Hledání nejkratších cest v grafu

Tomáš Duda a Artemij Pozdňakov
dudatom2@fit.cvut.cz a pozdnart@fit.cvut.cz

18. listopadu 2014

Obsah

1 Řešený problém a základní implementace

1.1 Definice problému

Naším úkolem v rámci semestrálního projektu v předmětu BI-EIA je implementace a optimalizace dvou algoritmů pro hledání nejkratších cest. Vstupem programu je tedy graf zadaný výčtem hran grafu a výstupem délka nejkratší cesty pro každou dvojici uzlů.

Pro řešení problému jsme implementovali dva algoritmy, první je Dijkstrův, druhý Floyd-Warshallův. Jejich stručný popis a informace o základní implementaci jsou obsahem následujících částí.

1.2 Dijkstrův algoritmus

Dijkstrův algoritmus funguje obdobně jako prohledávání do šířky, jenom místo obyčejné fronty používá prioritní frontu. Do té jsou před během algoritmu přesunuty všechny uzlu, počáteční z nulovou, zbylé s ∞ vzdáleností. Poté se až do vyprázdnění fronty vybírá nejbližší uzel a pro všechny jeho sousedy se vyzkouší, zda byla nalezena zkracující cesta (relaxace).

Běžná verze Dijkstrova algoritmu je určena pro hledání nejkratších cest z jednoho uzlu do všech ostatních, proto ho musíme v našem řešení volat $|V|$-krát, tedy z každého uzlu. Asymptotická složitost Dijkstrova algoritmu závisí na dvou faktorech. Jednak jde o vnitřní reprezentaci grafu (seznam uzlů a jejich sousedů nebo matice sousednosti) a dále o způsob implementace prioritní fronty, kterou algoritmus využívá.

V rámci snahy o kompromis mezi rozumnou rychlostí a dostatečnou možností kód následně optimalizovat jsme použili kombinaci reprezentace grafu seznamem sousedů a prioritní fronty řešené pomocí binární haldy. Tato kombinace má při hledání nejkratších cest mezi všemi dvojicemi uzlů asymptotickou složitost $\mathcal{O}(|V|(|E| + |V|) \log |V|)$, kde V je množina uzlů a E je množina hran.

1.3 Floyd-Warshallův algoritmus

Floyd-Warshallův algoritmus hledá nejkratší cesty metodou postupné konstrukce. Skládá se ze tří do sebe vnořených cyklů, které iterují přes všechny uzly grafu. Vnější cyklus určuje prostředníka, přes které se algoritmus právě snaží nalézt zlepšující cestu, dva vnitřní cykly poté určují dvojici koncových uzlů. Jako vnitřní reprezentace grafu je použita matice sousednosti, která je postupem algoritmu přepsaná na výslednou distanční matici. Asymptotická složitost Floyd-Warshallova algoritmu je $\mathcal{O}(|V|^3)$

1.4 Porovnání algoritmů

Pro řídké grafy ($|E| \sim |V|$) má Dijkstrův algoritmus (s použitím binární haldy) nižší teoretickou složitost než Floyd-Warshallův. Dá se však očekávat, že při reálném použití u grafů, které jsme schopni v rozumném čase na poskytnutém HW upočítat (řádově tisíce uzlů), bude Floyd-Warshallův algoritmus rychlejší, jelikož potřebuje vykonat v nejvnitřnějším cyklu mnohem méně operací než Dijkstrův.

1.5 Popis souborů

- **main.cpp** obsahuje zpracování argumentů z příkazové řádky a volání algoritmů.
- **mygraph.cpp** obsahuje třídu MyGraph, která jednak slouží pro vnitřní reprezentaci grafu (podporuje jak matici sousednosti tak i reprezentaci pomocí seznamů uzlů a jejich sousedů). Dále realizuje zpracování vstupního grafu ze souboru.
- **node.cpp** a **edge.cpp** obsahují pomocné třídy pro uložení uzlu nebo hrany v grafu.
- **floydwarshall.cpp** implementuje Floyd-Warshallův algoritmus.
- **dijkstra.cpp** implementuje Dijkstrův algoritmus.
- **binary_heap.cpp** poskytuje prioritní frontu přes binární haldu.
- **exception.cpp** definuje výjimku, která je použita při spuštění programu s chybnými argumenty.

2 Optimalizovaná verze sekvenčního algoritmu

Následující kapitola popisuje optimalizace provedené v sekvenčních verzích implementace a následné výsledky meření různých verzí.

