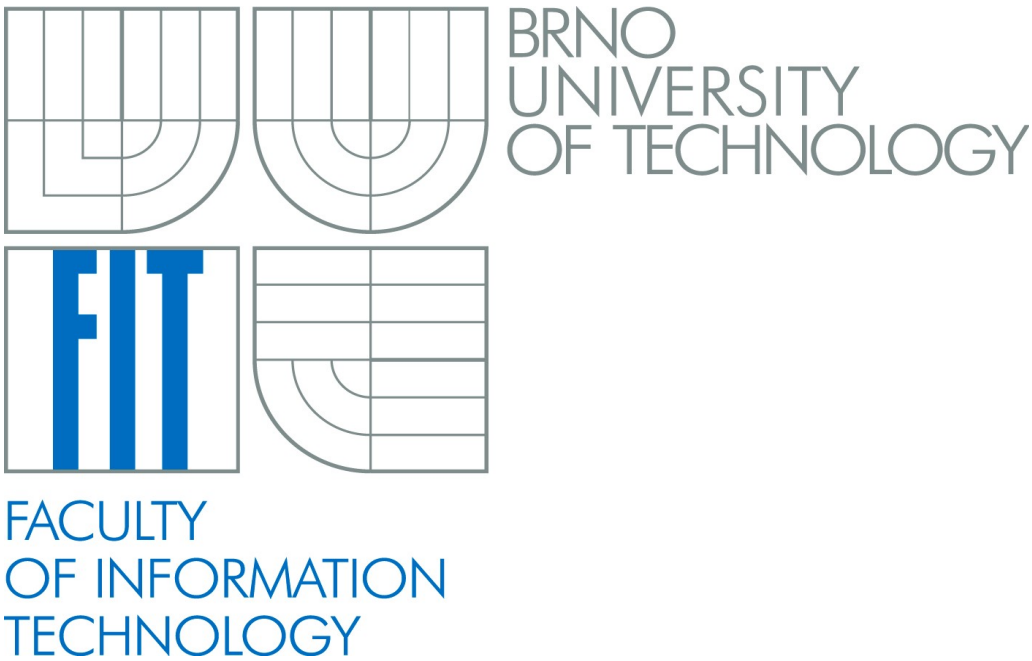


Dokumentácia k projektu do predmetu GAL
2014/2015

Porovnání - Vyhledání silně souvislých komponent (Gabowův algoritmus a Tarjanův algoritmus)

Tomáš Varga, xvarga00@stud.fit.vutbr.cz
Jan Wrona, xwrona00@stud.fit.vutbr.cz

1 Úvod

Cieľom tohoto projektu bolo vytvoriť konzolovú aplikáciu na porovnanie dvoch algoritmov na vyhľadanie silne súvislých komponent, a to konkrétne porovnanie Gabowho a Tarjanovho algoritmu. Kapitola 2 sa venuje analýze a kapitola 3 návrhu a implementácii daného problému. Ďalej v kapitole 4 sú zhrnuté experimenty nad spomínanými algoritmami.

2 Analýza

V tejto kapitole si priblížime niektoré základné pojmy z teórie grafov, pre lepšie pochopenie nasledujúceho textu. Definície týchto základných pojmov sú čerpané z [2]. Ďalej rozoberieme Gabowov a Tarjanov algoritmus.

2.1 Úvod do teórie grafov – základné pojmy

Orientovaný graf *G* je dvojica *G* = (*V*, *E*), kde:

- V je konečná množina uzlov
- $E \subseteq V^2$ je množina hrán.

Graf *G′* = (*V′*, *E′*) je **podgraf** grafu *G* = (*V*, *E*), keď:

- $V' \subseteq V \; a \; E' \subseteq E$.

Postupnosť uzlov $\langle v_0, v_1, v_2, \ldots, v_k \rangle$, kde $(v_i - 1, v_i) \in E$ pre *i* = 1,2,...,k, sa nazýva **sled** dĺžky *k* z v_0 do v_k.

Cesta je sled, v ktorom sa neopakujú uzly. **Orientovaná cesta** je cesta v orientovanom grafe.

Orientovaný graf sa nazýva **silne súvislý**, keď medzi ľubovoľnými dvoma uzlami existuje orientovaná cesta.

Silne súvislá komponenta je taký maximálny podgraf orientovaného grafu, v ktorom pre všetky dvojice vrcholov *u*, *v* existuje cesta z u do v a zároveň z *v* do *u* [1].

Definícia:

Nech *G* = (*V*, *E*) je orientovaný graf. Podgraf *G′* grafu *G* sa nazýva silne súvislá komponenta grafu *G*, keď platí:

1. *G′* je silne súvislý
2. *G′* je maximálny, tj. neexisuje žiadny silne súvislý podgraf *G″* rôzny od *G′* , ktorý by obsahoval podgraf *G′*.

2.3 Prehľadávanie do hĺbky (DFS)

Prehľadávanie do hĺbky (DFS – depth-first search) funguje tak, že sa začína od počiatočného uzlu, a prehľadáva sa prvý potomok, ak tento potomok má ďalších potomkov, tak prehľadávanie pokračuje v potomkovi daného potomka, atď. až raz vojde do uzlu, z ktorého sa už nedá nikam ďalej dostať. V takom prípade sa vráti do uzlu, v ktorom bolo prehľadávanie naposledy a vojde do ďalšieho uzlu, do ktorého sa odtiaľ dá dostať. Týmto spôsobom prehľadá všetky uzly daného grafu.

2.2 Gabowov algoritmus

Gabowov algoritmus je založený na algoritme DFS. Vstupom tohoto algoritmu je orientovaný graf a výstupom je zoznam silne súvislých komponent. Každá takáto komponenta je reprezentovaná zoznamom uzlov. Táto metóda využíva dva zásobníky (S, P). Jeden obsahuje uzly, ktoré zatiaľ nie sú súčasťou žiadnej silne súvislej komponenty. Ďalší obsahuje uzly, ktoré nepatria rôznym silne súvislým komponentám. Gabowov algoritmus ďalej používa zoznam (preorder), v ktorom je o každom vrchole uložená informácia o tom, či vrchol už bol navštívený alebo nie. Ďalším

pomocným zoznamom je zoznam následníkov (Adj), kde pre každý uzol je uložený zoznam jeho potomkov. Pseudokód pre tento algoritmus je nasledovný:

```
GABOW-VISIT:
      preorder[v] ← c; c ← c + 1; push(S, v); push(P, v)
      for w in Adj[v] do
            if preorder[w] = ∞ then
                  GABOW-VISIT(w)
            else if w is in S then
                  repeat
                        pop(P)
                  until preorder [peak(P)] ≤ preorder[w]
            end if
      end for

      if v = peak(P) then
            vytvor novú silne súvislú komponentu c
            repeat
                  w ← pop(S)
                  pridaj w do komponenty c
            until v = w
            pop(P)
            return c
      end if
```

Asymptotická zložitosť tohoto algoritmu je $O(|V|+|E|)$.

2.3 Tarjanov algoritmus

Tarjanov algoritmus je podobne ako Gabowov algoritmus založený na metóde DFS. Uzly sa pri prehľadávaní indexujú podľa poradia, v ktorom boli objavené. Pri návrate z rekurzie sa každému vrcholu priradí uzol s najnižším indexom na aký vie dosiahnuť. Všetky vrcholy, ktoré majú rovnaký cieľový uzol (index), sa nachádzaju v rovnakej komponente. Psedokód pre tento algoritmus je nasledovný:

```
TARJAN-VISIT:
      index[v] ← index; lowlink[v] ← index; index ← index + 1; push(S, v)
      for w in Adj[v] do
            if index[w] = ∞ then
                  TARJAN-VISIT(w)
            else if w is in S then
                  lowlink[v] ← min(lowlink[v], index[w])
            end if
      end for

      if lowlink[v] = index[v] then
            vytvoR novú silnE súvislú komponentu c
            repeat
                  w ← pop(S)
                  pridaj w do komponenty c
            until v = w
            return c
      end if
```

Asymptotická zložitosť tohoto algoritmu je $O(|V|+|E|)$.

3 Návrh a implementácia

Najprv bolo potrebné vymyslieť vhodnú reprezentáciu grafu, štruktúru, ktorá by zabaľovala všetky uzly, hrany grafu a operácie nad ním. Následne bolo potrebné, aby poskytovala informácie o následníkoch pre každý uzol. Ďalej trebalo premyslieť zmeranie zložitosti daných algoritmov.

Na implementáciu bol zvolený programovací jazyk Python (verzia 2.7) a boli použité neštandardné knižnice NetworkX a psutil.

3.1 Knižnice

NetworkX je balíček pre vytváranie a manipuláciu štruktúr, dynamiky a funkcií s komplexnými sieťami [5]. Táto knižnica bola použitá pre generovanie grafov rôznych druhov (riedke a husté grafy). Ďalej bola použitá pre overenie správnosti implementovaných algoritmov a na porovnanie pri experimentoch.

Psutil je multiplatformná knižnica pre získavanie informácií o spustených procesoch a využití systému (CPU, pameť, disky, siete) v Pythonu. To je užitočné hlavne pri monitorování systému, profilovaní a obmedzení procesných prostriedkov a riadení bežiacich procesov [6].

3.2. Reprezentácia grafu

Štruktúru grafu reprezentuje objekt *Graph* v module *graph.py*. Inštanciou tohoto objektu je slovník, ktorý ako kľúče obsahuje všetky uzly a ku každému kľúču (uzlu) je priradený zoznam uzlov s ktorými susedí. V podstate nám tieto zoznamy reprezentujú zároveň aj zoznamy následníkov. Ďalej tento modul obsahuje aj metódy na prevod medzi štruktúrami knižnice *NetworkX* a zmienenou vlastnou štruktúrou. Tieto metódy sa využívajú pri prevode vygenerovaných grafov pomocou knižnice *NetworkX*.

3.3 Algoritmy

Implementácia Gabowho algoritmu je obsiahnutá v module *gabow.py* a implementácia Tarjanovho algoritmu zas v module *tarjan.py*. Ich implementácia vychádza z popisu v kapitolách 2.2 resp. 2.3 a slúži na to metóda *strongly_connected_components*, ktorá využíva rekurzívnu metódu *__scc_visit* v oboch moduloch, ktorá postupne generuje silne súvislé komponenty. Tieto komponenty sú reprezentované zoznamom čísiel, kde každé číslo predstavuje uzol.

