Západočeská univerzita v Plzni
Fakulta aplikovaných věd

Implementace paralelního algoritmu pro hledání optimální cesty závislé na čase

Bakalářská práce

František Kolovský

Vedoucí práce:
Ing. Jan JEŽEK, Ph.D.

Plzeň, jaro 2015

Prohlášení

Prohlašuji, že tato diplomová práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

V Plzni dne ....

František Kolovský

Poděkování

Chtěl bych poděkovat vedoucímu práce Janu Ježkovi za metodické vedení a věcné připomínky. V práci jsem používal výpočetní výkon poskytovaný přes virtuální organizaci MetaCentrum v rámci projektu "Projects of Large Infrastructure for Research, Development, and Innovations" (LM2010005), za což velmi děkuji.

Acknowledgment

I thank supervisor for comments and methodical guidance. Access to computing and storage facilities owned by parties and projects contributing to the National Grid Infrastructure MetaCentrum, provided under the programme "Projects of Large Infrastructure for Research, Development, and Innovations" (LM2010005), is greatly appreciated.

Abstrakt

Tato práce řeší problém hledání nejkratší cesty v silniční síti, kde doba průjezdu úsekem závisí na čase, konkrétně problém volby doby výjezdu pro dosažení nejkratšího času cesty. Cílem této práce je vyvinout a implementovat paralelní algoritmus. Zaměřil jsem se na algoritmus pro distribuované prostředí na bázi modelu MapReduce.

Práce představuje MapReduce algoritmus pracující ve spojitém čase a založený na LCA (Label Corecting Algorithm), který byl implementován v prostředí Apache Spark za pomocí nástavby GraphX určené pro grafové analýzy. Jako graf byla použita silniční síť z OSM a transportní funkce byly vygenerovány náhodně.

Byl navržen a implementován paralelní algoritmus se složitostí $O(n^2)$ a dobrou škálovatelností. Dále byly provedeny výkonnostní testy, které ukázaly, že vyvinutý algoritmus je vhodný pro velmi velké grafy (které se nevejdou do paměti jednoho počítače), protože režie distribuovaného systému u malých grafů tvoří velké procento výpočetního času.

Klíčová slova

Apache Spark, nejrychlejší cesta v grafu, časová závislost, distribuované prostředí, MapReduce

Obsah

1 Teoretický úvod a kategorizace **6**

1.1 Graf 6

1.2 Funkce transportního času 6

1.3 Kategorizace problému 7

1.4 MapReduce systém 9

2 Použitý algoritmus **11**

2.1 Formulace problému 11

2.2 Label corecting algoritmus (LCA) 11

2.3 Implementace operací minimum, porovnání a vnoření pro po částech lineární funkci 13

2.3.1 Minimum ze dvou funkcí 14

2.3.2 Vnoření dvou funkcí 14

2.3.3 Porovnání dvou funkcí 15

3 Implementace v MapReduce **17**

3.1 Použité technologie a data 17

3.1.1 Apache Spark - GraphX 17

3.1.2 Cassandra 18

3.1.3 Zkušební dataset 18

3.2 Implementace 18

3.3 MapReduce algoritmus pro všechny odjezdové časy 21

3.3.1 Inicializace 22

3.3.2 Hlavní cyklus 22

4 Složitost a výkon **24**

4.1 Testy 25

4.1.1 Závislost výpočetního času na velikosti sítě 25

4.1.2 Závislost výpočetního času na počtu vláken (procesorů) 26

A Obsah přiloženého CD **31**

Úvod

Nalezení nejrychlejší cesty pro automobil z místa A do místa B závisí na mnoha parametrech jako jsou vzdálenost, povolená rychlost, hustota dopravy na komunikacích a počasí. Většina těchto jevů je proměnná během denní doby. Z tohoto důvodu je důležité počítat nejrychlejší trasu s ohledem na časově proměnnou průjezdnost pozemních komunikací.

V této práci se budu zabývat právě hledáním nejrychlejší cesty silniční sítí závislé na času (Time-Dependent Shortest Path). Konkrétně hledáním optimálního času pro odjezd během dne tak, aby cesta trvala co nejkratší dobu. Tento problém je v dané oblasti jedním z nejobecnějších.

Data pro tento problém jsou často velká a špatně uchopitelná na jednom počítači, a proto se využívá clusteru. Jednou z nejmodernějších metod jsou systémy na bázi MapReduce, které umožňují výpočet zpracovat paralelně na několika procesorech.

V první části práce se budu zabývat kategorizací a definicí problému, v druhé vývojem MapReduce algoritmu založeném na LCA (Label corecting algoritmus) a jeho implementací v distribuované výpočetním prostředí Apache Spark.

Kapitola 1

Teoretický úvod a kategorizace

V této kapitole jsou shrnuty různé metody pro hledání nejrychlejší cesty v silniční síti závislé na čase a jejich dělení. Dále zde jsou definovány základní pojmy a popsány jejich základní vlastnosti, které budou využívány dále v textu.

1.1 Graf

Graf je množina hran a vrcholů $G = (E, V)$, kde E je množina hran a V je množina vrcholů. V našem případě jsou hrany ulice a části sinic a vrcholy (uzly) jsou křižovatky.

Každá hrana $(i, j) \in E$ má počáteční a koncový vrchol ($i, j \in V$) tj. spojuje dva vrcholy. Každý vrchol může mít libovolný počet sousedních vrcholů.

Rozlišujeme orientovaný a neorientovaný graf. V orientovaném grafu jsou hrany průchodné pouze v jednom směru. V reálné silniční síti to jsou jednosměrné ulice. Vzhledem k tomu, že průjezdnost v každém směru je jiná, tak graf bude vždy orientovaný.

1.2 Funkce transportního času

Pro každou hranu v grafu je definována takzvaná cena hrany. V tomto případě je cena čas, za který se dá překonat. Vzhledem k tomu, že graf má být závislý na odjezdovém čase během dne, tak i cena hrany musí být závislá na čase.

Pro každou hranu $(i, j) \in E$ je tedy definována příjezdová funkce $a_{ij}(t_d)$. Tato funkce vrací příjezdový čas do vrcholu j, který je závislý na odjezdovém čase z vrcholu i. Když od této funkce odečteme odjezdový čas (t_d), dostaneme takzvanou

transportní funkci $(a_{ij}(t_d) - t_d)$. Tato funkce udává kolik času potřebujeme na překonání hrany.

Tyto funkce by měly splňovat podmínku [FRIESZ93]:

$$\frac{d}{dt_d} a_{ij}(t_d) > 0 \Rightarrow \frac{d}{dt_d}(a_{ij}(t_d) - t_d) > -1 \tag{1.1}$$

Tato podmínka vychází v vlastnosti silniční sítě FIFO (First In, First Out). Vlastnost FIFO znamená, že když do sítě vjede vozidlo A a po něm vozidlo B, tak vyjedou ve stejném pořadí jako do sítě vjely. Nemůže vozidlo B předjet vozidlo A.

Další podmínka je, že:

$$a_{ij}(t_d) > t_d \Rightarrow (a_{ij}(t_d) - t_d) > 0 \tag{1.2}$$

Tato podmínka vyjadřuje, že vozidlo nemůže dojet do vrcholu j dříve, než odjede z vrcholu i (Nelze cestovat v čase).

