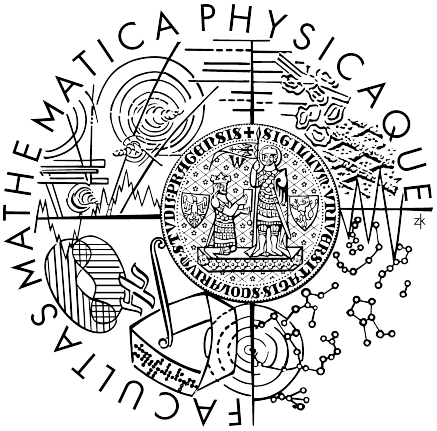
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Peter Truchlý

Experimentální analýza algoritmů pro hledání nejkratších cest

Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Alena Koubková, CSc.

Studijní program: Informatika, Softwarové systémy

Praha, 2009

Poďakovanie

Alene Koubkovej, vedúcej tejto práce, za poskytnuté materiály, poradné slovo a pomoc pri získavaní podkladov.

Pawan Harish a P. J. Narayanan, za poskytnutie CUDA implementácie pre otestovanie.

Všetkým, ktorí mi pomohli v priebehu štúdia a písania tejto práce.

Prehlasujem, že som svoju diplomovú prácu napísal samostatne a výhradne s použitím citovaných prameňov. Súhlasím s vypožičiavaním práce.

V Prahe dňa: 17.04.2009 Peter Truchlý

Obsah

Názov práce: Experimentální analýza algoritmů pro hledání nejkratších cest

Autor: Peter Truchlý

Katedra: Katedra softwarového inženýrství

Vedúci diplomovej práce: RNDr. Alena Koubková, CSc.

E-mail vedúceho: Alena.Koubkova@mff.cuni.cz

Abstrakt: Hľadanie najkratších ciest v grafe, je často riešenou úlohou programovania v mnohých podobách, zvyčajne ako súčasť riešenia iného problému. Vhodnosť algoritmu či implementácie, na riešenie konkrétnej skupiny problémov, nemusí byť na prvý pohľad zrejmá. V praxi preto môže nastať situácia, keď použitý algoritmus z hľadiska správnosti zodpovedá riešenej úlohe, avšak výkonovo o niekoľko rádov zaostáva. Cieľom diplomovej práce je poskytnutie aktuálneho, prakticky použiteľného prehľadu algoritmov, ktorý je doplnený o experimentálne zistenia a odporúčania vhodnosti pre jednotlivé typy úloh. Značná časť uvedených algoritmov bola otestovaná na spoločnej platforme, čím došlo k zjednoteniu a rozšíreniu predošlých výsledkov. Zahrnuté sú predovšetkým algoritmy triedy SSSP, implementovateľné na bežne dostupnom hardware, zmienené sú však aj algoritmy iných tried, napríklad OPSP a APSP. Špeciálna pozornosť je venovaná aktuálnemu trendu zvyšovania paralelizmu, či už vo forme viacjadrových CPU, alebo masívne paralelných výpočtov na platformách odvodených od GPU.

Kľúčové slová: najkratšie cesty, SSSP

Title: Experimental analysis of shortest paths algorithms

Author: Peter Truchlý

Department: Department of Software Engineering

Supervisor: RNDr. Alena Koubková, CSc.

Supervisor's e-mail address: Alena.Koubkova@mff.cuni.cz

Abstract: Shortest paths problem is one of the most encountered graph problems, which is commonly solved as subroutine in large variety of other, more complex tasks. If some algorithm or implementation fits the specific purpose, may, or may not, be completely obvious in practice. In some instances, theoretically correct solution behaves poorly in practice, lacking by more than order of magnitude after concurrent solution. Main goal of thesis, is to provide up to date overview of current algorithms, extended by experimentally obtained data and guidelines for their best usage. Majority of listed algorithms was tested on the same system, to provide wide and consistent comparison. Mainly, listed algorithms belongs to class SSSP, and are implementable to commodity hardware. Algorithms belonging to other classes, like OPSP or APSP are also mentioned. Special attention is dedicated to current growth of parallelism on hardware side, such as multi-core CPUs and massively parallel computing environments derived from GPU.

Keywords: shortest paths, SSSP

1 Úvod

High-performance code should ideally run so fast, that any further improvement in the code would be pointless.
Michael Abrash, [9]

Efektívne algoritmy a dátové štruktúry sú zvyčajne kľúčom k úspechu každého softvérového diela. Mnohé, ako napríklad matematické funkcie, triedenie, zoznamy, mapy, stromové štruktúry a iné, sú priamo súčasťou knižníc moderných platforiem pre tvorbu aplikácií. Spoločnou vlastnosťou je univerzálna použiteľnosť a často i dokázaná optimálnosť poskytovaných implementácií. Jedinou úlohou programátora je správne použitie a integrácia hotového riešenia, ktorá je vykúpená iba minimálnym, alebo žiadnym, výkonnostným dopadom.
Naneštastie, situácia v oblasti algoritmov pre hľadanie najkratších ciest je odlišná. Existuje niekoľko knižníc (napríklad [44]), ktoré implementujú tieto algoritmy a súvisiace dátové štruktúry efektívne. Konečná voľba riešenia však zostáva na pleciach programátora. Nesprávny výber algoritmu, použitej dátovej štruktúry, alebo implementácie, môže viesť k nepoužiteľnému výsledku. Hoci teoretické práce v tejto oblasti sú zaujímavé, v praxi nie je ojedinelým javom, keď horší algoritmus poskytuje nezanedbateľne lepší výkon, ako jeho teoreticky lepší konkurent.
Do rovnice, ktorá udáva výslednú efektivitu odvodenej implementácie, vstupuje v praxi viacero premenlivých faktorov. Pre spoľahlivé teoretické skúmanie by bolo nutné uvažovať rôznu cenu inštrukcií, vplyv cache, usporiadanie pamäťového subsystému, parametre a vlastnosti vstupných dát, predpokladaný model použitia a v neposlednej rade i paralelnosť cieľovej platformy. Logickým dôsledkom je využitie metód experimentálnej algoritmiky, ako prostriedku porovnania známych riešení na rôznych typoch problémov a prepojenie záverov experimentov s teoretickými znalosťami a výsledkami.

1.1 Ciele práce

Motiváciou tejto práce je poskytnúť prehľad riešení, doplnený o experimentálne získané dáta. Aktualizovať a zjednotiť výsledky rôznych prác z rôznych období. Sformulovať odporúčania, uľahčujúce rozhodnutie, keď je nutná implementácia niektorého z algoritmov.

1.2 Štruktúra práce

Kapitoly práce je možné rozdeliť na dve skupiny, praktické a teoretické. Delenie nie je striktne dodržané, v záujme lepšej prehľadnosti. Hlavný cieľ práce vyžaduje poskytnutie pohľadu na algoritmy z praktického hľadiska, niektoré experimentálne zistenia sú preto odkazované aj v teoretických kapitolách.

Teoretická časť začína kapitolou 2. Uvedené sú najprv definície problému a požadovaného riešenia v takej forme, aká je používaná v celej práci. Nasleduje rozdelenie známych algoritmov so stručnou charakteristikou. Kapitola 3 obsahuje popisy algoritmov vrátane uvedenia známych teoretických výsledkov časovej a pamäťovej zložitosti. Kapitola je rozdelená do častí zoradených podľa mien autorov, prípadne zaužívaného názvu popisovaného algoritmu. Detailný popis algoritmu je doplnený pseudokódom a popisom významných implementácií. Prechod do praktickej časti je tvorený kapitolou 4, ktorá najprv pojednáva všeobecne o aktuálne rozšírených platformách a výhľadom do blízkej budúcnosti. Pokračuje popisom problematiky presného merania času na PC, metodiky testov a nakoniec popisuje testovaciu platformu.

Praktická časť (kapitola 5) začína popisom generátorov, statických vstupov a skúmaných tried problémov. Ďalšia kapitola (6) patrí popisu vykonaných testov a výsledkov. Uvedené sú závery zistené v jednotlivých testoch, ale i celých skupinách testov. Zdôraznené (a podporené doplňujúcimi testami) sú rôzne pozorovania. Kapitola 7 obsahuje súhrnný pohľad na všetky experimenty, uvádza odporúčania a ponúka témy pre ďalšiu možnú prácu v tejto oblasti. Výsledky, vo forme tabuliek, aj nanesené do grafov, sú obsahom prílohy A. Príloha B je popisom obsahu priloženého DVD. Príloha C popisuje zdroje, ktoré môžu pomôcť pri praktických pokusoch s priloženými dátami, alebo pri realizácií podobnej štúdie. Po prílohe C nasleduje zoznam použitej literatúry.

2 Problém hľadania najkratších ciest

2.1 Základné definície a formulácia

2.1.1 Orientovaný graf

Graf $G = (V, E)$ pozostáva z množiny vrcholov V a hrán E. Veľkosti množín V a E sú vždy väčšie ako 0 a označované ako $n = |V|$ a $m = |E|$. V celej práci sa implicitne považuje G za orientovaný, čo znamená, že hrany sú tvorené usporiadanými dvojicami $e=(u, v)$ $u \in V \wedge v \in V$, u sa označuje za počiatočný a v za koncový vrchol hrany e. V prípade, že sa na vrcholy grafu odkazuje číslami, ide o očíslovanie vrcholov prirodzenými číslami z rozsahu 1 až n.

2.1.2 Ohodnotenie hrán

Ohodnotenie hrán je zobrazenie $l:E \rightarrow M$, nazývané tiež cena alebo dĺžka hrán. V závislosti od verzie algoritmu a jeho implementácie môže byť M množina celých čísiel, reálnych čísiel, alebo ich podmnožinou (najčastejšie sa jedná o nezáporné ohodnotenie hrán, alebo maximálnu prípustnú dĺžku hrany). Dĺžkou hrany e sa rozumie hodnota $l(e)$.

2.1.3 Cesta, cyklus, dĺžka

Cesta v grafe G je každá postupnosť vrcholov a hrán tvaru $v_0, e_1, v_1, \ldots, e_k, v_k$ pre $0 \leq k \leq n$, pre ktorú platí, že v_0 až v_k sú navzájom rôzne a súčasne $e_i = (v_{i-1}, v_i)$, pre $k \geq i \geq 1$, $e_i \in E$. .
Dĺžka cesty je súčtom dĺžok hrán patriacich ceste.
Cyklus, na rozdiel od definície cesty, povoľuje a zároveň požaduje rovnosť $v_0 = v_k$.
Dĺžka cyklu je súčtom dĺžok hrán patriacich cyklu.

2.1.4 Problém

Zadanie, taktiež označované ako **problém**, alebo **sieť**, je trojica (G, l, s), G je graf, l je ohodnotenie hrán a $s, s \in V$, **počiatočný vrchol** (zdroj).
Vrchol $u, u \in V$ je **dosiahnuteľný**, ak existuje cesta z s do vrcholu u. Pre všetky riešené problémy sa ďalej predpokladá, že neobsahujú cyklus zápornej dĺžky a že všetky vrcholy sú dosiahnuteľné. Rovnako je možné predpokladať, že vstupný problém neobsahuje slučky (hrany tvaru (v, v), $v \in V$) a násobné hrany, nakoľko tieto prípadné vlastnosti vstupu je možné odstrániť bez dopadu na výsledné riešenie (z násobných hrán stačí ponechať hranu s najmenším ohodnotením). Ak sa navyše v grafe nevyskytuje žiadny cyklus, problém sa označuje ako **acyklický**.

2.1.5 Riešenie

Riešením problému sa rozumie nájdenie zobrazenia $d:V \rightarrow M$, ktoré udáva dĺžku najkratšej cesty začínajúcej v s pre každý vrchol. Hodnota $d(s)$ je vždy 0. Súčasným požiadavkom riešenia, môže byť i nájdenie zobrazenia $pred:\{V\text{-}s\} \rightarrow V$, ktoré pre každý vrchol udáva predchodcu na najkratšej ceste z s do tohto vrcholu. Výpočet *pred* súčasne s d však nie je nutný, nakoľko ťažisko problému spočíva práve vo výpočte d, dopočítanie predchodcov je následne možné na základe tejto funkcie v čase $O(m)$.