Pre porovnanie správnosti našej implementácie boli použité metódy z knižnice *NetworkX*, ktoré implementujú Tarjanov algoritmus s Nuutilovou modifikáciou a Kosarajov algoritmus. V kapitole 4 budú označené s príponou *_nx*.

3.4 Ovládanie aplikácie

Keď v systéme nie sú prítomné neštandardnýé knižnice zmienené v kapitole 3.1, tak je možné ich nainštalovať spustením skriptu *INSTALL.sh*, ktorý vykoná ich inštaláciu do domovského adresára užívateľa.

Súbor, ktorý vykonáva samotné meranie a porovanie sa volá *scc_analysis.py*. Jeho hlavným účelom je vykonanie meraní časovej a priestorovej zložitosti jedného, či viac algoritmov zároveň. Výstupom sú vždy dva súbory. Prvý s príponou *_time.csv* a druhý s príponou *_memory.csv*, ktoré

obsahujú namerané hodnoty spotrebovaného procesorového času a spotrebovanej pamäte. Program môže byť spustený v dvoch režimoch:

1. *single* pre meranie jedného algoritmu, pričom parametrom je hustota grafu
2. *multi* meria jeden či viac algoritmov pre rovnaký graf

Spoločné prepínače pre oba režimy sú:

- *-o* parameter OUTPUT predstavuje prefix zmieňovaných vstupných súborov
- *-s* týmto prepínačom je možné ovplyvniť rozsah veľkosti grafu, ktoré sú predávané na vstup meraným algoritmom. Možný je jeden parameter z množiny {low, mid, high}
- *-r* ovplyvňuje počet opakovaní všetkých meraní. Vyššie číslo znamená presnejšie meranie, ale taktiež výrazne predĺži celkovú dobu behu programu
- *-m* určuje počet opakovaní algoritmu behom jedného merania. Funkcia meriaca čas neposkytuje také rozlíšenie, aby bolo možné presne zmerať dobu jediného behu algoritmu
- *-c* umožňuje riadiť počet procesov, ktoré budú pri meraní použité. Bez tohoto prepínača bude paralelne spustených práve toľko meriacich procesov, koľko je dostupných procesorov.
- *-g* zapne garbage collector behom merania. Garbage collector môže ovplyvniť dobu behu programu. Na systémoch s malým množstvom pamäti, ale môže jeho vypnutie znamenať jej úplne zaplnenie. Preto je možnosť garbage collector prístupná užívateľovi

Ovládanie režimu *single*:

- --algorithm prijíma ako parameter jeden algoritmus z množiny {kosaraju_nx, tarjan, tarjan_nx, gabow}. Jedná sa o algoritmus, na ktorom bude prevádzané meranie
- --densities prijíma ako parameter čiarkou oddelené hodnoty, ktoré znamenajú hustoty grafu, na ktorých bude altoritmus zmeraný. Jedná sa o reálne čisla v rozsahu <0,1>, pričom 0 je graf bez akýchkoľvek hrán a 1 je úplny graf.

Ovládanie režimu *multi*:

- --algorithms prijíma ako parameter čiarkou oddelené názvy algoritmov z množiny {kosaraju_nx, tarjan, tarjan_nx, gabow}. Jedná sa o algoritmy, ktoré budú vzájomne porovnávané
- --density v režime *multi* je meraných viac algoritmov nad grafmi s rozdielnou veľkosťou, ale vždy rovnakou hustotou. Parametrom tohoto prepínača je možné túto hustotu nastaviť. Jedná sa o reálne číslo, ktorého význam je bližšie popísaný pri prepínači --*densities* v režime *single*

Ďalšou súčasťou vytváranej aplikácie je skript *run_measurement.sh*, ktorý postupne spúšťa program *scc_analysis.py* s rôznym nastavením. Jeho beh môže trvať veľmi dlho, v závislosti na dostupnom hardware.

4 Experimenty

Režimy meriaceho programu zmieneného v kapitole 3.4 napovedajú, že výsledky experimentu sú rozdelené do dvoch častí. Tou prvou je meranie jednotlivých algoritmov, každého zvlášť, v závislosti na veľkosti grafu a jeho hustote. Z týchto výsledkov sa dá vyčítať predovšetkým pre akú hustotu grafu algoritmus pracuje rýchlejšie. Druhá časť experimentov algoritmy vzájomne porovnáva a je teda podstatne relevantnejší čo sa týka zadania tohoto projektu.

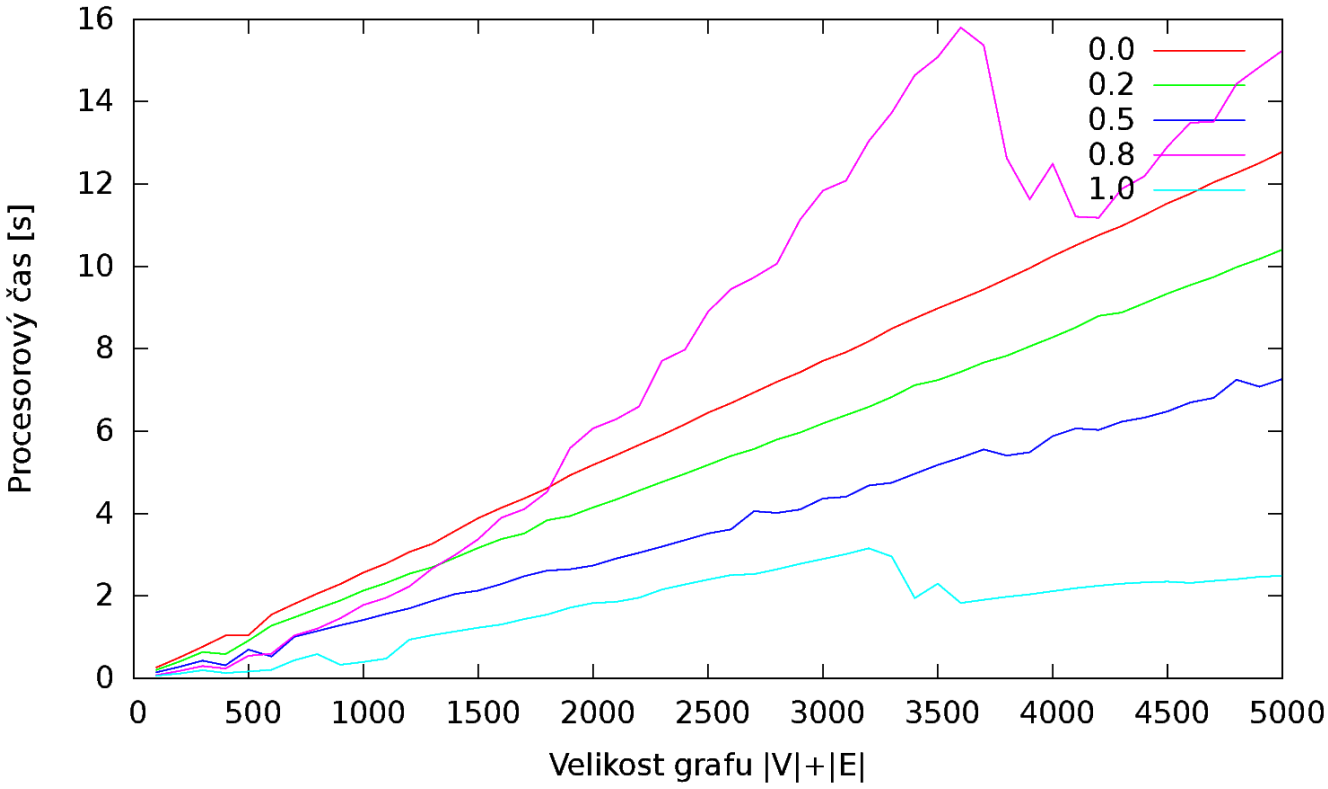
Aj keď sme časovú zložitosť merali ako spotrebovaný procesorový čas (nie reálny), stále môžu byť výsledky skreslené faktormi, ktoré sme nemohli ovplyvniť. Jedná sa predovšetkým o vplyv iných procesov, ktoré na systému bežia súčasne. Tento faktor sme minimalizovali spustením meraní na vyhradenom stroji, na ktorom okrem procesov nevyhnutných pre fungovanie operačného systému a administráciu stroja, nebežali žiadne iné procesy. Takéto možnosti školné servery neposkytujú a osobné počítače neposkytuju dostatočný výkon ani potrebné množstvo operačnej pamäte. Preto sme využili služieb *Národnej Gridovej Infraštruktúry, MetaCentrum*. Pre prvé meranie bol exkluzívne vyhradený stroj z clusteru *doom.metacentrum.cz*, druhé meranie bolo prevádzané na stroji *eru2.ruk.cuni.cz*. Druhý zmieňovaný je osadený menej výkonnými procesormi, čo je na výsledkoch vidieť (krivky majú rovnaký priebeh, ale dá sa pozorovať väčšia spotreba procesorového času pri zhodnej veľkosti grafu).

Pri experimentovaní sa každá metóda volala určitý krát, aby sme eliminovali nízke rozlíšenie meraného procesorového času (pri našom experimentovaní bola táto hodnota nastavená na 1000). Toto chovanie ide nastaviť parametrom *-m* zmieneným v kapitole 3.4. Celé meranie sa opakovalo viackrát, aby sme zmenšili chybu merania (pri našom experimentovaní bola táto hodnota nastavená na 3) a napokon sa vybrala najmenšia hodnota.

