ševčík), kteří mi předali cenné zkušenosti.

**Abstrakt**

Práce se zabývá směrovacím algoritmem (Plateau Algorithm) pro alternativní trasy a jeho optimalizacemi. Cílem tohoto algoritmu je získat několik možných tras z bodu A do bodu B, které mohou být dále využity pro distribuci dopravního provozu. V práci je prezentováno několik vylepšení algoritmu Plateau, které sníží časovou náročnost výpočtu a přitom zásadně nezhorší jeho výsledek. Součástí práce je také provedená optimalizace vybraných částí řešení výpočtu pomocí jednoduché paralelizace. Výsledná implementace algoritmu je následně testována, jak na notebooku, tak i na výpočetním nodu HPC clusteru.

**Klíčová slova**: směrování, Plateau algoritmus, optimalizace, vlákna, hierarchie, HPC

**Abstract**

The thesis deals with routing algorithm (Plateau Algorithm) for alternative paths and its optimization. The aim of this algorithm is to get several possible routes from point A to point B, which can be then used to distribute traffic. Several improvements of the Plateau algorithm, which lower time demands of computation and at the same time does not affect its result, are presented in the thesis. A part of the thesis is optimization of selected parts of solution with simple parallelization. Final algorithm implementation is then tested both on notebook and HPC cluster node.

**Key Words**: routing, Plateau Algorithm, optimization, threads, hierarchy, HPC

# Obsah

# Seznam použitých zkratek a symbolů

| | | |
|---|---|---|
| GPS | – | Global Positioning System |
| HPC | – | High-performance Computing |
| 2D | – | Two-dimensional Space |

# Seznam obrázků

# Seznam tabulek

# Seznam výpisů zdrojového kódu

# 1 Úvod

Motivací pro vznik této práce byl fakt, že v dnešní době se často setkáváme s klasickým problémem, transferu dat či osob z bodu A do bodu B a většinou hledáme tu nejkratší respektive nejrychlejší cestu. Je zde několik úloh, které se typicky tímto zabývají, ať už je to směrování paketů v počítačové sítí, nebo hledání ideální cesty pro navigaci vozidel [3] po silniční síti. Tyto dvě zmiňované úlohy využívají jiných algoritmů, což je dáno rozdílnými vstupními parametry a aktéry celého procesu. Co zůstává stejné pro oba algoritmy, je to, že když najdeme ideální cestu v komunikační síti a pošleme po ní velké množství paketů, tak zahltí směrovač, nebo v případě silniční sítě to budou auta, která ucpou průjezd cesty. Proto by bylo ideální předcházet těmto situacím a distribuovat provoz i nějakou alternativní cestou, která může být o nějaký kus delší, ale ve výsledku to bude pro všechny účastníky výhodnější řešení. Je následně na zvážení, jak přerozdělení aplikovat, zda tuto možnost necháme na samotném iniciátorovi požadavku, či budeme v nějakém cyklu kombinovat například deset různých cest.

Hlavním cílem práce je implementovat Plateau Algorithm [5], který je určen k vyhledávání alternativní trasy mezi počátečním a koncovým bodem v silniční síti. Dále bude v práci řešena optimalizace výpočtu včetně využití více vláknové architektury.

Tato práce nepřímo navazuje na již vypracovanou bakalářskou práci [6], která řešila podobné problémy u klasických směrovacích algoritmů a dále rozšiřuje sadu algoritmů prezentovaných v této práci a to včetně datových struktur potřebných pro ukládání routovacích dat.

Důležitou části práce bylo navrhnout algoritmy a provést jejich implementaci tak, aby mohly být spouštěny na superpočítačové infrastruktuře a následně mohlo být provedeno jejich profilování pomocí standardních nástrojů používaných na superpočítačích - konkrétně na clusteru Salomon. Proto výsledné testy, které jsou prezentovány v textu práce, obsahují pro porovnání čas potřebný pro výpočet algoritmu jak na notebooku, tak i na výpočetním nodu clusteru. Z výše zmíněných důvodů byl pro implementaci použit jazyk C++.

V 2 kapitole si přiblížíme obecný principy pro směrovací algoritmy a několik základních algoritmů i popíšu včetně jejich výhod a nevýhod.

Kapitola číslo 3 popisuje dva algoritmy pro hledání alternativních cest, kterými jsem se zabýval a přiblíží důvody, proč jsem si nakonec vybral Plateau algoritmus.

Popisem vlastní implementace Plateau algoritmu se budu zabývat v sekci 4, která bude mimo jiné obsahovat i ukázky se zdrojovým kódem.

Kapitola číslo 5 obsahuje informace o potřebném času na výpočet jednotlivých částí Plateau algoritmu a následně popisuje možnosti vylepšení jednotlivých sekcí algoritmu. Součástí je i část věnovaná paralelizaci.

Poslední kapitola 6 je věnovaná testům, které porovnávají rychlosti v několika verzích samotného algoritmu. Jsou zde časy na HPC clusteru a notebooku pro porovnání. V této kapitole se také nachází obrázky alternativních cest, které jsou výsledkem algoritmu.

# 2 Směrovací algoritmy

Transformací silniční sítě do grafu [7], vytvoříme datovou strukturu, nad kterou lze spouštět algoritmy. Grafem rozumíme matematickou strukturu, která má přesně definovaný počet vrcholů a mezi těmito vrcholy existují hrany, tyto hrany jsou ohodnocené (třeba jejich délka) a mají svou orientaci. V našem případě bude orientace reprezentovat, zda je cesta jednosměrná, nebo se jezdí v obou směrech. Grafy, které budu využívat ve své práci, budou vždy orientované jen jedním směrem. Aby algoritmy fungovaly, tak jak mají, musí být tento graf spojitý, to znamená, že z každého vrcholu A jsem schopný přes N vrcholů a N+1 hran dostat se do vrcholu B.

(a) OpenStreetMap (b) Graf

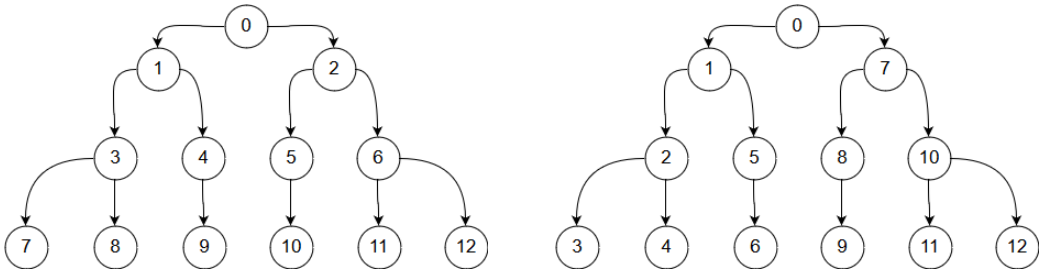
Obrázek 1: Ukázka převodu mapy na graf v oblasti Ostrava-Poruba

## 2.1 Algoritmy průchodu grafem

Jelikož jsme si řekli, že silniční síť můžeme transformovat do grafu, tak lze využít všechny algoritmy, které nám slouží k hledání prvků v grafu. Typickými zástupci jsou Prohledávání do šířky [2] (Breadth-first search - BFS) a Prohledávání do hloubky [2] (Depth-first search - DFS). Oba dva tyto algoritmy mají podobný postup:

1. Objevíme počáteční vrchol (U grafu s kořenem je to právě on, jinde vezmeme nějaký náhodný).
2. Objevíme jeho sousedy, tedy vrcholy, do kterých je možno se dostat z tohoto vrcholu a tyto vrcholy vložíme do struktury, která reprezentuje algoritmus (pro BFS je to fronta, pro DFS je to zásobník). Samotný vrchol vložíme do struktury, která nám reprezentuje navštívené vrcholy.
3. Vezmeme vrchol, který je na vrcholu dané struktury.
4. Vracíme se na bod 2, pokud jsme nenašli hledaný prvek nebo existuje ještě nenavštívený vrchol, jinak končíme.

Jak je patrné z obrázku 2, kde je příklad stromové struktury o čtyřech úrovních, tak tato metoda by nebyla moc efektivní, jelikož algoritmus nepočítá vůbec s nějakou váhou hrany grafu.




(a) BFS (b) DFS

Obrázek 2: Pořadí zpracování jednotlivých vrcholů v daném algoritmu (BFS, DFS)

Ve výsledku by to znamenalo, že kdybychom měli cestu A, která měří 10 km a ta měla po cestě dva vrcholy, pak bychom měli cestu B, která měří 5 km avšak cesta by obsahovala 10 vrcholu, tak podle toho algoritmu by vyhrála cesta typu A, což rozhodně není nejkratší možnost.

Výhodou je rozhodně jednoduchost implemetance a lze tyto algoritmy použít, kdyby nás jen zajímalo, jestli existuje nějaká cesta a už by nám bylo jedno, jestli je to ta nejlepší. Mínusy jsou tedy zřejmé, nejsme schopni najít nejlepší cestu a rychlost těchto algoritmů bude velmi špatná, jelikož jsou navrženy tak, že prochází celý graf.

## 2.2 Dijkstrův algoritmus

Jedná se o typického zástupce algoritmu pro směrování. Jestliže existuje cesta mezi body A a B, máme zaručeno, že výsledkem bude nejkratší cesta mezi těmito uzly. Jedná se vlastně o vylepšenou verzi BFS, která už je schopná brát v potaz i váhu hrany. Využívá prioritní frontu, což je struktura, která obsahuje objevené vrcholy, avšak ještě ne navštívené. Na vrcholu fronty je vždy uzel, který má nejkratší vzdálenost od startovacího vrcholu. Tuto frontu lze implementovat jako binární haldu, případně se dá vždy najít nejmenší prvek ve struktuře, ale to je velmi neefektivní. Každý vrchol má tři příznaky - zda byl navštíven, vzdálenost od startovního uzlu a jeho předchůdce (aby bylo možné následně poskládat celou cestu). Implementace algoritmu není nijak složitá a průběh algoritmu vypadá takto:

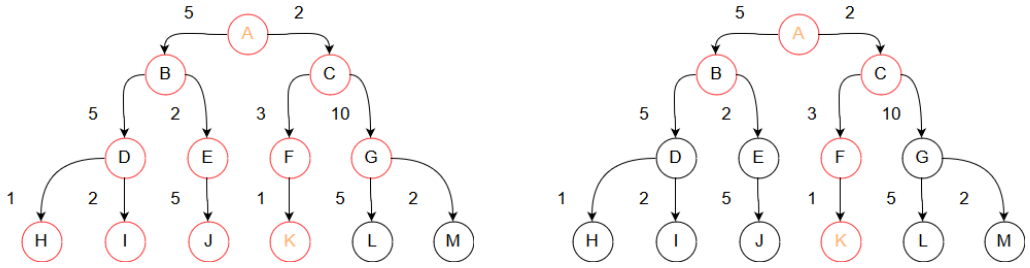
1. Všem vrcholům nastavíme příznak, zda byl navštíven, na zápornou hodnotu, vzdálenost na nekonečno a předešlý uzel také na zápornou hodnotu.

2. Vybereme jako první startovní vrchol (nyní aktuální vrchol) a vzdálenost nastavíme na nulu.

3. Z aktuálního vrcholu najdeme všechny hrany, které vedou do sousedních vrcholů, pokud tyto vrcholy nejsou označené jako navštívené, tak je přidáme do prioritní fronty (případně pokud se v ní již nacházejí, tak jen upravíme hodnotu vzdálenosti od startovacího bodu, jestliže je vzdálenost nyní kratší) a nastavíme hodnotu předešlého uzlu na aktuální uzel. Proměnná vzdálenost bude vzdálenost aktuálního uzlu plus váha hrany.

4. Aktuální uzel označíme za navštívený, pokud to byl cílový vrchol nebo prioritní fronta je prázdná, tak končíme.

5. Z prioritní fronty vybere vrchol, který má zatím nejkratší vzdálenost, tento vrchol bude zatím označen jako aktuální a pokračujeme pravidlem číslo 3.




(a) BFS (b) Dijkstra

Obrázek 3: Průchod grafu daným algoritmem (BFS, Dijkstra)

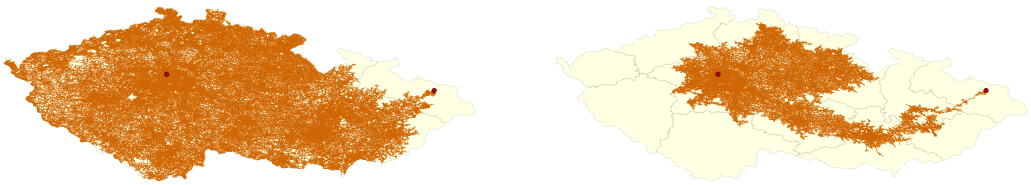
Na obrázku 3 lze vidět, jak se prioritní fronta projeví a jak moc vylepšuje základní variantu algoritmu BFS. Startovní uzel je uzel A a cílem je uzel K, oba uzly mají zvýrazněné písmeno oranžově. Červeně znázorněné vrcholy jsou ty, které byly navštívené.