2.1.6 Hlavné triedy problémov

Takto určená trieda problémov a algoritmov sa označuje ako **SSSP** – Single Source Shortest Paths, čiže cesty so spoločným počiatkom. Ak je vstupný problém rozšírený o špecifikáciu koncového vrcholu a požadované riešenie zúžené na výpočet *d,* pre vrcholy ležiace na najkratšej ceste z s do tohto vrcholu, jedná sa o triedu problémov **OPSP** – One Pair Shortest Path(s), alebo najkratšia cesta pre jeden pár. Naopak, ak požadované riešenie má obsahovať dĺžku najkratšej cesty pre všetky dvojice vrcholov, ide o triedu **APSP** – All Pairs Shortest Paths, alebo najkratšie cesty pre všetky páry.

2.2 Prehľad algoritmov a implementácií

Vzhľadom k rozsahu problematiky a plynúceho množstva rôznych špecializovaných algoritmov, nemôže byť tento prehľad úplný. Obsahuje však všetky známe sekvenčné algoritmy skupiny SSSP, mnohé významné algoritmy ostatných skupín a taktiež množstvo ďalších, menej známych.

Zoznam obsahuje abecedne zoradené algoritmy a významné modifikácie. Pri každom algoritme je stručný, charakterizujúci popis a referencie na literatúru. Podčiarknutím sú vyznačené položky, o ktorých podrobnejšie pojednáva tretia kapitola. Ak nie je uvedené inak, jedná sa o algoritmus riešiaci problém SSSP.

- **A* (A-Star)** Je heuristicky zlepšený Dijkstrov algoritmus pre riešenie OPSP v grafoch, kde je známy relevantný optimistický odhad (nesmie presiahnuť skutočnú dĺžku) najkratšej cesty z každého skúmaného vrcholu do cieľa. [22]
- **Acyclic (ACC)** Rieši acyklické siete v lineárnom čase, popísaný napríklad v [10]. Je založený na topologickom triedení vrcholov podľa orientácií hrán.
- **B* (B-Star)** Rozširuje myšlienku A-Star algoritmu (rovnako sa jedná o OPSP) a každému skúmanému uzlu priraďuje okrem optimistického i pesimistický odhad (nesmie byť nikdy menší než skutočná dĺžka) najkratšej cesty do cieľa. [23]
- **Bellman-Ford-Moore (BF)** Prehľadávanie grafu do hĺbky pomocou fronty (FIFO), posledné dosiahnuté vrcholy sa presúvajú vždy na koniec a prípadné predošlé vloženie je odstránené.
- **Bellman-Ford-Moore s kontrolou predka (BFP)** Rozširuje BF o heuristiku, v ktorej sa vrchol nepridá do fronty, ak je v nej zaradený predok vrcholu.
- **D***, tiež **Stentz (D-Star)** Je špecializovaný algoritmus pre OPSP, odvodený od A-Star, určený najmä pre využitie v robotike. Prehľadávanie prebieha v opačnom smere, od cieľa k počiatku a algoritmus dokáže dynamicky prehodnocovať najkratšiu cestu na základe zmien vstupného problému. [26, 30] Pre štvorcové siete s adaptívnym delením v [31].
- **Delta-Stepping (Δ-Stepping)** Je teoreticky dobre paralelizovateľný algoritmus, ktorý

preberá princípy z DIKB, pri výpočte rozdeľuje hrany podľa Δ na „ľahké“ a „ťažké“, pričom ľahké hrany sú preskúmané prednostne. [16]

- **Dijkstrov algoritmus (DIKQ)** Prehľadáva vždy dosiahnutý vrchol s minimálnou vzdialenosťou od zdroja, v prípade nezáporných hrán tak skenuje každý vrchol najviac raz. Základný variant používa pre hľadanie minima lineárny priechod zoznamu dosiahnutých vrcholov.
- **Dijkstrov algoritmus s priehradkami (DIKB)** Uchováva dosiahnuté vrcholy v priehradkách, podľa ich vzdialenosti od zdroja. Vyžaduje navýšenie pamäte úmerné maximálnej dĺžke hrany a zavádza obmedzenie na celočíselnosť ohodnotenia.
- **Dijkstrov algoritmus s priehradkami a 'vrecom' (DIKBM)** Pracuje ako DIKB, ale zmenšuje pamäťové nároky zmenšením počtu priehradiek, vrcholy mimo rozsah sú odložené bokom, kým nedôjde k posunu rozsahu spracovaných vzdialeností.
- **Dijkstrov algoritmus s 'približnými' priehradkami (DIKBA)** Znižuje pamäťové nároky DIKB pomocou zväčšenia rozsahu pre každú priehradku. Priehradka tak, na rozdiel od DIKB, nie je charakterizovaná iba jedinou hodnotou, ale rozsahom.
- **Dijkstrov algoritmus s dvojúrovňovými priehradkami (DIKBD)** Je vylepšením DIKBA. Obsahuje dve sady(úrovne) priehradiek, pričom prvá je zhodná s priehradkami DIKBA a druhá, iba pre rozsah jednej priehradky z prvej sady, je zhodná s priehradkami DIKB. Vrcholy sú pri priechode algoritmu postupne napĺňané z priehradiek prvej úrovne do priehradiek nižšej úrovne.
- **Dijkstrov algoritmus s Fibonacciho haldou (DIKF)** Uchováva Dosiahnuté vrcholy vo Fibonacciho halde.
- **Dijkstrov algoritmus s k-regulárnou haldou (DIKH)** Využíva haldu, kde každý prvok má najviac k potomkov. (Hodnota k je minimálne 2, zvyčajne je vyššia.)
- **Dijkstrov algoritmus – ďalšie modifikácie** Vzhľadom k modularite Dijkstrovho algoritmu, existuje celý rad ďalších implementácií, používajúcich ešte zložitejšie dátové štruktúry pre výber vrcholu s minimálnou dosiahnutou vzdialenosťou. Implementácia s R-haldou používa kombináciu priehradiek a haldy [11], implementácie s viacúrovňovými priehradkami [50] zovšeobecňujú DIKBD na ľubovolný počet úrovní priehradiek. Použitie kombinovaných, adaptívnych štruktúr, založených na haldách, priehradkách a prípadne zoznamoch vedie k implementáciám označovaným ako „smart queues“, čiže inteligentné (prioritné) fronty [48,49,51,52].
- **Dvojsmerné heuristické prehľadávanie** Je kombinácia heuristického prístupu a dvojsmerného prehľadávania grafu pre riešenie OPSP. Algoritmus využíva heuristiku pre orezanie prehľadávania a súčasne postupuje (striedaním) v prehľadávaní od zdroja i cieľa. Popis tohto prístupu je možné nájsť v [24] a [25].
- **Floyd–Warshallov algoritmus (WFI)** Rieši problém APSP pomocou postupného zlepšovania dĺžok najkratších ciest v matici rozmerov $n \times n$. [28]
- **Gausovskou eliminációu riešený APSP na GPU (GEAPSP)** Využíva efektívne hrubý výpočetný výkon, ktorý poskytuje GPU. Algoritmus pracuje na mieste a nepotrebuje počas výpočtu premiestňovať dáta z pamäte GPU do operačnej pamäte, čo prispieva k jeho efektivite. [33]
- **Geometricky zrýchlený Dijkstrov algoritmus** Využíva predvýpočty na urýchlenie opakovaných dotazov OPSP. Pre vstup sa predpokladá dostupnosť vopred daného

geometrického rozloženia vrcholov. Algoritmus a experimentálne výsledky popisujú práce [20, 32].

- **Goldberg-Radzik (GOR)** Rozširuje heuristiku BFP až na hľadanie topologického zotriedenia dosiahnutej časti grafu, následne prehľadáva vrcholy v nájdenom poradí.
- **Harish-Narayanan (CUDASSSP)** Je paralelný algoritmus, prispôsobený implementácií na platformu CUDA. [15]
- **Harish-Narayanan APSP** Využíva opakovaný výpočet CUDASSSP pre riešenie problému APSP.
- **Johnsonov algoritmus** Rieši problém APSP, na riedkych grafoch dosahuje lepší časový odhad ako WFI. [29]
- **Open Shortest Path First (OSPF)** Je sieťový smerovací protokol, ktorý obsahuje zabudovaný algoritmus pre hľadanie najkratšej cesty. Tento algoritmus pracuje výlučne na základe údajov o smerovaní obsiahnutých v paketoch a dokáže sa vyrovnať so zmenami topológie a výpadkami hrán. Algoritmus je založený na Dijkstrovom algoritme. [27]
- **Pallottinov algoritmus (TWO_Q)** Nadväzuje na algoritmus PAPE a zlepšuje časový odhad pre najhorší prípad. [13]
- **Pape-Levit (PAPE)** Patrí do skupiny „graph growth“, algoritmus udržiava (monotónnu) množinu vrcholov na ktorej pracuje. [12,13]
- **Parallel Degree SSSP (PDSSSP)** Je paralelný algoritmus využívajúci špeciálnu dátovú štruktúru, popísaný v [17]. Algoritmus je popisovaný na modeli PRAM, ktorý je iba teoretický (s výnimkou prototypu [18]).
- **Parallel Dijkstra (PDSPA)** Rozdeľuje Dijkstrov algoritmus na fázy, ktoré je možné vykonať paralelne. Tento algoritmus existuje pre teoretický model PRAM a je efektívny iba na istých množinách grafov (primárne náhodných). [21]
- **Rozdeľovanie grafu urýchľujúce Dijkstrov algoritmus** Využíva rôzne metódy rozdelenia vstupného grafu pre opakované riešenie OPSP nad tou istou sieťou. Rozdelenie umožní výpočet pre konkrétny dotaz obmedziť na podmnožinu vrcholov pôvodného grafu. Popis a experimentálne výsledky rôznych metód rozdelenia popisuje práca [19].
- **Threshold algoritmus (THRESH)** Je kombináciou prístupov BF, DIKQ a TWO_Q. Algoritmus počas výpočtu prerozdeľuje vrcholy podľa parametru t (threshold). [14]

3 Algoritmy

Riešením problému SSSP je nájdenie zobrazenia *d*, tak ako je popísané v 2.1, takéto zobrazenie spĺňa:

(P1) $$\forall e \in E, e=(v,u): d(v)+l(e) \geq d(u)$$

Algoritmy pre hľadanie najkratších ciest sú zvyčajne založené na postupnom zlepšovaní horného odhadu pre *d*, označené ako *td*. Na začiatku je *td*(*s*)=0 a pre ostatné vrcholy $v \in V, v \neq s: td(v) = \infty$. Výpočet spočíva v zlepšovaní odhadu podľa hrán, ktoré nespĺňajú P1.

Pri zlepšovaní *td*(*u*) podľa hrany *e*=(*v*, *u*), je tiež obvyklé uchovávanie referencie na *v*. Zobrazenie *pred*(*u*) je definované pre tie vrcholy, ktorých *td*(*u*) je konečné, *pred*(*s*) = *s*.

Veľká časť algoritmov využíva operáciu nazývanú *scan*(*v*), $v \in V$, ktorá je realizovaná ako priechod zoznamom výstupných hrán vrcholu (hrany tvaru *e*=(*v*, *u*), $u \in V$) a v prípade, že je *td*(*v*) + *l*(*e*) < *td*(*u*), nastaví sa *td*(*u*) = *td*(*v*) + *l*(*e*). Zvyčajne je zároveň vrchol u nejakým spôsobom označený pre ďalšie spracovanie.

A3.0 Operácia *scan*(*v*)

```
1: for e z E, e=(v, u) do
2:     if td(v) + l(e) < td(u) then
3:         td(u) := td(v) + l(e)     // zlepšenie odhadu
4:         pred(u) := v              // nastavenie predchodcu
5:     fi
6: od
```

3.1 Acc

Algoritmus Acc je vhodný na hľadanie najkratších ciest v acyklickom orientovanom grafe (DAG). Práca algoritmu pozostáva z dvoch činností: 1. nájdenie topologického usporiadania vstupného grafu počnúc od zdroja, 2. vykonanie operácie *scan* pre každý vrchol podľa poradia topologického usporiadania.

Obe operácie je možné vykonávať buď súčasne, vykonaním *scan* po zaradení vrcholu do zoznamu usporiadaných vrcholov, alebo za sebou, keď je vykonaný jeden priechod Bellman-Fordovho algoritmu na výsledný zoznam. Do vrcholov ktoré neboli zaradené pri činnosti 1 do

zoznamu, neexistuje cesta zo zdroja. Vždy tam musia patriť vrcholy, z ktorých existuje cesta do zdroja, inak by graf nebol acyklický. Algoritmus ACC preto predpokladá, že do zdroja nesmerujú žiadne príchodzie hrany.