4.1 Jednotlivé algoritmy – časová zložitosť

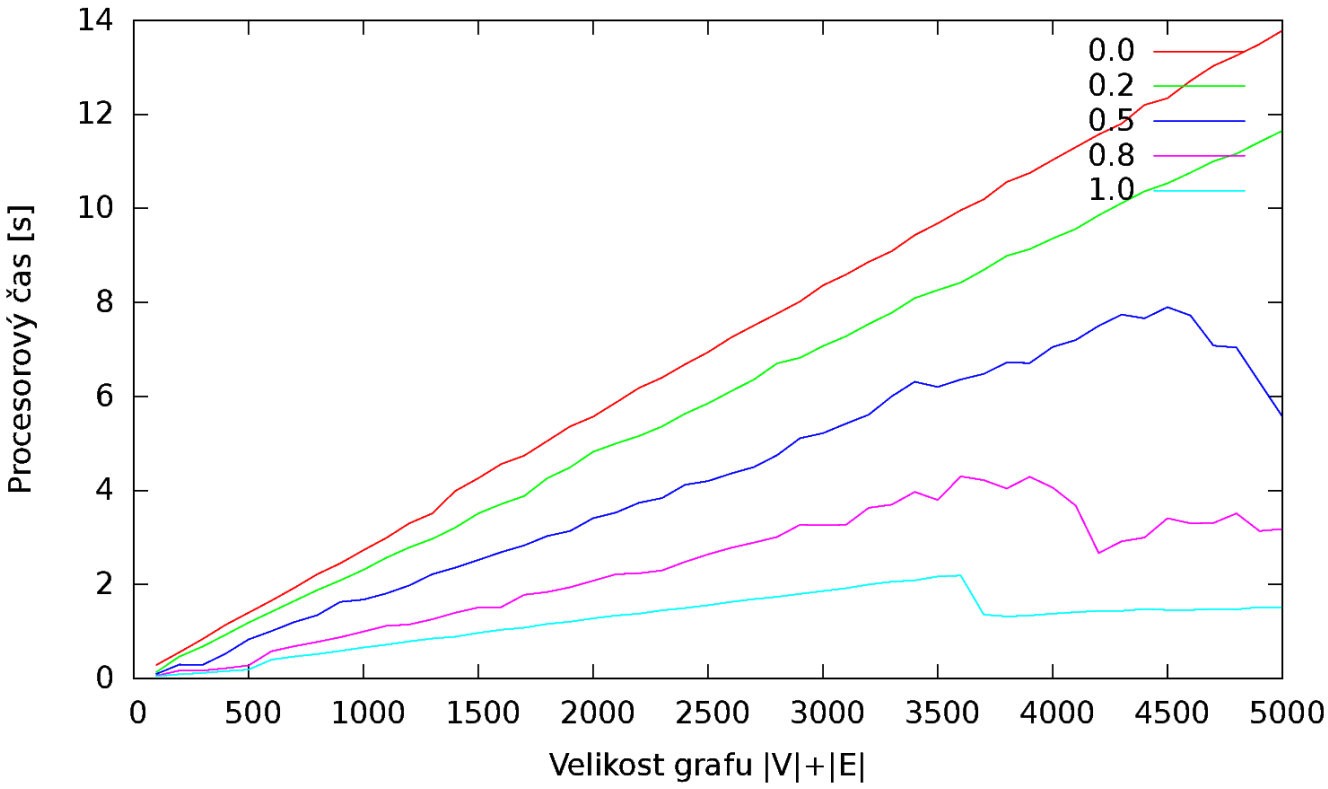
Gabowov algoritmus

Zväčšujúcim sa grafom, časová zložitosť rastie lineárne. Pro redšie grafy je sklon priamky väčší, čo je spôsobené vyprázdňovaním zásobníku a priradzovaním všetkých uzlov do silne súvislej komponenty. Pre grafy s hustotou 0.8 sa pre veľkosť grafu 1000 a viac algoritmus chová mierne odlišne – časová náročnosť je vyššia. Že sa nejedná o chybu merania dokazuje veľmi podobný výsledok merania č.2, ktorého výsledky nájdete v prílohe.



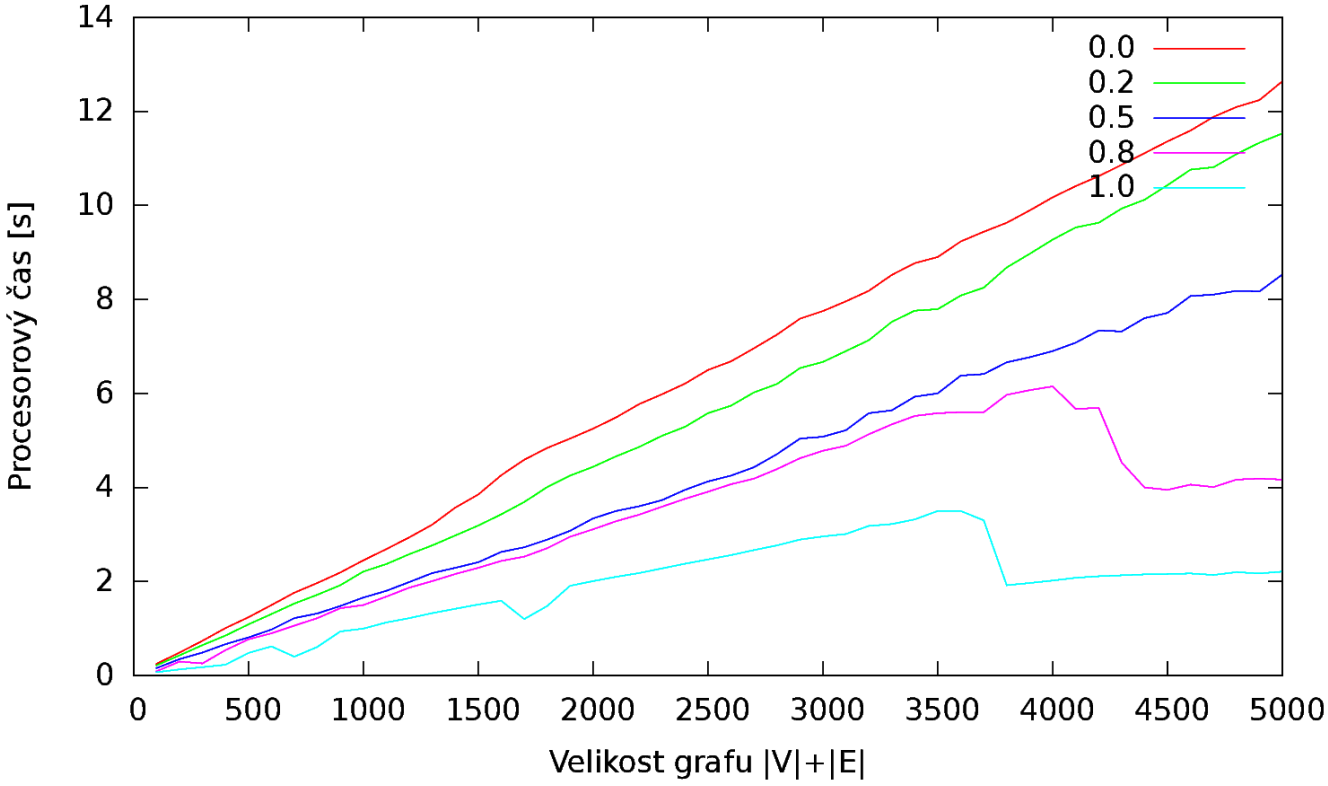
Tarjanov algoritmus

Ani meranie Tarjanovho algoritmu neprinieslo žiadne prekvapenie, čo sa týka jeho časovej zložitosti. Opäť je vidieť lineárny priebeh v závislosti na veľkosti grafu, pričom redšie grafy znamenajú väčšiu časovú náročnosť.



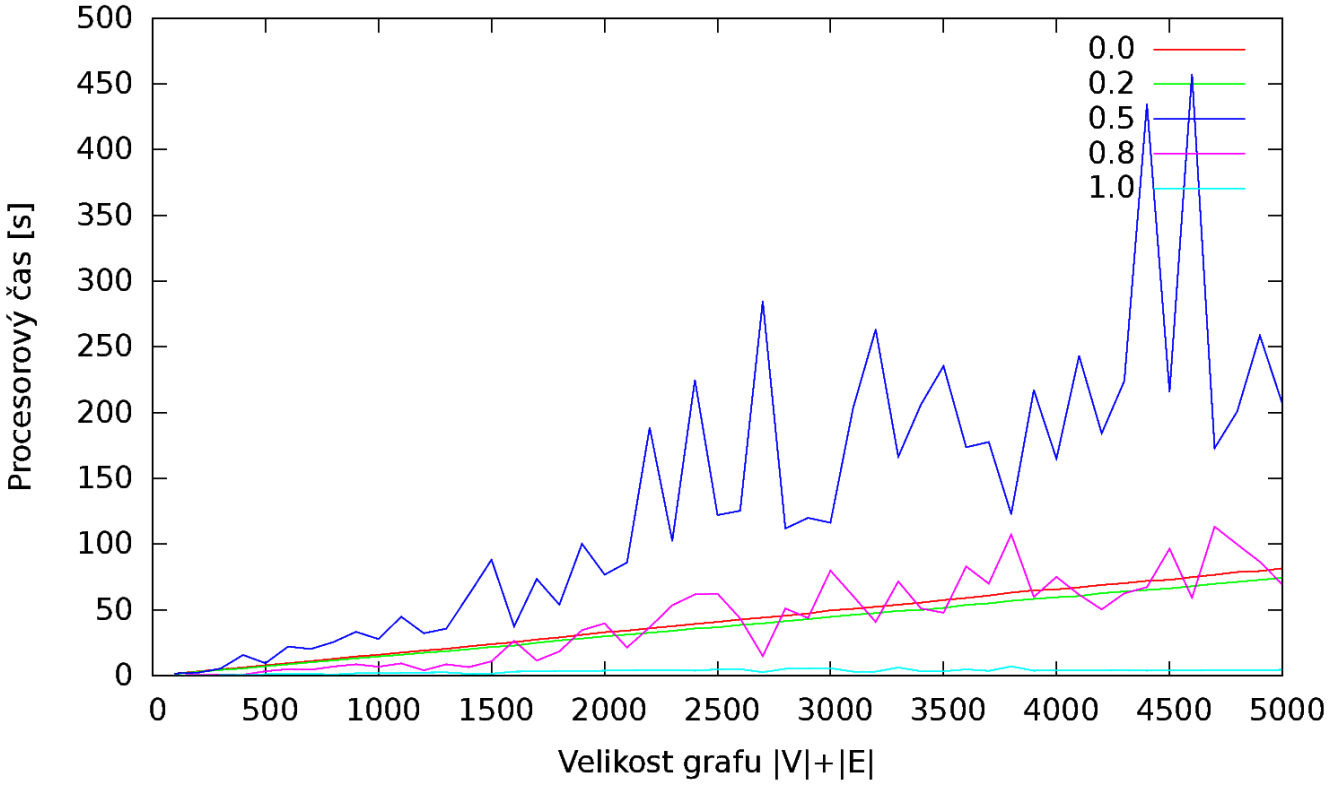
Tarjanov algoritmus s Nuutilovou modifikáciou

Na prvý pohľad tento graf kopíruje výsledky Tarjanovho algoritmu bez modifikácie. Dá sa ale vidieť, že při porovnaní algoritmu v nasledujúcej sekcii, nie je tomu tak a krivky sú vzájomne posunuté hore či dole, v závislosti na hustote.