Výhodou tohoto algoritmu je oproti BFS a DFS zmiňovaná nejkratší cesta Avšak i když by se mohlo na první pohled zdát, že oproti BFS jsme si mnohonásobně pomohli, co se týče časové náročnosti, není tomu tak. Na velkém grafu bychom většinou navštívili podobný počet vrcholů, takže stále máme pomalý algoritmus.

## 2.3 A*

Jelikož Dijkstrův algoritmus je sám o sobě dosti pomalý, bylo nutné ho nějak modifikovat, aby vznikl mnohem efektivnější způsob vyhledávání, ale aby stále byl velmi lehký na implementaci. Přesně z toho důvodu vznikl A* [6], který do celého procesu přináší další složku, se kterou se musí počítat v prioritní frontě. A* na úkor této složky neposkytuje vždy nejlepší výsledek. Jedná se tedy o heuristiku a není garantováno, že nalezená cesta bude tou nejlepší, která existuje.

Vytvoření správné heuristické funkce a nastavení jejich parametrů může být a většinou je ten největší problém celého algoritmu. Optimalizace A* vychází z toho, že všechny vrcholy mají své GPS souřadnice a tím pádem bychom byli schopni vypočítat jejich leteckou vzdálenost a tu přičíst k váze jednotlivých hran. Je zapotřebí vybalancovat koeficient tak, aby obě složky měly stejnou váhu.



(a) Dijkstra (b) A*

Obrázek 4: Ukázka toho, jak moc je rozdílná prohledávaná plocha grafu jednotlivými algoritmy (Dijkstra, A*). Hledaná cesta vede z Prahy do Ostravy

# 3 Směrovací algoritmy pro alternativní trasy

Algoritmus slouží pro vyhledávání nejkratší cesty mezi body A a B, ale vrátí X různých výsledků a ideální by bylo, aby se tyto cesty co nejvíce lišily, tedy uzly tvořící jednu cestu se nebudou shodovat s uzly, které budou tvořit cestu dalšího výsledku, avšak délka těchto dvou a více cest by měla být stále stejná, případně může být rozdílná v rámci nějaké tolerované odchylky. Jak si každý hned domyslí, tato možnost je opravdu ideální ale v praxi nereálná, jelikož tolik jiných cest ani nemusí existovat.

Budou nás zajímat dva druhy algoritmů, se kterými jsem pracoval během návrhů. Jejich výhodou je, že oba tyto algoritmy vycházejí ze základního Dijkstrova algoritmu a jen nabalují nějaké inovace a myšlenky. První z nich byla jen prvotní myšlenka, a proto jsem algoritmu dal pracovní název Multi-Dijkstra Alternatives, podle hledání jsem nenašel žádný oficiální název pro tuto metodu a druhý, kterým se budeme zabývat podrobně, je Plateu algoritmus, jež vychází z vědecké publikace [5].

## 3.1 Multi-Dijkstra Alternatives

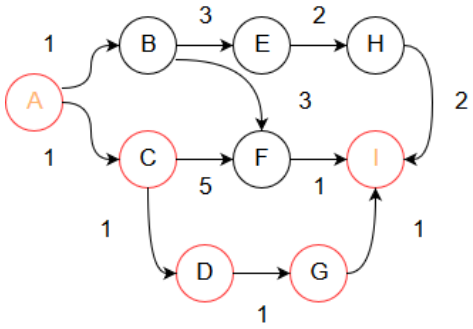
Za tímto nápadem je velmi jednoduchá myšlenka. Proč bychom nemohli pustit víckrát obyčejného Dijkstru, aby nám našel více cest. Trošku nám to nabourává podstata deterministického požadavku na algoritmus, takže je zapotřebí změnit vstupní parametry.

Nejlehčí možnost je po každém průchodu vložit do nějaké datové struktury všechny uzly, které jsou ve výsledném průchodu. Před dalším průchodem všechny tyto uzly smazat ze vstupního grafu, aby již nadále nebylo možné je využívat. Postup lze napsat následovně:

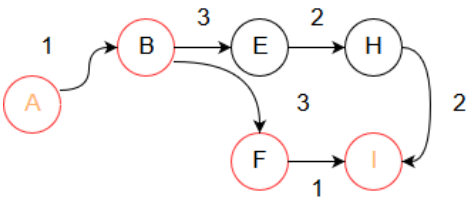
1. Proveď Dijkstru s grafem G a nastav počet opakování na o = 1.
2. Ulož si výslednou cestu do množiny S (množina hran grafu G, která obsahuje výsledné cesty).
3. Vytvoř podgraf $G_x$ grafu G tak, že z grafu G odstraníš všechny vrcholy z množiny S.
4. Proveď Dijkstru s grafem $G_x$ pokud o se nerovná požadovaný počet opakování.
5. Zvedni o o jedna a pokračuj krokem 2.

Na obrázku 5 je vyobrazen startovní bod A a cílový bod I. Červeně zvýrazněné vrcholy tvoří cestu v jednotlivých průchodech. Už z tohoto obrázku je však patrné, že s tímto řešením bychom nedostali přijatelné výsledky, jelikož hned při dalším průchodu by graf již neobsahoval vrchol B a tím pádem bychom nebyli schopni najít teoreticky další potencionální alternativní cestu, která se tam ještě nachází.

Na obrázku 6 si můžeme povšimnout, že nemá stejný problém jako předešlá varianta 5. Je to dáno tím, že můžeme vylepšit naše přidávání do struktury o to, že nebude přidávat n prvních a z posledních vrcholů. V tomto případě je n = z = 2. Díky tomu bychom zde nalezli až čtyři

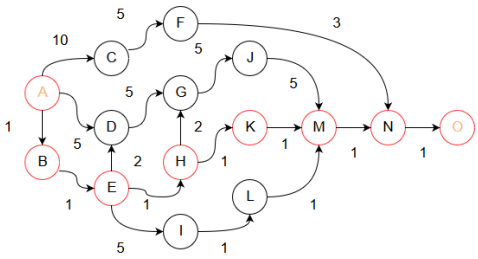



(a) První iterace algoritmu

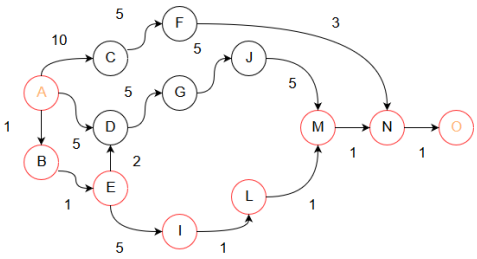



(b) Druhá iterace algoritmu

Obrázek 5: Multi-Dijkstra Alternatives - Nalezení cest a následné zmenšení grafu




(a) První iterace algoritmu




(b) Druhá iterace algoritmu

Obrázek 6: Multi-Dijkstra Alternatives - Vylepšená verze algoritmu o pravidlo zachování vrcholů

alternativní řešení, bez tohoto vylepšení by to byla však nula. Bohužel je potřeba si uvědomit, že takovéto řešení funguje opět jen na umělém, malém grafu, jelikož na reálné mapě by to dopadlo tak, že bychom měli několik cest namačkaných na sebe v úzkém pruhu a to nejspíše není náš cíl. Proto by se tato myšlenka musela rozšířit o komplikovanější věci. Opět bychom mohli využít již zmiňované GPS souřadnice a pomocí nich vytvořit nějaký filtr, který by efektivně eliminoval více uzlů v nějaké oblasti, jenže celý tento proces by ještě víc zhoršoval už tak neefektivního Dijsktru.

Bohužel tohle není jediný problém tohoto řešení. Bylo by vhodné, kdybychom mohli každý cyklus tohoto algoritmu pustit na jednom vlákně. Jenže to by nebylo opět tak jednoduché. Ideální řešení, že bychom pustili každého Dijkstru zvlášť nelze realizovat, jelikož jsou závislí jeden na druhém, další se může spustit až tehdy, kdy se ten předešlí dokončil a upravil graf pro vstup. Jedinou šancí je implementace Bidirection-Dijkstry [6], což je vylepšená verze obyčejného tím, že se vydáme jak ze startu, tak i z cíle a čekáme, než se někde tyto dvě cesty spojí. Zde je opět potřeba si uvědomit, že nestačí pustit jedno vlákno ze startu a následně jedno z cíle, jelikož je problém, že by potřebovali přistupovat ke stejným zdrojům, aby kontrolovali, zda se nenachází v jejich navštívených vrcholech nějaký takový, který má už i ten druhý. Z tohoto důvodu by byla potřebná implementace nějakého omezeného přístupu k jistým zdrojům, třeba omezení pomocí semaforů, jenže takto omezený provoz by mohl být spíš přítěží než užitečný.

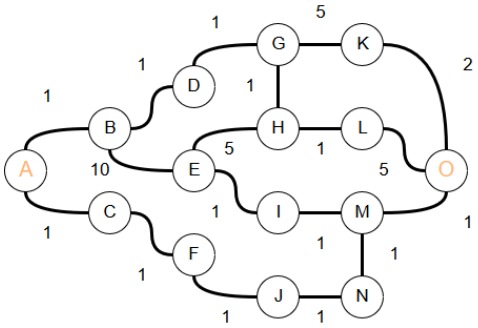
Takže když si to shrneme, tak na nějakém malém grafu, který má stovky vrcholů by se dal tento postup realizovat a fungoval by spolehlivě a nespornou výhodou je fakt, že implementace vylepšeného řešení bez GPS, je opravdu na stejné úrovni obtížnosti implementace jako samotný Dijkstra.

## 3.2 Plateau algoritmus

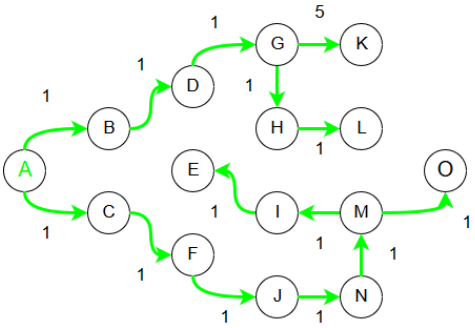
Plateau algoritmus je založen na myšlence, že jsme schopni poměrně jednoduše najít nejkratší cesty z bodu A do všech ostatních vrcholů za pomocí upraveného Dijkstry, který nemá žádnou ukončující podmínku v podobě cílového vrcholu, ale naopak takto projde celý graf, který mu byl dán jako vstupní parametr. Během procházení značí jednotlivé vrcholy tak, aby po skončení tohoto procesu bylo jasné, jak jsme schopni se z bodu A dostat do jakéhokoliv bodu v grafu, tedy nastavuje u každého vrcholu grafu jeho předchůdce. Další zásadní rozdíl je, že tento postup aplikujeme z bodu B, který byl na začátku označen jako cíl cesty. V této fázi jej však využijeme jako startovací vrchol a opět vyhledáme nejkratší cesty do všech ostatních vrcholů grafu. Je zřejmé, že tyto dva grafy nebudou totožné. Kdybychom je pak překryli jeden přes druhý, tak bychom mohli pozorovat zvláštnost, že se nám překrývají některé hrany. A přesně tohle je naším cílem hledání, tyto tzv. Plošiny (Plateaus), podle kterých je tento postup pojmenován. Oblasti, které se takto překrývají, jsou pro nás jasným znamením, že tudy povede alternativní cesta.

Pojďme si to vysvětlit na těchto obrázcích 7. Na prvním obrázku 7a máme znázorněný počáteční ohodnocený graf, kde se snažíme najít alternativní cesty z vrcholu A do vrcholu O. Jak bylo popsáno výše, nejdříve musíme z počátečního vrcholu najít nejkratší cesty do všech ostatních vrcholů grafu, což je obrázek 7b. Tímto nám vznikne kostra původního grafu, což je podgraf původního grafu, který obsahuje všechny původní vrcholy, je spojitý a do každého vrcholu může vstupovat právě jedna hrana, což tedy znamená, že je acyklický. Orientaci hran, tedy směr, ve kterém jsme se dostali do daného uzlu, znázorňují šipky. Stejný postup aplikujeme z bodu, který je cílový, tedy v našem případě je to O. Tento postup je znázorněn na obrázku číslo 7c. Na posledním obrázku 7d máme překrytí těchto dvou podgrafů. Oranžově zabarvené hrany jsou ty, které odpovídají oběma podgrafům a jejich směry šly proti sobě. Tyto hrany jsou ty, které nás zajímají, a díky nim víme, že dvojice dvou vrcholů, mezi kterýma se nachází tato hrana, jsou součástí cesty, a že tedy nějaká cesta vůbec existuje. Zeleně a červeně jsou označeny původní hrany, které se nepřekrývají, a tudíž nám nic neprozrazují. Modře jsem zvýraznil ty hrany, na které chci upozornit. Tyto hrany se sice překrývají, směr je však ve stejném směru a tím pádem se nevyhodnotí, že by zde mohla vést alternativní cesta. Tohle je velmi důležitý fakt. Musíme si uvědomit, že již nemáme znalosti o původním grafu a tím pádem nevíme, že existuje nějaká hrana mezi B a E, tudíž pro nás je v této situaci spojení hran E - I - M nějaká slepá cesta, kde nemá žádný smysl jezdit. Podobně to platí pro hranu A - B. Nesmíte se nechat zmást, že tudy nakonec povede alternativní cesta, rozhodně za to nemůže tato hrana.