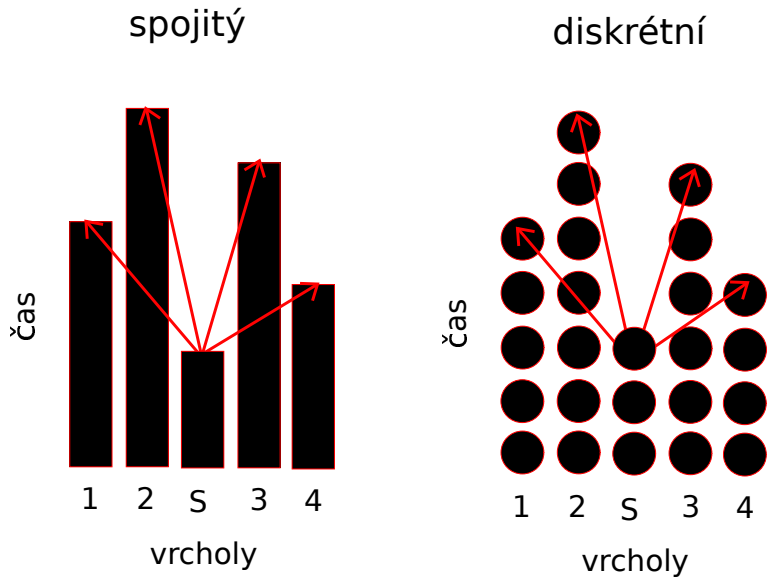
1.3 Kategorizace problému

Nejdříve je třeba vysvětlit, jak funguje nejzákladnější a nejpoužívanější algoritmus pro hledání statické nejkratší cesty. Je to Dijkstrův algoritmus. Při inicializaci se do počátečního vrcholu (s) nastaví hodnota 0 $(dist(s) = 0)$ a do ostatních ∞ $(\forall i \in V : dist(i) = \infty)$. Počáteční vrchol se vloží do prioritní fronty. V hlavním cyklu se vybere vrchol z prioritní fronty a pro všechny jeho sousedy (j) se spočte nová hodnota jako $dist(j) = min(dist(i) + w(i, j), dist(j))$, kde $w(i, j)$ je cena hrany. Tyto vrcholy se vloží do prioritní fronty a celý cyklus se opakuje.

Tento algoritmus, tak jak je, nelze použít pro graf závislý na čase, protože nelze použít prioritní frontu. Existuje mnoho metod jak tento problém vyřešit.

Řešení se dají rozdělit na dvě hlavní skupiny a to jsou algoritmy pracující v diskrétním čase a algoritmy pracující ve spojitém čase. To znamená, že závislost každé hrany v grafu na času je buď vyjádřena diskrétní funkcí, nebo spojitou funkcí (viz obrázek 1.1).

Diskrétní algoritmy jsou podrobně popsány v [CHABINI02] nebo [CHABINI98]. Nejrozšířenější jsou algoritmy DOT a IOT. Výhoda těchto algoritmů je, že se v nich dají v určité míře využít rychlé statické algoritmy. Algoritmy založenými na spojitém čase se podrobně zabývá B. Dean ve své dizertační práci [DEAN99] a pozdějších článcích (2004). V těchto pracích jsou především popsány algoritmy Label Corecting (LCA) a Label Setting. Modifikace právě zmíněného LCA byla použita v této práci.



Obrázek 1.1: Znázornění spojitého a diskrétního grafu (strom nejkratších cest z počátečního vrcholu) [DEAN99]

.

Výhoda řešení problému ve spojitém čase je lepší aproximace reálné příjezdové funkce a tím přesnější řešení.

Další rozdělení řešení tohoto problému můžeme udělat podle statických algoritmů, na kterých je algoritmus založen. Takto jde využít téměř každý statický algoritmus. Řešení založeno na Dijkstra je použito například v práci [DING08]. Řešení pomocí A* je popsáno v článcích [ZHAO08] a [OHSHIMA06]. V mnoha dalších pracích se používá například Bi-directional, Bellman-Ford [DING08], ALT [ZHAO08].

Zajímavé řešení odlišné od ostatních je popsáno v práci [HAGHANI05], kde autor používá genetické algoritmy.

Podmínkou pro paralelní řešení je dobrá paralelizovatelnost sekvenčního algoritmu. Práce zabývající se paralelním řešením v diskrétním čase je [GANUGPATI98] a [CHABINI02]. Paralelní algoritmus založený na geografické dekompozici je popsán v práci [AYED11]. LCA (Label Correcting Algorithm) je použit v práci [LAWSON13] v prostředí C/C++, Matlab a MPI (Message Passing Interface).

Zajímavou implementaci popsal ve své dizertační práci R. C. Sperb [SPERB10]. Autor používá databázovou nástavbu PostGIS a jazyk PL/pgSQL.

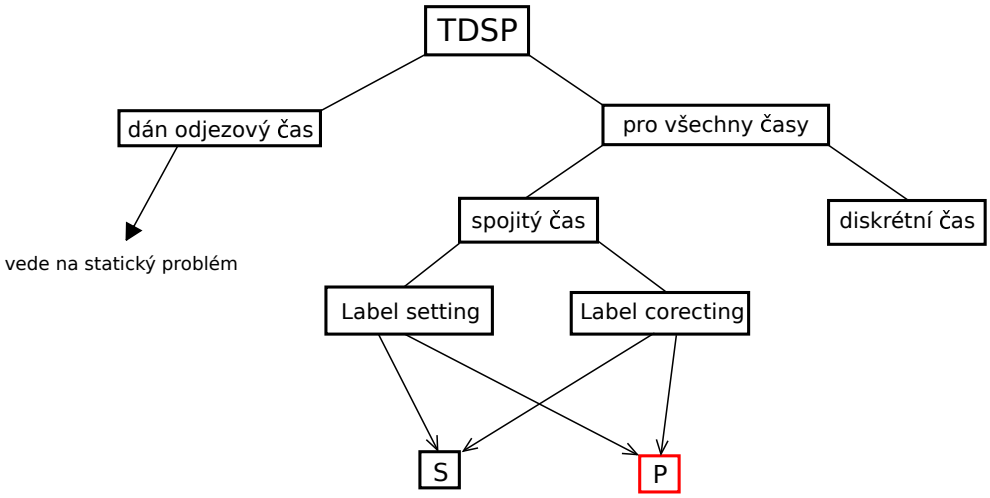
Zde budu uvádět kategorizaci tohoto problému podle práce [DEAN99]. Algoritmy dělíme následujícím způsobem:

- pro jeden odjezdový čas

  - lehká modifikace statických algoritmů

- pro všechny odjezdové časy
  - řešení ve spojitém čase
  - řešení v diskrétním čase
  - one-to-all: z jednoho vrcholu do všech vrcholů
  - all-to-one: ze všech vrcholů do jednoho vrcholu
  - all-to-all: ze všech vrcholů do všech vrcholů
  - one-to one: z jednoho vrcholu do jednoho vrcholu
  - paralelní
  - sériový

Schematicky je rozdělení znázorněno na obrázku 1.2.