Kosarajov algoritmus:

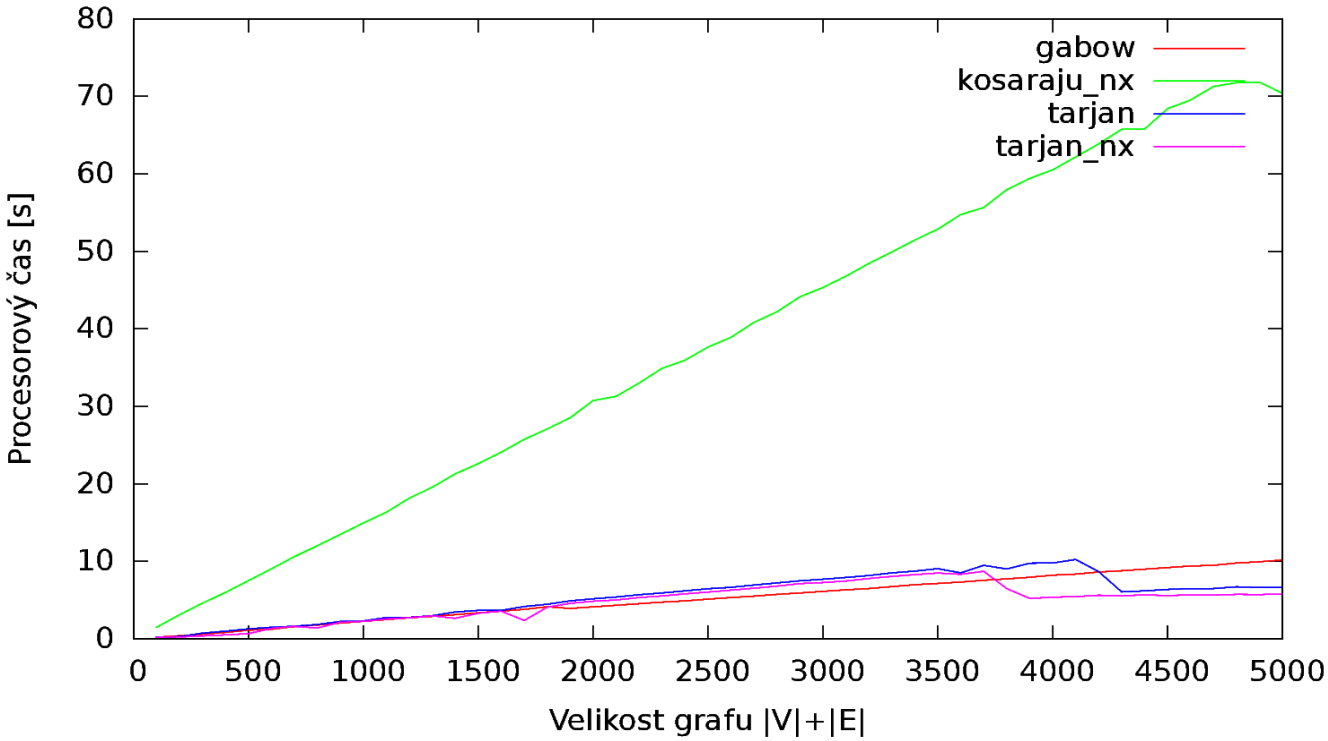
Pre hustoty 0.0, 0.2 a 1.0 nie sú výsledky príliž prekvapivé. Naopak pre hustoty 0.5 a predovšetkým 0.8 sú výsledky veľmi zaujímavé. Aj keď interpoláciou hodnôt by vznikla približne priamka (lineárna zložitosť je zachovaná), pre grafy s podobnou veľkosťou algoritmus podáva prekvapivo rozdielne výsledky. Príčina tohto chovania nám nie je celkom jasná, ani v tomto prípade sa ale nejedná o chybu merania. Druhé meranie, ktoré nájdete v prílohe poskytlo obdobné hodnoty.



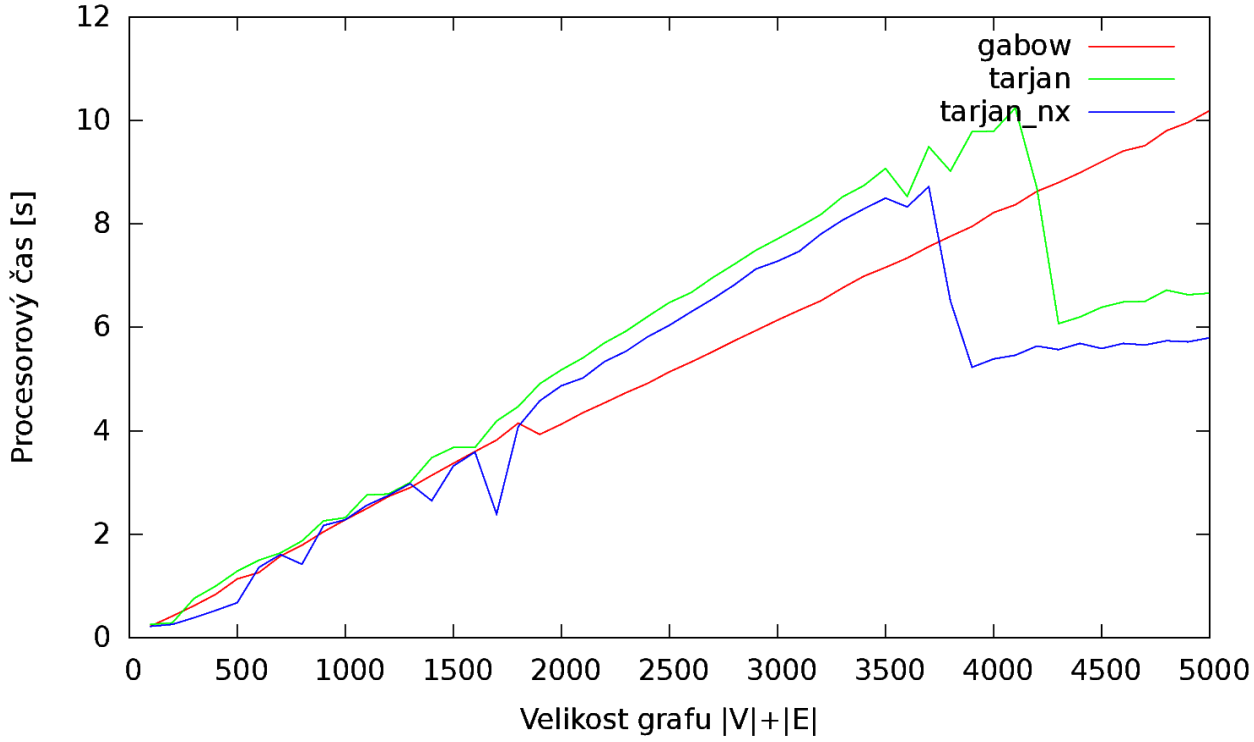
4.2 Porovnanie algoritmov – časová zložitosť

Hustota 0.2 – riedky graf

Kosarajov algoritmus podáva podstatne horšie výsledky ako ostatné algoritmy, dokonca je potreba graf vykresliť bez neho, aby bolo možné ostatné algoritmy porovnať.

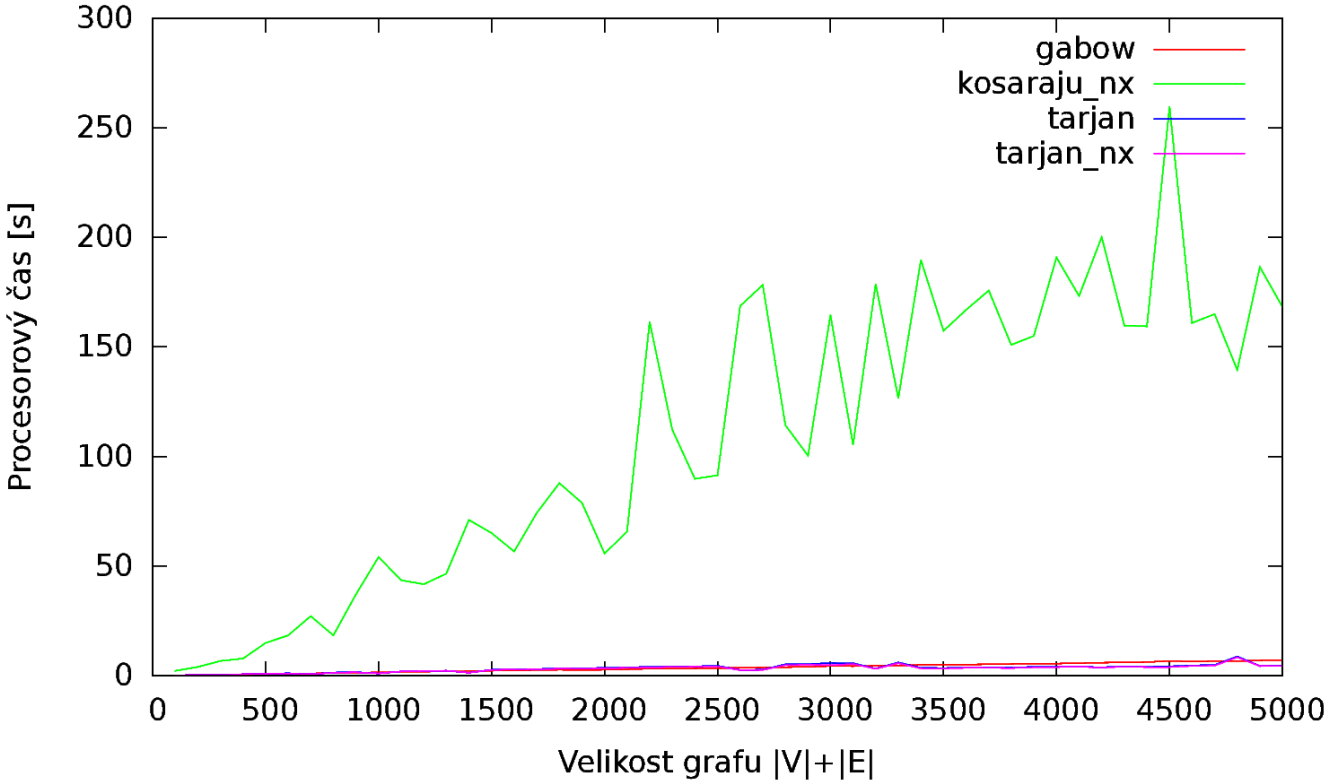


Z nasledujúceho grafu sa dá vyčítať predovšetkým to, že Tarjanov algoritmus s Nuutilovou modifikáciou vážne podáva lepšie výsledky pre riedke grafy. Gabowov algoritmus je pre menšie grafy mierne efektívnejší, naopak je tomu pri grafe s veľkosťou nad 4500.