A3.1 Algoritmus ACC

```
 1: Vytvor prázdny zoznam L // zoznam pre topologické usporiadanie
 2: Vytvor prázdnu množinu S; S := {s} // v množine je iba zdroj
 3: while S neprázdna do
 4:    vyber u z S
 5:    vlož u na koniec L
 6:    for v kde existuje hrana e=(u, v) do
 7:       odober hranu e z grafu
 8:       if v nemá žiadne príchodzie hrany then
 9:          vlož v do S fi
10:    od
11: od
12: while zoznam L neprázdny do
13:    vyber v z L
14:    scan(v)
15: od
```

Algoritmus pracuje v čase O(n+m) a vyžaduje pamäť O(n+m).

Ak sú hodnoty funkcie ohodnotenia hrán vynásobené -1, algoritmus hľadá najdlhšie cesty v pôvodnej sieti, aj takúto úlohu rieši v lineárnom čase. Vďaka podmienke acyklickosti grafu, je taktiež vylúčená existencia záporného cyklu.

Implementácia testovaná v tejto práci (ACC) vykonáva zotriedenie vrcholov a *scan* v dvoch fázach, najprv topologické triedenie a následne jeden priechod cez vrcholy, tak ako uvádza pseudokód A3.1.

3.2 Bellman – Ford – Moore

Algoritmus pre SSSP podľa autorov Richard Bellman a Lester Ford, Jr.. Pôvodná verzia algoritmu používa n priechodov cez m hrán, pričom sú vykonané úpravy hodnôt *td* a *pred* u tých vrcholov, kde dochádza ku zlepšeniu. Táto verzia je príliš neefektívna z časového hľadiska v priemernom prípade, ktorý je rovný dolnému i hornému odhadu O(n·m). Algoritmus nekladie žiadne obmedzenia pre dĺžky hrán, ktoré môžu byť aj záporné a desatinné čísla. Jednoduché zlepšenie je možné v podobe zastavenia vonkajšieho cyklu, ktorý sa opakuje n krát vtedy, ak vo vnútornom cykle pre hrany nedôjde k žiadnemu zlepšeniu *td*. Rovnako je ale zbytočné, aj skúmanie všetkých hrán v každom priechode. V tejto práci je popísaný zlepšený variant, používajúci frontu (FIFO pamäť) pre vrcholy. Horný odhad časovej zložitosti zostane nezmenený, ale v praxi je toto riešenie podstatne efektívnejšie. Pamäťová zložitosť je v oboch prípadoch O(n+m). Dolný odhad pre časovú náročnosť tak klesne na O(n+m), tento prípad nastáva napríklad vtedy, ak je vstupný graf strom.
Výpočet začína pridaním zdroja s do fronty. Algoritmus opakovane vyberá vrcholy z fronty až dokým nezostane prázdna. Pre každý vybraný vrchol je vykonaná operácia *scan*, vrcholy pre ktoré dôjde ku zlepšeniu *td* sú pridané na koniec fronty, ale iba v prípade že ešte nie sú vo fronte obsiahnuté. Pridanie a odobratie vrcholu z fronty je realizovateľné v konštantnom čase. Kontrola prítomnosti vrcholu vo fronte je možná taktiež v konštantnom čase, napríklad

pomocou bitového príznaku pre každý vrchol.

A3.2 Algoritmus Bellman-Ford-Moore

```
1: Vytvor prázdnu frontu Q; Q:={s} // do fronty je vložený zdroj
2: while Q neprázdna do
3:     vyber v z Q
4:     scan(v, Q)
5: od
```

Pseudo kód A3.2 využíva metódu *scan* ktorá je rozšírená o pridávanie vrcholov do fronty v prípade, ak došlo ku zlepšeniu *td*, upravený kód obsahuje výpis A3.3.

A3.3 Operácia *scan*(*v*, *Q*) pre vrchol a frontu

```
1: for e z E, e=(v, u) do
2:     if td(v) + l(e) < td(u) then
3:         td(u) := td(v) + l(e)
4:         pred(u) := v
5:         if Q neobsahuje u then
6:             u pridaj do Q
7:         fi
8:     fi
9: od
```

Ďalšie zlepšenie, ktoré je možné vykonať s konštantným navýšením času pri spracovaní vrcholu, je kontrola na predchodcu. Pri operácii *scan* je skontrolovaný predchodca (podľa *pred*) preberaného vrcholu na prítomnosť vo fronte, ak je predok vrcholu prítomný, *scan* sa nevykoná. Toto opatrenie šetrí prácu pri zlepšovaní *td*, pretože určite dôjde znovu k prehľadávaniu aktuálneho vrcholu po prehľadaní jeho predka.

V práci je základná (zefektívnená) implementácia označená ako BF, verzia využívajúca kontrolu predka ako BFP. Obe použité implementácie vyžadujú O(n+m) pamäte a O(n·m) času v najhoršom prípade. Algoritmus používajúci zásobník (LIFO) je zahrnutý pod názvom STACK, časová zložitosť je však exponenciálna, v najhoršom prípade O(n^n).

3.3 Dijkstra

Algoritmus pre hľadanie najkratších ciest, ktorý navrhol Edsger Wybe Dijkstra, je primárne určený pre grafy z nezáporným ohodnotením hrán. Niekedy je taktiež uvádzané, že tento algoritmus je možné použiť iba pre nezáporné dĺžky hrán, je však implementovateľný tak, aby správne pracoval aj so záporným ohodnotením hrán. V prípade vstupov so záporným ohodnotením sa však zhoršujú časové nároky algoritmu, nakoľko nie sú vylúčené opakované volania operácie *scan* pre ten istý vrchol. V najhoršom prípade sú časové nároky O(n·m).

Základnou ideou algoritmu je výber vrcholu pre spracovanie podľa najnižšej dosiahnutej vzdialenosti od zdroja. Toto kritérium je výhodné najviac v prípadoch, keď nemôže dôjsť k poklesu hodnoty *td* pre žiadny skenovaný vrchol, alebo ak ku poklesom nedochádza príliš často. Pre nezáporné ohodnotenia hrán je zaručené, že každý dosiahnuteľný vrchol z počiatku je spracovaný iba raz.

Obmedzenia pre dĺžky hrán konkrétnych implementácií neurčuje samotný algoritmus, ale použitá dátová štruktúra, pre uchovávanie dosiahnutých vrcholov. Priamočiara, naivná

implementácia, by totiž vyžadovala pre nájdenie vrcholu s minimálnou vzdialenosťou od počiatku, v najhoršom prípade až n operácií. To by zhoršilo časovú zložitosť algoritmu na $O(n^2)$. V praxi sa ako najlepšie preukázali implementácie využívajúce priehradky, čo v závislosti od konkrétnej verzie vyžaduje celočíselné ohodnotenia, alebo ohodnotenia s vopred známym maximálnym rozsahom a počtom hodnôt.

Vo všeobecnosti je možné popísať algoritmus pomocou štruktúry, ktorá podporuje výber prvku s minimálnou hodnotou *td*, vloženie prvku, zistenie prítomnosti prvku a vyrovná sa s úpravou *td* pre vložený prvok.

A3.4 Dijkstrov Algoritmus

```
1: Vytvor prázdnu štruktúru R; R:= {s} // prvý preberaný je počiatočný vrchol
2: while R neprázdna do
3:     vyber v z R                    // kde td(v) je najmenšia spomedzi prvkov R
4:     scan(v, R)
5: od
```

Použitá operácia *scan* je rovnaká, ako popisuje A3.3 v kapitole 3.2 – Bellman-Ford-Moore.

Pamäťové nároky algoritmu sú dané aj nárokmi použitej štruktúry R, zvyčajne sa jedná o $O(n+m)$, v prípade priehradiek je to $O(n+m+C)$, kde C je počet priehradiek.

Implementácia priamočiarej verzie s lineárnym hľadaním minima je označená ako DIKQ. Názov DIKH nesie implementácia používajúca k-regulárnu haldu s konštantným k (pre DIKH k=4, pre DIKHk, kde k=2,4,8 a 16 je to príslušná hodnota k). Časová zložitosť verzie s haldou je $O(m \cdot \log n)$. Implementácia pomocou Fibonacciho haldy, vedie k časovej zložitosti $O(m+n \cdot \log n)$, táto je ale v praxi horšia, pretože potrebné operácie na jej udržiavanie sú zvyčajne drahšie, než teoreticky získaný rozdiel (často je napríklad $m \sim 4n$, alebo $\log n \sim 30$).

Použitie priehradiek je závislé od obmedzenosti množiny hodnôt funkcie ohodnotenia hrán. Všetky skúmané problémy majú ohodnotenia celočíselné. Obmedzenie na maximálnu dĺžku hrany – C, je potrebné (a postačujúce) z toho dôvodu, aby sa do priehradiek zmestili všetky dosiahnuté vrcholy (nie z hľadiska veľkosti pamäte ale hodnoty *td*). Z princípu fungovania Dijkstrovho algoritmu – skúmanie vrcholu s najnižším *td*, nemôže byť pri nezáporných ohodnoteniach hrán potrebných viac, ako C+1, za sebou nasledujúcich priehradiek. Implementácia DIKB používajúca C priehradiek, má časové nároky $O(m+n \cdot C)$.

Pamäťové nároky DIKB znižujú implementácie DIKBM a DIKBA. DIKBM pomocou zmenšenia počtu priehradiek na B ($B < C$) a umiestňovaním vrcholov, ktoré sa nezmestia do aktuálneho mapovaného rozsahu, do náhradnej štruktúry. Takto je možné znížiť pamäťové nároky, na úkor zvýšenia časovej zložitosti, ktorá stúpne na $O(m+n((C/B)+B))$ pre kladné ohodnotenia hrán. Táto implementácia je tiež nazývaná implementáciou s priehradkami a vrecom. DIKBA využíva fakt, že pri základnej verzii je veľká časť z C priehradiek prázdna, zväčšením rozsahu pre každú priehradku, zvyčajne nepríde k ich prílišnému zaplneniu a výber minima z priehradky bude iba o málo pomalší ako pri DIKB, navyše je ušetrený istý čas pri preskakovaní prázdnych priehradiek. Ak je pre rozsah priehradky zvolená Δ (celé číslo), potom jedna priehradka nahradí Δ pôvodných priehradiek a celkový počet klesne na $C/\Delta+1$. Časová zložitosť je horšia ako pri DIKB, pre nezáporné ohodnotenia je $O(m \cdot \Delta + n \cdot (\Delta + C/\Delta))$.

Kombináciou prístupov DIKBA a DIKB, je možné odvodiť implementácie s viacstupňovými priehradkami. Pri tomto prístupe sú všetky úrovne priehradiek pre rozsah, okrem poslednej úrovne. Rozsah priehradiek nižšej úrovne vždy pokrýva rozsah jedinej priehradky úrovne vyššej. Pri vyprázdnení najnižšej priehradky sa zmení pracovný rozsah a je znovu naplnená

vrcholmi z niektorej priehradky vyššej úrovne. Testovaná je implementácia s dvojúrovňovými priehradkami – DIKBD. Vnorenie priehradiek je možné ľubovolne zvyšovať, v praxi sa však ukazujú ako rozumné maximálne 4 úrovne priehradiek. Časové nároky pri nezápornom ohodnotení hrán, pre implementáciu s dvomi úrovňami priehradiek, kde prvá úroveň má priehradky veľkosti Δ sú $O(m + n\cdot(\Delta+C/\Delta))$. Optimálna hodnota Δ pre prvú úroveň priehradiek je $\sqrt{C}$. Špecificky viac úrovňovými priehradkami v implementácii Dijkstrovho algoritmu sa zaoberajú práce [48, 49, 50]. Pre k úrovňové priehradky je časová zložitosť algoritmu $O(m+n(k+C^{1/k}))$ a vyžaduje pamäť $O(m+n+kC^{1/k})$.