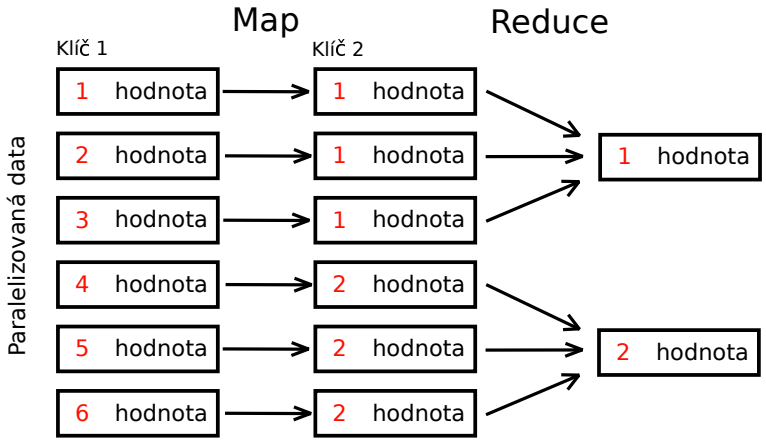
Obrázek 1.2: Schematické rozdělení algoritmů pro Time-Depend shortest path problem (TDSP) (větve v práci neřešené jsou zanedbané). Červeně je označen použitý algoritmus. P - paralelní S - sériový

1.4 MapReduce systém

MapReduce je model pro paralelní zpracování velkého množství dat. V tomto modelu se používají dvě základní funkce Map a Reduce.

Výhoda tohoto modelu je, že obě základní funkce probíhají paralelně. To znamená, že aplikace, napsaná v systému implementující tento model, může běžet na mnoha počítačích najednou (až tisíce počítačů v clusteru), a tudíž je zpracování dat rychlé a efektivní. Takto se dají rychle a jednoduše zpracovávat datasety o velikostech v řádech petabajtů. Nejznámějšími zástupci v této oblasti zpracování dat jsou systémy jako Hadoop a Spark.

Vstupem funkce Map je dvojice (klíč 1, hodnota 1) a výstupem je dvojice (klíč 2, hodnota 2). K tomuto převodu se používá mapovací funkce (Map), která se aplikuje na každou dvojici na vstupu. Funkce Reduce zkombinuje (spojí) všechny hodnoty se stejným klíčem (obrázek 1.3) [DEAN04].



Obrázek 1.3: Schéma modelu MapReduce

Kapitola 2

Použitý algoritmus

2.1 Formulace problému

V rámci této práce bude řešen problém one-to-all pro všechny odjezdové časy. Na tento problém vede většina ostatních podproblémů.

Vstupem pro algoritmus je graf ($G = (E, V)$), příjezdové funkce pro každou hranu ($\forall(i,j) \in E : \exists a_{ij}(t_d)$) a počáteční vrchol. Výstupem jsou příjezdové funkce ke každému vrcholu (funkce času odjezdu z počátečního vrcholu)($\forall i \in V : \exists EA_{si}$), kde EA_{si} je příjezdový čas ($a_{si}(t_d)$) po nejrychlejší cestě, s je počáteční vrchol.

Hledáme optimální čas odjezdu během dne tak, aby cesta trvala trvala co nejkratší dobu.

Pro řešení tohoto problému byl vybrán algoritmus, který pracuje ve spojitém čase, protože spojitý čas více odpovídá realitě. Dále byl vybrán Label corecting algorimus (LCA). Je to poměrně jednoduchý algoritmus, ale je vhodný, protože se dobře paralelizuje a pro MapReduce model je nejvhodnější. Ztráty na výkonu se nahradí výpočetní silou.

2.2 Label corecting algoritmus (LCA)

Label Correcting algoritmus je jeden z nejjednodušších algoritmů pro hledání nejkratší cesty v grafu. Nejprve je do každého vrcholu přiřazena hodnota ∞ a do počátečního vrcholu funkce t ($EA_{ss} = t$). Poté se vybere libovolný vrchol (A) (nemusí to být sousední bod počátečního) a pro všechny sousední vrcholy se spočte hodnota příjezdové funkce (EA_{sj}) jako kombinace příjezdové funkce ve vrcholu A (vnitřní funkce) a příjezdové funkce příslušné hrany ($f_j(t) = a_{ij}(EA_{si})$). Zjistí se, zda

takto spočtená příjezdová funkce je alespoň na nějakém intervalu menší, než původní funkce ve vrcholu ($EA_{sj}(t) \geq f_j(t)$). Když ano, tak se spočte minimum z této ($f_j(t)$) a původní funkce (EA_{sj}) a uloží se do vrcholu. Toto se provádí tak dlouho, dokud pro všechny vrcholy nebude platit ukončovací podmínka tj. $\forall(i,j) \in E$: $EA_{sj} \leq a_{ij}(EA_{si})$ (algoritmus 2.2). Celý algoritmus je napsán v pseudokódu (algoritmus 2.1).

Na tomto algoritmu je postaven výsledný MapReduce algoritmus, který je popsán dále.

Algorithm 2.1 LCA pro všechny odjezdové časy

```
1: For all i ∈ V : EA_si(t) = ∞
2: EA_ss(t) = t
3: while viz ukončovací podmínka do
4:    Select some i ∈ V
5:    for all neighbors j do
6:       f(t) = min(EA_sj(t), a_ij(EA_si(t))
7:       if EA_sj(t) ≠ f(t) then
8:          EA_sj(t) = f(t)
9:       end if
10:   end for
11: end while
```

Algorithm 2.2 Podmínka ukončení cyklu

```
1: for all i ∈ V do
2:    for all neighbors j do
3:       if EA_sj(t) ≥ a_ij(EA_si) then
4:          return FALSE
5:       end if
6:    end for
7: end for
8: return TRUE
```

Jeho složitost je oproti Dijkstra algoritmu mnohem větší. Ale Dijkstra algoritmus nelze použít, protože se v něm používá prioritní fronta. Prioritní frontu nelze použít, protože nelze jednoznačně porovnat dvě funkce. Na nějakém intervalu může být menší jedna a na druhém druhá funkce. Tohoto problému se zbavíme, když dosadíme odjezdový čas. Cíl práce je ale implementovat algoritmus pro všechny časy.

Jak je vidět v algoritmu 2.2 a 2.1. Je zde použito několik operací:

- minimum ($min(f(t), g(t))$)
- porovnání ($f(t) < g(t)$)
- vnoření (kumulace) ($f(g(t))$)

Tyto operace jsou pro skalární hodnoty jednoduše implementovatelné. Například minimum:

```
1: if a < b then
2:     return a
3: end if
4: return b
```

Ostatní operace se implementují podobně jednoduše. Problém nastává při implementaci těchto operací pro funkce. Už nebudou takto jednoduché a jejich implementace závisí na implementaci příjezdové funkce. Tak se dostáváme k problému, jak realizovat příjezdovou funkci v paměti počítače.

Realizace příjezdové funkce by měla vypadat tak, aby splňovala následující podmínky:

- Provádění operací (minimum, porovnání a vnoření) by mělo být výpočetně nenáročné
- Výsledek operace by měl být uložitelný stejným způsoben jako vstupní funkce
- Výsledek operace by měl zabírat přibližně stejně paměti

Ve stávajících řešeních se setkáme s "po částech"lineární funkcí. Dá se diskutovat, jestli je to ideální řešení, ale vyskytuje se v literatuře [DING08], [DEAN99]. Samozřejmě pro tuto implementaci musí platit stejné teoretické podmínky uvedené výše.

2.3 Implementace operací minimum, porovnání a vnoření pro po částech lineární funkci

Na implementaci těchto funkcí závisí celková rychlost a stabilita algoritmu, proto je třeba implementaci věnovat pozornost.