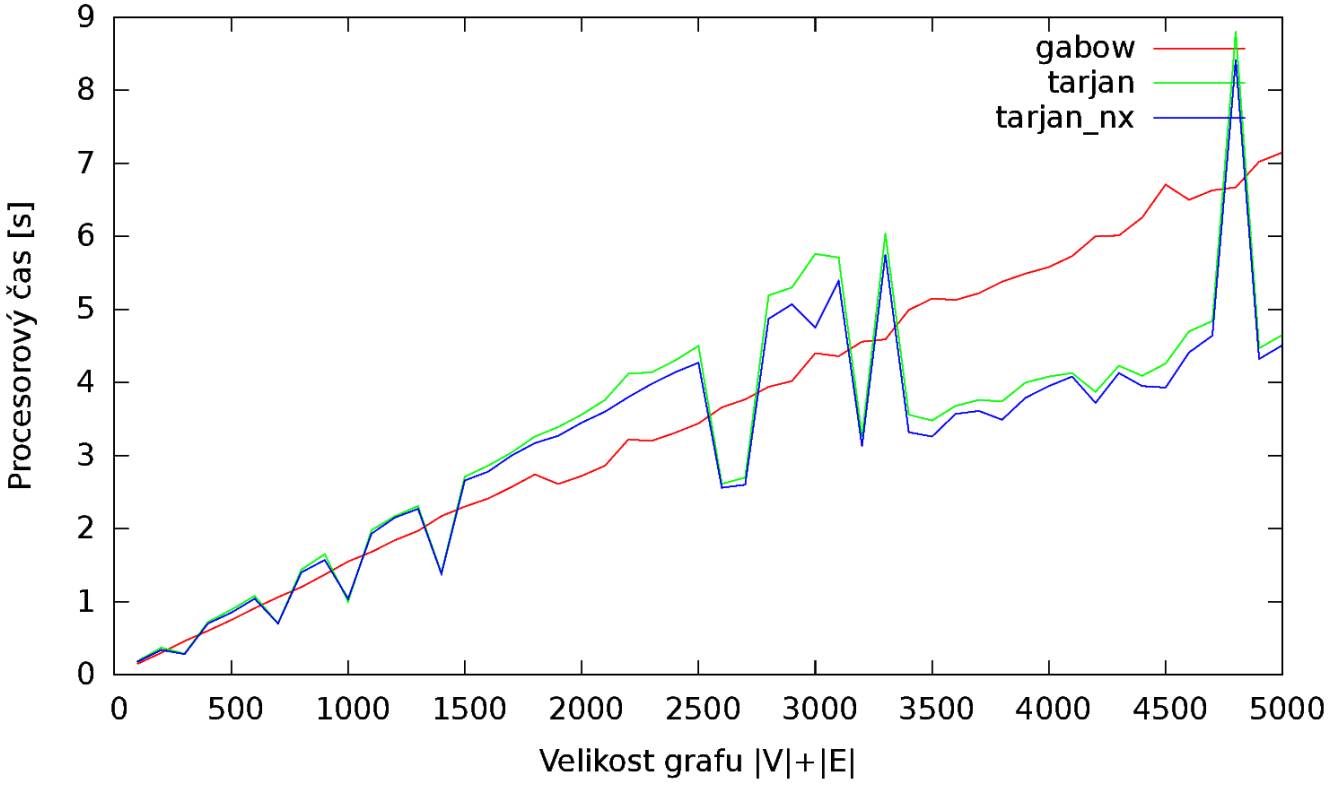


Hustota 0.5 – rovnaký počet hrán ako uzlov

Situácia s Kosarajovým algoritmom sa opakuje, nižšie uvedený graf je vykreslený bez neho.

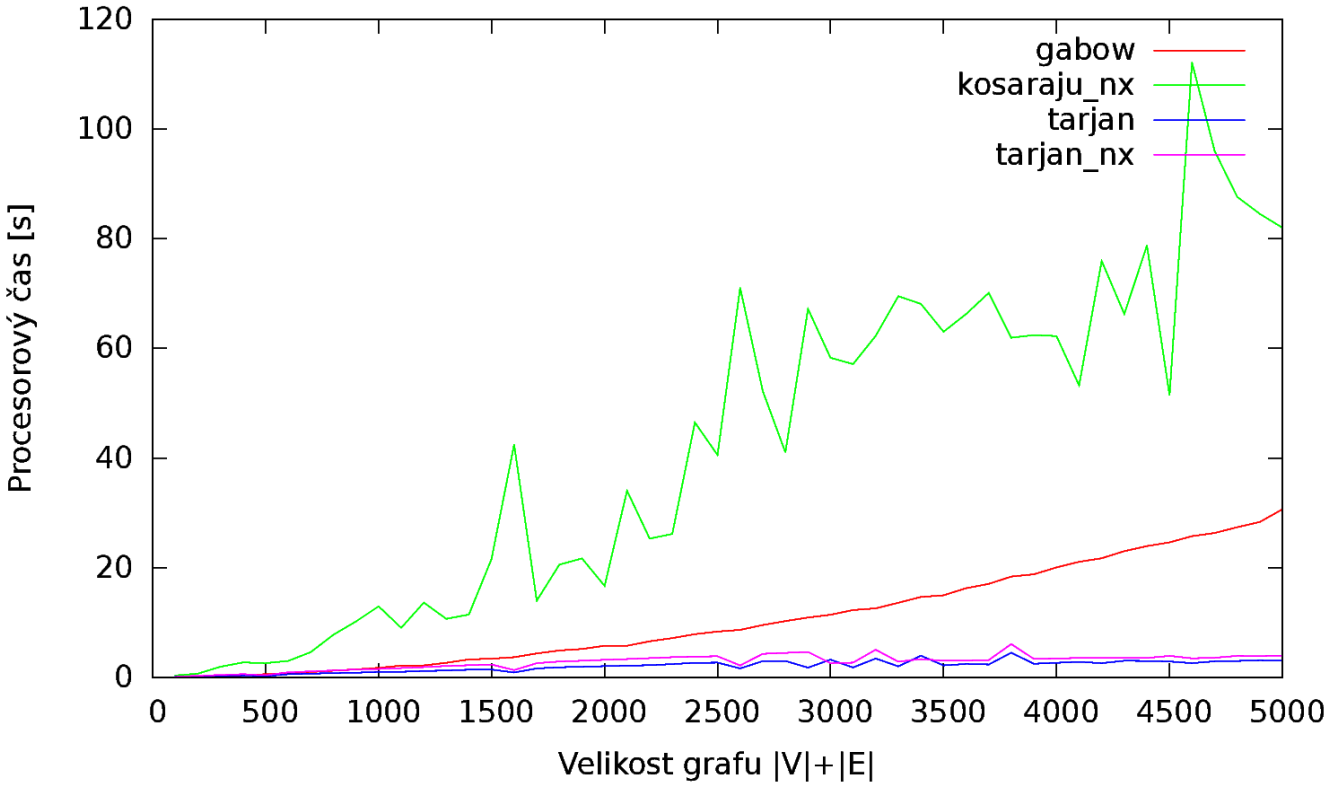


Pre graf s rovnakým počtom uzlov i hrán sa pôvodný aj modifikovaný Tarjanov algoritmus vyrovnávajú. Situácia ohľadne Gabowovho algoritmu je totožná s hustotou 0.2.

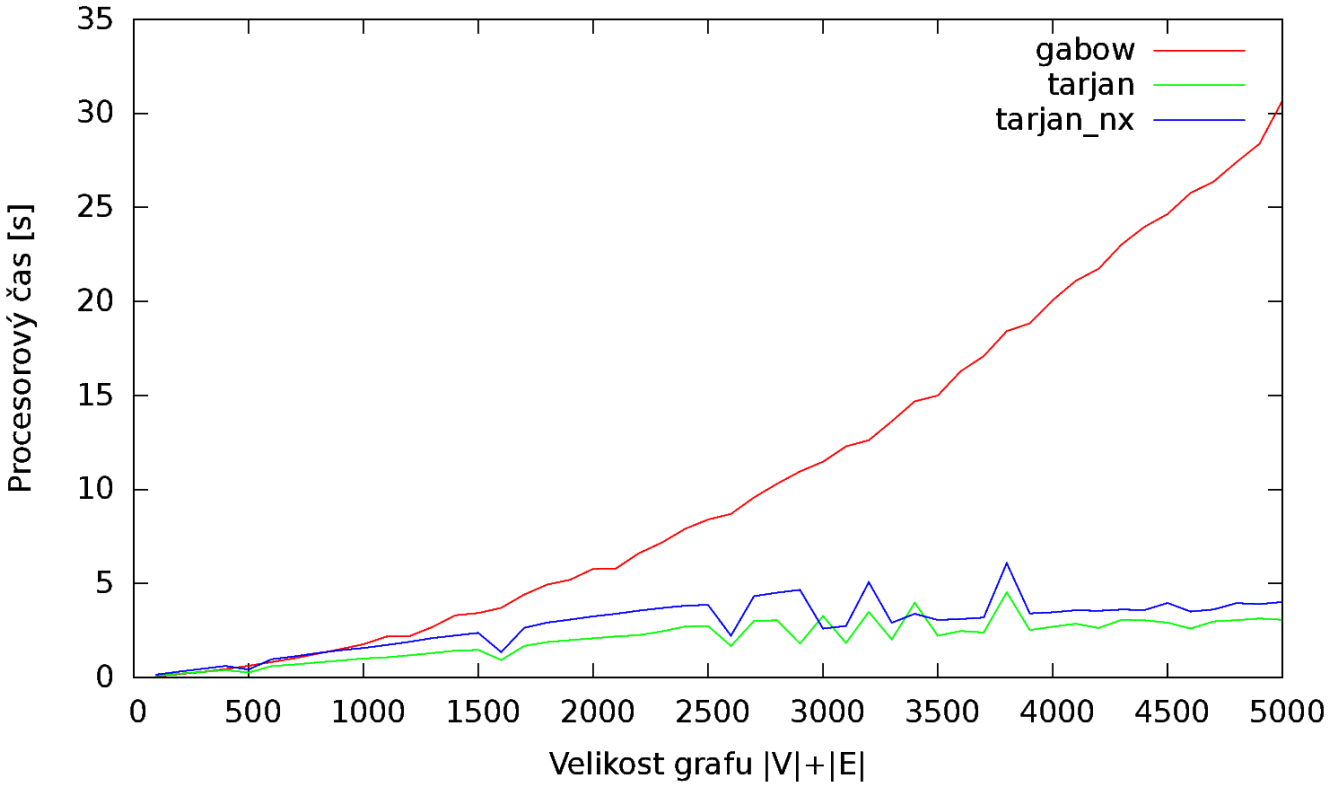


Hustota 0.8 – hustý graf

Ani pre hustý graf sa Kosarajov algoritmus efektivitou nepribližuje svojím konkurentom. Znova je nižšie uvedený výsledok zobrazený bez neho.