Komplikovanejším využitím haldy a priehradiek, je možné implementovať ešte efektívnejšie dátové štruktúry, nazývané tiež HOT queues a smart queues. Tieto štruktúry sú podrobne popísané v [51] a [48,52]. Ich implementácia je technicky náročná a praktický prínos nemusí vždy zodpovedať vynaloženému úsiliu, experimentálne dáta ale potvrdzujú že tieto implementácie sú robustnejšie a poskytujú o niečo lepšie výsledky než samostatné haldy alebo viacúrovňové priehradky. Zlepšené sú tiež teoretické výsledky pre horné odhady časovej zložitosti, ktoré sú pre implementácie využívajúce HOT queues $O(m+n\cdot(\log C\log\log C)^{1/3})$ a pre smart queues $O(m+n\cdot\log C/\log\log C)$. V druhom prípade, navyše, ak sú dĺžky hrán rozložené rovnomerne v intervale $1..M$ (M je celé číslo), časová zložitosť algoritmu je s vysokou pravdepodobnosťou v priemernom prípade $O(n+m)$.

3.4 Goldberg – Radzik

V algoritme Bellman-Ford-Moore sú prehľadávané vrcholy v tom poradí, v akom sú pridávané do fronty. Heuristika v algoritme BFP pri spracovaní vrcholov, vynecháva *scan* pre tie vrcholy, ktoré majú predchodcu podľa *pred* zaradené do fronty. Túto heuristiku by bolo možné rozšíriť na viacerých predchodcov spracovaného vrcholu, pretože vždy, keď je vrchol pridaný do fronty, došlo preň k zlepšeniu hodnoty *td* a následne dôjde nevyhnutne ku zmene *td* aj pre potomkov (nieje vylúčená ani zmena *pred*), potomkovia takéhoto vrchola teda nie sú spracovaní, pretože ešte budú určite zaradení do fronty po spracovaní ich predka.
Pomocou funkcie *td*, ktorá udáva najlepšiu dosiahnutú vzdialenosť od počiatku pre každý vrchol, je možné definovať redukovanú dĺžku hrany $l_{td}(u, v) = l(u, v) + td(u) – td(v)$. Je zrejmé, že pre hrany, ktoré môžu zlepšiť dosiahnutú vzdialenosť pre vrchol v, je hodnota redukovanej dĺžky hrany záporná. Redukovaná dĺžka je definovaná iba pre tie hrany, ktorých počiatočný vrchol je dosiahnutý. Ak množina E_{td} obsahuje hrany, ktorých l_{td} je záporná, alebo rovná nule, potom $G_{td} = G(V, E_{td})$ je graf, ktorý obsahuje pôvodnú množinu vrcholov V a množina hrán je redukovaná na hrany s nulovou alebo zápornou redukovanou dĺžkou.
Algoritmus Goldberg-Radzik [53, 1], zahrnutý ako GOR a GOR1. Popis využíva bez ujmy na obecnosti predpoklad, že G_{td} je acyklický. To nastáva ak vstupný graf neobsahuje cykly nulovej alebo zápornej dĺžky. Rozšírením heuristiky BFP na maximálny možný počet predkov vrcholu, vzniká s využitím topologického triedenia preberaných vrcholov algoritmus, ktorého časová zložitosť je $O(n\cdot m)$ a pamäťová $O(n+m)$ pre ľubovolné ohodnotenia hrán. V prípade acyklického vstupu, je časová zložitosť algoritmu v najhoršom prípade $O(m)$.
Vrcholy sú pri spracovaní označené tromi stavmi: nedosiahnuté, dosiahnuté a prebrané. Algoritmus pracuje s dvomi množinami, A a B. na začiatku je množina A prázdna, B obsahuje s. Práca algoritmu je rozdelená do priechodov. Na začiatku každého priechodu, sa z množiny dosiahnutých vrcholov vytvorí utriedený zoznam (obsahuje prvky A) a následne sú vrcholy tohto zoznamu spracované pomocou *scan*, nové dosiahnuté vrcholy sú pridávané do B.

A3.5 Algoritmus Goldberg-Radzik

```
 1: Vytvor prázdny zoznam A
 2: Vytvor prázdnu množinu B; B:={s}
 3: Označ všetky vrcholy ako nedosiahnuté
 4: Označ s ako dosiahnutý
 5: while B neprázdna do
 6:    z B odober vrcholy, z ktorých nevedie hrana so zápornou l_td
 7:    A naplň vrcholmi dosiahnuteľnými z B v G_td
 8:    označ vrcholy v A za dosiahnuté
 9:    aplikuj topologické triedenie na A podľa E_td
10:    vyprázdni B
11:    while A neprázdna do
12:       vyber u z A
13:       scan(u, B)
14:    od
15: od
```

Dosiahnuteľnosť na riadku 7 sa skúma priechodom do hĺbky. Topologické triedenie je popísané pri algoritme Acyclic v A3.1, riadky 1 až 11. Priechod cyklu na riadkoch 11 až 14 prechádza vrcholy v poradí, kde pre každú hranu (v, u) z E_{td}, v je v A pred u. Každý odobraný vrchol na riadku 6 je označený ako prebraný (nie je vylúčené opakované preberanie).
Predpoklad, že graf G_{td} neobsahuje cyklus nulovej, alebo zápornej dĺžky, je možné odstrániť na základe dvoch pravidiel. Pre cyklus zápornej dĺžky je možné ukončenie výpočtu. Pre cyklus nulovej dĺžky je možné vykonať kontrakciu tohto cyklu a pokračovať vo výpočte, ako uvádza [53, strana 4].

A3.6 Operácia *scan*(v, B)

```
1: for e z E, e=(v, u) do
2:    if td(v) + l(e) < td(u) then
3:       td(u) := td(v) + l(e)
4:       pred(u) := v
5:       v označ ako prebraný
6:       u označ ako dosiahnutý
7:       u pridaj do B
8:    fi
9: od
```

Variant algoritmu GOR1, uvedený v [1], využíva pri priechode do hĺbky (riadok 9 pôvodného algoritmu) zároveň operáciu zhodnú s vnútrom operácie *scan* (riadky 2 až 8). Tento postup v praxi zrýchľuje algoritmus, pretože zväčšuje dosiahnutú časť grafu bez zhoršenia celkovej časovej zložitosti algoritmu, vo výsledku je tak potrebných menej, alebo toľko isto priechodov vonkajšieho cyklu, ako v pôvodnom algoritme.

3.5 Harish – Narayanan

Algoritmus publikovaný v [15], ktorého autormi sú Pawan Harish a P. J. Narayanan, neobsahuje nový prístup riešenia SSSP, ale upravuje sekvenčný Bellman-Ford-Moore pre použitie na GPU. Kým Dijkstrov algoritmus výberom prvku s minimálnou hodnotou *td*

neposkytuje veľký priestor paralelizmu, BF nieje efektívny vzhľadom k celkovej časovej náročnosti. Teoreticky efektívnejší Δ-Stepping, je podstatne zložitejšie implementovateľný. Vzhľadom ku krátkej existencii masovo dostupného, štandardizovaného GPGPU výpočetného prostredia, je algoritmus Harish-Narayanan aktuálne jediným, ku ktorému existuje použiteľná implementácia pre GPU. Použiteľnosť implementácie je chápaná, ako schopnosť vyriešiť aspoň jednu triedu úloh nezanedbateľne rýchlejšie, než podobne nákladné CPU. Testovaná implementácia má názov CUDASSSP.

***Poznámka**: Existuje efektívna implementácia pre GPU a problém APSP [33], ktorá poskytuje urýchlenie až v rozsahu 10^2 násobku, oproti efektívnym sekvenčným algoritmom pre CPU.*

Algoritmus, na rozdiel od BF, nepoužíva žiadnu štruktúru pre zoznam dosiahnutých vrcholov. Vrcholy označené na prebratie, majú nastavený bitový príznak v poli, ktoré je indexované číslom vrcholu. Výpočet algoritmu prebieha v krokoch. V každom kroku je najprv vykonaná operácia *scan* (KERNEL1), pre každý vrchol s nastaveným príznakom. Všetky vrcholy sú spracované paralelne vlastným vláknom. Pre nájdené hrany, zlepšujúce hodnotu *td* pre cieľový vrchol, nie sú hodnoty okamžite aktualizované, ale iba zapísané do poľa nových hodnôt. Keďže každá hrana má práve 1 počiatočný vrchol a operácia *scan* prehľadáva iba výstupné hrany vrcholu, nedochádza pri paralelnom spustení *scan* ku splneniu *„race conditions"*. V prípade zápisu nových hodnôt však nie je vylúčený konkurenčný zápis, preto sa využíva operácia atomického minima, ktorá nastaví hodnotu prvého argumentu na menšiu z dvoch zadaných hodnôt atomicky. V druhej časti fázy dochádza k aktualizácií *td* podľa zlepšujúcich hodnôt. Na konci fázy má *td* pre každý vrchol správnu hodnotu, nie je ale súčasne možné nastavovať hodnotu *pred*. Predchodcovia vrcholu sa z tohto dôvodu nepočítajú na GPU, je však možné ich dopočítať v čase O(*m*) po skončení výpočtu.

A3.7 Algoritmus Harish–Narayanan

```
1: Vytvorenie polí Va, Ea, Wa, Na a načítanie vstupu
2: Ma:= false, Ma[s]:=true
3: Ua = Ca = td
4: while Ma aspoň pre 1 index = true do
5:    <pre všetky vrcholy> KERNEL1(Va;Ea;Wa;Ma;Ca;Ua)
6:    <pre všetky vrcholy> KERNEL2(Va;Ea;Wa;Ma;Ca;Ua)
7: od
```

Implementácia využíva pole V_a, v ktorom sú uložené stupne vrcholov (podľa počtu vychádzajúcich hrán), E_a - indexy do poľa hrán a C_a – hodnota *td* pre vrchol. Ďalej bitové pole M_a, pole dĺžok hrán W_a a zoznam nasledovníkov vrcholu N_a. Veľkosti polí V_a, C_a, E_a a M_a sú n, veľkosť W_a je m. V každej iterácii je spustených n vlákien, pričom každé vlákno dostane pridelené číslo svojho vrcholu.

A3.8 Operácia *scan*: KERNEL1 – *n* vlákien, každé dostane číslo vrcholu *i*

```
1: if Ma[i] then                    // i je pridelené číslo vrcholu
2:    Ma[i]:=false
3:    for index:=1 to Va[i] do
4: |     if Ua[Na[Ea[i]+index]] > Ca[i] + Wa[Ea[i]+index] then
5: |        Ua[Na[Ea[i]+index]] := Ca[i] + Wa[Ea[i]+index]
6: |     fi                          // riadky 4, 5, 6 prebehnú atomicky vzhľadom k prvku Ua
7:    od
```

Ak je hodnota M_a nastavená na true, výpočet pokračuje operáciou *scan*, inak vlákno skončí. Operácia *scan* využíva stupeň vrcholu z V_a, počiatočný index do zoznamu nasledovníkov E_a a prechádza pomocou polí W_a a N_a vrcholy, kam vedie hrana, ako pri bežnej operácii *scan* v ostatných algoritmoch. Aktualizácie C_a sa nevykonávajú priamo, ale do poľa U_a, pomocou operácie atomického minima. Po dobehnutí všetkých vlákien, je spustených n nových, ktoré porovnajú či $U_a < C_a$ pre každý vrchol a nastavia príznaky do poľa M_a pre tie vrcholy, kde bola hodnota $U_a < C_a$. Súčasne nastavujú aj príznak udávajúci, či došlo aspoň v jednom prípade ku zlepšeniu do spoločnej premennej.

A3.9 KERNEL2 – n vlákien, každé dostane číslo vrcholu i

```
1: if Ca[i] > Ua[i] then
2:     Ca[i]:=Ua[i]
3:     Ma[i]:=true
4: fi
5: Ua[i]:=Ca[i]
```

Časová zložitosť tohto algoritmu v najhoršom prípade je $O(n \cdot m)$ (z hľadiska celkovej práce), čas potrebný pre vyriešenie problému závisí od maximálneho počtu hrán na najkratšej ceste a najvyššieho stupňa vrcholu. V najhoršom prípade výpočet trvá $O(n \cdot m)$. Pamäťové nároky algoritmu sú $O(n+m)$.