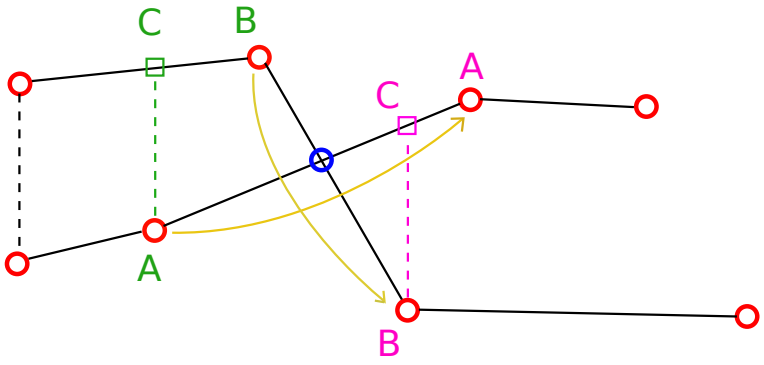
Je třeba, aby tyto operace měly konstantní složitost (nesmí záviset na velikosti grafu) a výsledná příjezdová funkce nesmí nabírat na složitosti (musí zabírat stejně paměti).

2.3.1 Minimum ze dvou funkcí

Jak je vidět na obrázku 2.1, nejdříve je třeba najít průsečíky funkcí a pak překopírovat příslušné části těchto funkcí. Zde bude uvedena pouze základní myšlenka procesu pro získání minima.

Vstupem jsou dvě funkce. Když jedna z nich má funkční hodnotu na celém definičním oboru nekonečno, tak funkce vrátí tu druhou. Pokud jsou obě funkce konečné, tak program vezme první bod obou funkcí (A, B) a porovná jejich x souřadnice. K bodu s menší x souřadnicí (například A) najde bod na druhé funkci o stejné x souřadnici (C). Potom tyto dvě funkční hodnoty porovná. Pokud je menší funkční hodnota v bodě A, tak do výsledné funkce je přidán tento bod. Pokud je to naopak, tak se nepřidá nic. Najde novou dvojici bodů, tak že vezme bod B a další bod na funkci po A. Pokud se x souřadnice rovnaly, posouváme oba body. Zároveň se hlídá, která funkce je aktuálně menší, když se to změní počítá se průsečík příslušných lineárních funkcí.

V zeleném případě na obrázku vidíte výše popisovaný případ. Po dvou iteracích algoritmus dojde do fialového případu a zjistí, že se funkce prohodily, spočítá průsečík (modré kolečko) a zařadí ho do výstupu.



Obrázek 2.1: Popis algoritmu funkce minimum

2.3.2 Vnoření dvou funkcí

Vnoření dvou příjezdových funkcí je nejproblémovější operace. Například kdyby příjezdová funkce byla reprezentována polynomem 5 stupně, tak výstup bude polynom

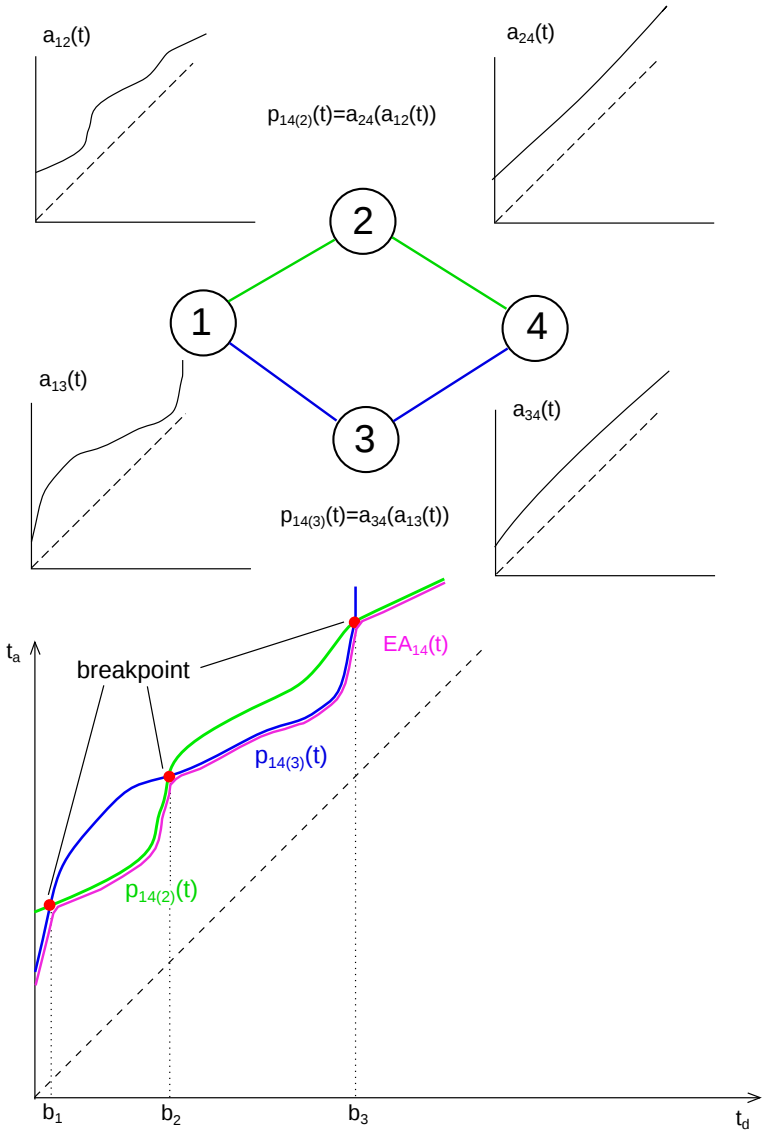
25 stupně! Stejná situace je i s implementací po částech lineární funkce. Do výsledné funkce se promítnou lomové body jak z vnitřní, tak z vnější. To má za následek neustálé zvyšování počtu lomových bodů velmi rychlím tempem (počet hran na cestě * počet lomových bodů). S tím je spojeno zvýšení složitosti algoritmu.

Z toho vyplývá požadavek, aby výsledná funkce měla přibližně stejný počet lomových bodů jako vstupní funkce. To znamená, že výsledek nebude exaktně správný. Je tedy nutné zkonstruovat funkci přibližně. V našem případě byl zvolen zjednodušený postup. Spočtou se hodnoty pouze pro lomové body vnitřní funkce (za předpokladu, že vnitřní a vnější funkce mají přibližně stejný počet lomových bodů).

2.3.3 Porovnání dvou funkcí

Algoritmus pro porovnání funguje stejně jako pro počítání minima. Tento úkon se provádí na vrcholech při porovnání (aktualizaci příjezdové funkce na vrcholu). Chceme vědět, zda je alespoň část nové funkce pod původní. Tedy algoritmus postupuje jako u minima, ale neukládá body a čeká až se nová funkce dostane pod startu funkci ve vrcholu. Až se tak stane, vrátí *true*, jestliže se tak nestane vrátí *false*.

Na obrázku 2.2 je ukázka algoritmu a operací na jednoduchém grafu. Jak je vidět, nejprve se spočte příjezdová funkce do vrcholu 4 pomocí vrcholu 2 a 3 ($p_{14(2)}$ a $p_{14(3)}$). Pak se provede minimum z těchto funkcí a to je výsledná příjezdová funkce do vrcholu 4 (EA_{14}).