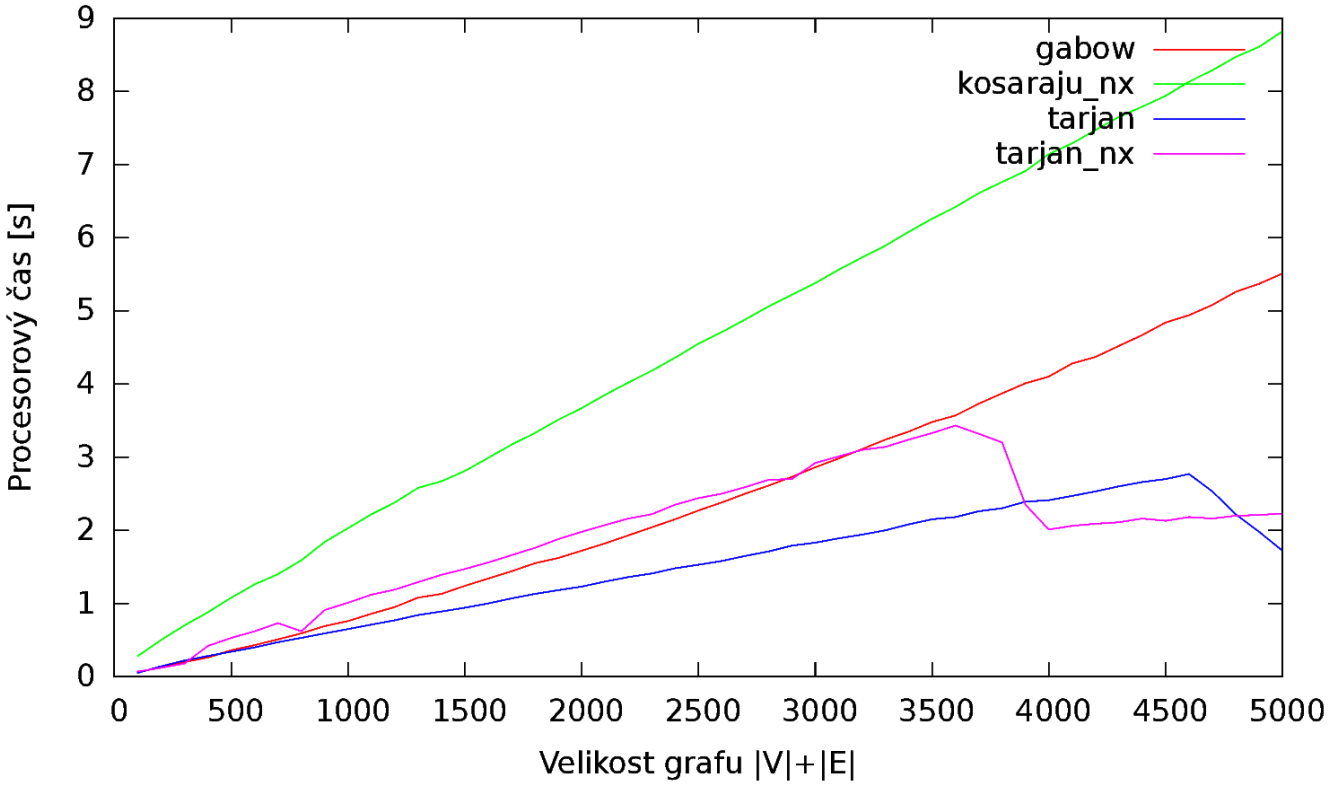


Pre hustý graf dosahuje Tarjanov algoritmus dokonca horšie časy, než jeho pôvodná verzia. Gabowov algoritmus sa viditeľne odchyluje. Pre husté grafy je podstatne menej efektívny.



Hustota 1.0 – úplny graf

Kosarajov algoritmus pre úplny graf poskytuje tiež porovnateľnú časovú zložitost s ostatnými algoritmami. Ale aj tak je stále najmenej efektívny. Na opačnom konci sa nachádza Tarjanov algoritmus, ktorý dosahuje najlepších výsledkov z testovaných algoritmov.

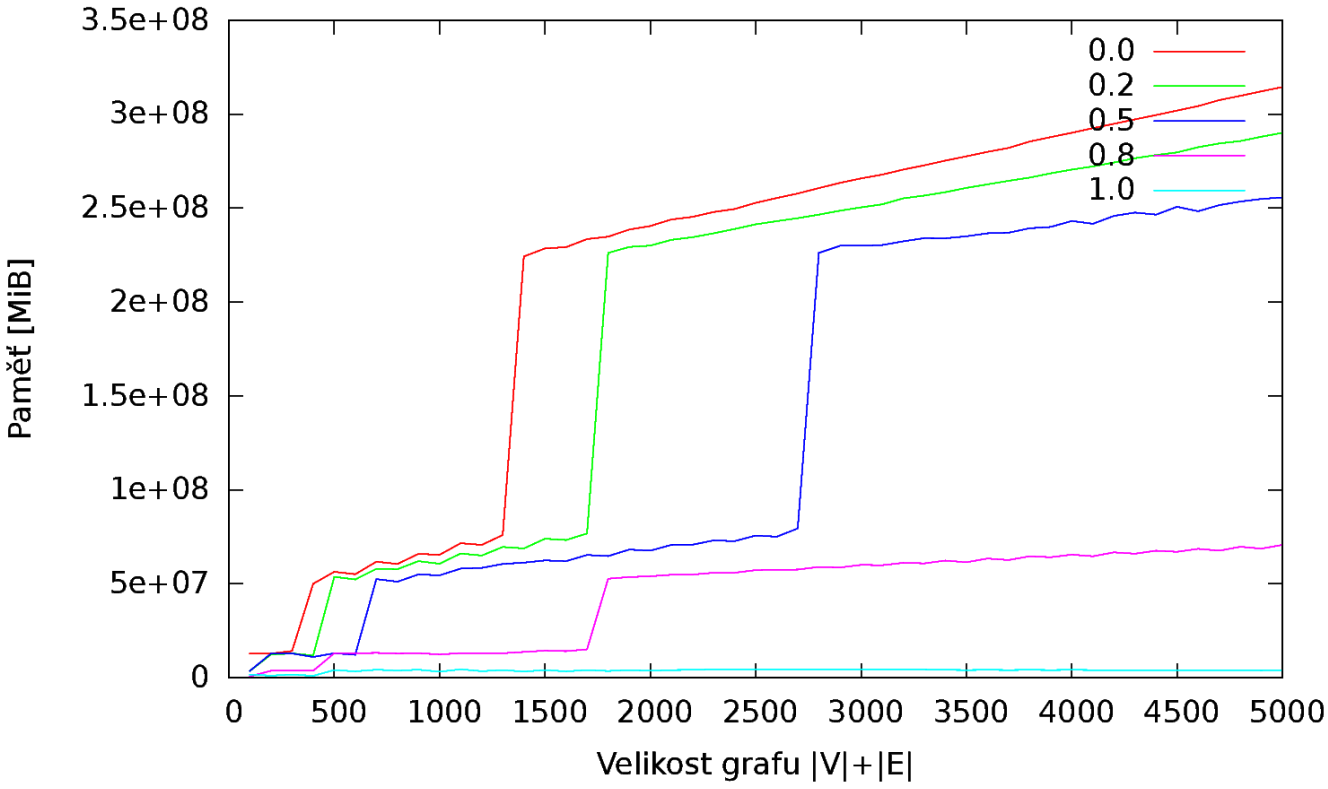


4.3 Jednotlivé algoritmy – priestorová zložitosť

Na úvod sekcie priestorovej zložitosti je potrebné zmieniť, že pri implementácii boli použité abstraktné datové typy, ktoré si alokujú viac pamäti, než skutočne potrebujú. Veľkost takejto alokovanej pamäti rastie skokovo, aj keď veľkosť grafu sa pri experimentoch zväčšovala lineárne. Výsledky sú týmto značne ovplyvnené.