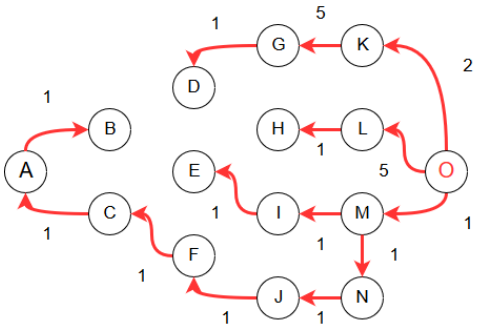
Alternativní cestu získáme tak, že se podíváme na nějakou plošinu a od jednoho ze dvou vrcholů zpět, tedy v protisměru startovacího uzlu, hledáme předchůdce v podgrafu (v našem



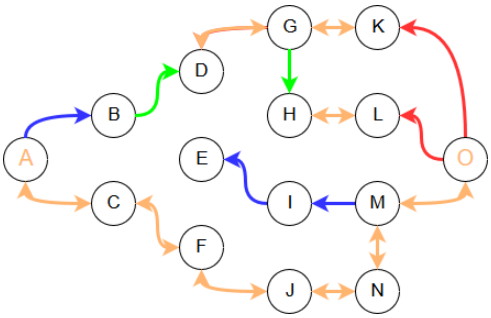

(a) Vstupní graf




(b) Podgraf z vrcholu A




(c) Podgraf z vrcholu O




(d) Překrytí obou podgrafů

Obrázek 7: Plateau Algorithm - Ukázka vytvořených obou podgrafů z počátečního grafu a jejich vyhodnocení

případě by to byl podgraf na obrázku 7b). Zastavíme se, až dojdeme k našemu startovacímu vrcholu. Následně uděláme podobnou věc, jen hledáme ve druhém směru, tedy v protisměru druhého podgrafu (obrázek 7c).

Abychom si to ukázali prakticky, tak to vysvětlím na plošině H – L:

1. Vyberu vrchol H.
2. Najdu jeho předchůdce podle zeleného grafu (G).
3. Opakuji krok 2, než se dostanu do startujícího vrcholu (G -> D -> B-> A).
4. Vrátím se k vrcholu H.
5. Najdu jeho následovníka podle červeného grafu (L).
6. Opakuji krok 5, než dorazím do koncového vrcholu (L -> O)
7. Výsledek dostanu tak, že první část přetočím, připojím vrchol H a druhou část

Výsledné cesty a jejich plošina:

- A -> C -> F -> J -> N -> M -> O (A, C, F, J, N, M, O)

- A -> B -> D -> G -> K -> O (D, G, K)
- A -> B -> D -> G -> H -> L -> O (H, L)

Další věc, na kterou bych rád upozornil, je fakt, že když se podíváme na červený graf, tedy obrázek třetí, tak si můžeme všimnout, že z vrcholu O do G vedou dvě stejně dlouhé cesty, ale díky správnému vyhodnocení na základě Dijkstrova algoritmu se do bodu G dostáváme přes K. Ale představme si situaci, kde by se z nějakého důvodu vyhodnotila nejkratší cesta přes H, tím pádem bychom neměli nadále v tomto grafu hranu mezi G – K a ve výsledku bychom ztratili úplně jednu alternativní cestu. Stejně tak bychom nějak předpokládali, podle pohledu na původní graf, že bude existovat alternativní cesta A -> B -> E -> I -> M -> O, avšak jak již víme, nám žádná taková nevypadla. Je to dáno tím, že váha hrany B – E je příliš vysoká. Mohlo by nás tedy napadnout, že ji rozdělíme na několik stejně malých, nebo si říct rovnou, že délky hran se bude snažit dělat co nejvíc stejně dlouhé. Tohle by sice pomohlo hledat víc alternativních cest, avšak kompletně bychom zpomalili rychlost celého běhu. O tom však více v další sekci. Jen jsem chtěl upozornit na možný problém a může se hodit o tom vědět. Avšak tentokrát je pro nás výhodou, že silniční síť má několika násobně víc vrcholů, než je v ukázce a proto nás nebude trápit, když nám pár cest tímto způsobem vypadne.

# 4 Implementace Plateau algoritmu

Nyní se budeme zabývat již samotnou programátorskou implementací tohoto problému. Celý postup bych shrnul do tří kroků. Začíná to tím, že pomocí upraveného Dijkstrova algoritmu, kterého jsem popisoval v minulé kapitole, vytvoříme opět již zmiňované dva podgrafy, které následně předáme ke zpracování. Druhým krokem je nalezení samotných plošin a ve třetím kroku musíme zpracovat takto vzniklé cesty, tedy spojit ve správném pořadí obě části a následně tyto cesty filtrovat, podle zadaných parametrů.

## 4.1 První část – Dijkstra

Ještě jednou upozorňuji, že se jedná o upravenou verzi Dijkstrova algoritmu, která hledá nejkratší cesty z jednoho bodu do všech ostatních nad daným grafem. Tedy oproti klasickému Dijkstrovu algoritmu zde není ukončující podmínka, že najdu hledaný vrchol, ale je potřeba projít celý graf. Nakonec nevracím jednu výslednou cestu, ale strukturu, která obsahuje všechny vrcholy a jejich předchůdce, pomocí kterého jsem se do daného uzlu nakonec dostal. Tohle vše musí proběhnout dvakrát, tedy jednou pro vrchol, který bude pro nás startovní a druhý, který bude cílový. Je nutné si uvědomit, že musíme implementovat dvě různé verze, jelikož každá využívá jiných hran, jestliže je máme rozdělené, jak jsem na začátku práce zmiňoval a to tedy tak, že obou směrně cesty rozdělíme na dvě hrany, kde každá má jinou orientaci a vlastně z nich vzniknou takové jednosměrky.

```
void DijkstraForth(BinaryHeap<float, int> &openSetForth, nodeMap &
    closedSetForth {
    while (openSetForth.Count()!= 0) {
        int actualId = openSetForth.Remove();
        VisitedNodeLocation &tmp = closedSetForth[actualId];
        tmp.SetWasUsed(true);
        VisitedNodeLocation actualNode = tmp;
        const auto &edges = this->routingGraph->GetEdgesOut(actualId);
        for (const auto &edge : edges) {
            Node node2 = this->routingGraph->GetEndNodeByEdge(*edge);
            float totalCost = actualNode.TotalCost + costCalculator->
                GetTravelCost(**);
            auto node2Find = closedSetForth.find(node2.id);
            if (node2Find == closedSetForth.end()) {
                closedSetForth[node2.id] = VisitedNodeLocation(**);
                openSetForth.Add(totalCost, node2.id);
            } else if (!node2Find->second.WasUsed) {
                if (node2Find->second.TotalCost > totalCost) {
```

```
                openSetForth.UpdatePriority(**);
                node2Find->second.Update(**);
            }
        }
    }
}
```

Výpis 1: Ukázka kódu pro průchod grafu Dijkstrou

Na zdrojovém kódu výpis 1, který je záměrně upravený, aby nebyl příliš dlouhý, lze vidět řešení upraveného Dijkstrova algoritmu. Parametry funkcí, které jsou označené dvěma hvězdičkami (**), jsou pro pochopení a případné přepsání naprosto zbytečné, jelikož se odvíjejí od implementace samotné funkce, takže jejich název je dostačující. Jak lze i podle některých názvů funkcí vydedukovat, je tato metoda pro průchod ze startovacího uzlu.

Jak je vidět, tak do této fáze vstupují dvě datové struktury. **OpenSetForth** je datová struktura typu binární halda a slouží zde pro uchovávání vrcholů, které byly objevené, ale nebyly navštívené. Je inicializovaná vně této metody a po návratu je dealokována. Binární haldu jako takovou jsem sám neimplementoval, jelikož již jedna byla přítomná v projektu, na kterém jsem pracoval. Další důležitou strukturou je **closedSetForth**, který je representován pomocí hashmapy. Abych byl přesný, využívá se Google dense_hash_map [9] pro C++, která opět podle předešlého testování je ideální pro tak velká data. **ClosedSet** je pro nás ta nejdůležitější část, jelikož v něm se nakonec budou nacházet vrcholy a jejich předchůdci, takže se zde budeme opravdu často doptávat. Třetí důležitou částí je graf, který je třídní proměnnou, takže není potřeba jej posílat jako parametr metody.

Další část, která zde stojí za vysvětlení, je řádek, kde se snažíme do **node2Find** přiřadit hodnotu. Funkce **find** je implementována nad zmiňovanou hashmapou a vrací prvek když ho najde, pokud ne, tak se proměnná nastaví na speciální příznak, který hned v následujícím kroku využíváme k tomu, abychom zjistili, zda naše struktura obsahuje tento prvek, jestliže jej neobsahuje, tak jej tam vložíme. Pokud by tam byla, tak první zjistíme, zda je už uzavřen tento uzel (příznak **WasUsed**) a pokud nebude uzavřen, tak zjistíme, zda jsme náhodou nenašli kratší cestu k němu.

## 4.2 Druhá část – Nalezení plošin

Nyní máme předpřipravené podgrafy pro další část algoritmu, která musí předešlé dva podgrafy projít a najít v nich vzorce, které by nám identifikovaly, že zde se nachází část grafu, která nás opravdu zajímá. To jsou tedy ty situace, které byly vysvětleny v kapitole 3.2, která popisuje princip tohoto algoritmu.

```
vector<RouteSolution> FindPlateaus(nodeMap &closedSetForth, nodeMap &
    closedSetBack, int startId, int endId) {
```

```
    int intersectionId;
    google::dense_hash_set<int> processedNodeIds;
    std::vector<RouteSolution> solutions;
    for (auto &intersectionForthNode : closedSetForth) {
        intersectionId = intersectionForthNode.first;
        if (processedNodeIds.find(intersectionId) != processedNodeIds.end()){
            continue;
        }
        processedNodeIds.insert(intersectionId);
        const auto &intersectionBackNode = closedSetBack.find(intersectionId);
        if (intersectionBackNode != closedSetBack.end()) {
            float plateauTime = 0;
            if (intersectionForthNode.second.PreviousNodeId ==
                intersectionBackNode->second.PreviousNodeId){
                continue;
            }
            ForthPartPlateau(**);
            BackPartPlateau(**);
            if (plateauTime > 0 ) {
               AddAlternativeSolution(**);
            }
        }
    }
    return solutions;
}
```

Výpis 2: Ukázka kódu pro vyhodnocení plošin

Pojďme si vysvětlit zdrojový kód 2, který je opět záměrně poupraven jako v předešlém případě. Potřebné parametry pro chod této metody jsou zmiňované dvě struktury, které nám budou nějakým způsobem reprezentovat graf. Pořád využíváme strukturu, která je vytvořená společností Google. Další dva důležité parametry jsou počáteční vrchol a cílový. Ty nám budou později vstupovat jako parametr do další metody.

