GAL projekt – Algoritmus s frontou podezřelých

Dvořáková Lucie (xdvora1f)

16. února 2017

1 Popis algoritmu

Algoritmus s frontou podezřelých slouží k řešení problému nejkratší cesty z jednoho do všech vrcholů. Vstupem problému je orientovný ohodnocený graf $G = (V, E)$ s ohodnocovací funkcí $w : E \to \mathbb{R}$. Graf neobsahuje negativní cykly, avšak může obsahovat negativní hrany. Algoritmus hledá cesty, které mají od počátečního vrcholu u k jednotlivým vrcholům v cestu s nejmenší váhou w,

$$w(\langle v_0, v_1, ..., v_k \rangle) = \sum_{i=1}^{k} w(v_{i-1}, v_1)$$

kde v_i je váha jednotlivých cest. Nejkratší cesta mezi počátečním uzlem u a jednotlivými vrcholy v se definuje jako $\delta(u, v)$:

$$\delta(u, v) = \begin{cases} min\{w(p) : u \rightsquigarrow^p v\} & pokud\ existuje\ cesta\ z\ u\ do\ v \\ \infty & jinak \end{cases}$$

Výstupem algoritmu je pak strom tvořený nejkratšími cestami, kde pro libovolnou cestu p z u do v platí $w(p) = \delta(u, v)$.

2 Analýza algoritmu

Algoritmus prochází vrcholy a relaxuje hrany obdobně jako Bellman-Ford a Dijkstrův algoritmus. Hlavním rozdílem je výběr, v jakém pořadí bude algoritmus uzly procházet a které uzly při kontrole úplně vynechá. Narozdíl od Dijkstrova algoritmu je schopen pracovat se zápornými hranami.

Algoritmus má obdobně jako Bellman-Ford časovou složitost $O(|V| \cdot |E|)$ avšak ve většině případů se jedná o algoritmus rychlejší. Důvodem toho je fronta podezřelých, do které se vkládají pouze podezřelé vrcholy na rozdíl od Bellman-Ford algoritmu, který kontroluje opakovaně všechny uzly. Jestliže při relaxaci hrany nedojde ke změně ceny cesty do daného uzlu, můžeme předpokládat, že nedojde ani ke změně ceny cesty jdoucí do vrcholu následujícího. Následující vrchol se tedy nemusí zařazovat do fronty podezřelých, a tím se vynechají zbytečné kontroly. Naopak, dojde-li při relaxaci ke změně ceny cesty do daného uzlu, všechny sousedící uzly budou do fronty zařazeny za předpokladu, že se ve frontě již nevyskytují.

Fronta je implementovaná pomocí množiny, ze které se vybíra prvek nacházející se v množině nejdelší dobu, tedy se jedná o FIFO frontu. Narozdíl od prioritní fronty využité v Dijkstrově algoritmu má vyjmutí prvku z FIFO fronty konstantní časovou složitost $O(1)$ oproti vyjmutí

prvku z prioritní fronty, kde je logaritmická časová složitost $O(log \cdot |V|)$, při využití haldy. U řídkých grafů, kde počet hran nepřesahuje nějaký nevelký násobek počtu vrcholů, bývá často algoritmus s frontou podezřelých rychlejší.

function INITIALIZE-SINGLE-SOURCE(G, s)
  for každý uzel $v \in (V - s)$ **do**
    $d[v] \leftarrow \infty$
    $\pi[v] \leftarrow NILL$
  $d[s] \leftarrow 0$
  $Q \leftarrow \varnothing$
  ENQUEUE(Q, s)

function RELAX(u, v, w)
  if $d[v] > d[u] + w(u, v)$ **then**
    $d[v] \leftarrow d[u] + w(u, v)$
    $\pi[v] \leftarrow u$
    if $v \notin Q$ **then**
      ENQUEUE(Q, v)

function ALGORITMUS S FRONTOU PODEZŘELÝCH($\mathbf{G}, \mathbf{w}, \mathbf{s}$)
  INITIALIZE-SINGLE-SOURCE(G, s)
  while $Q \neq \varnothing$ **do**
    $u \leftarrow$ DEQUEUE(Q)
    for každý uzel $v \in Adj[u]$ **do**
      RELAX(u, v, w)

3 Paralelní verze algoritmu

Přímočarým způsobem jak lze paralelizovat algoritmus s frontou podezřelých je rozdělit vrcholy čekající ve frontě mezi jednotlivá vlákna. V následujicím kódu je patrné, kde byl zařazen paralelní `for` cyklus pomocí `#pragma omp for` a zajištění výlučného přístupu do fronty Q pomocí `#pragma omp critical`.