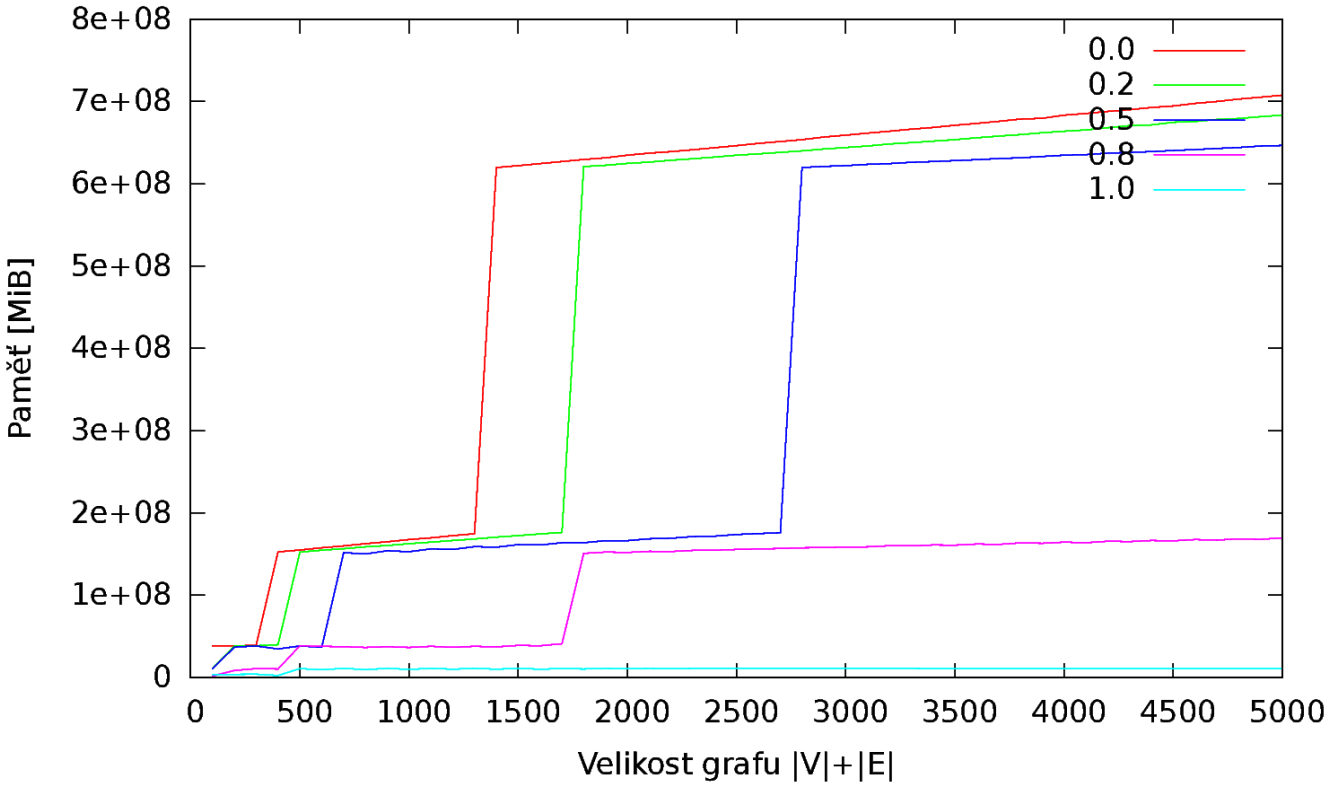
Gabowov algoritmus

Výsledky sú podobne ako pri časovej zložitosti. Riedke grafy poskytujú horšie výsledky než grafy husté. Výpočet pre úplne grafy je až prekvapivo priestorovo nenáročný.



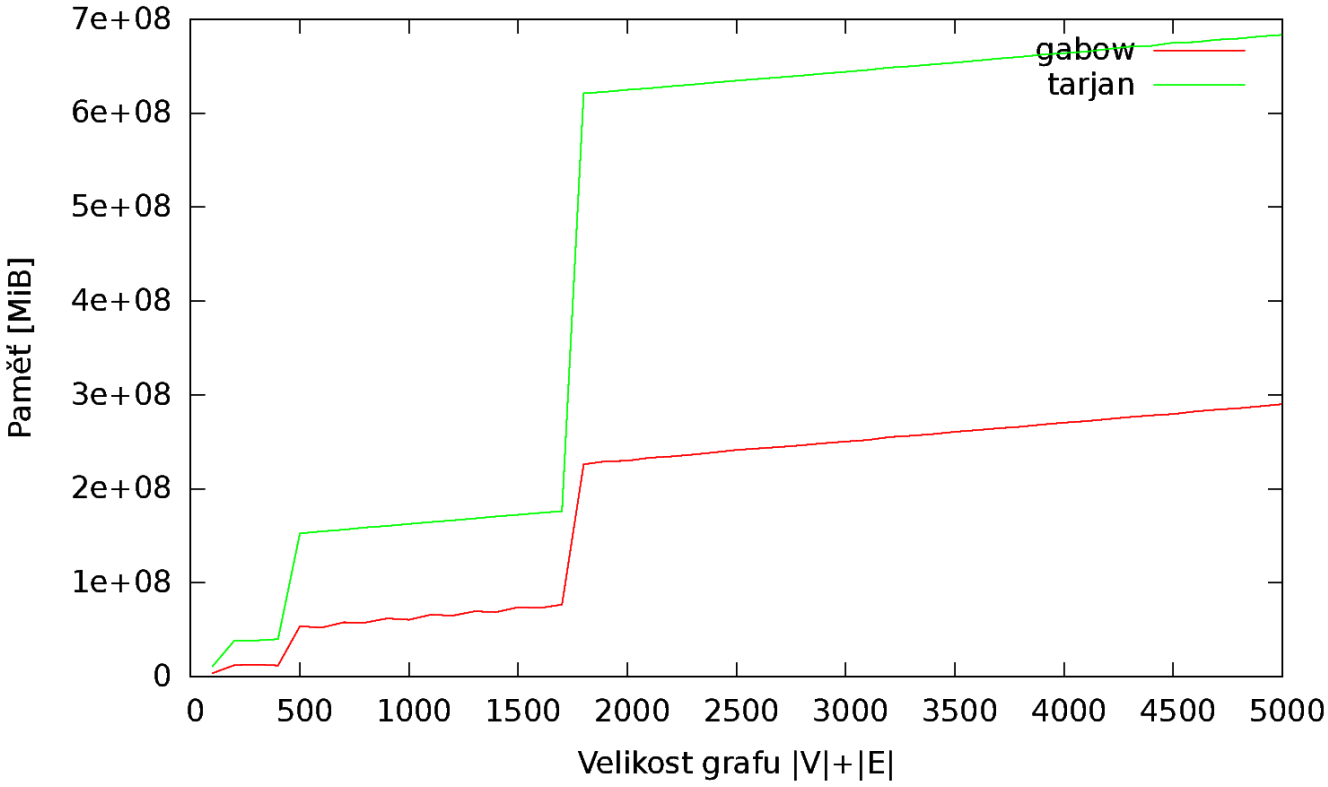
Tarjanov algoritmus

Výsledok experimentu pre Tarjanov algoritmus nepriniesol žiadne nečakané výsledky. Situácia je obdobná ako u Gabowovho algoritmu.

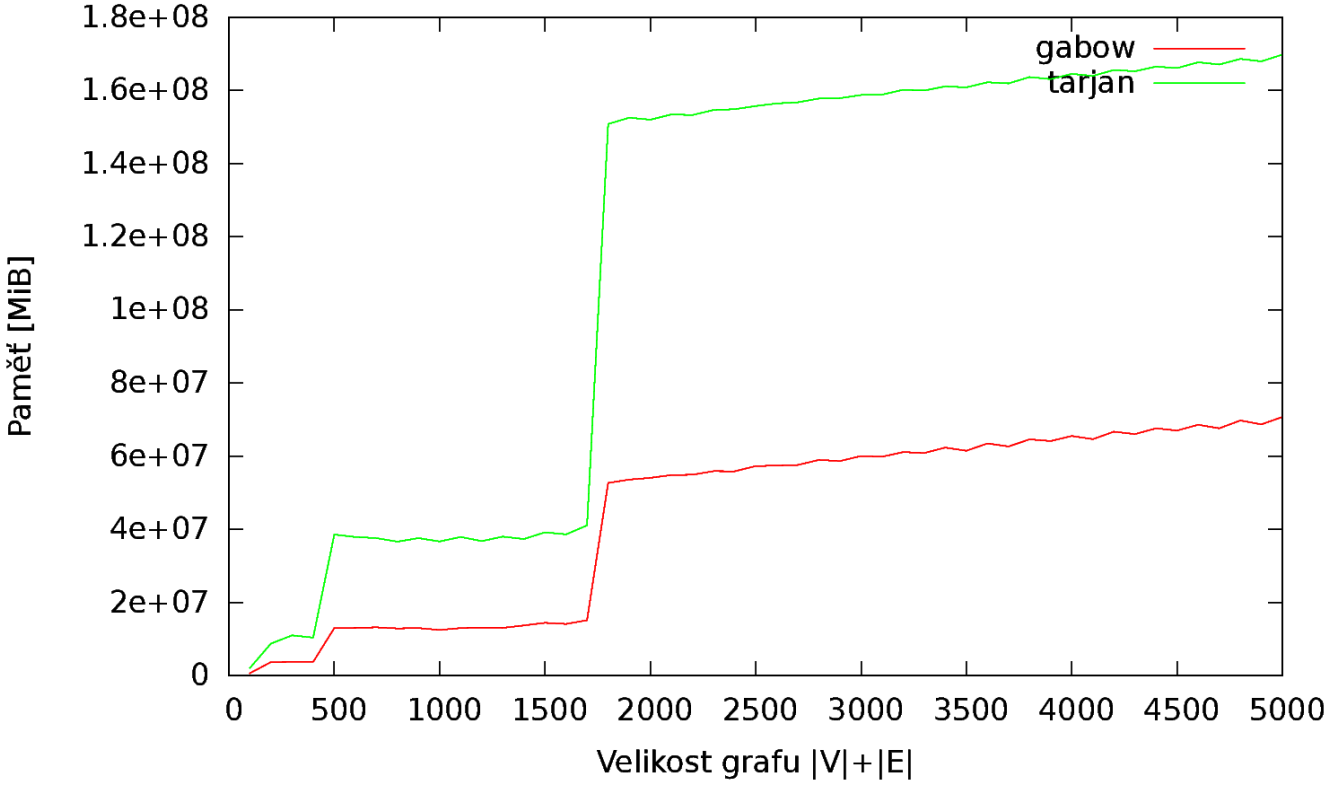


4.4 Porovnanie algoritmu – priestorová zložitosť

Hustota 0.2 – riedky graf



Hustota 0.8 – hustý graf



Porovnanie priestorovej zložitosti vychádza lepšie pre Gabowov algoritmus, a to ako pre riedke tak aj pre husté grafy.

Záver

Pri výsledkoch z experimentov pre jednotlivé algoritmy je vidieť, že teoretická zložitosť algoritmov spomínaná v kapitole 2 odpovedá nameranej zložitosti pri experimentoch z kapitoly 4, a to lineárnej zložitosti.