Nyní si popíšeme tělo metody. Na začátku si deklarujeme proměnou **intersectinId**, která nám bude do budoucna reprezentovat vrchol, který by měl být hraničním a v jednom směru bychom získali část cesty ke startu a ve druhém směru pak k cíli. Následně si deklarujeme **processedNodeIds**, což je struktura, která je velmi příbuzná naší již používané dense_hash_map, jen s tím rozdílem, že zde si ukládáme jen klíče, žádná další přidaná hodnota zde není. Ale jelikož jediným úkolem této struktury je to, že budu v sobě uchovávat ID již zpracovaných vrcholů, tak je to přesně to, co hledáme. Ve zdrojovém kódu chybí průběh její inicializace, který je popsán v

její dokumentaci na webu [9]. První *for* cyklus nám slouží k procházení všech vrcholu v grafu, který měl za počáteční bod startující uzel. Po vybrání jednoho z těchto vrcholů si jej uložíme jako náš možný středový bod. Následně je potřeba kontroly, zda jsme již s tímto vrchol nepracovali, pokud ne, tak dále pokračujeme. V opačném případě vybereme další z možných, dokud neprojdeme celou kolekci. Pokud se tedy tento vrchol ještě nenacházel v **processedNodeIds**, tak jej tam vložíme, jelikož jej právě budeme testovat, jestli se kolem něj nachází plošina. Nyní provedeme kontrolu, zda se náš kandidát nachází i ve druhém grafu, který má za výchozí bod cíl. Nyní je potřeba kontroly, zda se do tohoto vrcholu nedostávám ze stejného uzlu v obou případech, jestliže by tomu tak bylo, tak by to znamenalo, že jsme nejspíše narazili na slepou cestu a tahle situace je nám k ničemu (Viz obrázek 7 d) - modré šipky). Jestliže jsme prošli i touto kontrolou, následuje už samotné testování předchozích vrcholů, zda existuje cesta, která je v obou podgrafech. Tuto část řeší následující dvě metody **ForthPartPlateau** a **BackPartPlateau**. Jelikož těla těchto dvou funkcí jsou dost podobné, vysvětlím průběh jen na jedné z nich.

```
void ForthPartPlateau(nodeMap &closedSetForth, nodeMap &closedSetBack, google::
    dense_hash_set<int> &processedNodeIds, int actualId, int nextId, float &
    plateauTime){
    while (nextId != -1) {
        const auto &nextForthNode = closedSetForth.find(nextId);
        if (nextForthNode != closedSetForth.end()) {
            if (nextForthNode->second.PreviousNodeId == actualId) {
                processedNodeIds.insert(nextId);
                plateauTime += nextForthNode->second.TravelTime;
                actualId = nextId;
                const auto &nextBackNode = closedSetBack.find(nextId);
                nextId = nextBackNode->second.PreviousNodeId;

            } else {
                break;
            }
        } else {
            break;
        }
    }
}
```

Výpis 3: Ukázka kódu metody ForthPartPlateau

**ForthPartPlateau** výpis 3 má jako vstupní parametry, které jsou nutné pro chod, oba dva grafy, seznam již testovaných vrcholů, délku celé plošiny, ID aktuálního vrcholu a jeho

předchůdce, ale je to ID předchůdce vůči grafu, který je v **closedSetBack**. Následující postup provádíme tak dlouho, než nenarazíme na startující uzel, který má speciální příznak -1 jako předchozí uzel. Nyní vyhledáme konkrétní uzel v grafu, který nám tvoří cestu od startu k mezi vrcholu. Jestliže jsme jej našli a jeho následovník je aktuální vrchol (v prvních průchodu je to ten vrchol, který jsme označili jako prostřední), tak jsme úspěšně našli první část plošiny. Vložíme jej k ostatním zpracovaným uzlům. Zjistíme váhu hrany, která spojuje vrcholy, a přičteme ji k celkovému času plošiny, jestliže váhou hrany je čas, za který jsme schopni danou vzdálenost urazit. Nyní si zjistíme v **closedSetBack** další následující vrchol, který by mohl být částí plošiny, a toto celé testování opakujeme pro tento nový prvek, dokud tedy nedojdeme do počátečního uzlu, nebo nezjistíme, že už nesplňují vrcholy požadované pravidla a tím pádem máme konec levé části plošiny. Následně tedy provedeme to samé pro pravou část plošiny, která směřuje k cíli. K tomu nám slouží tedy druhá metoda **BackPartPlateau**.

```
int actualNodeId = endNodeId;
float totalTime = 0;
do {
   const VisitedNodeLocation &actualVisitedNode = closedSet[actualNodeId];
   totalTime += actualVisitedNode.TravelTime;
   actualNodeId = actualVisitedNode.PreviousNodeId;
} while (actualNodeId != startNodeId);
```

Výpis 4: Ukázka kódu pro sečtení času jedné části cesty

Nakonec se tedy vyhodnotí, zda se našla plošina díky tomu, že její velikost bude větší než nula. Jeli tomu tak, tak ji přidáme do možných řešení pomocí metody **AddAlternativeSolution**. Parametry této metody se mohou velmi lišit na základě celé implementace. V mém případě zde byly potřeba ty (oba dva grafy, start, cíl a prostřední vrchol, což je nějaký uzel, který tvoří plošinu), které nám poslouží k sestavení cesty ze startu až k cíli. Ještě je pak potřeba v této metodě spočítat celkovou délku cesty, která je následně další možností pro výběr trasy. Potřebné informace o délce cesty zjistíme tak, že spojíme výsledky dvou metod, kterým předáme vstupní parametr startovní, koncový vrchol a opět graf. Ve směru ze startu do prostředního uzlu, to budou přesně tyto dva body, které budeme potřebovat a navíc ještě graf, který měl jako počáteční uzel start. Pro druhou metodu to bude opět prostřední uzel, který bude stále na pozici cílového uzlu, a počáteční vrchol bude ten, který je opravdovým koncem cesty a také využijeme jeho graf. Tělo metody 4 je znázorněno na části kódu. Sečtením výsledků získáme celkovou délku cesty.

## 4.3 Třetí část – Poskládání cest a jejich seřazení

Po kompletním projití obou grafů a nalezení všech plošin, můžeme na nalezené výsledky aplikovat námi zvolený filtr tak, aby nám vypadli nejlépe odpovídající výsledky na prvních místech. Následně je jen na naší implementaci, jak sestavíme kompletní cestu. Ale v základu bude dosti

podobná metodě 4, která nám pomáhala získat informace o délce cesty. V jednoduché podobě to pak může vypadat tak, že to tedy projdeme stejně, jen si uložíme pořadí ID. Nutno si uvědomit, že když bychom takto spojili dvě části cesty, tak jedna z nich by byla ve špatném pořadí a proto jednu z nich musíme přeskládat, v mém případě to byla právě ta první.

# 5 Paralelizace a optimalizace Plateau algoritmu

Nyní máme naimplementovaný základ celého algoritmu. Pomocí grafického rozhraní nad databází jsem schopen si i ověřit, že vrácené cesty mi vytváří jednu spojitou čáru a dá se předpokládat, že tedy nalezené cesty jsou správné. Teď se pojďme podívat, jaké úsilí nás stojí výpočet, tedy měřeno v jednotkách času.

Tabulka 1: Plateau Algorithm - Test počátečního řešení.

| Odkud | Kam | Délka [km] | Počet vrcholů grafu | Čas [s] |
|---|---|---|---|---|
| Praha | Praha | 8 | 945314 | 10.857 |
| Plzeň | Ostrava | 353 | 945314 | 11.846 |
| Praha | Ostrava | 270 | 945314 | 11.247 |

Z naměřených časů (viz tabulka 1), jasně vyplývá, že je relativně jedno, jestli je start a cíl od sebe vzdálen 10 km nebo 100 km, časy si jsou dosti podobné. Je to dáno podstatou algoritmu. Jak jsem už několikrát zmiňoval, v první fázi je potřeba prohledat celý graf, takže se nedá předpokládat, že by reálná vzdálenost dvou bodů mohla mít vliv na samotnou rychlost algoritmu.

Nyní si pojďme rozdělit celý algoritmus na několik částí, ve kterých budeme měřit čas, jak dlouhou dobu procesor potřebujeme na to, aby byl schopen tuto část vypočítat. Nejvhodnější rozdělení jsem už využíval v kapitole 3.2 o implementaci, takže budou tedy tři úseky.

Tabulka 2: Plateau Algorithm - Časy tří základních sekcí

| Sekce | Čas [s] |
|---|---|
| Dijkstra | 8.412 |
| Nalezení plošin | 3.551 |
| Poskládání cest a jejich seřazení | 0.023 |

Jak lze vidět v tabulce 2, nejvíce problematickou částí je samotné prohledávání grafů a značkování předchůdců pro nalezení nejkratších cest do všech vrcholů. Další nezanedbatelnou částí je hledání plošin, které je tedy také v rámci několika sekund a za jedinou uspokojující se dá považovat až poslední část, která je oproti dvěma minulým výrazně menší. Nyní tedy víme, na kterou část se máme primárně zaměřit a pokusit se o její vylepšení.

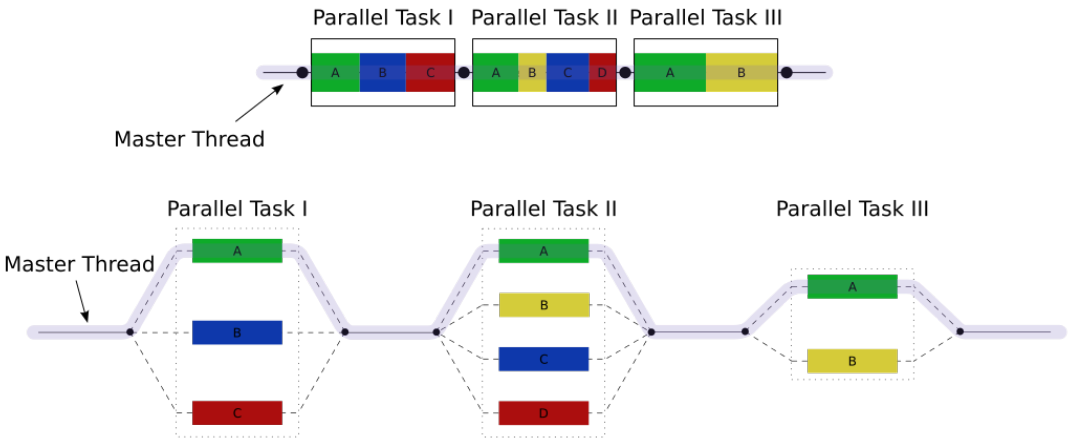
## 5.1 Paralelizace

Nejlepším možným zlepšením by bylo, kdybychom mohli provést paralelizaci kódu, abychom byli schopni využít více jádrovou architekturu počítače. Jelikož jsem vyvíjel software, který poběží nakonec na superpočítači, tak jsem se řídil některými doporučeními, které mi byly řečeny. Jedno

z nich bylo to, že nemusím využívat C++ funkce pro vlákna, ale že bude lepší, když budu využívat OpenMP [1], což je standart, který zajišťuje lehčí práci s vlákny.

### 5.1.1 OpenMP

Je to soustava direktiv pro překladač a knihovních procedur pro paralelní programování. Usnadňuje implementaci těchto problémů a zároveň obstará částečnou optimalizaci. Je vytvořen pro počítače se sdílenou pamětí. Jazyky, které podporuje tato knihovna, jsou C, C++ a Fortran. Samotná knihovna však neřeší problémy, které mohou vzniknout, když využíváme paralelizaci. Je potřeba počítat s tím, že situace, kdy vlákna budou potřebovat pracovat se stejnou částí paměti, budeme stále muset ošetřit pomocí metod, které jsou k tomu speciálně určeny, v případě C/C++ to jsou třeba semafory. Proto tedy může stále docházet k uváznutím (Deadlock), když budeme špatně přistupovat k omezeným zdrojům a dalším podobným problémům.




Obrázek 8: OpenMP - Ukázka dělení vláken do více paralelních sekcí

Program začíná hlavním vláknem, které nám následně OpenMP umožňuje pomocí několika klíčových slov rozvětvit na paralelní úseky, kde každému vláknu je přiřazen čas procesoru podle toho, kolik máme jader v procesoru, případně kolik jsme jich uvolnili na tuto práci. Knihovna se nám sama postará o takzvaný Fork hlavního vlákna a následné spojení. Pomocí tohoto řešení se velmi jednoduše předávají parametry metod, jelikož jediné co musíme udělat je to, že samotné volání metody, které by probíhala bez paralelizace, obalíme do direktiv a máme hotovo.

```
#pragma omp parallel sections num_threads(2)
    {
        #pragma omp section
        { Metoda1(); }
        #pragma omp section
        { Metoda2(); }
    }
```

Výpis 5: Ukázka zápisu sekcí v OpenMP

Na zdrojovém kódu 5 jde vidět, jak je velmi lehké užití OpenMP v jeho nejjednodušší podobě. Na prvním řádku definujeme překladači, že se zde bude nacházet sekce, kde budeme chtít spouštět vlákna a pomocí parametru **num_threads** mu řekneme, kolik vláken požadujeme. Následně **omp section** nám ohraničuje sekci, která běží v jednom vlákně. Takovýchto sekcí můžeme mít, kolik chceme, a nevadí, že máme třeba jen dvě vlákna přidělena, v takovém případě se prostě vykonávají jednotlivé sekce jedna za druhou. Tedy jakmile je sekce dokončena, tak je přiřazeno její vlákno sekci, která by měla být další v pořadí.