Obrázek 2.2: Ukázka algoritmu na jednoduchém grafu, kde $p_{ij(k)}(t)$ je cesta z i do j pomocí vrcholu k.

Kapitola 3

Implementace v MapReduce

V této kapitole se budu zabývat konkrétní implementací algoritmu v MapReduce modelu. Konkrétně byl použit systém Apache Spark využívající distribuovanou databázi Cassandra jako úložiště vstupních dat. Dále se v této kapitole budu zabývat použitelností dostupností dat pro algoritmus.

Všechny zdrojové kódy jsou na přiloženém CD.

3.1 Použité technologie a data

3.1.1 Apache Spark - GraphX

Apache Spark je open-source (Apache License 2.0) framework pro počítání na clusteru používající MapReduce model. Vývoj tohoto nástroje začal v roce 2010 na UC Berkeley.

Základním principem je, že výpočty (operace) nad daty jsou prováděny paralelně na mnoha uzlech clusteru (jak mapovací tak reduce funkce). Je tudíž důležité mít i data uložena distribuovaně na jednotlivcích výpočetních uzlech. K tomu slouží distribuovaná úložiště.

Spark může být spuštěn na různých typech clastrů a to na Hadoop YARN, Apache Mesos a na standardním Spark clusteru. Dále může pracovat s distribuovanými úložišti Hadoop Distributed File System (HDFS), Cassandra a Hbase. Aplikace lze psát v programovacích jazycích Scala, Java a Python. Pro práci s grafy je určena nástavba GraphX (distributed graph processing framework). Tato nástavba podporuje pouze jazyk Scala. Přes tento framework jdou jednoduše psát aplikace zaměřené na analýzu nad distribuovaným grafem.

Základním prvkem je objekt reprezentující datovou sadu, která je uložena distribuovaně na jednotlivých uzlech clusteru. Nazývá se Resilient Distributed Datasets (RDD). RDD může být vytvořen z externího datového zdroje jako Hbase, Cassandra, HDFS. Na RDD můžeme aplikovat transformace (např. map, reduce, filter) [Foundation15].

3.1.2 Cassandra

Cassandra je open-source distribuovaná databáze pro velká data. Používá dotazovací jazyk CQL (Cassandra Query Language). Podporuje replikace a vyznačuje se dobrou škálovatelností. Podporuje propojení s Hadoop MapReduce. Tuto databázi jsem využil pro uložení silniční sítě [DataStax15].

3.1.3 Zkušební dataset

Pro vyzkoušení algoritmu byla použita silniční síť z Open Street Map (OSM), protože jsou to nejlépe dostupná data silniční sítě (obrázek 3.3). Dataset obsahuje kolem 16000 hran (Plzeň a okolí). Nejprve byly silnice importovány pomocí nástroje OSM2PO do PostGIS databáze. Poté byly k silnicím náhodně vygenerovány transportní funkce a uloženy do databáze Cassandra (postup vytvoření na obrázku 3.2).

Tento postup byl zvolen, protože reálné transportní funkce nebyly k dispozici v době psaní práce. Byl proveden pokus o jejich sestrojení z reálných dat z provozu, ale ani těchto dat nebyl dostatek pro zkonstruování funkcí. Další možností, jak tyto funkce získat, je teoretické modelování dopravy v dané oblasti. Této problematice se věnuje projekt OpenTransportNet (OTN), řešený na oddělení Geomatiky ZČU[1].

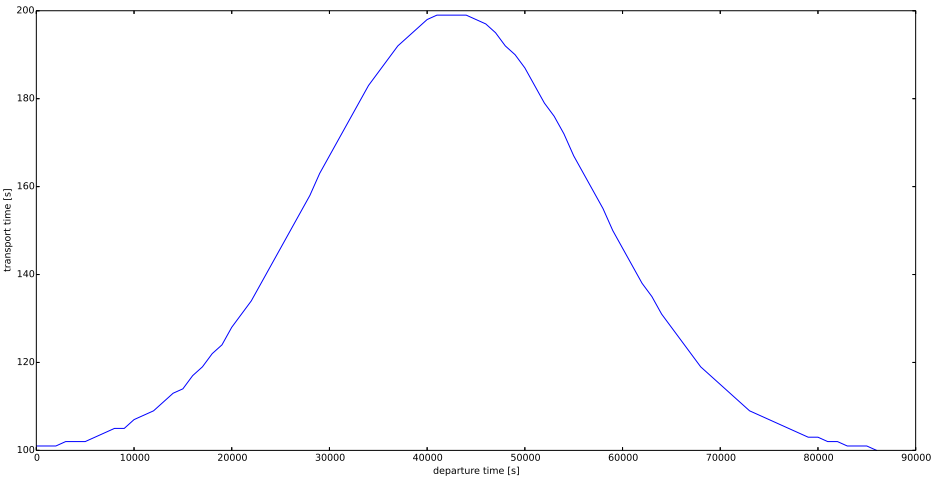
Transportní funkce byly tedy generovány náhodným posunem funkce e^{-t^2} po ose x a vynásobením délkou úseku a transformováním na správný interval (24 hodin). Tento postup byl zvolen s ohledem na reálný předpoklad vývoje dopravy (viz obrázek 3.1).

3.2 Implementace

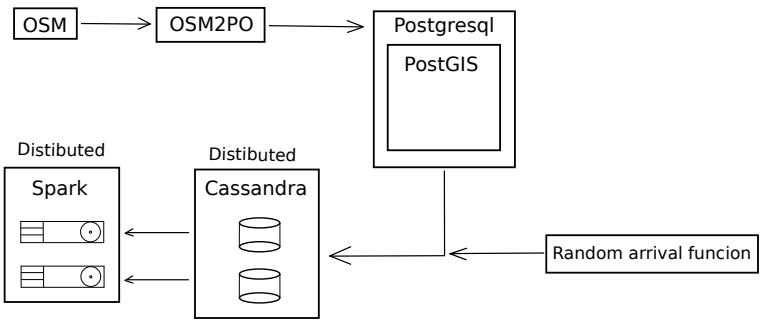
Nejprve je třeba se seznámit s prostředky frameworku GraphX.

Základním prvkem je objekt *Graph*. Parametry pro vytvoření objektu *Graph* jsou RDD hran a RDD vrcholů grafu. RDD hran tvoří objekty typu *Edge* (parametry

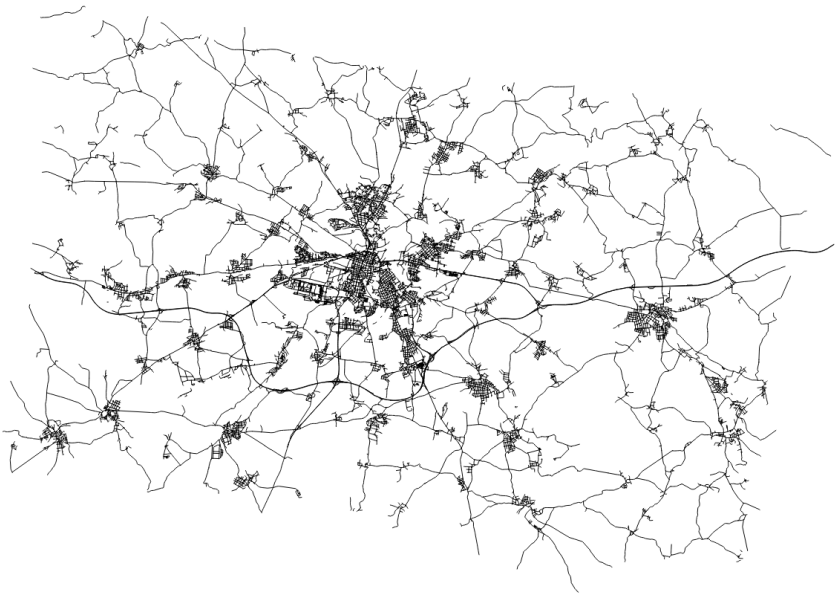
[1]Tento projekt je řešen ve spolupráci s firmu EDIP, která se zabývá mimo jiné modelováním dopravních intenzit