Pri hodnotení časovej zložitosti jednotlivých algoritmov sa dá z experimentov vyčítať aj to, že získanie silne súvislých komponent z riedkých grafov je časovo náročnejšie oproti ich ziskaniu z hustých grafov. Je to spôsobené tým, že pri každom volaní metódy *__scc_visit* sa do zásobníku pridá aktuálny uzol, takže ich tam bude presne *n*. Zásobník sa musí v druhom cykle celý vyprázdniť, takže čím viac uzlov, tým viackrát sa druhý cyklus vykoná. Pri porovnávaní algoritmov medzi sebou sme zistili, že Kosarajov algoritmus podáva podstatne horšie výsledky ako ostatné algoritmy a to aj pre riedke aj pre husté grafy. Pre riedke grafy podával celkovo najlepšie výsledky Tarjanov algoritmus s Nuutilovou modifikáciou. Gabowov algoritmus podával lepšie výsledky len pre menšie grafy. Pre grafy s rovnakými počtami hrán a uzlov sa pôvodný aj modifikovaný Tarjanov algoritmus vyrovnávajú. Gabowov algoritmus je aj v tomto prípade efektívnejší len pre menšie grafy. Pre hustý graf dosahuje Tarjanov algoritmus dokonca horšie časy, než jeho pôvodná verzia. Gabowov algoritmus je pre husté grafy podstatne menej efektívny.

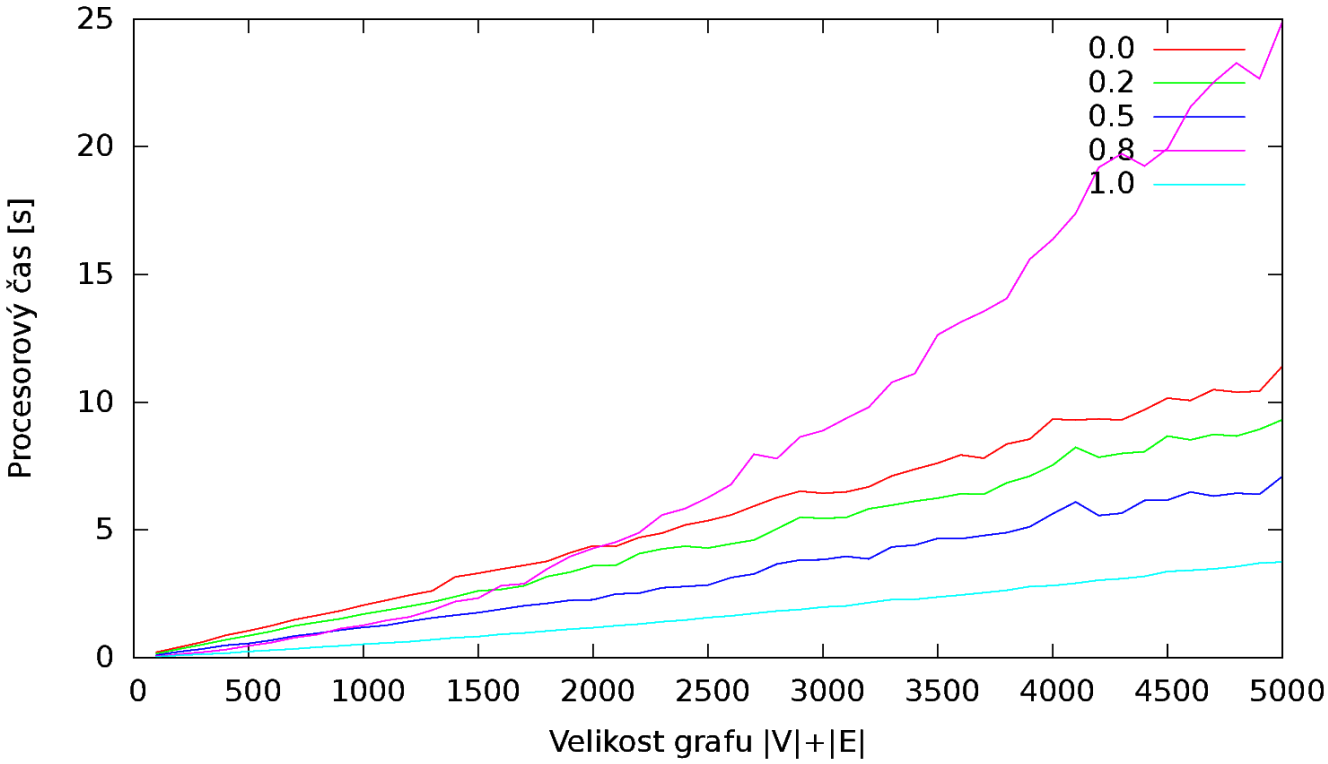
Pri hodnotení pamäťovej zložitosti jednotlivých algoritmov sme prišli na to, že Python si nejakým spôsobom predalokuje viac pamäti, ako skutočne potrebuje. Čo pri experimentoch dáva trošku skreslené výsledky. Aj napriek tomu sú výsledky podobné ako pri časovej zložitosti. Pre Gabowov algoritmus nad riedkymi grafmi poskytuje horšie výsledky než nad hustými grafmi. Výpočet pre úplne grafy je až prekvapivo priestorovo nenáročný. Výsledok experimentov pre Tarjanov algoritmus je značne podobný výsledku z Gabowho algoritmu. Porovnanie priestorovej zložitosti vychádza lepšie pre Gabowov algoritmus, a to ako pre riedke tak aj pre husté grafy.

Na záver môžeme skonštatovať, že kvôli skresleným hodnotám z experimentovania nad pamäťovou zložitosťou je vo výsledku Tarjanov algoritmus efektívnejši ako Gabowov algoritmus.

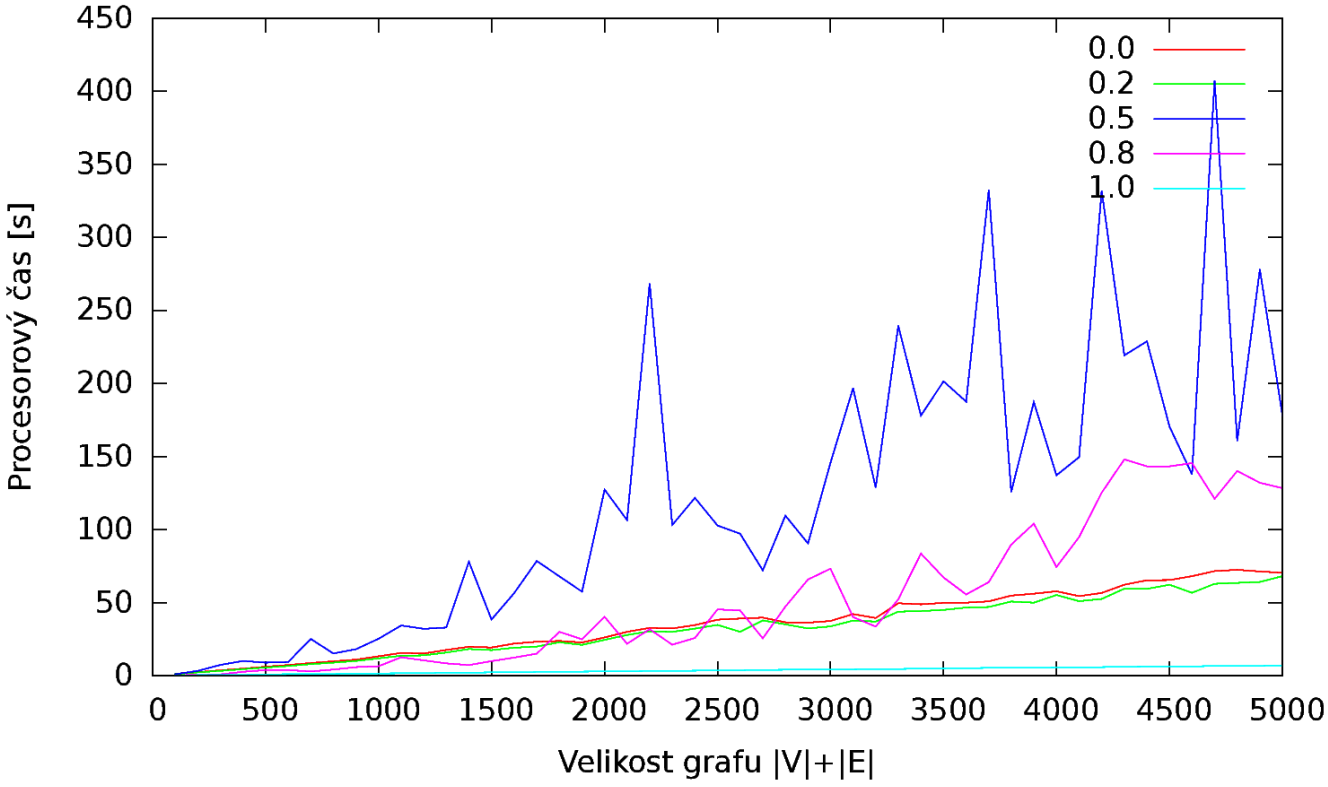
Príloha A

Niektoré výsledky druhého merania, ktoré bolo spustené v rovnakom rozsahu ako meranie č.1. Tu sú uvedené len tie výsledky, pri ktorých meranie č.1 dopadlo nečakane a je potrebné prekázať ich správnosť.

Gabowov algoritmus:



Kosarajov algoritmus:



Zdroje

[1] Silně souvislá komponenta. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2014 [cit. 2014-12-13]. Dostupné z: http://cs.wikipedia.org/wiki/Siln%C4%9B_souvisl%C3%A1_komponenta

[2] KŘIVKA, Zbyněk a Tomáš MASOPUST. Grafové algoritmy. Brno, 2014. Slajdy. FIT VUT.

[3] Path-based strong component algorithm. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2012, 2014 [cit. 2014-12-13]. Dostupné z: http://en.wikipedia.org/wiki/Path-based_strong_component_algorithm

[4] GABOW, Harold N. *Computer Science Technical Report*. University of Colorado Boulder, 1999. Dostupné z:

http://scholar.colorado.edu/cgi/viewcontent.cgi?article=1836&context=csci_techreports

[5] NETWORKX, Developer team. NetworkX. *NetworkX* [online]. 2014 [cit. 2014-12-13]. Dostupné z: https://networkx.github.io/

[6] RODOLA, Giampaolo. Psutil. *Psutil* [online]. 2008 [cit. 2014-12-13]. Dostupné z: https://github.com/giampaolo/psutil