Popsal jsem zde jen základní princip, další dokumentace [1] je dostupná online a jsou k ní vždy praktické ukázky. Jedním z dalších zajímavých možností zde bylo procházení nějakého problému v cyklu a nad ním spouštět jednotlivá vlákna, která se optimalizují v rámci každé inkrementace proměnné. Pro náš případ tato optimalizace nebyla nutná.

### 5.1.2 Plateau

Nyní přejděme zpět k našemu problému. Abychom vytvořili dva potřebné podgrafy, museli jsme implementovat dvě podobné metody, které mají za úkol najít nejkratší cesty v našem grafu. V mém případě, když se podívám na jejich parametry, tak vím, že jsou jiné a to mi indikuje, že zde nejspíše nedochází k přístupu ke společným zdrojům a proto můžu bez velkých obav tyto dvě funkce obalit již zmiňovaným OpenMP, aby běžely paralelně.

Tabulka 3: Plateau Algorithm - Časy tří základních sekcí vylepšené o vlákna v Dijkstrovi

| Sekce | Čas před vylepšením [s] | Čas po vylepšení [s] |
|---|---|---|
| Dijkstra | 8.412 | 4.756 |
| Nalezení plošin | 3.551 | 3.689 |
| Poskládání cest a jejich seřazení | 0.023 | 0.024 |

Jak je patrné z výsledků v tabulce 3, tato jednoduchá úprava nám zrychlila chod první části téměř o polovinu. Což se dalo předpokládat. Bohužel, toto zrychlení nemá vliv na žádnou další část algoritmu, jelikož se vstupní parametry do dalších částí nezměnily. Na jednu stranu je to dobře, jelikož máme stále jistotu, že výsledné cesty budou ty nejlepší, na druhou stranu je pořád časová zátěž až moc velká. Můžu rovnou prozradit, že nikde jinde se nám už nepodaří pomocí vláken zrychlit o tolik výkon. Co se týče samotné metody pro Dijkstru, tak uvnitř ní nelze moc využít dalších vláken. Jediná další část, která by se zde nabízela, je ta, kde si zjistíme, které hrany z daného vrcholu vystupují a jejich koncové vrcholy. Jediný problém je, že zde musíme vkládat informace do sktruktury, která by byla společná pro všechny, jinými slovy to je společný zdroj. Když jsem předpokládal, že při vkládání do hashmapy, která by měla mít jasně dané pravidla, kam konkrétní ID uzlu vloží, tak se ukázalo, že můj předpokládat je špatný. Někde

docházelo k přepisování v paměti a tím pádem jsem tuto možnost zavrhl. Další problém, který se zde nachází, je ten, že i tak režie pro vznik vlákna (podle testu) by byla mnohem větší, než jeho samotný přínos. Takže nám nezbývá nic jiného, než se přesunout k další části algoritmu.

Ve druhé části, hledání plošin, se nacházelo opět místo, kde bychom mohli uplatnit vlákna. Ve fázi, kdy jsme již našli náš prostřední uzel, tak se vydáváme jednou stranou směrem ke startu a pak tou druhou směrem k cíli. Bohužel se zde objevuje stejný problém, že by bylo nutno implementovat omezení zdrojů, což by mělo za následek, že by žádného zlepšení nebylo využito a jen bychom zbytečně měli dvě vlákna aktivní, avšak jen jedno by ve skutečnosti mohlo pracovat efektivně.

Naštěstí byla ještě jedna sekce ve druhé části algoritmu, která umožňovala implementaci vláken. Ve chvíli, kdy máme již plošinu, tak bylo potřeba dopočítat informace. Jelikož každá část se provádí nad jedním ze dvou podgrafů, tak zde můžeme opět využít již zmiňovanou nejjednodušší verzi OpenMP. Avšak musím zde upozornit na jeden velký problém. Jelikož náš prostřední uzel nemá vždy stejnou vzdálenost ke startu a cíli, tak se dost často stane, že jedno vlákno bude rychlejší a musí čekat na to druhé. Zmiňuji to proto, že tato situace se vyobrazí na většině nástrojů, které zkoumají chod programu, velmi negativní červenou barvou, která indikuje to, že vlákna nejsou vytížena na 100 % a bylo by vhodné s tím něco udělat. Bohužel v našem případě by to znamenalo úplně smazání vláken, což určitě nechceme a musíme se smířit s tím, že občas nebude využit maximální výkon.

Tabulka 4: Plateau Algorithm - Časy tří základních sekcí vylepšené o vlákna ve druhé části algoritmu (včetně předešlých vylepšení)

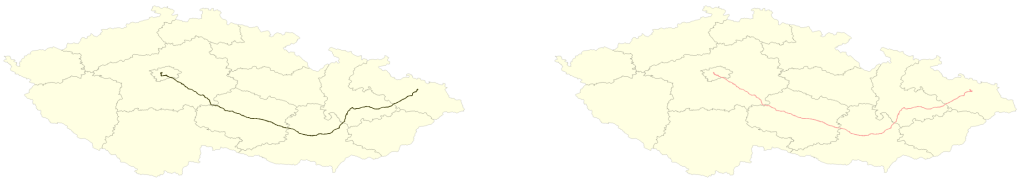
| Sekce | Čas před vylepšením [s] | Čas po vylepšení [s] |
|---|---|---|
| Dijkstra | 4.756 | 4.695 |
| Nalezení plošin | 3.689 | 3.156 |
| Poskládání cest a jejich seřazení | 0.024 | 0.025 |

Jak je vidět v tabulce 4, tak tato poslední úprava nám přidala něco málo času, ale celková doba je stále špatná. Tohle však byla poslední možná část, kde se daly ještě implementovat vlákna. Když by někdo implementoval vlastní filtr, který by mu byl schopný nějak efektivně seřadit výsledky a byl schopný zde zakomponovat vlákna, tak by tohle bylo poslední místo, kde by to ještě mohlo jít. Avšak já využíval C++ funkci **Sort**, takže jsem se nemohl uvnitř metody o něco takového pokoušet.

Paralelizace nám pomohla snížit potřebný čas na celý algoritmus, avšak ještě není stále dostatečně rychlý, abychom s tím mohli být spokojeni. Nemluvě o tom, že zatím velikost grafu odpovídá jen celé České Republice, avšak kdybychom k ní připojili další stát, tak se nám protáhne potřebná doba a to výrazně.

## 5.2 Optimalizace

Až do teď všechna vylepšení měly za následek zkrácení potřebné doby na výpočet algoritmu a nemělo to nejmenší vliv na výsledek, ten byl stále stejný. Avšak nyní už některé úpravy mohou mít na nepřesnost výsledku a tím pádem se bude jednat o heuristiky. V testech budu využívat již implementované vlákna a trasa bude vždy z Prahy do Ostravy.



(a) Výsledek (b) Alternativní cesta

Obrázek 9: Plateau Algorithm - Ukázka problému podobných výsledků

### 5.2.1 Redukce výsledků

Jedním ze zásadních problémů (viz obrázek 9), který existuje v nynější podobě implementace, byl fakt, že nalezené cesty se občas od sebe lišily jen o několik metrů (méně než 1 % uzlů se jen lišilo) na vzdálenosti 200 km a více. Díky tomu jsem měl problém i s filtrováním možných cest, které trvalo zbytečně dlouho a vracelo mi výsledky, se kterými jsem nebyl spokojen. Jak lze názorně vidět na obrázku, který reprezentuje cestu z Prahy do Ostravy. Jediné co se liší je kousek začátku a ještě menší kousek konce cesty. Proto jsem se rozhodl, že by bylo vhodné, kdybych vytvořil podmínku, která mi zajistí, že k této situaci nebude docházet.

Jednoduše lze vyčíst z přiloženého kódu to, že celé vylepšení spočívá v tom, že si určíme nějakou vzdálenost cesty, která nám bude určovat, zda je vhodné toto řešení vyhodnotit jako požadované či nikoliv. Docílil jsem toho jednoduchou úpravou. Když najdu první řešení, které mi najde cestu od stratu až k cíli, tak jej přidám jako správné řešení, ale ještě si z něj odvodím délku, podlé které budu nakonec filtrovat výsledky. Kolik by měla být ta správná délka, nemám nijak exaktně dokázáno. Prostě jsem pouštěl testy a podle výsledků jsem určil, že 10 % původní délky je relativně ideální. Všechny výsledné trasy stálé zůstávaly stejné a jen rychlost se zlepšovala. Samozřejmě je diskutabilní, že třeba 20 % by mohlo ještě víc urychlit výpočet, avšak podle výsledků se již často nenašla stejná trasa. Ve výsledku to znamenalo, že pro nadcházející korektní přidání bylo potřeba, aby velikost plošiny, byla alespoň jednou desetinou délky prvního řešení. Což tedy znamená, že délka alternativní cesty, která by nám vznikla, je alespoň o námi danou hodnotu jiná.

Tabulka 5: Plateau Algorithm - Časy tří základních sekcí vylepšené o redukci výsledků (včetně předešlých vylepšení)

| Sekce | Čas před vylepšením [s] | Čas po vylepšení [s] |
|---|---|---|
| Dijkstra | 4.695 | 4.591 |
| Nalezení plošin | 3.156 | 1.705 |
| Poskládání cest a jejich seřazení | 0.025 | 0.023 |

Tabulka 6: Plateau Algorithm - Ukázka redukce výsledků při použití optimalizace pro snížení tohoto počtu

| Odkud | Kam | Délka [km] | Počet řešení před vylepšením | Počet řešení po vylepšení |
|---|---|---|---|---|
| Praha | Ostrava | 275 | 15732 | 13 |
| Praha | Brno | 185 | 14394 | 16 |
| Brno | Olomouc | 64 | 13392 | 7 |
| Pardubice | Hradec Králové | 18 | 12018 | 8 |

Jak je patrné z výsledků v tabulkách 5 a 6, tak toto řešení nám poskytlo vylepšení hned na dvou místech. Ukrátilo nám tolik drahocenný čas a také nám zpřesnilo výsledky, takže si můžeme být více jistí tím, že vybereme-li si danou alternativní cestu, bude se jednat o dosti jiné řešení, než to, které by nám našel obyčejný Dijkstra.

### 5.2.2 Filtr grafu - ohraničení

Jak jsem již na začátku podotkl, tak je docela velký problém v tom, že rychlost algoritmu není nijak zvlášť ovlivněna tím, jak moc jsou dva body vzdálené od sebe, ale ovlivňuje to samotná velikost grafu. Proto je nutné, abychom nějakým způsobem ovlivnili jen tu část, kterou chceme prohledávat. K tomu využijeme GPS souřadnice jednotlivých bodů. Následně je potřeba najít nějakou funkci F, která by nám vracela, zda se daný bod nachází stále v naší požadované části grafu. Jednou z možností, která již byla částečně naimplementována, byla funkce, která by reprezentovala to, zda se nachází bod uvnitř kruhu. Střed tohoto kruhu se vypočítával opět pomocí GPS souřadnic startovacího a koncového vrcholu. Následně se určila nějaká vzdálenost od tohoto bodu a podle ní se vyhodnocoval filtr. Ovšem já jsem zvolil jiný geometrický útvar. Dle mého názoru je elipsa mnohem výstižnější pro trajektorii cesty.

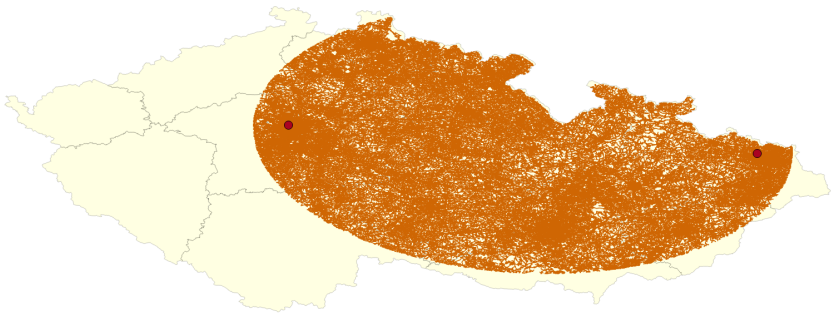
$$|xe| + |xf| = k$$

Využívám faktu, že když sečtu vzdálenost jakéhokoliv bodu $x$ na obvodu elipsy od ohniska $e$ a vzdálenost bodu $x$ k ohnisku $f$, dostanu výsledek, který je stejný pro všechny body, které tvoří obvod elipsy. V mém případě to znamená, že ohniska budou GPS souřadnice mých počátečních bodů a proměnou $k$ získám tak, že vypočítám vzdálenost mezi počátečními body a následně ji

Tabulka 7: Plateau Algorithm - Časy tří základních sekcí vylepšené o filtr grafu elipsou (včetně předešlých vylepšení)

| Sekce | Čas před vylepšením [s] | Čas po vylepšení [s]) |
|---|---|---|
| Dijkstra | 4.591 | 4.146 |
| Nalezení plošin | 1.705 | 0.413 |
| Poskládání cest a jejich seřazení | 0.023 | 0.023 |

zvětším o nějaký koeficient. Opět jsem testoval několik možností a nejvíce se mi osvědčilo, když jsem pro vzdálenosti nad 100 km nastavil tuto proměnnou na 1.15 původní délky. Což tedy prakticky znamená, že vždy budu vyhledávat ve vzdálenosti v proti směru k druhému bodu alespoň 15 km a to je pro většinu cest dostatečné. Opět se nezměnily výsledky testů, a proto jsem toto nastavení zachoval. Následně je potřeba ještě nastavit koeficient pro vzdálenosti, které jsou menší než zmiňovaná stovka. Pro střední vzdálenost (20-100 km) jsem vyhodnotil jako optimální 1.5 a pro menší vzdálenosti je koeficient nastaven na 2.