3.6 Pallottino, Pape – Levit

Inkrementálne algoritmy, pre hľadanie najkratších ciest, zjednotil vo svojej práci Stefano Pallottino [13]. Princíp fungovania týchto algoritmov je založený na rozdelení vrcholov grafu do dvoch množín. Prvá množina, taktiež označovaná ako množina s vysokou prioritou, obsahuje vrcholy, ktoré už boli dosiahnuté z počiatku a najmenej raz prehľadané operáciou *scan*. Druhá množina obsahuje dosiahnuté vrcholy, ktoré ešte neboli spracované ani raz. Všetky algoritmy prehľadávajú najprv vrcholy prvej množiny a v prípade, že je prázdna, preskúmajú vrchol z druhej množiny.

Rozdiel, medzi jednotlivými implementáciami, je iba v štruktúrach, ktoré sú použité pre uchovávanie vrcholov oboch množín. Pamäťové nároky všetkých variánt sú $O(n+m)$. Algoritmy Pape a Levit používajú pre množinu s vysokou prioritou zásobník (LIFO) a pre druhú množinu frontu (FIFO), testovaná implementácia je označená názvom PAPE. Horný odhad časovej náročnosti PAPE je $O(n \cdot 2^n)$. Implementácia TWO_Q používa pre obe množiny frontu, časový odhad v najhoršom prípade je $O(n^2 \cdot m)$.

A3.10 Algoritmus TWO_Q / PAPE

```
1: Vytvor prázdnu frontu (zásobník pre PAPE) Q1
2: Vytvor prázdnu frontu Q2; vlož s do Q2
3: while Q1 neprázdna or Q2 neprázdna do
4:     if Q1 neprázdna then
5:         vyber v z Q1 else
6:         vyber v z Q2
7:     fi
8:     scan(v, Q1, Q2)
9: od
```

Z hľadiska implementácie je výhodné, označovať vrcholy na prítomnosť vo fronte, podobne, ako pri algoritme Bellman-Ford-Moore. Vrcholy sú označené ako „zaradené vo fronte“, a „nezaradené“, zároveň však aj ako „najmenej raz prehľadané“ a „neprehľadané“. Pre vrchol vo fronte potom platí, že ak bol najmenej raz prehľadaný, musí sa nachádzať v Q_1, v opačnom prípade v Q_2. Pre dosiahnuté vrcholy (počas scan), ktoré nie sú zaradené vo fronte, je pre neprehľadané vrcholy použitá fronta Q_2 a pre najmenej raz prehľadané Q_1.

A3.11 Operácia *scan*(v, Q_1, Q_2)

```
 1: for e z E, e=(v, u) do
 2:    if td(v) + l(e) < td(u) then
 3:       td(u) := td(v) + l(e)
 4:       pred(u) := v
 5:       if u je „nezaradený“ then
 6:          if u je „neprehľadaný“ then
 7:             vlož u do Q2
 8:          else
 9:             vlož u do Q1
10:          fi
11:       fi
12:    fi
13: od
```

Algoritmus A3.11 Popisuje operáciu *scan* použitú v TWO_Q a PAPE.

3.7 Threshold a Δ-Stepping

Medzi publikáciou algoritmu Threshold [14] a Δ-Stepping [16] ubehlo 14 rokov. Napriek tomu sú tieto dva algoritmy veľmi blízke svojou základnou myšlienkou. Dá sa na ne pozerať ako na zvoľnenie Dijkstrovho algoritmu, alebo na sprísnenie Bellman-Fordovho algoritmu a to v podmienke, podľa ktorej sú volené vrcholy pre spracovanie.

Algoritmus Threshold, zaradený v práci pod názvom THRESH, rozdeľuje výpočet na fázy. V každej fáze, je na začiatku vybraná podmnožina vrcholov, ktorých hodnota *td* je nižšia ako zvolené *t*. Voľba *t* je laditeľný parameter, ktorý sa vyberá medzi minimálnou hodnotou a váženým priemerom hodnoty *td* všetkých dosiahnutých vrcholov. Nasleduje spracovanie vybraných vrcholov operáciou *scan*. Výpočet končí, ak už nezostanú žiadne dosiahnuté vrcholy na prebratie. Implementácia THRESH uchováva oba zoznamy (preberaných a dosiahnutých) vrcholov ako fronty. Pre nezáporné ohodnotenie hrán, je časová zložitosť tohto algoritmu v najhoršom prípade O($n \cdot m$).

V prípade algoritmu Δ-Stepping, sú dosiahnuté vrcholy uchovávané v priehradkách o rozsahu Δ. Skúmané sú následne vrcholy z najnižšej priehradky ktorá je neprázdna. Tento algoritmus je veľmi podobný DIKBA, nevyberá však vrcholy v poradí podľa ich minimálnej hodnoty *td*, ale v takom poradí, ako boli vložené. Tento prístup nemá nijakú zvláštnu výhodu oproti DIKBA, práve naopak, je menej efektívny v prípade, ak nastáva opakovaný *scan*, je ale lepšie paralelizovateľný. Najhoršia časová zložitosť algoritmu je rovnaká, ako pri BFP O($n \cdot m$). V špeciálnom prípade a pri implementácii na PRAM, ako uvádza [16], je možné dosiahnutie priemernej časovej zložitosti $O(\log^3 n / loglogn)$, na náhodných grafoch. Celková práca pritom zostáva lineárna O($n+m$). Pamäťové nároky oboch algoritmov sú O($n+m$). (u Δ-Stepping, iba ak platia podmienky kladené v [16], z ktorých plynie obmedzenie pre celkový počet priehradiek < m).

A3.12 Algoritmus Threshold

```
 1: Vytvor prázdnu frontu Q2; vlož s do Q2
 2: Vytvor prázdnu frontu Q1
 3: t := 0;
 4: while Q1 neprázdna or Q2 neprázdna do
 5:    naplň Q1 vrcholmi v z Q2 pre td(v) ≤ t
 6:    while Q1 neprázdna do
 7:       vyber v z Q1
 8:       scan(v, Q2)
 9:    od
10:    aktualizuj hodnotu t podľa Q2
11: od
```

Operácia *scan* je rovnaká, ako pri algoritme BFP (A3.3). Do fronty sú vkladané vrcholy najviac raz. Aktualizácia hodnoty t je možná i efektívnejšie, než priechodom prvkov fronty (postupným aktualizovaním), na výslednú časovú zložitosť algoritmu však tento postup nemá vplyv, pretože aj výber vrcholov do Q_1 sa realizuje priechodom Q_2.

4 Prostredie a metodika

4.1 Aktuálne výpočetné platformy

Technický pokrok, vo výrobe a vývoji procesorov, umožnil rozšírenie paralelných architektúr do všetkých segmentov trhu výpočtových zariadení. Použitie viacerých procesorov v jednom systéme nie je nové, v posledných rokoch sa však stalo štandardom i v osobných počítačoch a mobilných zariadeniach. Migráciou technológie sa mení význam zo zvládania záťaže produkovanej viacerými užívateľmi, na nový – podporený očakávaniami: zrýchlenie prevádzkovaných aplikácií jedného užívateľa.

Problém hľadania najkratších ciest, ktorý je predmetom tejto štúdie, je riešený na všetkých triedach zariadení, od prenosných GPS navigácií, telefónov, herných konzol, až po viac soketové servery. Na rozdiel od iných, nie je tento problém triviálne paralelizovateľný. Vysoká cena synchronizačných primitív a nízky počet výpočetných jednotiek (2 – 4) do veľkej miery znemožňujú adaptáciu súčasných algoritmov. Existujúce riešenia, ktoré ponúkajú dobrú škálovateľnosť, sú buď obmedzené na špeciálne architektúry [60], neposkytujú výhodu oproti najrýchlejšiemu sekvenčnému algoritmu [15] (pre väčšinu prípadov), alebo sú výhodné iba pre špeciálnu triedu úloh [55, 17].

Na druhej strane stoja aplikácie, ktoré kladú vysoké nároky na výpočetný výkon a sú zároveň dobre paralelizovateľné. Pre tieto v minulosti nepostačovali CPU a preto vznikol špecializovaný, vysoko paralelizovaný hardvér, akým je GPU/grafická karta, fyzikálny procesor a iné. Postupným vývojom sa zvýšila programovateľnosť natoľko, že umožňujú beh ľubovolných výpočtov, implementovateľných napríklad v jazyku C. Tieto zariadenia sú široko dostupné a objavujú sa produkty, ktoré kombinujú oba prístupy.

Dôkazom prebiehajúcej konvergencie CPU, GPU a mnoho procesorových počítačov, sú aktuálne, ale i ohlásené produkty ako Cell [61, 63], Tesla [62] a Larabee [64]. Zjednocujúcou silou sa možno stane OpenCL [65], alebo niektoré z komerčných riešení, ktoré sa pozerajú na všetky dostupné CPU, GPU a ostatné procesory v systéme, ako na množinu výpočetných zariadení a poskytujú jednotný prístup a prostredie pre ich využitie.

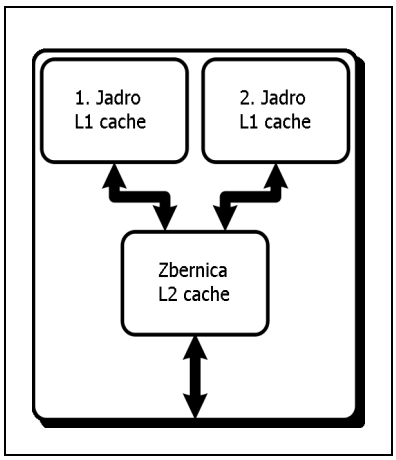
Z teoretického hľadiska, nie je korektné priame porovnávanie výsledného času, ktorý dosahuje sekvenčný algoritmus na CPU a vysoko paralelizovaný algoritmus na GPU, najmä v prípade, keď sa líšia celkovou vykonanou prácou. Z hľadiska koncovej ceny, počtu tranzistorov, výrobných nákladov či spotreby, sú však tieto riešenia porovnateľné, preto v praxi k tomuto porovnávaniu dochádza. Často je na takýchto platformách pri požadovaných

rozsahoch vstupu výrazne rýchlejší algoritmus, ktorý vyžaduje napríklad O($n \cdot m$) celkovej práce, než sekvenčný algoritmus pre CPU, s časovou zložitosťou O($m+n \cdot \log n$).

Výpočty v rámci tejto štúdie sú realizované na dvoch aktuálnych výpočetných platformách – viac jadrové CPU a programovateľná grafická karta s mnoho jadrovým GPU.

4.1.1 Dvoj jadrové CPU

Viac jadrové (2, 4 a v blízkej budúcnosti 6 a 8) CPU využívajú výhody pokročilého výrobného procesu, ktorý umožňuje integráciu viacerých plnohodnotných moderných CPU na plochu, ktorú pôvodne zaberalo jediné jadro. U aktuálnych dvoj jadrových a lepších štvor-jadrových procesorov je zdieľaná vyrovnávacia pamäť druhej úrovne (L2, prípadne L3), pričom každé z jadier má vlastnú, menšiu pamäť prvej úrovne (L1). Pre využitie plného výkonu jednou aplikáciou, je nutné vytvorenie najmenej 2 vlákien (resp. 4 a viac podľa počtu jadier).



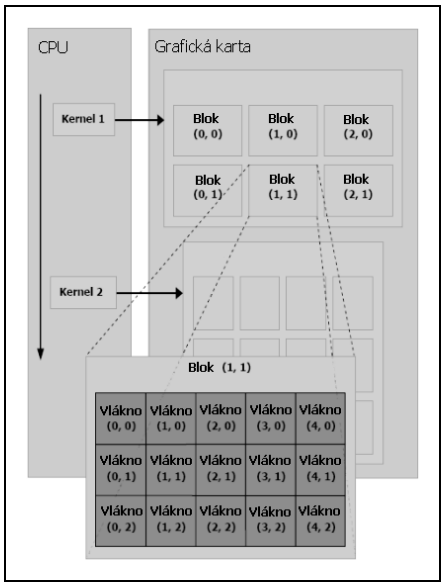
O4.1.1 Vysoko úrovňový diagram dvoj jadrového CPU. [57]

Programovací model je spätne kompatibilný s jednoduchým CPU. Jednovláknová aplikácia môže vždy bežať iba na jednom z jadier a teda nikdy nedosiahne maximálny možný výkon. Pri využití viacerých vlákien, sú tieto vykonávané súčasne, tam kde dochádza k úpravám spoločných štruktúr, je však nutné využitie synchronizačných primitív. Celková cena spojená s udržiavaním a synchronizáciou vlákien je pomerne vysoká, preto sú efektívne paralelizovateľné iba málo previazané úlohy.