Obrázek 3.1: Transportní funkce simuluje denní špičku



Obrázek 3.2: Schéma vytvoření zkušebního datasetu

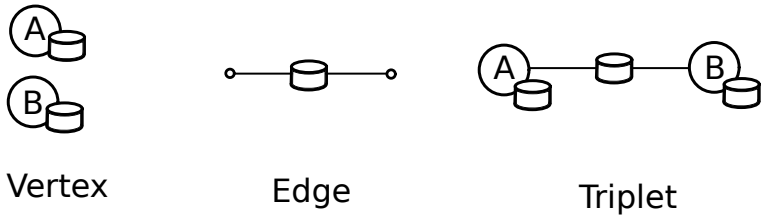


Obrázek 3.3: Náhled datasetu (S-JTSK)

jsou: počáteční vrchol, koncový vrchol a atributy). RDD vrcholů tvoří objekty typu *Tuple*. Na prvním místě je ID vrcholu (*VertexID*) a na druhém atributy. Více se dozvíte ve Spark dokumentaci [Foundation15].

V GraphX existují 3 pohledy na graf (obrázek 3.4):

- VertexRDD - RDD vrcholů grafu (Každý vrchol obsahuje ID a atributy.),
- EdgeRDD - RDD hran grafu (Každá hrana obsahuje ID počátečního vrcholu, ID koncového vrcholu a atributy.),
- triplets - RDD tripletů (Obsahuje kompletní informace o hraně a obou vrcholech včetně atributů.).



Obrázek 3.4: Tři pohledy na graf

Graph má několik pro nás důležitých základní metod odvíjejících se od pohledů na graf (viz výše). Transformační funkce:

- mapVertices,
- mapEdges,
- mapTriplets.

Join operace:

- outerJoinVertices - Spojí RDD vrcholů s grafem podle pravidla.

Pro nás nejdůležitější je metoda *mapReduceTriplets*. V novějších verzích programu Spark se tato funkce nahrazuje metodou *aggregateMessages*. Tato metoda v principu dělá to samé, co *mapReduceTriplets*. Parametry metody jsou:

- Mapovací funkce pro triplety - Výstup jsou takzvané *Messages* (zprávy) pro jednotlivé vrcholy.
- Funkce pro kombinování *Messages* (reduce).

Tato metoda vrací aktualizované RDD vrcholů. Atributy vrcholů mohou být tedy pozměněny na základě vlastností sousedních vrcholů. Tato funkce je stavební kámen všech algoritmů v GraphX.

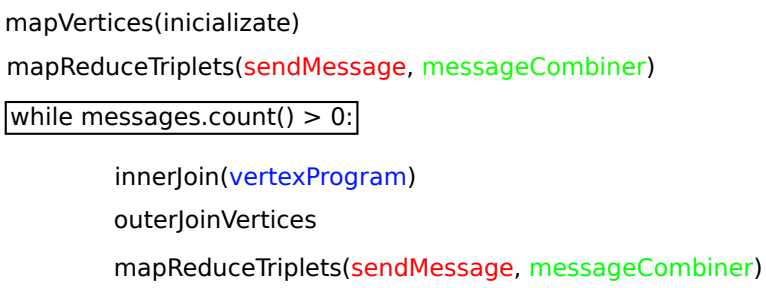
Například pro zjištění počtu sousedů každého vrcholu bude mapovací funkce vracet vždy 1 (Message (zpráva)) a reduce funkce bude tyto zprávy sčítat. V jazyce Scala:

```
mapReduceTriplets(triplet => 1, (a,b) => a + b)
```

Pro jednoduší psaní algoritmů je v GraphX takzvané Pregel API. Je to abstraktní třída pro grafové počítání. Přes tuto třídu jsou napsané všechny algoritmy již implementované v GraphX. Bohužel pro můj účel tato třída neumožňuje dynamické řízení počtu iterací, takže nebyl použita.

3.3 MapReduce algoritmus pro všechny odjezdové časy

Jak již bylo psáno výše, algoritmus vychází z LCA. Nejprve se graf inicializuje stejným způsobem jako u LCA. Poté se pro všechny hrany zjistí, zda je cesta do koncového vrcholu hrany právě přes tuto hranu výhodnější (map). Z těchto výsledků spočteme novou příjezdovou funkci v každém vrcholu (reduce). Toto opakuji tak dlouho, dokud nebude co aktualizovat (opravovat).



Obrázek 3.5: Grafické znázornění MapReduce algoritmu, červeně - mapovací funkce, zeleně - reduce funkce, modře - join funkce.

3.3.1 Inicializace

Při inicializace se do počátečního vrcholu zapíše funkce $f(x) = x$ a do ostatních nekonečno (implementováno jako prázdná funkce). Inicializační mapovací funkce:

1: **if** vertex == source vertex **then**
2: **return** Arival function $f(x) = x$
3: **else**
4: **return** ∞
5: **end if**

```
graph.mapVertices(inicializate)
```

3.3.2 Hlavní cyklus

V hlavním cyklu se nejprve provede metoda *mapReduceTriplets*, která vrátí nové příjezdové funkce (atributy) k vrcholům (*messages*). Spočítá se počet nových (aktualizovaných) příjezdových funkcí. Když nebude žádná nová příjezdová funkce, tak se algoritmus ukončí. Pokud počet nových (aktualizovaných) příjezdových funkcí (*messages*) bude nenulový, tak se spojí nové funkce se starými (*innerJoin*) a nakonec se nové vrcholy spojí s grafem (*outerJoinVertices*). Takto se pokračuje dokud nebude nulový počet aktualizovaných (nových) příjezdových funkcí. Složitá ukončovaní podmínka zformulovaná výše je tady hlídána právě počtem počtem aktualizací (*messages*). Na obrázku 3.5 vidíte graficky znázorněný algoritmus.

vertexProgram

VertexProgram je funkce, která je parametrem metody (*vertices.innerJoin*). Tato funkce zajišťuje spojení starých a nových atributů vrcholů. Parametry jsou ID vr-

cholu, starý a nový atribut. V našem případě jde o příjezdové funkce. Funkce vrací nový atribut (příjezdovou funkci) vrcholu. Výsledná příjezdová funkce musí být minimum ze staré a nové funkce. Tedy:

```
1: return minimum(old, new)
```

sendMessage

SendMessage je mapovací funkce, která vrací příjezdové funkce pro určitý vrchol za použití příslušné hrany. Parametr je tedy pouze triplet,

```
1: cumul = a_ij(EA_si)
2: if piece of cumul is < EA_sj then
3:     return cumul
4: else
5:     return nothing
6: end if
```

kde i je počáteční vrchol a j je koncový vrchol. Nyní existuje pro každý vrchol několik příjezdových funkcí (EA_{si}). Minimálně 0 a maximálně jako počet sousedů.