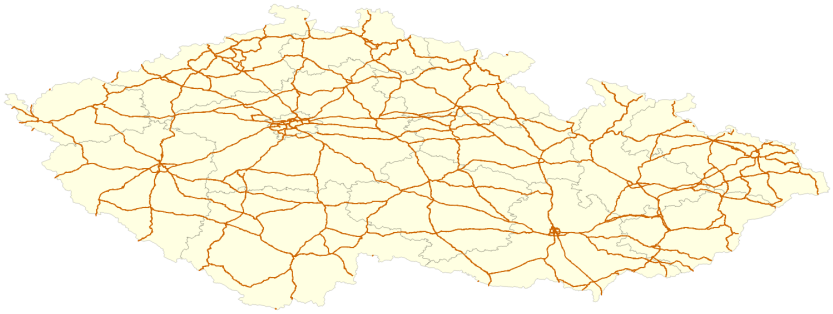
Obrázek 10: Plateau Algorithm - Ukázka ohraničeného grafu elipsou

Na obrázku 10 můžeme vidět, že se nám opravdu podařil zmenšit výsledný graf a má tedy podobu elipsy. Což je pro nás velmi pozitivní, jelikož konečně nejsme nuceni procházet celý graf, i když je vzdálenost třeba pouhých 30 km. Je však potřeba si uvědomit, že takto získané řešení nemusí být to nejlepší a nemusí zahrnovat všechny ideální alternativní trasy, avšak nárůst rychlosti (viz tabulka 7) samotného provedení je obrovský, hlavně pro kratší vzdálenosti.

### 5.2.3 Filtr grafu - hierarchie

Další možností, jak omezit velikost grafu, je to, že do vyhledávaného prostoru nezahrneme úplně všechny kategorie cest. Jak víme, tak máme několik základních dělení cest podle jejich třídy. Tohoto faktu se dá dobře využít, ale je podmínkou, aby naše hrany grafu tuto informaci měly.

V mém případě jsou kategorie dálnice a silnice první třídy zahrnuty pod čísly 0 až 2. Následují silnice druhé třídy, ty mají čísla 3 až 5 a čísly 6 a 7 jsou označené silnice třetí třídy a místní komunikace. Dělení jsem nevytvářel já sám (rozdělení provedeno v OpenStreetMap), jen využívám těchto tří rozdílných skupin a vím, čemu odpovídají.



Obrázek 11: Ukázka velikosti grafu silniční sítě první třídy a dálnic

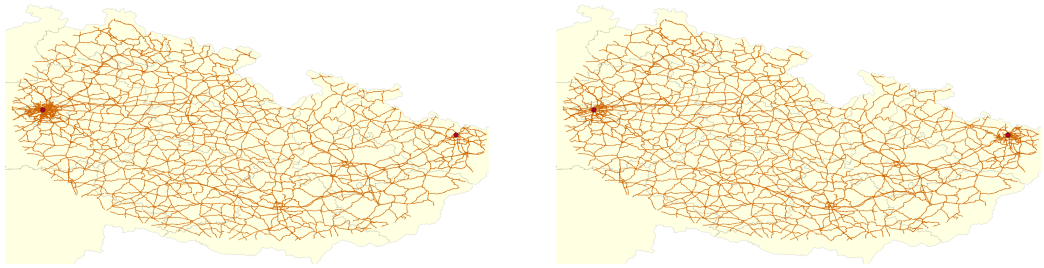
Obrázek 11 znázorňuje první skupinu cest, kterou jsem popisoval výše. Je to vlastně hlavní tepna provozu po celé České Republice. Kdybychom byli schopni procházet jen takto velký graf, tak bychom získávali výsledky v setinách sekund bez nutnosti dalšího upravování grafu. Avšak jak si každý hned domyslí, tak jsme velmi omezení tím, odkud kam jsme vůbec schopni se dostat. Tento graf se dá využít v několika speciálních případech. Když bychom třeba potřebovali najít cestu mezi dvěma z deseti nejlidnatějších měst u nás a nezáleželo by nám, že cesta začíná třeba až někde na okraji města, kde je například nějaký nájezd na dálnici. K tomuto bodu bychom následně mohli dopočítat cestu obyčejným Dijkstrou. To by mohlo platit třeba i pro vyhledání cesty z nějaké blízké vesnice. Tohle je jedno z řešení, kterým se budu v budoucnu věnovat více, avšak nyní nejsou obsahem této práce.

Další, co mě napadlo, bylo to, že když jednou najedu na vyšší kategorii, tak automaticky nemusím prohledávat už ty cesty, které jsou nižší kategorie. Bohužel tohle se ukázalo jako obrovský omyl. Jelikož samotná cesta docela dost často skáče o jednu kategorii níž a výš co 5 km, což byl tedy první problém, ale důležitější bylo to, že když jsem si nastavil nějakou obecnou proměnou, která tedy určovala ty kategorie, co splňují podmínku, tak to rozbilo Dijkstru. Výsledek byl asi takový, že se nakonec propagovala jen jediná cesta a ten zbytek jsem nestihl najít. Bylo možné tam dát nějakou další proměnnou, která by mi navýšení kategorie udělala až o několik kroků později, abych měl tedy cest víc, avšak tady jsem nebyl opravdu nijak schopen určit alespoň trochu rozumné číslo a ani ho nijak odvodit.

### 5.2.4 Filtr grafu – hierarchie ve vzdálenosti od počátečního bodu

Jelikož jsem nebyl v základu schopen nijak využít potenciál toho, že máme nějaké třídy silnic, musel jsem rozvinout tuto myšlenku dál. Vycházel jsem z předešlých poznatků, že je potřeba, abych na začátku mohl využívat všechny kategorie cest, jelikož s největší pravděpodobnostní bude má cesta nakonec vycházet někde od paneláku ve městě a to jsou cesty nízkých kategorií. Ale čím delší trasu jsem ujel, tím méně budu využívat nižší třídy.

Rozhodl jsem se znovu využít GPS souřadnice vrcholů. Rozdělil jsem si graf na tři různé části podle jejich vzdálenosti od počátečního bodu v km. Jednou části jsou všechny třídy, další část je ochuzena o cesty poslední třídy a první část tvoří rozšířenou síť hlavní tepny. Tři části se ukázaly jako ideální, jelikož když se mi mění mé číslo (0-7) kategorie hrany, tak se většinou alespoň drží právě v mých pomyslných hranicích. Dále bylo potřeba vyřešit to, že parametr vzdálenosti, kdy se aktivuje nový filtr, bude jiný pro cestu, která má délku 20 km a která má délku 300 km. Proto jsem zde ještě zakomponoval to, že podle letecké vzdálenosti startu a cíle ovlivním právě tento parametr. Opět metodou pokusů jsem dospěl k tomu, že číslo 100 je ideální.



(a) Podgraf s počátečním vrcholem v Praze (b) Podgraf s počátečním vrcholem v Ostravě

Obrázek 12: Plateau Algorithm - Ukázka podgrafů vzniklých za pomocí více úrovňového dělení cest

Na přiložených obrázcích 12 vidíme, jak to může vypadat ve finále, když aplikujeme tento nový filtr. Na levém obrázku vidíme, jak v okolí Prahy končí již druhá vrstva filtru, tedy prostřední část. Část, která pokrývá všechny kategorie tam nelze ani pořádně rozeznat, ale byla by to tak, kterou překrývá červený puntík a kousek kolem něj. Ale čeho je potřeba si všimnout, že kolem cílového bodu (Ostrava), již žádný takový velký prostor není zabarven, což vyplývá z toho, že tam hledáme jen cesty, které spadají do nejvyšší kategorie. Tohle má za následek to, že počáteční cesta a koncová budou vždy stejné a až od nejbližšího napojení na silnice vyšší třídy budou existovat alternativní cesty. Je to dáno tím, že tento krátký úsek bude vždy jen v jednom z podgrafů a proto tam nelze najít žádnou alternativu. Je to taková daň za toto vylepšení. Ale ve výsledku nám to nemusí úplně vadit, jelikož algoritmus splní účel. Vyvede nás z města a následně máme možnost volby.

Zde je přiložen kód funkce, která mi spočítala vzdálenost dvou GPS bodů v km. Na začátku jsem chybně využíval euklidovské vzdálenosti, ale ta je jen pro 2D prostor. Země je kulatá a proto

Tabulka 8: Plateau Algorithm - Časy tří základních sekcí vylepšené o filtr grafu podle třídy silnic (včetně předešlých vylepšení)

| Sekce | Čas před vylepšením [s] | Čas po vylepšení [s] |
|---|---|---|
| Dijkstra | 4.146 | 0.846 |
| Nalezení plošin | 0.413 | 0.086 |
| Poskládání cest a jejich seřazení | 0.023 | 0.024 |

bylo potřeba využít složitější vzorec pro kouli [4]. Této chyby jsem si všiml až od vzdálenosti zhruba cca 80 km a více. Do té doby to vycházelo relativně pěkně, avšak následně to skákalo neuvěřitelně rychle nahoru a najednou vzdálenost Praha – Ostrava byla mnohem větší než nejdelší letecká vzdálenost od nejvýchodnějšího bodu v České Republice k tomu nejzápadnějšímu. Byla to docela častá chyba i u jiných řešení, které jsem si četl, proto na to raději upozorňuji.

Jak lze vidět z testu v tabulce 8, tak toto vylepšení přináší zásadní zrychlení a to tedy hlavně pro delší vzdálenosti, kde nám moc nepomůže oříznout jen velikost grafu pomocí elipsy, nebo nějakého dalšího útvaru. Ale jak jsem již upozorňoval, jedná se jen o heuristiku, a proto nalezené výsledky nemusí být nutně vždy ty nejlepší. Nalezení nejkratší cesty je zde stále garantováno, ale o těch alternativních to vždy s jistotou nelze říct.

### 5.2.5 Binární halda

Díky výsledkům z nástrojů na prohlížení běhu programu jsem zjistil, že tohle je celkově největší obtíž celého algoritmu. Abych byl konkrétní, tak největší problém dělá to, že hledám již zpracovaný uzel a kontroluji, zda nová cesta k němu není kratší, abych tak vytvořil novou vazbu. Bohužel binární halda neoplývá vlastností, že by se v ní dobře vyhledávalo, ale mnohem důležitější vlastností, že na prvním místě máme vždy nejmenší prvek. Proto když v ní následně hledáme prvek, tak nám nezbývá nic jiného, než to vzít sekvenčním vyhledáváním od začátku a pokračovat dokud jej nenajdeme. Snažil jsem se vymyslet alespoň nějaké urychlení.

Pokusil jsem se ukládat ke všem vrcholům pozici, na které se nacházeli, když jsem je do binární haldy vložil a od ní následně vyhledávat. Podle naměřených experimentů se nacházely vrcholy v zhruba 15 % případů na místě, kde byly vloženy nebo případně o jednu úroveň zanoření nahoru nebo dolů. Tohle mělo za následek zlepšení rychlosti cca o 1 %. Bohužel to mělo mnohem víc negativních přírůstků. Paměťová složitost celého řešení se zvýšila o několik proměnných pro každý vrchol a hlavně bylo potřeba dopsati několik matoucích řádků, které byly ve výsledku matoucí, a činili stávající kód nepřehledným. Z těchto negativních dopadů jsem se rozhodl neimplementovat toto vylepšení do závěrečné verze.

## 5.3 Programátorské techniky a programy použité při práci

Během práce jsem se musel seznámit s několika novými technologiemi pro mě, které mi ve výsledku opravdu ulehčili práci a také umožňovali spolupráci v týmu.