function RELAX(u, v, w)
  if $d[v] > d[u] + w(u, v)$ **then**
    $d[v] \leftarrow d[u] + w(u, v)$
    $\pi[v] \leftarrow u$
    if $v \notin Q$ **then**
      #pragma omp critical
      ENQUEUE(Q, v)

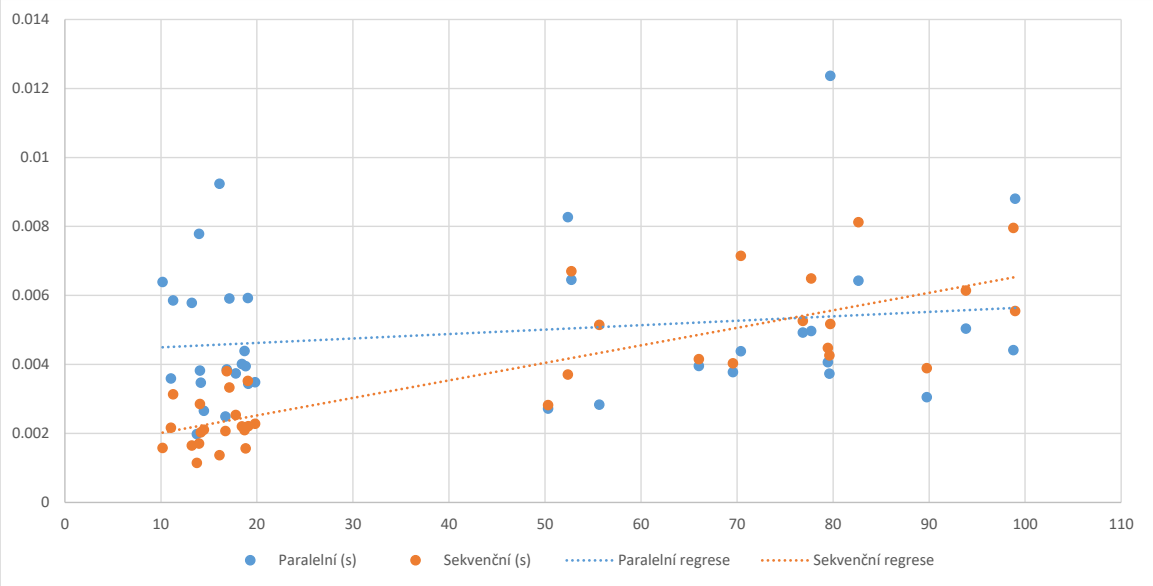
function ALGORITMUS S FRONTOU PODEZŘELÝCH($\mathbf{G}, \mathbf{w}, \mathbf{s}$)
  INITIALIZE-SINGLE-SOURCE(G, s)
  while $Q \neq \varnothing$ **do**
    #pragma omp for
    for každý uzel $u \in Q$ **do**
      #pragma omp critical
      $u \leftarrow$ DEQUEUE(Q)
      for každý uzel $v \in Adj[u]$ **do**
        RELAX(u, v, w)

Získáváme složitost $O\left(\frac{|V| \cdot |E|}{p}\right)$, kde p je počet vláken. Čím více bude vrcholů ve frontě, tím více by měl být výpočet paralelní a algoritmus tak bude efektivnější oproti své sekvenční verzi. Můžeme tedy předpokládat, že výhoda paralelní implementace se zvláště projeví u hustých grafů. Proto se na tento aspekt zaměříme při experimentech.

4 Experimenty

V experimentech je především pozorován vztah rychlosti algoritmu vzhledem hustotě, takže počet uzlů v grafu je generován náhodně v rozmezí 1250 až 2500, nezávisle na počtu hran. Když tedy budeme počet vrcholů považovat jako konstantu, měla by při nárůstu počtu hran grafu lineárně stoupat časová složitost. Prostorová složitost algoritmu je $O(|V|)$, nejen z důvodu využití fronty pro ukládání podezřelých vrcholů, ale také kvůli pomocnému poli, díky kterému můžeme v konstantním čase zjisťovat, zda se vrchol ve frontě nachází.