4.1.2 Mnoho jadrový procesor - GPU

Súčasné GPU obsahujú rádovo stovky výpočetných jednotiek, často označovaných ako unifikované shadery. Tieto sú schopné vykonávania bežných celo číselných operácií, riadiacich inštrukcií a desatinných operácií s jednoduchou presnosťou (32bit), prípadne dvojitou presnosťou (64bit) u najnovších GPU. Pre tento model je teda možná kompilácia bežného kódu štrukturovaného jazyka. V prípade platformy CUDA [58, 59], ktorá je použitá pri experimentoch, je použitý jazyk C s rozšíreniami pre riadenie vlákien aplikácie. Programovanie pozostáva z vytvorenia „hostiteľského kódu“ a „kernelov“. Hostiteľský kód beží na CPU a vykonáva tie úlohy, ktoré nie sú paralelizované: vstup, výstup a sekvenčné časti algoritmu. Paralelizovaná časť implementácie je tvorená kernelom (prípadne viacerými), ktorý približne zodpovedá krátkej procedúre bežného algoritmu (napríklad operácia *scan*, použitá v popisoch kapitoly 3). Kernely sú vždy spúšťané na GPU, všetky dáta s ktorými pracujú, musia byť preto pripravené hostiteľským kódom v pamäti grafickej karty. Výsledky sú po skončení výpočtu kopírované späť. Spustenie kernelu je súčasne spojené s vytvorením veľkého počtu vlákien (pre vyťaženie G92 je nutných minimálne 128, v praxi však omnoho

viac), tieto sú na rozdiel od vlákien na CPU nízko nákladové, takže ich spustenie a udržiavanie nespôsobuje zásadný problému, sú naopak potrebné pre plné vyťaženie všetkých jednotiek.



O4.1.2 Schéma behu programu na platforme CUDA. Zdroj [58].

Mnohými vláknami súčasne spúšťaný kernel, reprezentuje základný princíp programovania GPU, kde je výhodné, aby každá výpočtová jednotka realizovala rovnakú operáciu. Organizácia do blokov, je rozšírením abstrakcie modelu PRAM, každý blok je možné adresovať viacerými súradnicami a vlákna v rámci bloku, je možné znovu adresovať. Vlákno tak nemusí obdržať iba pridelené jedno celé číslo, ale niekoľko čísiel, ktoré zodpovedajú súradniciam bloku a vlákna v rámci bloku. Synchronizačné primitíva buď nie sú podporované vôbec (GPU pred G92), alebo sú podporované v rámci poskytovaných operácií (atomické funkcie – NVIDIA [58]).

4.2 Meranie času v experimentoch

4.2.1 Požiadavky a problémy merania času

Experimentálna analýza algoritmov sa opiera o empiricky získané údaje. Pri skúmaní časovej náročnosti algoritmu je nevyhnutné samotné meranie času ktorý je spotrebovaný riešením úlohy. Meranie času je v súčasnosti samostatným problémom.

1. Merané časové intervaly a rozdiely medzi implementáciami sú malé, rádovo až na hranici nanosekúnd, teda 10^{-9} s. Táto skutočnosť plynie z aktuálnych pracovných frekvencií procesorov, ktoré sú v násobkoch GHz.
2. Bod v čase, od ktorého je potrebné merať čas a tak isto bod, v ktorom je potrebné meranie ukončiť, nie je vonkajším pozorovaním rozlíšiteľný. Pri skúmaní algoritmu je potrebné zmerať iba interval, ktorý je spotrebovaný riešením úlohy. Praktický test ale vyžaduje, aby implementácia nejakým spôsobom načítala, alebo vytvorila zadanie, taktiež je potrebné vytvorenie výstupu, za účelom kontroly a zozbierania relevantných informácií.
3. Existuje niekoľko možností merania času, pričom žiadna z nich nie je dostupná na každej platforme a každá z nich má svoje výhody a nevýhody.

Bod 1 vylučuje možnosť merania človekom, nakoľko reakčná doba je minimálne o 6 rádov väčšia než presnosť ktorá sa vyžaduje. Bod 2 v spojení s bodom 1 vylučuje aj všetky ostatné externé možnosti. Ak by bolo meranie času založené na predpoklade, že program signalizuje začiatok a ukončenie výpočtu dohodnutou výstupnou operáciou, muselo by existovať zariadenie, ktorého latencia je dostatočne nízka. Z praktického hľadiska sú ale všetky zariadenia ktoré by pripadali do úvahy (sieťový interface, zvukový výstup, obrazový výstup, mechaniky pre pamäťové médiá), konštruované tak, že sú zaťažené nekonštantným oneskorením. Oneskorenie na menovaných výstupoch je v rádoch ms. Čas je preto možné merať s dostatočnou presnosťou, iba za pomoci samotnej implementácie.

4.2.2 Meranie času za pomoci testovacieho stroja

Ako praktická a dostupná metóda merania je využitie inštrukcie, alebo API, určených na meranie času priamo v implementácií testovacieho programu. Principiálne je tento prístup zachytený kódom:

```
údaj1 = začiatok_merania();
meraná_úloha();
údaj2 = koniec_merania();
```

Technicky je táto metóda založená na počítaní cyklov obvodu, ktorý pracuje na stálej frekvencií. Najlepšiu presnosť môže v typickom PC poskytnúť CPU, stúpa tiež rozšírenosť špecializovaného hardware na meranie času a časovanie, ktorý býva súčasťou základnej dosky.

4.2.3 Inštrukcia RDTSC

Súčasné procesory obsahujú vstavané počítadlo cyklov, ktoré počíta presný počet vykonaných cyklov procesoru. Odčítanie hodnoty na tomto počítadle je možné s presnosťou na desiatky cyklov, čo je zapríčinené spôsobom akým sú spracovávané inštrukcie a akým je nastavovaná samotná hodnota počítadla.
Výsledný čas, ktorý zabralo vykonanie meranej úlohy je potom možné určiť výpočtom, v ktorom je najprv odčítané trvanie odčítania stavu počítadla (toto je v meranom úseku započítané presne 1x) a následne je hodnota vynásobená dĺžkou trvania jedného cyklu. Dĺžku trvania jedného odčítania hodnoty je možné určiť ako priemer dostatočne veľkého počtu odčítaní, pričom celkový počet cyklov sa spočíta samotnou metódou.
Získanie hodnoty ubehnutých cyklov procesoru bolo v minulosti problematické, ak nie nemožné. Počítače neobsahovali špecializovaný hardware pre túto úlohu. S rozšírením multimediálnych a interaktívnych aplikácií v posledných desaťročiach stúpli požiadavky na presné časovanie a meranie času u osobných počítačov, z vývojárskej komunity tiež prišiel požiadavok na možnosť presného profilovania častí programov.
S príchodom architektúry Pentium® od spoločnosti Intel® bola pridaná inštrukcia RDTSC [5]. Inštrukcia dokáže načítať 64 bitovú hodnotu (do dvoch 32 bitových registrov u CPU Pentium), ktorá je inkrementovaná každým pracovným cyklom procesora a pri spustení začína na 0. Táto inštrukcia nieje serializujúca, takže zistená hodnota môže predbehnúť niekoľko nasledujúcich inštrukcií. Vzhľadom ku konštantnej dĺžke pipeline a jej veľkosti (31 inštrukcií u CPU Pentium 4, 14 inštrukcií u CPU Core2) je možné túto vlastnosť zanedbať. Tento spôsob merania je rýchly a presný, za predpokladu, že sa kód vykonáva na jednom CPU a jeho frekvencia je počas vykonávania nemenná. Jedno CPU a nemenná frekvencia boli v období po uvedení procesorov Pentium® takmer 10 rokov ich stabilnou vlastnosťou. Rovnaká situácia bola aj u ostatných výrobcov CPU pre PC a tak sa využívanie tejto inštrukcie široko zaužívalo. V súčasnosti sú používané takmer výlučne procesory s 2 a viac jadrami ktorých frekvencia sa mení podľa záťaže. Synchronizácia hodnôt získavaných pomocou RDTSC

naprieč jadrami nebola zaručená a frekvencia procesoru môže kolísať v závislosti od nastavenia operačného systému, BIOSu, teploty a vyťaženia CPU. Tieto zmeny v architektúre HW nielen sťažili využitie tejto hodnoty ale zároveň spôsobili nefunkčnosť mnohých API operačných systémov, ktoré boli na týchto predpokladoch založené.
Riešeniu vzniknutej situácie sa venovalo úsilie zo strany výrobcov hardware ako i softvérových spoločností. Okrem vytvorenia alternatív, boli implementované i opatrenia ktoré zabezpečujú kompatibilitu nového hardware s existujúcim software. Praktické overenie súčasného stavu je popísané a vyhodnotené v druhej časti tejto kapitoly.

4.2.4 Systémové API a špecializovaný časovací hardware

Radikálnym riešením problematiky sledovania času a časovania udalostí je štandardizovanie a zavedenie nového hardware. Tento prístup sa začal objavovať najprv u notebookov, kde sa objavili problémy spôsobené meniacou sa frekvenciou CPU skôr, než na desktopových PC. Ako pomerne rozšírené riešenie, sa z praktických skúseností ukazuje použitie hardvérového časovača s frekvenciou 3,579,545 Hz. Pravdepodobne sa jedná o rovnaký generátor frekvencie ako pri štandarde vysielania NTSC [6]. Novšie riešenie je napríklad HPET (High Precision Event Timer), ktorý má v štandarde odporúčanú minimálnu frekvenciu 10MHz [7].
Využitie časovacieho hardware zastrešuje operačný systém. Pre účel tejto práce bolo preskúmané API systému Windows™, odporúčané spoločnosťou Microsoft®: QueryPerformanceCounter popísané na MSDN [4]. Toto API podľa dostupného hardware použije špecializované zariadenie, alebo využije kontrolovanú inštrukciu RDTSC. API pracuje s frekvenciou časovača a počtom napočítaných cyklov. Výpočet času pomocou tohto API je analogický k použitiu RDTSC s tým rozdielom, že frekvencia časovača je zisťovaná priamo v API, kým u RDTSC je nutné frekvenciu buď odmerať, alebo zadať ručne.

4.2.5 Test aktuálnych systémov

Za účelom praktického overenia metód merania času, navrhnutých pre túto prácu, bol vykonaný test na niekoľkých platformách. Tento test bol implementovaný v jazyku C++, skompilovaný pomocou VS2008 a výsledky boli minimálne 1x overované pre každú platformu.

Test prebiehal spustením exe súboru na nezaťaženom počítači. Výstup programu sú textové informácie zistené behom programu pomocou API a krátkymi experimentmi.
Test pozostáva z nasledujúcich častí.

- Zistenie počtu jadier CPU tak ako ich hlási OS (tento údaj môže byť skresľujúci na platformách ktoré emulujú na jednom jadre viacej logických CPU, ako sú procesory s podporou HT od spoločnosti Intel® - Pentium 4 a Core i7)
- Odčítanie frekvencie CPU bez záťaže. Toto odčítanie je realizované pomocou blokujúceho uspania procesu na 10 sekúnd a následné prepočítanie hodnôt počítadla pred a po uspaní. Dĺžka časového úseku, ktorý bol meraný sa určuje pomocou systémového časovača a presnosť jeho zmerania je približne 16ms.
- Odčítanie frekvencie CPU so záťažou. Je realizované rovnako ako meranie bez záťaže, ale miesto uspania procesu sa aktívne čaká na ubehnutie požadovaného časového intervalu. (Vyťaženie CPU a jeho frekvencia bola počas vykonania meraní súčasne pozorovaná, za pomoci výrobcom dodávaného monitorovacieho software pre každú z platforiem. Pri odčítaní frekvencie bez záťaže dochádzalo skutočne k zníženiu pracovnej frekvencie CPU a pri meraní so záťažou bežali všetky procesory na svojej maximálnej frekvencií.)
- Meranie trvania prázdneho cyklu pre 100 tisíc, 1 milión a 10 miliónov opakovaní. Kód použitý pre toto meranie je „*int tmp=0;for (int i = 0; i< iterations; i++){tmp = tmp ^ i;}*„. Vykonávaná bitová operácia v tele cyklu je potrebná pre zabránenie odstránenia

cyklu optimalizáciami. Testovací program bol skompilovaný s maximálnou úrovňou optimalizácií. Takto boli kompilované všetky implementácie v tejto práci, pretože rovnako budú bežať/bežia aj skutočné programy založené na týchto algoritmoch.