### 5.3.1 CMake

Jedná se o svobodný software, který umožní automatizovaný překlad programu v operačních systémech. Jelikož jsem využíval vývojové studio, od společnosti JetBrains, CLion, které při vytváření projektu využívá přesně tohohle řešení, tak jsem se s ním opravdu často potkával. CMake využívá toho, že do hlavního konfiguračního souboru CMakeLists.txt napíšeme naší konfiguraci, která se následně aplikuje při skládání projektu. Obrovskou výhodou následně je to, že pro přenesení a spuštění projektu na jiném stroji je potřeba jen kopie celého řešení a aby daný stroj podporoval CMake.

### 5.3.2 Git

Jelikož se jednalo o můj první velký týmový projekt, tak jsem se poprvé setkal i s verzovacím systémem. Ukázalo se, že je nezbytnou součástí dnešních moderních technik. Na začátku je to opravdu jednoduchý způsob, jak dostat na váš pracovní stroj nějakou rozpracovanou verzi projektu a následně neskutečně mocný nástroj proto, aby mohl tým mezi sebou sdílet své řešení, ale zároveň si nepřepisovali svá řešení, když to zrovna ještě nechtějí.

### 5.3.3 Unit test

Pro mě obzvlášť nejužitečnější poznatek. Osobně jsem podceňoval význam testů, ale po mé zkušenosti už nikdy nebudu. Bylo neskutečnou výhodou, když jsem si napsal unit testy pro můj algoritmus a následně po každé úpravě jsem mohl testovat, zda se mi změnily mé očekávané výsledky a případně jak moc, jestli jsou ještě v rámci nějaké únosné meze. Také mi to úplně na začátku pomohlo se dostat do struktury celého projektu a nějakým užitečným způsobem jsem si jej mohl osahat. Opět jsme zde využívali otestovaný způsob společnosti Google.

### 5.3.4 HPC

Jelikož bylo potřeba testovat i vznikají software na clusteru, tak jsem si procvičil i práci v Linuxovém prostředí a přihlašovaní na vzdálený server. Obrovskou výhodou bylo to, že jsem měl díky tomu přístup k nástrojům, které jsou jinak velmi vysoce zpoplatněné, jako je třeba VTune od společnosti Intel. Je to profiler, tedy nástroj k analýze běhu programu a využívání jeho prostředků. Z něj jsem zjistil zmiňované problémy s nedokonalým využíváním vláken v některých případech a také to, že samotnému Dijkstrovi nejvíce trvá proces vyhledávání v Binární haldě.

### 5.3.5 QGIS

Jedná se o geografický informační systém, který nám umožňuje pracovat s mapovými podklady. Díky němu jsem byl schopen vytvářet SQL dotaz nad databází a ta mi vrátila požadovaný výsledek, který jsem si byl schopen v tomto programu vykreslit. Pomocí tohoto řešení jsem si mohl vizuálně kontrolovat své výsledky a hlavně vytvářet potřebné obrázky do své práce.

# 6 Experimenty

Nyní ukážu několik případů užití v různých situacích, jejich časovou náročnost a výsledky. Čas budu uvádět v sekundách a vzdálenost je vzdušnou čarou. Testy budu provádět na notebooku a případně výpočetním clusteru Salomon. V obou případech poběží řešení maximálně na dvou jádrech a operační systém je typu Linux. Notebook využívá procesoru Intel Core i5 2520M Sandy Bridg s taktem 2.5 GHz a pamětí RAM 12 GB. Na HPC clusteru jsem využíval vždy jeden node, který obsahuje dva procesory Intel Xeon E5-2680v3 s taktem 2.5 GHz. Každý z nich má 12 jader a mají přístup k paměti o velikosti 128 GB.

Ještě než začnu ukazovat samotné výsledky, tak bych rád upozornil na jeden podstatnou věc související s hardwarem, na který jsem během testování narazil, a to je to, že když spustíme dva stejné testy hned po sobě, tak natažená data v L1-L3 cache paměti procesoru z prvního pokusu se tam stále nachází a výsledný čas pak může být rozdílný od prvního o 10-30 %. Větší počet procent, je pak dán, když jsou dva počáteční body daleko od sebe a hlavně jak moc filtru na graf bylo aplikováno. Proto jsem se snažil vždy ukazovat jen právě první spuštění.

## 6.1 Testy

V této tabulce 9 vidíme několik výsledků, které již využívají všech vylepšení, které jsem ve své práci prezentoval. Když porovnáme tyto naměřené časy a čas 11.247 sec., který jsem naměřil při obyčejné implementaci bez jakéhokoliv vylepšení a vláken, tak se mi podařilo snížit výsledný čas o dva řády. Také zde můžete porovnat čas mezi notebookem a superpočítačem. Všimněme si také toho, že v některých situacích (např. Brno - Olomouc) je vidět velký rozdíl mezi počtem vrcholů v jednom a druhém grafu. Toto zapříčiní, že jedno vlákno skončí dříve a je nuceno čekat na to druhé, a jak jsem psal, tento fakt se nelíbí nástrojům na vyhodnocení běhu programu.

Nyní následují tři testy v tabulce 10 11 12, které ukazují, jak působí na danou vzdálenost jednotlivá vylepšení.

Tabulkou 13 jsem chtěl poukázat na význam hierarchického filtru na velkou vzdálenost. Jak lze vidět tak na vzdálenostech do 10 km se vůbec nic nemění, výsledky zůstávají stejné a mohlo by se i zdát, že je tam další část filtru nadbytečná. Toto má jednoduché vysvětlení, jelikož v obou případech spadne celá oblast do prohledávání všech kategorií cest. Kde je to už mnohem víc zajímavé, jsou oblasti nad 100 km, tam lze pozorovat velké rozdíly v tom, zda byl aplikován filtr pouze elipsou, nebo se využívají i hierarchie.

## 6.2 Výsledné cesty

Na obrázku 13 vidíme pět možných cest z Prahy do Ostravy. Jak lze vidět tak cesty jsou nakonec rozdělené do dvou pomyslných oblastí. Jedna vede kolem Brna a ta druhá vede přes Hradec Králové. Na začátku se rozdělí do dvou různých směrů, ale konec už vychází pro všechny cesty stejně. Řekl bych, že čtyři z pěti cest v této situaci se dají pokládat za dostatečně alternativní

Tabulka 9: Plateau Algorithm - Test několika tras v ČR, kde využívám již všechny optimalizace, které jsem popsal. Jsou zde i časy pro srovnání mezi notebookem a jedním vypočtením uzlem Salomonu (HPC Cluster) [Vrcholy 1 = počet navštívených vrcholů v grafu ze startovního bodu; Vrcholy 2 = počet navštívených vrcholů v grafu z koncového bodu; čas 1 = notebook; čas 2 = jeden výpočetní uzel clusteru]

| Odkud | Kam | Vzdálenost [km] | Vrcholy 1 | Vrcholy 2 | Alternativy | Čas 1 [s] | Čas 2 [s] |
|---|---|---|---|---|---|---|---|
| Praha | Praha | 8 | 33217 | 33195 | 42 | 0.258 | 0.079 |
| Ostrava | Ostrava | 6 | 10364 | 10363 | 44 | 0.073 | 0.023 |
| Brno | Brno | 6 | 8075 | 8059 | 77 | 0.054 | 0.017 |
| Liberec | Liberec | 3 | 4542 | 4537 | 24 | 0.035 | 0.011 |
| Pardubice | Pardubice | 1 | 2309 | 2308 | 7 | 0.017 | 0.005 |
| Praha | Brno | 185 | 45052 | 42865 | 10 | 0.333 | 0.109 |
| Praha | Ostrava | 275 | 81007 | 78444 | 19 | 0.597 | 0.201 |
| Brno | Olomouc | 64 | 21688 | 17095 | 26 | 0.223 | 0.048 |
| Pardubice | Olomouc | 114 | 27106 | 27397 | 29 | 0.375 | 0.065 |
| Pardubice | Opava | 149 | 21812 | 20525 | 24 | 0.201 | 0.049 |

Tabulka 10: Plateau Algorithm - Test jednotlivých vylepšení na krátké trase. Vždy se aplikuje jen jedno vylepšení. Cesta je v rámci Ostravy. Její délka je 6 km. Čas potřebný pro algoritmus bez jakéhokoliv vylepšení je pro notebook 11.032 s a pro výpočetní uzel Salomonu 4.333 s. [Vrcholy 1 = počet navštívených vrcholů v grafu ze startovního bodu; Vrcholy 2 = počet navštívených vrcholů v grafu z koncového bodu; Notebook = čas algoritmu na notebooku; Salomon = čas na jednom výpočetním uzlu clusteru]

| Typ vylepšení | Vrcholy 1 | Vrcholy 2 | Alternativy | Notebook [s] | Salomon [s] |
|---|---|---|---|---|---|
| Vlákna | 945314 | 945462 | 12060 | 6.817 | 2.969 |
| Redukce výsledků | 945314 | 945462 | 3 | 9.668 | 3.596 |
| Filtr grafu - Elipsa | 10364 | 10363 | 396 | 0.138 | 0.033 |
| Filtr grafu – Hierarchie | 127211 | 126897 | 3014 | 2.263 | 0.557 |

Tabulka 11: Plateau Algorithm - Test jednotlivých vylepšení na středně dlouhé trase. Vždy se aplikuje jen jedno vylepšení. Cesta je Brno - Olomouc. Její délka je 64 km. Čas potřebný pro algoritmus bez jakéhokoliv vylepšení je pro notebook 12.290 s a pro výpočetní uzel Salomonu 4.301 s. [Vrcholy 1 = počet navštívených vrcholů v grafu ze startovního bodu; Vrcholy 2 = počet navštívených vrcholů v grafu z koncového bodu; Notebook = čas algoritmu na notebooku; Salomon = čas na jednom výpočetním uzlu clusteru]

| Typ vylepšení | Vrcholy 1 | Vrcholy 2 | Alternativy | Notebook [s] | Salomon [s] |
|---|---|---|---|---|---|
| Vlákna | 945314 | 945462 | 13392 | 7.215 | 3.021 |
| Redukce výsledků | 945314 | 945462 | 7 | 10.371 | 3.715 |
| Filtr grafu - Elipsa | 51656 | 51507 | 1249 | 0.821 | 0.189 |
| Filtr grafu – Hierarchie | 131036 | 131013 | 3037 | 1.948 | 0.5611 |

Tabulka 12: Plateau Algorithm - Test jednotlivých vylepšení na dlouhé trase. Vždy se aplikuje jen jedno vylepšení. Cesta je Praha - Ostrava. Její délka je 275 km. Čas potřebný pro algoritmus bez jakéhokoliv vylepšení je pro notebook 11.657 s a pro výpočetní uzel Salomonu 4.427 s. [Vrcholy 1 = počet navštívených vrcholů v grafu ze startovního bodu; Vrcholy 2 = počet navštívených vrcholů v grafu z koncového bodu; Notebook = čas algoritmu na notebooku; Salomon = čas na jednom výpočetním uzlu clusteru]

| Typ vylepšení | Vrcholy 1 | Vrcholy 2 | Alternativy | Notebook [s] | Salomon [s] |
|---|---|---|---|---|---|
| Vlákna | 945314 | 945462 | 15732 | 7.511 | 3.221 |
| Redukce výsledků | 945314 | 945462 | 13 | 9.931 | 3.704 |
| Filtr grafu - Elipsa | 637456 | 637375 | 12025 | 8.168 | 3.224 |
| Filtr grafu – Hierarchie | 116606 | 114091 | 3098 | 1.804 | 0.511 |