2.1 Dijkstrův algoritmus

Jelikož nebylo Dijkstrův algoritmus možné optimalizovat klasickými metodami (vysoká datová provázanost, nemožnost rozbalit vnitřní cyklus kvůli komplikované datové struktuře), pokusili jsme se řešení optimalizovat více jinými způsoby.

2.1.1 Úprava algoritmu

Prvním byla výměna původně použité prioritní fronty z STL za vlastní implementaci, která navíc podporuje operaci decreaseKey a tudíž není potřeba u každého uzlu vyňatého z fronty testovat, zda je jeho hodnota klíče aktuální (zkrátka není nutné používat reinserting).

2.1.2 Přepínače gcc

Druhý pokus o optimalizaci proběhnul pomocí použití různých přepínačů při kompilaci v gcc.

- **-O3** - zapnutí plných optimalizací cílového kódu.
- **-mffast-math** - zrychlené vyhodnocení matematických výrazů.
- **-msse4.2** - zapnutí vektorových instrukcí.
- **-mfpmath=sse** - použití SSE instrukcí místo klasických FP.
- **-msseregparm** - předávání FP dat přes SSE registry.
- **-mpc32** - zaokrouhlení FP výpočtů.
- **-ffast-math** - urychlené provádění výpočtů.

Naopak testované, ale nakonec nepoužité zůstaly (neměly pozitivní vliv na rychlost):

- **-march=opteron** i **-march=barcelona** - využití všech instrukcí na cílovém procesoru.
- **-regparm1..3** - jejich použití vyvalávalo na Staru segfault.

2.1.3 Profiling

V rámci profilingu jsme měřili čas strávený v jednotlivých řádkách kódu. Jako velmi neefektivní se ukázaly být vectory a iterátory z STL, které jsme nahradili obyčejným polem. Dále jsme zkusili odbourat zapouzdření v datových strukturách reprezentující graf, pomocí výčtu sousedů. Poslední úpravou, kterou se povedlo zlepšit čas běhu, bylo důsledné předpočítání všech položek (vzdáleností a indexů) před operací relaxace (v nejvnitřnějším cyklu Dijkstrova algoritmu).

Po všech úpravách profiler ukazuje, že přes 80 % veškerého času tráví program přístupem do pole, ve kterém jsou uloženy aktuální vzdálenosti. To už se nám nepodařilo vylepšit. Přístup do tohoto pole není sekvenční (získáváme z něj vzdálenosti sousedů uzlu, který je aktuálně vytažen z prioritní fronty, tyto vzdálenosti mohou být na libovolném místě), a proto nemůžeme data z toho pole nějak "předpřipravit" před samotným výpočtem.

2.2 Floyd-Warshallův algoritmus

Loop-tiling, loop unrolling, vektorizace? TODO.

2.3 Popis testovacích instancí

Výběr testovacích instancí pro testování a porovnání obou implementovaných algoritmů byl poměrně komplikovaný. Dijkstrův algoritmus je narozdíl od Floyd-Warhsallova citlivý na vstupní data, tudíž bylo nutné, aby se jednotlivé testovací grafy lišily nejenom v počtu uzlů, ale i v počtu hran.

Floyd-Warshallův algoritmus se navíc ukázal být o hodně výkonnějším, tudíž zatímco neoptimalizovaná implementace Dijkstrova algoritmu na největší instanci za 60 minut na testovacím serveru nedoběhla, Floyd-Warshall ji upočítal za 10 minut.

Vybrali jsme tedy grafy o čtyřech různých velikostech, co do počtu uzlů (800, 1600, 2400 s 3200) a u každého z nich tři různé instance lišící se poštem hran. První z trojice je vždy řídký graf (obsahuje přibližně desetinu hran, co graf úplný), druhý obsahuje přibližně polovinu hran úplného grafu a třetí je hustý, obsahuje 90 % hran úplného grafu.

Název souboru	Počet uzlů	Počet hran
graf800_80	800	31521
graf800_400	800	160035
graf800_720	800	287590
graf1600_160	1600	128022
graf1600_800	1600	639663
graf1600_1400	1600	1151713
graf2400_240	2400	289420
graf2400_1200	2400	1439793
graf2400_2160	2400	2591136
graf3200_320	3200	513016
graf3200_1600	3200	2560196
graf3200_2880	3200	4608704

Tabulka 1: Vlastnosti grafů, na kterých byly testovány sekvenční implementace algoritmů.