- Sledovanie neklesania postupnosti odčítavanej pomocou RDTSC bez nastavenia afinity procesu a s nastavením afinity na prvé jadro CPU.
- Sledovanie najväčšieho rozdielu medzi hodnotami odčítavanými pomocou RDTSC bez nastavenia afinity a s nastavením afinity na prvé jadro.
- Meranie trvania jedného zistenia stavu počítadla pomocou inštrukcie RDTSC použitím cyklu dĺžky 100 tisíc, 1 milión a 10 miliónov opakovaní.
- Zistenie parametru metódy Windows® API nazývanej QueryPerformanceCounter. Zisťovaná je frekvencia časovača udávaného týmto API a meraný je tiež príspevok samotnej metódy k nameranému času pre 100 tisíc, 1 milión a 10 miliónov opakovaní.

Kód ktorý meria príspevok samotného merania k výslednému času je pre obe metódy založený na rovnakej myšlienke.

```
for (int i = 0; i< iterations; i++){
        QueryPerformanceCounter(&start);
        QueryPerformanceCounter(&stop);
        tmp += stop.QuadPart - start.QuadPart;
}
```

Meria sa prázdny úsek pričom sa kumulatívne sčíta čas, nameraný medzi spustením merania a zastavením merania, teda začiatkom a koncom merania bez akejkoľvek ďalšej operácie medzi týmito úkonmi.

	Puma (ZM82)	Centrino (T7500)	Kentsfield (Q6600)	Conroe (E6300)	Winchester (A64_3200)	Prescott (P4_550)
počet logických CPU	2	2	4	2	1	2
počet fyzických CPU	2	2	4	2	1	1
frekvencia CPU bez záťaže / MHz	2,200	2,194	2,400	1,864	2,009	3,412
frekvencia CPU so záťažou / MHz	2,200	2,194	2,400	1,864	2,009	3,412
trvanie prázdneho cyklu (počet pre 1M)	910,730	834,742	711,480	914,385	997,892	700,292
neklesanie RDTSC	neklesá	neklesá	neklesá	neklesá	neklesá	neklesá
najväčšia medzera pri afinite na všetky jadrá CPU	11,275,522	3,508,120	429,273	271,019	125,764,674	26,730,291
najväčšia medzera pri afinite na prvé jadro CPU	18,878,645	52,422,007	1,319,058	14,890,512	126,326,646	49,042,535
trvanie RDTSC (počet pre 1M)	165,193,325	68,071,322	67,412,646	67,374,258	9,111,584	111,919,504
frekvencia QPC / Hz	3,579,545	3,579,545	25,000,000	1,864,820,000	3,579,545	3,412,890,000
trvanie QPC (počet pre 1M)	1,648,967,396	1,960,806,471	1,271,774,160	277,710,914	2,381,378,918	338,045,041

T4.2.5 Prehľad zistených hodnôt na šiestich platformách.

Názvy platforiem sú uvádzané ako kódové označenia procesorov alebo celých platforiem,

ktoré boli pre porovnanie vybrané všetky rôzne. Niektoré názvy možno nájsť v [8], ďalšie informácie je možné vyhľadať napríklad pod heslami „Centrino T7500 CPU“.

Prvé dva systémy sú zástupcami notebookových platforiem Intel-Centrino a AMD-Puma, prostredná dvojica sú desktopové systémy s viacjadrovými procesormi (2 a 4 jadrá) a posledné 2 systémy sú staršie, jedno jadrové CPU, Athlon 64 od AMD a Intel Pentium 4.

Z uvedeného prehľadu plynie niekoľko kľúčových zistení.

- Všetky testované platformy až na Winchester pri nečinnosti znižujú frekvenciu CPU, inštrukcia RDTSC ale vracia vždy počet cyklov, ako keby CPU bežalo po celý čas na maximálnej frekvencii.
- Prázdny cyklus u všetkých platforiem zabral výrazne menej cyklov procesoru, než bol samotný počet vykonaných inštrukcií, toto zodpovedá skutočnosti, že súčasné procesory majú IPC > 1.
- Pokles hodnoty načítanej pomocou RDTSC sa nepotvrdil ani v jednom prípade.
- Najväčšia medzera medzi neustále za sebou načítanými hodnotami RDTSC vznikla vždy pri spracovaní na 1 jadre. Medzera v hodnotách pri odčítaní počtu cyklov väčšia, než je doba samotného merania znamená, že testovací proces bol na nejaký čas pozastavený. Najväčší dopad je pozorovateľný na platforme Winchester, ktorá má iba jeden, jedno jadrový procesor.
- Najlepšie sa z hľadiska dostupnosti CPU javia desktopové viacjadrové procesory, pri ktorých sú medzery v načítaných hodnotách rádovo nižšie. Na týchto procesoroch je teda najmenšie nebezpečenstvo významného ovplyvnenia výsledku pri meraní.
- Odčítanie samotnej hodnoty počítadla cyklov zaberie na všetkých procesoroch rádovo 100 cyklov s výnimkou 9 cyklov na platforme Winchester. Táto skutočnosť je pravdepodobne zapríčinená tým, že platforma Winchester neznižuje frekvenciu CPU a má iba jedno jadro, preto realizuje túto inštrukciu jednoduchšie.
- U troch procesorov bola frekvencia QPC hlásená presne 3,579,545 Hz. U jedného systému šlo o 25MHz, čo zodpovedá špecifikácií HPET. Posledné dva systémy hlásili frekvenciu časovača zhodnú s frekvenciou CPU, pravdepodobne sa teda na týchto platformách využíva aj v tomto API inštrukcia RDTSC.
- Trvanie jedného odčítania času trvá pomocou API QPC tam, kde je prítomný samostatný časovač rádovo 1000 až 2500 cyklov CPU, na platformách Prescott a Conroe to bolo približne 300 cyklov.

4.3 Metodika

Všetky testy implementácií prebiehali podľa jednotnej metodiky, ktorej popis je predmetom tejto kapitoly. Pre zníženie objemu práce u často opakovaných úkonov, bola vytvorená vlastná implementácia testovacieho prostredia. S podobným výsledkom by bolo možné napríklad využitie utility make a skriptov na systéme Unix/Linux, použité prostredie je však prenositeľné a v spojení s niektorým IDE (Eclipse) ponúka o niečo lepší užívateľský komfort. (Pre použitie sú potrebné aspoň minimálne znalosti programovania, ktoré sú i tak predpokladom realizácie podobnej štúdie.)

4.3.1 Použité implementácie

Implementácie algoritmov boli prispôsobené pre beh na platforme Windows, kompiláciou v Microsoft Visual C++ 2008 Express Edition. Vlastná implementácia merania času je zakompilovaná do všetkých implementácií a vždy obaľuje iba volanie samotnej metódy

výpočtu, bez započítania vstupu a výstupu. V prípade implementácie využívajúcej GPU, je započítaný aj čas spotrebovaný presúvaním dát do pamäte grafickej karty, nakoľko tento krok je v každom prípade nevyhnutný a ostatné implementácie bežiace na CPU ho nepotrebujú. Všetky implementácie sú kompilované ako „release“ s nasledujúcimi nastaveniami:

- optimalizácie: maximálna rýchlosť (O2)
- platforma: x86, SSE2
- kompilácia: kód jazyka C
- vypnutie rozšírených znakových funkcií: vypnutý wchar_t
- rozvinutie funkcií: vždy ak je to možné
- ostatné optimalizácie: uprednostnenie rýchleho kódu, generovanie kódu pri linkovaní

Pre presné informácie o použitých nastaveniach, je na DVD priložený súbor, ktorý bol použitý ako základ pre všetky projekty testovaných implementácií. (src/windows/template.vcproj)

Generátory problémov nemajú z hľadiska ich kompilácie vplyv na výsledky merania, boli preto kompilované s rôznymi nastaveniami, v závislosti od potrebných úprav pre spustenie na systéme Windows.

Testovacie prostredie je založené na platforme Java a skladá sa z vlastnej implementácie pomocných nástrojov. Pre ich použitie je nutná modifikácia tried „Main“, umiestnených v najvyššom balíku. Program „Main03“ slúži ako vizualizácia pre grafy, „Main04“ vyhodnocuje testovaciu dávku a „Main06“ spája výsledky meraní do grafov a tabuliek.

4.3.2 Priebeh experimentov

Testy prebiehali v dávkach, na stroji nezaťaženom žiadnou inou úlohou. Každá dávka je definovaná pomocou niekoľkých súborov s príponou „template“ - šablón uložených v adresári dávky. Spustenie programu „Main04“, ktorý načíta a spúšťa spracovanie dávky, bolo realizované pomocou prostredia Eclipse. Životný cyklus spracovania je nasledovný:

1. Vytvorenie výstupného adresára so zhodným názvom ako je adresár dávky na preddefinovanej ceste.
2. Rozloženie súborov template na definície jednotlivých problémov. Súbory šablón obsahujú jednoduchú syntax, kde jeden riadok obsahujúci ľubovolný text a sekvenciu „text1[variant1|variant2|...|variantN]text2“ vytvorí N variácií problémov postupným vytváraním riadkov „text1variant1text2“ až „text1variantNtext2“ s ostatnými riadkami súboru.
3. Vytvorené súbory problémov (prípona „properties“) sú znovu načítané. Obsahujú vždy určenie práve jedného generátoru a jedného algoritmu, spolu s ostatnými parametrami, ktoré sú špecifické pre použitý generátor.
4. Pre každý problém je spustený generátor. Vytvorená sieť je uložená v spoločnom formáte v súbore s príponou „input“.
5. Pre každý problém je spustené riešenie pomocou testovanej implementácie. Vstup vytvorený v kroku 4 je konvertovaný do tvaru, ktorý vyžaduje implementácia.
6. Výstup riešenia (neupravovaný v súbore s príponou „output“) a výsledok riešenia (čas behu, počet cyklov a pod. v súbore s príponou „result“) sú uložené na disk.

Krok 5 je realizovaný opakovane, pre ten istý vstup a algoritmus sa použije iba minimálna hodnota nameraného času z piatich pokusov. Po prečítaní celého vstupu algoritmom, začne

bežať časový limit stanovený na 1 minútu skutočného času. Po celý čas je hlavné vlákno blokované čítaním výstupu programu (na ten sa počas výpočtu nič nevypisuje). Časovacie vlákno spí, prebúdza sa iba 1x za sekundu, aby vpísalo do konzoly znak „.“ a skontrolovalo uplynutie časového limitu. Ak implementácia vypíše na výstup riadok, ktorý obsahuje informácie o nameranom čase, odpočítavanie je zastavené a program má neobmedzený čas na to, aby zapísal na výstup nájdené riešenie. Všetky implementácie vypisujú výsledný časový údaj pred tým, ako samotné riešenie. Ak uplynie časový limit a implementácia nestihne spracovať zadaný vstup, výpočet sa ukončí a odmeraný čas sa uvedie 0. To znamená pri spracovaní vypršanie času na danom vstupe.

Dávky experimentov sú vytvorené tak, aby pre každú výslednú hodnotu, existovalo najmenej 5 meraní pre rovnaký typ problému riešeného tým istým algoritmom. Vo výsledku teda znamená otestovanie A implementácií pre B veľkostí problému a piatich meraniach pre každý výsledok $A \cdot B \cdot 5^2$ meraní na $A \cdot B \cdot 5$ vstupoch.

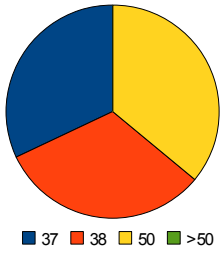
Implementácie algoritmov nemajú nastavenú afinitu na jedno jadro CPU, systém ich tak môže plánovať ľubovolne, čo sa ukázalo ako najlepšia možnosť z hľadiska presnosti merania času.

Aby pri prerušení spracovania dávky, nebolo nutné celý výpočet opakovať, výsledky a vstupy sú ukladané priebežne na disk. Pri opakovanom spustení sú vytvárané iba tie súbory, ktoré chýbajú, alebo je čas ich zmeny starší, než súbor na ktorom závisia.