messageCombiner

MessageCombiner je reduce funkce zajišťující zkombinování několika příjezdových funkcí k jednomu vrcholu. Musí to být minimum ze všech těchto funkcí. Tedy:

```
1: return minimum(a, b)
```

Nyní máme ke každému vrcholu jednu nebo žádnou příjezdovou funkci. Je třeba pro ukončovací podmínku spočítat jejich počet. Není-li již žádná příjezdová funkce, je podmínka splněna. Tedy pro každou hranu platí podmínka:

$$a_{ij}(EA_{si}) \geq EA_{sj} \tag{3.1}$$

Kapitola 4

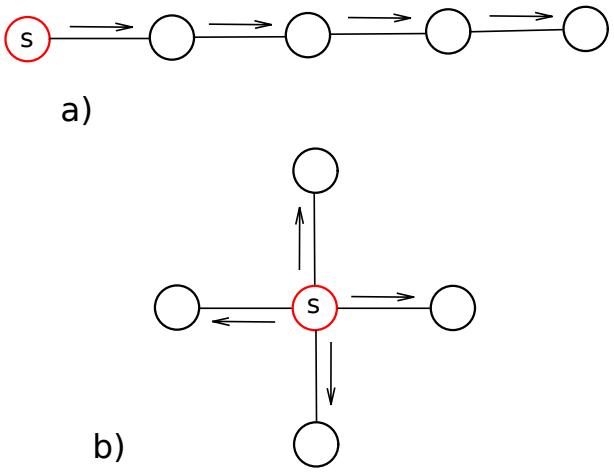
Složitost a výkon

Z předchozí kapitoly víme, že nejrychlejší trasy nalezneme, když bude splněna podmínka ukončení algoritmu. Otázka zní, kolik potřebujeme iterací k tomu, aby tato podmínka platila. Vycházíme z předpokladu, že po jedné iteraci nalezneme pouze nejkratší cesty o délce 1 hrany. Tedy po n iteracích nalezneme nejkratší cesty o n hranách. Problém je v tom, že nevíme, kolik hran obsahuje nejkratší cesta do daného vrcholu [DEAN99].

V nejhorším případě, kdy graf zdegeneruje v řetězec hran (přímku), potřebujeme k nalezení nejkratší cesty tolik iterací, jako je hran (n - počet hran). V jedné iteraci provádíme funkci *sendMessage* pro každou hranu, tedy n-krát. Výsledná složitost pro nejhorší případ je $O(n^2)$ [DEAN99]. Výhodou je, že funkce *sendMessage* je provádí paralelně. V reálné silniční síti samozřejmě je potřeba mnohem méně iterací než n^2.

Na obrázku 4.1 je vidět nejhorší a nejlepší případ. V nejhorším případě (a) algoritmus najde nejkratší cestu do všech vrcholů po 5 iteracích. V nejlepším ji najde už po jedné iteraci (b).

Rychlost algoritmu dále závisí na časové složitosti dílčích funkcí jako minimum, vnoření a porovnání. V implementaci "po částech"lineární funkce tato časová složitost závisí na počtu lomových bodů lomené čáry. Čím více lomových bodů, tím je tato funkce náročnější. Počet těchto bodů je tedy kompromis mezi rychlostí a přesností.

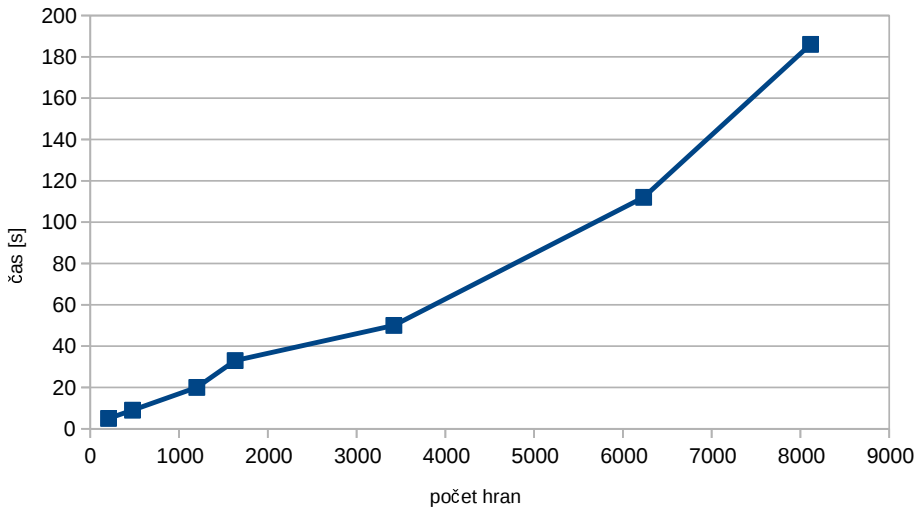


Obrázek 4.1: Nejhorší a nejlepší případ

4.1 Testy

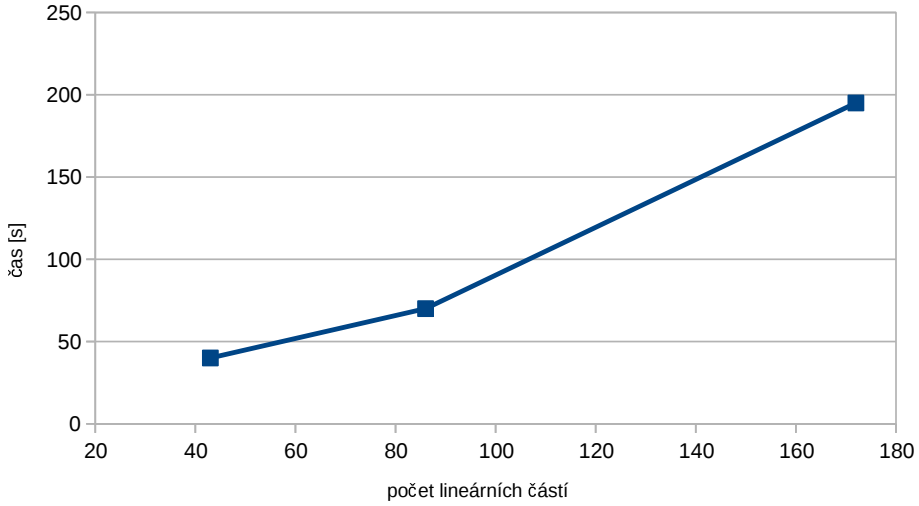
4.1.1 Závislost výpočetního času na velikosti sítě

Testy byly prováděny na notebooku s 8 Gb RAM a dvou-jádrovým procesorem Intel(R) Core(TM) i5-4300M CPU @ 2.60GHz. Částečně byly testy prováděny i na clusteru, ale bez větších úspěchů (viz dále). Jeden z nejdůležitějších testů je závislost výpočetního času na velikosti grafu (4.2). Pro tento test byly vytvořeny různé datasety o různých velikostech od 206 do 8116 hran (reálná silniční síť - části Plzně). Interval mezi lomovými body příjezdové funkce je 1000 s, tedy 86 lomových bodů.



Obrázek 4.2: Závislost výpočetního času na velikosti grafu

Další důležitý test je závislost výpočetního výkonu na počtu lomových bodů v příjezdové funkci (obrázek 4.3). Tento test byl proveden s grafem o velikosti 16158 hran postupně pro intervaly mezi body příjezdové funkce 500 s, 1000 s a 2000 s.