2.4 Měření a porovnání výkonosti různých sekvenčních verzí

V následující části jsou uvedeny výsledky meření jednotlivých typů implementací. Údaje jsou uváděny v MFPLOPS. Zatímco i Floyd-Warshallova algoritmu můžeme počítat s $2|V|^3$ operací v plovoucí čárce na jeden běh, u Dijkstrova algoritmu je situace o trochu komplikovanější, protože počet operací v FP závisí na podobě vstupního grafu a odhady pomocí složitosti nebyly příliš přesné. Změřili jsme tedy počet FP operací pro jednotlivé testovací instance.

Název souboru	FP operací
graf800_80	115,283 M
graf800_400	524,098 M
graf800_720	932,637 M
graf1600_160	879,441 M
graf1600_800	4,139 G
graf1600_1400	7,414 G
graf2400_240	2,912 G
graf2400_1200	13,929 G
graf2400_2160	24,956 G
graf3200_320	6,800 G
graf3200_1600	32,963 G
graf3200_2880	59,136 G

Tabulka 2: Počet FP operací potřebných k běhu Dijkstrova algoritmu pro jednotlivé instance.

2.4.1 Měření Dijkstrova algoritmu

V následující tabulce jsou naměřené hodnoty pro Dijkstrův algoritmus. Měřena byla jednak verze, která využívá prioritní frontu z STL, poté verze používající prioritní frontu s podporou operace decreaseKey, třetí verze byla kompilovaná s přepínačem -O3, ve čtvrté verzi byly použity další přepínače vypsané v sekci Optimalizovaná verze sekvenčních algoritmů. Nakonec jsou uvedeny hodnoty měření pro kód po profilingu.

Z měření jde vypozorovat několik souvislostí:

- Největší vliv na výkon mělo profilování kódu, kde byly odstraněny některé nadbytečné přístupy do paměti. Nicméně program i tak tráví prací s pamětí mnohonásobně více času, než samotnými výpočty. To je ale dáno potřebnou složitější datovou strukturou.
- Změna prioritní fronty z STL implementace na vlastní implementaci program mírně zrychlila.
- Při použití více přepínačů (než -O3) program pro většinu instancí zrychlí, ale výkony jsou mnohem méně stabilní.

Instance	STL	BH	-O3	k. opt	prof
graf800_80	11,9	20,8	50,8	50,1	96,0
graf800_400	15,5	16,7	25,5	27,9	55,0
graf800_720	13,0	15,9	28,9	32,7	52,0
graf1600_160	12,3	16,9	28,4	27,5	47,5
graf1600_800	14,4	15,6	26,6	30,0	50,9
graf1600_1400	14,8	16,4	25,4	29,7	52,1
graf2400_240	11,8	15,3	25,1	23,2	35,8
graf2400_1200	14,2	15,4	24,8	24,9	51,6
graf2400_2160	14,8	12,6	25,1	26,9	49,5
graf3200_320	12,3	12,7	23,7	24,9	39,0
graf3200_1600	14,3	14,9	25,0	28,0	52,7
graf3200_2880	x	x	25,1	27,1	50,0

Tabulka 3: Naměřené výsledky pro Dijkstrův algoritmus. Hodnoty jsou v MFLOPS. Políčka, ve kterých je uvedeno x značí, že pro ně algoritmus nedokázal doběhnout ve stanoveném limitu 60 minut.

2.4.2 Měření Floyd-Warshallova algoritmu

Ve prvním sloupci NoOpt jsou výsledky meření výchozího programu bez použití žádných optimalizací. V dalších je kompilace se základní optimalizací kompilátoru -O3. Další měřená verze je po loop tilingu (kompilovaná s -O3 a -ffast-math).

Instance	NoOpt	-O3	Tiling	???
graf800_80	129,0	975,2	1177,0	
graf800_400	130,4	890,4	1248,7	
graf800_720	129,3	1402,7	1190,7	
graf1600_160	125,2	686,7	1233,7	
graf1600_800	125,3	688,4	1180,4	
graf1600_1400	125,2	680,9	1177,0	
graf2400_240	126,0	670,7	1201,0	
graf2400_1200	126,1	671,0	1209,9	
graf2400_2160	126,4	670,0	1326,0	
graf3200_320	125,8	665,6	1276,2	
graf3200_1600	126,0	663,6	1303,2	
graf3200_2880	125,9	662,9	1347,9	

Tabulka 4: Naměřené výsledky pro Floyd-Warshallův algoritmus. Hodnoty jsou v MFLOPS.

3 Vícevláknová implementace

4 Závěr

Reference

[1] KOLÁŘ, Josef. *Teoretická informatika.* Česká informatická společnost, Praha, 2004. 205s.