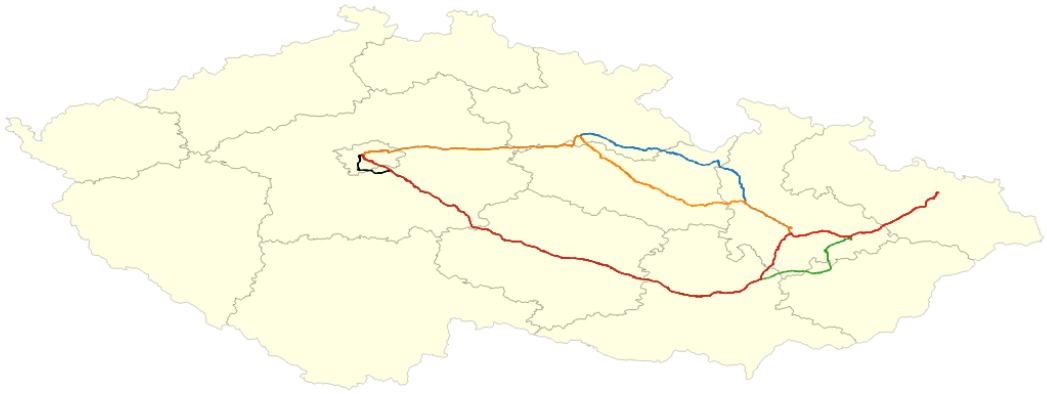
Tabulka 13: Plateau Algorithm - Test kompletního řešení s a bez hierarchického filtru. [(Obě informace o vrcholech se vztahují k testu bez hierarchického filtru) Vrcholy 1 = počet navštívených vrcholu v grafu ze startovního bodu; Vrcholy 2 = počet navštívených vrcholu v grafu z koncového bodu; Čas 1 = filtr obsahuje jen elipsu, hierarchie ignoruje, běží na notebooku; Čas 2 = kompletní řešení na notebooku]

| Odkud | Kam | Vzdálenost [km] | Vrcholy 1 | Vrcholy 2 | Alternativy | Čas 1 [s] | Čas 2 [s] |
|---|---|---|---|---|---|---|---|
| Praha | Praha | 8 | 33217 | 33195 | 42 | 0.240 | 0.258 |
| Ostrava | Ostrava | 6 | 10364 | 10363 | 44 | 0.069 | 0.073 |
| Brno | Brno | 6 | 8075 | 8059 | 77 | 0.056 | 0.054 |
| Liberec | Liberec | 3 | 4542 | 4537 | 24 | 0.034 | 0.035 |
| Pardubice | Pardubice | 1 | 2309 | 2308 | 7 | 0.016 | 0.017 |
| Praha | Brno | 185 | 312756 | 312536 | 11 | 2.151 | 0.333 |
| Praha | Ostrava | 275 | 637456 | 637375 | 19 | 4.411 | 0.597 |
| Brno | Olomouc | 64 | 51656 | 51507 | 27 | 0.335 | 0.223 |
| Pardubice | Olomouc | 114 | 108180 | 108138 | 17 | 0.687 | 0.375 |
| Pardubice | Opava | 149 | 150765 | 150714 | 14 | 0.988 | 0.201 |

cestu, respektive tři, když budu počítat, že červená trasa je ta nejlepší a tedy ostatní jsou právě k ní alternativou.

Ještě jsem vytvořil další dva obrázky s výslednou cestou, kde na prvním 14 lze vidět cesty z Plzně do Olomouce a na druhém 15 jsou vyobrazeny cesty z Liberce do Brna.

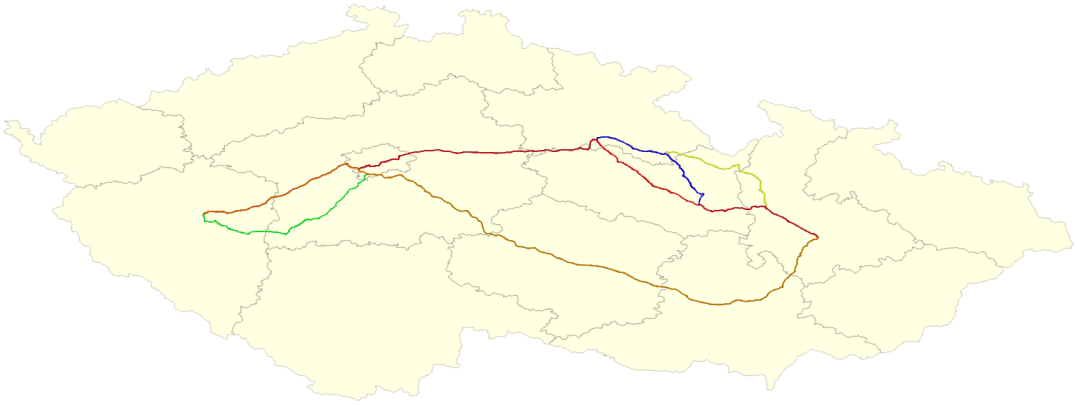
Na obrázku 16 lze vidět deset různých cest od budovy IT4Innovations v Ostravě s cílem ČEZ Arénou. Zde bych řekl, že se jedná o velmi povedený výsledek, jelikož jak je vidno z obrázku, tak existují jen dvě hlavní přístupové trasy a obě jsou využité. Následně je jen rozdíl mezi distribucí v oblasti Poruby a Arény. Na grafu velikosti většího města dostáváme dobré výsledky.



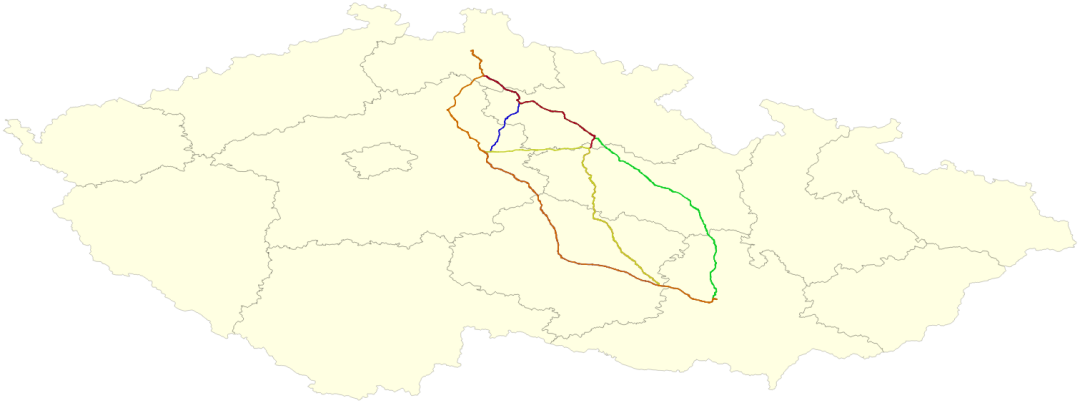
Obrázek 13: Ukázka výsledných cest z Prahy do Ostravy

## 6.3 Shrnutí

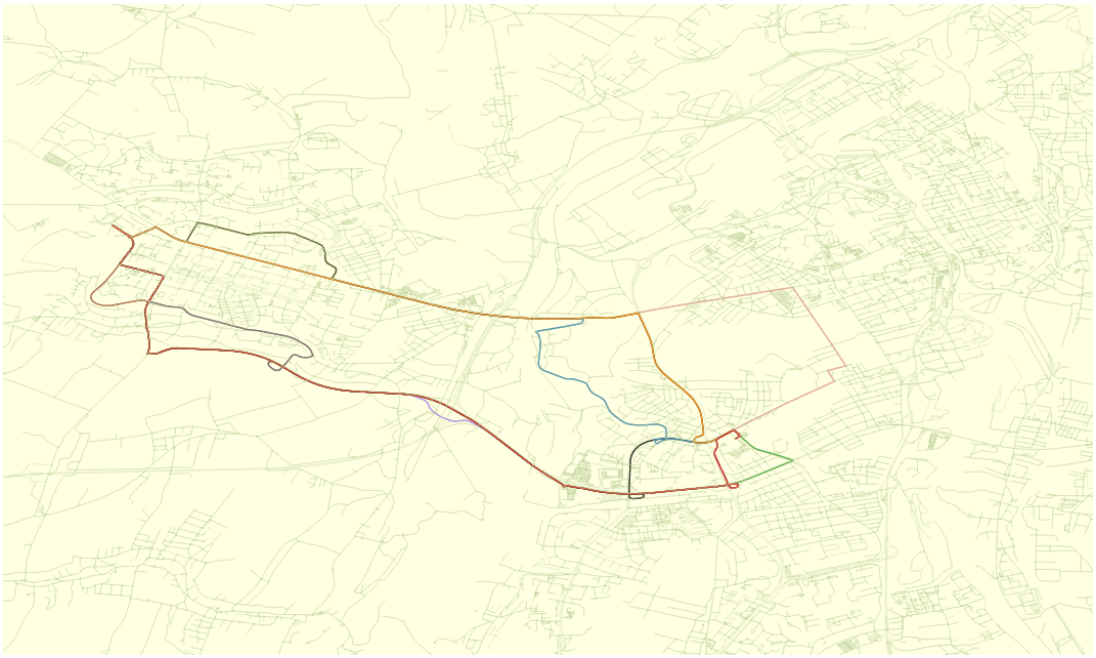
Pomocí testů se dokázalo, že všechny vylepšení zkracují potřebnou dobu na výpočet algoritmu. Dále je vidět, že čím delší vzdálenost mezi startem a cílem je, tím více se vyplatí implementovat hierarchický filtr. Využití vláken se vyplatilo. Výpočet na clusteru je podle očekávání mnohem rychlejší než na notebooku.



Obrázek 14: Ukázka výsledných cest z Plzně do Olomouce



Obrázek 15: Ukázka výsledných cest z Liberce do Brna



Obrázek 16: Ukázka deseti cest z IT4I do ČEZ Arény (Ostrava)

# 7 Závěr

Práce popisuje implementaci Plateau algoritmu, který je i v základní podobě schopen vyhledat požadovaná alternativní cesty, avšak cena tohoto výpočtu je velmi vysoká a navíc je potřeba implementovat filtr, který by nějak třídil výsledky. Proto je určitě vhodné využít některá vylepšení.

Při rozhodování, kterou optimalizaci Plateau algoritmu zvolit, můžeme využít výsledky testů z kapitoly 6. Kdybychom chtěli například prohledávat jen podgraf o velikosti města, určitě by stačila implementace filtru grafu elipsou, případně jen rovnou mít graf, který by obsahoval konkrétní část mapy. V případě, že by tento algoritmus prohledával graf, který by měl více než 150 km na délku (mimo jiné by záleželo i na počtu vrcholů v tomto grafu, ale předpokládám podobnou hustotu, jako měl můj testovací graf), doporučuji využít všechny vylepšení. Přestože optimalizace, která se týká filtru grafu, nezaručuje vždy to, že najdu ty nejlepší cesty, tak je vhodné je využít, jelikož to jsou právě ony, co nám nejvíce urychlí výpočet.

Ukázal jsem, že optimální je použít implementaci Plateau algoritmu, která využívá dvou jader procesoru. Přestože se může stát, že vlákna, která poběží, nebudou stejně vytížena. Je to daň za výrazné zlepšení první části algoritmu. To tedy znamená, že na mnou testovaném node HPC clusteru, který měl 24 jader, by najednou mohlo běžet 12 různých dotazů.

Výsledná implementace se bude využívat v H2020 projektu Antarex, kde VŠB-TU Ostrava, reprezentovaná IT4Innovations, optimalizuje kód pro Self-Adaptive Navigation System. Součástí tohoto systému je Server-Side Routing, který na několika výpočetních nodech musí zároveň zpracovat uživatelské požadavky na dotazované trasy více uživatelů.

Na tuto práci by se dalo navázat dalšími optimalizacemi, jako je například zmiňované řešení s hierarchiemi, kdy se cesta rozdělí na tří části. Prostřední část bude využívat jen silnice první třídy. První a třetí část musí následně najít optimální cestu k vrcholům, kterými prostřední část začíná, respektive končí.

# Literatura


[1] BARNEY, Blaise. OpenMP. Computing.llnl.gov [online]. Lawrence Livermore National Laboratory [cit. 2018-04-21]. Dostupné z: https://computing.llnl.gov/tutorials/openMP/

[2] ČERNÝ, Jakub. Základní grafové algoritmy [online]. [cit. 2018-04-23]. Dostupné z: https://kam.mff.cuni.cz/~kuba/ka/pruchod.pdf

[3] HAVÍČEK Petr. Algoritmy pro směrování dopravních vozidel. Master's thesis, VŠB - Technická univerzita Ostrava, 2010.

[4] KADLEČEK, Jiří. Geometrie v rovině a prostoru: pro střední školy. Praha: Prometheus, 1996. ISBN 80-7196-017-9.

[5] PARASKEVOPOULOS, Andreas a ZAROLIAGIS, Christos . Improved Alternative Route Planning [online]. Computer Technology Institute & Press "Diophantus" Patras University Campus, 2013 [cit. 2018-04-21]. Dostupné z: https://hal.inria.fr/hal-00871739/document

[6] TOMIS, Radek. Optimalizace směrování dopravních vozidel [online]. Ostrava, 2011 [cit. 2018-04-21]. Dostupné z: https://dspace.vsb.cz/handle/10084/87503. Bakalářská práce. VŠB Ostrava.

[7] VOLEK, Josef a LINDA, Bohdan. Teorie grafů - aplikace v dopravě a veřejné správě. Pardubice: Univerzita Pardubice, 2012. ISBN 978-80-7395-225-9.

[8] C/C++. Builder.cz [online]. 2000 [cit. 2018-04-21]. Dostupné z: http://www.builder.cz/rubriky/c/c--/zaklady-oop-v-c-od-c-k-c--155657cz

[9] GOOGLE-SPARSEHASH@GOOGLEGROUPS.COM. Sparsehash. GitHub [online]. 2005 [cit. 2018-04-21]. Dostupné z: https://github.com/sparsehash/sparsehash