Obrázek 4.3: Závislost výpočetního času na počtu lomových bodů příjezdové funkce

4.1.2 Závislost výpočetního času na počtu vláken (procesorů)

V době psaní práce virtuální organizace Metacentrum zřídila výpočetní cluster založený na softwaru Hadoop. Po mé iniciativě byl na cluster doinstalován i Apache Spark (Hadoop YARN - Spark používá výpočetní uzly Hadoopu). Byl tedy proveden pokus o stanovení závislosti výpočetního času na počtu vláken, ale vzhledem k tomu, že cluster byl uveden do provozu až v závěru psaní práce, tak se testy nepovedlo provést.

Závěr

V práci byl vyvinut a implementován MapReduce algoritmus pro hledání nejrychlejší cesty v silniční síti závislé na čase, konkrétně problém volby optimálního času výjezdu. Tento algoritmus je postavený na statickém LCA. Jeho složitost je proto n^2/k, kde n je počet hran a k počet procesorů ($k < n$).

Implementace v distribuovaném výpočetním prostředí Apache Spark je vhodná pro velmi velké grafy (které se nevejdou do paměti jednoho počítače), protože režie, která je potřeba pro rozdistribuování dat a komunikaci mezi jednotlivými výpočetními uzly u malých grafů, tvoří velké procento výpočetního času, a malé grafy se pohodlně vejdou do paměti jednoho počítače. U velkých grafů je toto procento mnohem menší.

Dalším problémem je dostupnost transportních funkcí pro tuto úlohu. Proběhl pokus tyto funkce odvodit z reálných dat z provozu, ale ani těchto dat nebyl dostatek. Další možností jak tyto funkce získat je teoretické modelování dopravních intenzit. Tímto problémem se zabývá projekt Open Transport Net (OTN), který právě probíhá na Západočeské univerzitě.

Dále jeden ne zcela dořešený problém je operace vnoření dvou příjezdových funkcí a vůbec jak implementovat tyto funkce. To je možnost, kam dále směřovat s výzkumem v této oblasti.

Literatura

[AYED11] AYED, H.; HABBAS, Z.; KHADRAOUI, D.: A parallel time-dependent multimodal shortest path algorithm based on geographical partitioning. In *Nature and Biologically Inspired Computing (NaBIC), 2011 Third World Congress on*, Oct 2011, s. 213–218, doi:10.1109/NaBIC.2011.6089461.

[CHABINI98] CHABINI, I.: Discrete Dynamic Shortest Path Problems In Transportation Applications: Complexity And Algorithms With Optimal Run Time. *Transportation Research Records*, ročník 1645, 1998: s. 170–175.

[CHABINI02] CHABINI, I.; GANUGAPATI, S.: Parallel algorithms for dynamic shortest path problems. *International Transactions in Operational Research*, 2002: s. 279–302.

[DataStax15] DataStax, I.: Apache Cassandra Documentation 2.0. 2015, [Online].
URL `<http://docs.datastax.com/en/cassandra/2.0/cassandra/gettingStartedCassandraIntro.html>`

[DEAN99] DEAN, C., Brian: Continuous-Time Dynamic Shortest Path Algorithms. 1999.
URL `<http://people.cs.clemson.edu/~bcdean/bdean\_masters\_thesis.pdf>`

[DEAN04] DEAN, J.; GHEMAWAT, S.: MapReduce: simplified data processing on large clusters. *Communications of the ACM*, ročník 51, č. 1, 2004: s. 107–113.

[DING08] DING, B.; YU, J. X.; QIN, L.: Finding Time-dependent Shortest Paths over Large Graphs. In *Proceedings of the 11th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '08, New York, NY, USA: ACM, 2008, ISBN 978-1-59593-926-5, s. 205–216, doi:10.1145/1353343.1353371.
URL <http://doi.acm.org/10.1145/1353343.1353371>

[Foundation15] Foundation, T. A. S.: Apache Spark Documentation 1.2.1. 2015, [Online].
URL <https://spark.apache.org/docs/1.2.1/index.html>

[FRIESZ93] FRIESZ, T. L.; ANANDALINGAM, G.; MEHTA, N. J.; aj.: The multiobjective equilibrium network design problem revisited: A simulated annealing approach. *European Journal of Operational Research*, ročník 65, č. 1, 1993: s. 44 – 57, ISSN 0377-2217, doi:http://dx.doi.org/10.1016/0377-2217(93)90143-B.
URL <http://www.sciencedirect.com/science/article/pii/037722179390143B>

[GANUGPATI98] GANUGPATI, S. V.: Dynamic shortest paths algorithms : parallel implementations and application to the solution of dynamic traffic assignment models. *Massachusetts Institute of Technology*, 1998, thesis (M.S.)–Massachusetts Institute of Technology, Dept. of Civil and Environmental Engineering, 1998.Includes bibliographical references (leaves 183-186).
URL <http://dspace.mit.edu/handle/1721.1/46478>

[HAGHANI05] HAGHANI, A.; JUNG, S.: A dynamic vehicle routing problem with time-dependent travel times. *Computers & Operations Research*, ročník 32, č. 11, 2005: s. 2959 – 2986, ISSN 0305-0548, doi:http://dx.doi.org/10.1016/j.cor.2004.04.013.
URL <http://www.sciencedirect.com/science/article/pii/S0305054804000887>

[LAWSON13] LAWSON, G.; ALLEN, S.; ROSE, G.; aj.: Parallel Label-Correcting Algorithms for Large-Scale Static and Dynamic

Transportation Networks on Laptop Personal Computers. *Transportation Research Board 92nd Annual Meeting*, 2013.
URL <http://docs.trb.org/prp/13-2103.pdf>

[OHSHIMA06] OHSHIMA, T.: A Landmark Algorithm for the Time-Dependent Shortest Path Problem. 2006.
URL <http://www-or.amp.i.kyoto-u.ac.jp/members/ohshima/Paper/MThesis/MThesis.pdf>

[SPERB10] SPERB, R. C.: Solving time-dependent shortest path problems in a database context. 2010.
URL <http://www.itc.nl/library/papers_2010/msc/gfm/sperb.pdf>

[ZHAO08] ZHAO, L.; OHSHIMA, T.; NAGAMOCHI, H.: A* Algorithm for the time-dependent shortest path problem. 2008.
URL <http://www-or.amp.i.kyoto-u.ac.jp/~liang/research/waac08.pdf>

Příloha A

Obsah přiloženého CD

- BP_Kolovsky.pdf - vlastní práce
- source - složka s projektem
  - src/main/scala - složka se zdrojovými kódy
    - ArivalFunction.scala - reprezentace příjezdové funkce
    - FunctionForAF.scala - operace pro příjezdové funkce
    - MyPregel.scala - modifikovaný objekt Pregel pro moje potřeby
    - DPFile.scala - třída pro spuštění na Hadoop YARN clusteru s načítáním ze souboru
    - DinamicPathIter.scala - třída pro spuštění na localhostu a načítání z Apache Cassandra
  - target/scala-2.10/spark_routing_2.10-1.0.jar - sestavený JAR soubor pomocí SBT
  - road_1000 - soubor se sítí pro načtení ze souboru v clusteru
  - readme.txt - instrukce ke spuštění na clusteru
  - simple2.sbt - soubor pro SBT pro sestavení JAR (Spark 1.2.0)