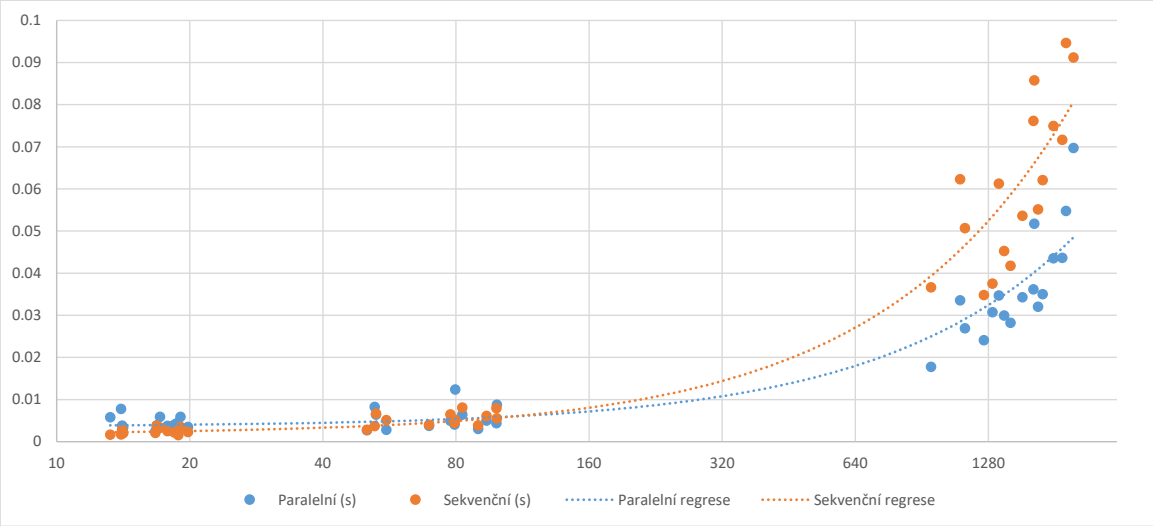
Algoritmus s frontou podezřelých je zejména rychlý u řídkých grafů, kde se nikdy ve frontě nenachází velký počet uzlů. Při srovnávání rychlosti sekvenčního a paralelního algoritmu nad velmi řídkými grafy (průměrně 15 hran pro každý vrchol), je sekvenční algoritmus rychlejší. Až při zvýšení průměrného počtu k 80 hranám na jeden uzel, začíná být paralelní verze výpočtu časově efektivnější (viz obrázek č. 1).



Obrázek 1: Porovnání sekvenčního a pararelního výpočtu pro řídké grafy

Ve shodě s teoretickou verzí se jedná o linearní časovou složitost.

Při zvyšování koncentrace hran až k grafu úplnému se začne výrazněji projevovat paralelizace algoritmu viz obrázek č. 2. Pro experimenty zabívající se hustotou grafů bylo celkově vygenerováno 50 grafů. Z důvodu selhání techniky byl algoritmus spouštěn pouze na čtyřjádrovém procesoru, i přesto je zvýšení rychlosti z grafu patrné.



Obrázek 2: Porovnání sekvenčního a pararelního výpočtu pro celou škálu grafů

5 Spuštění algoritmu

Projekt je naprogramovaný v jazyce C++ a využívá `OpenMP` (přeložitelné na serveru `merlin`). Obsahuje také `Makefile`, který generuje výstupní spustitelný soubor `main`. Při jeho spustění bez parametrů vypíše nápovědu:

```
Usage: [ -p | -s | -g ] [ <input.txt> | 1 | 2 | 3 | 4 | 5 | 6 ]
-p <input.txt> - calculates algorithm parallely with <input.txt>
-s <input.txt> - calculates algorithm sequentially with <input.txt>
-g 1 - generates graph nodes < 100, output input.txt
-g 2 - generates graph nodes < 500, output input.txt
-g 3 - generates graph nodes < 2500, output input.txt
-g 4 - generates graph paths < 20*nodes, output input.txt
-g 5 - generates graph paths < 100*nodes output input.txt
-g 6 - generates graph paths < nodes*nodes, output input.txt
```

Parametr `-p` a `-s` spouští algoritmus paralelně nebo sekvenčně na graf uložený v `input.txt` a vypisuje čas běhu samotného algoritmu (tj. bez času nutného pro měření nepodstatného načítání vstupního souboru). Parametr `-g` slouží k vytvoření jedné ze šesti variant náhodného grafu. Varianta `1` až `3` generuje řídké grafy s negativními hranami s postupně se navyšujícím počtem uzlů. Varianta `4` až `6` generuje grafy s postupně zvyšující koncentrací hran. Všechny tyto varianty generují svůj výstup do `input.txt`. Testovací sada vygenerovaná pro účely experimentů je v komprimované podobě dostupná na `https://bitbucket.org/lucidvoci/gal2016/src`.

Příklad postupu pro vygenerování řídkého grafu s velkým počtem vrcholů a následovným spuštěním algoritmu pomocí jak sekvenčního tak paralelního výpočtu:

```
> make
> ./main -g 3
> ./main -s input.txt
> ./main -p input.txt
```

Projekt také obsahuje jednoduchý skript `testing.py`, který slouží ke generování většího množství grafů, a to jak za pomoci sekvenčního, tak paralelního výpočtu.

Reference

[1] J. Demel *Grafy a jejich aplikace* (Academia, Praha, 2002).