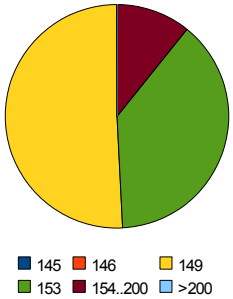
4.3.3 Kalibrácia merania času

Pri každom spustení dávky, bol pred spustením prvého experimentu, spúšťaný program „cal.exe“. Tento program je zostavený s identickými nastaveniami, ako testované implementácie a používa tú istú funkciu pre meranie času. Opakovane (10^6 krát) je spúšťané a zastavované meranie času na prázdny úsek. Zistené hodnoty sú priebežne započítavané do dvoch histogramov v rozsahoch 0 až 5000 cyklov / nanosekúnd.

Odčítania RDTSC pre jeden test (10^6 meraní)	
hodnota / počet cyklov	zastúpenie
<37	0
37	320154
38	319816
50	359992
>50	38



Odčítania QPC pre jeden test (10^6 meraní)	
hodnota / nanosekundy	zastúpenie
<145	0
145	105
146	118
149	508099
153	383701
154 až 200	107870
>200	107



T4.3.3 Zhrnutie výsledkov pre jeden beh kalibrácie.

Údaje získane v desiatich samostatných meraniach ukázali, že hodnoty ako napríklad 37 cyklov alebo 149ns, sa vo vysokom zastúpení vždy opakujú. Žiadna z 10^7 získaných hodnôt nebola nižšia ako 37 cyklov pre odčítanie RDTSC a 145ns pre odčítanie QPC. Za hodnoty vo význame príspevku samotnej metódy k nameranému času, boli na základe týchto zistení zvolené mediány príslušných zoznamov hodnôt. Odchýlky, ktoré pri tejto metóde môžu vzniknúť, sú štatisticky nevýznamné vzhľadom k meraným dobám behu.

4.4 Testovacia platforma

Všetky experimenty zahrnuté do finálnych výsledkov, prebehli na spoločnej platforme, podrobne popísanej v tejto kapitole. Niektoré predbežné testy a čiastkové experimenty boli vykonané aj na iných platformách. Súčasne s výsledkami získanými na inej platforme, je vždy uvedená jej stručná charakteristika, napríklad v tvare „Core 2 Duo E6300“, alebo „C2D 1.86GHz“. Veľkosť operačnej pamäte bola v každom prípade minimálne 2GB a operačný systém Windows® XP Professional Service Pack 3.

4.4.1 Hardvér

Komponenty testovacej zostavy sú volené s ohľadom na aktuálnosť tak, aby umožnili testovaným implementáciám dosiahnuť optimálny výkon. Pretože žiadna z testovaných implementácií nevyužíva efektívne viacero jadier CPU, bol použitý dvoj jadrový procesor s čo najvyššou pracovnou frekvenciou. Implementácia CUDASSSP využíva konštrukciu dostupnú natívne až od GPU revízie G92 a vyššie (výpočetný model sm_11, atomické funkcie). Pamäť o veľkosti 2GB postačovala pre súčasný beh systému, programov a dvojnásobné načítanie najväčšieho vstupu, bez požitia stránkovania na disk (najväčšie testované vstupy mali približne 750MB).

Komponenta	Typ
CPU	Intel® E8600 (3.33GHz, 6MB cache, 45nm)
Základná doska	DFI, DK P35-T2RS LGA 775
RAM	2x1GB Kingston, 800MHz (DDR2), CL5
Grafická karta	8800GTS 512(MB), G92, BIOS 62.92.16.00.43 frekvencia: jadro/shader/pamäť: 650/1625/970MHz
Pevný disk č. 1	WD 320GB, 7200 ot./min, 8MB cache
Pevný disk č. 2	WD 640GB, 7200 ot./min, 16MB cache
Zdroj	PC P&C 610W
Ostatné pripojené príslušenstvo	bežný LCD monitor, klávesnica, myš

T4.4.1 Komponenty testovacej zostavy.

Pevný disk č. 1 bol súčasne systémový a pracovný disk, bol rozdelený na dve partície, jednu systémovú o veľkosti 20GB a zvyšok disku. Disk č. 2 slúžil na odkladanie výsledkov po dobehnutí dávky. Pracovné frekvencie a napätie všetkých komponent boli nastavené na výrobcom udávané hodnoty. Funkčnosť a stabilita systému boli pred vykonaním

experimentov overené známymi utilitami (Memtest86, SuperPI, 3DMark2006). Účelom bolo najmä vylúčenie možnosti prehrievania CPU v dlhodobej záťaži, ktoré by mohlo mať za následok znižovanie výkonu [39] a nestabilitu spôsobenú nedostatočným napájaním. Napájanie a chladenie zostavy bolo dimenzované tak, aby zaistilo stabilitu aj za takýchto podmienok. Šetriace funkcie, ktoré pri nečinnosti znižujú spotrebu, alebo vypínajú časti systému boli v maximálnej možnej miere v BIOS i operačnom systéme zakázané.

4.4.2 Softvérové prostredie

Základ vybavenia tvoril operačný systém Windows® XP Professional Service Pack 3, ošetrený aktuálnymi záplatami. Systém bežal v základnom nastavení, bez vypínania akýchkoľvek služieb. Na systéme nebol inštalovaný žiadny software, ktorý by bežal počas experimentov, okrem softvéru, ktorý je uvedený ďalej.

Systémové prostredie bolo rozšírené o tieto súčasti:

- ovládače dodané výrobcom hardware pre základnú dosku
- ovládač grafickej karty NVIDIA® s podporou CUDA 2.1, verzia 181.20 (32bit)
- DirectX 9.0c
- Java SE [34] JDK, verzia 1.6.0 Update 10 až 13 a príslušná JVM
- Cygwin [35], verzia 1.5.25-15

Ďalej bol inštalovaný nasledujúci software:

- Eclipse IDE pre Javu [37]
- Visual C++ 2008 Express Edition [38] (verzia 9.0.30729.1)
- CUDA [36] toolkit 2.1 (verzia 9.01.429)
- NVIDIA SDK CUDA 2.1 (verzia 9.01.429)
- kompilátor GCC a G77 (GNU Fortran) (verzia 3.4.4)

Eclipse a Java boli použité ako platforma pre spúšťanie testov. V jazyku Java je implementované vlastné prostredie, ktoré integruje generovanie vstupov, spúšťanie testov a ukladanie výsledkov. Generátory vstupov boli kompilované pomocou Visual C++ 2008 a GCC. Kernely pre CUDA implementáciu boli kompilované pomocou nvcc (CUDA toolkit). Generátory GTgraph [40] vyžadujú kompiláciu pomocou Fortranu (G77) a GCC, pre beh týchto generátorov sú potrebné behové knižnice Cygwin.

4.4.3 Parametre platformy

Pre možnosť priameho porovnania dosiahnutých výsledkov s inými prácami, realizovanými na podobných testovacích zostavách (napríklad [15] a [33]), udáva tabuľka *T4.4.3a* prehľad teoretického maximálneho výkonu a pamäťovej priepustnosti CPU a GPU.

Komponenta	Priepustnosť pamäte v GB/s	Maximálny výkon v GFlops
GPU (8800GTS 512)	62.1	624.0
CPU (E8600)	6.4	13.32

T4.4.3a Prehľad teoretického výkonu a pamäťovej priepustnosti použitého CPU a GPU.

Pre testovaciu platformu bol vykonaný rovnaký experiment, ako popisuje kapitola 4.2.5.

	Wolfdale (E8600)
počet logických CPU	2
počet fyzických CPU	2
frekvencia CPU bez záťaže / MHz	3,338
frekvencia CPU so záťažou / MHz	3,338
trvanie prázdneho cyklu (počet pre 1M) / cykly	526,842
neklesanie RDTSC	neklesá
najväčšia medzera pri afinite na všetky jadrá CPU	610,750
najväčšia medzera pri afinite na prvé jadro CPU	13,781,837
trvanie RDTSC (počet pre 1M) / cykly	37,319,510
frekvencia QPC / Hz	3,338,760,000
trvanie QPC (počet pre 1M) / cykly	131,602,318

T4.4.3b Prehľad zistených hodnôt na testovacej platforme.

Zhodou okolností dosiahla testovacia platforma najlepšie hodnoty pri odčítaní RDTSC a QPC spomedzi všetkých viac jadrových CPU. Meranie pomocou API QPC navyše indikuje použitie RDTSC, vzhľadom ku rovnosti hlásenej frekvencie počítadla a pracovnej frekvencie CPU. Odčítania hodnôt bez záťaže (v stave keď sa automaticky zníži pracovná frekvencia CPU) a hodnôt so záťažou ukázali, že nedochádza k ovplyvneniu merania času pomocou tejto metódy pri kolísaní pracovnej frekvencie. K tomuto navyše typicky nedochádza, pretože implementácie používajúce CPU, vyťažia vždy jedno jadro na 100%. (Ku zníženiu frekvencie CPU počas merania, by mohlo teoreticky dôjsť v prípade CUDA implementácie.) Preplánovanie procesu počas výpočtu, taktiež nespôsobilo ani v jedinom prípade odčítanie dvoch, po sebe nasledujúcich klesajúcich hodnôt. Výsledky uvedené v tabuľke T4.4.3b, boli potvrdené opakovanými meraniami, bez zistenia zásadného rozdielu.

5 Triedy problémov

Triedy problémov a testy sú zvolené tak, aby pokryli čo najširšiu škálu reálne riešených problémov. S výnimkou jedného testu, na cestných sieťach, sú všetky grafy generované. Pôvod generátorov, parametre, vlastnosti a ukážky vstupov sú uvedené v zodpovedajúcich častiach tejto kapitoly pre každý generátor. Parametre sú zvýraznené **tučným** písmom, aby nedošlo k zámene so symbolmi zavedenými v kapitolách 2 a 3. Pomocou siedmich popísaných generátorov, boli vytvorené a otestované vstupy o celkovej veľkosti približne 400GB.

5.1 Generátor Grid

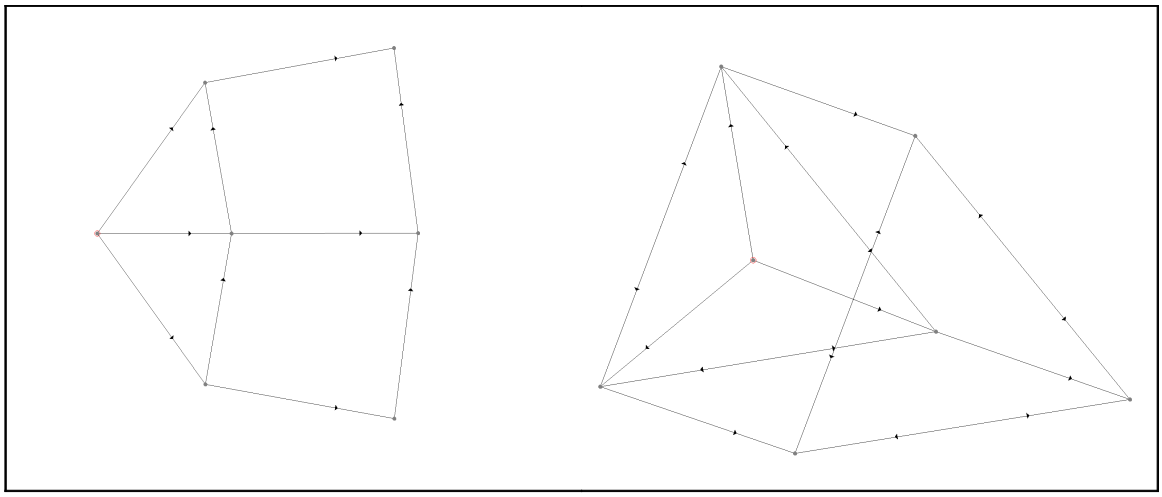
5.1.1 Pôvod generátora

Grid je prvým generátorom prebraným z práce [1]. Zdrojové kódy tejto práce sú dostupné na internete pod názvom *SPLIB* a ich ďalšie použitie pre výskumné účely nieje limitované. Autormi *SPLIB* sú Boris Cherkassky (cher@ch.comrel.msk.su), Andrew Goldberg (goldberg@cs.stanford.edu) a Tomasz Radzik (radzik@cs.cornell.edu).

5.1.2 Parametre a vlastnosti výstupu

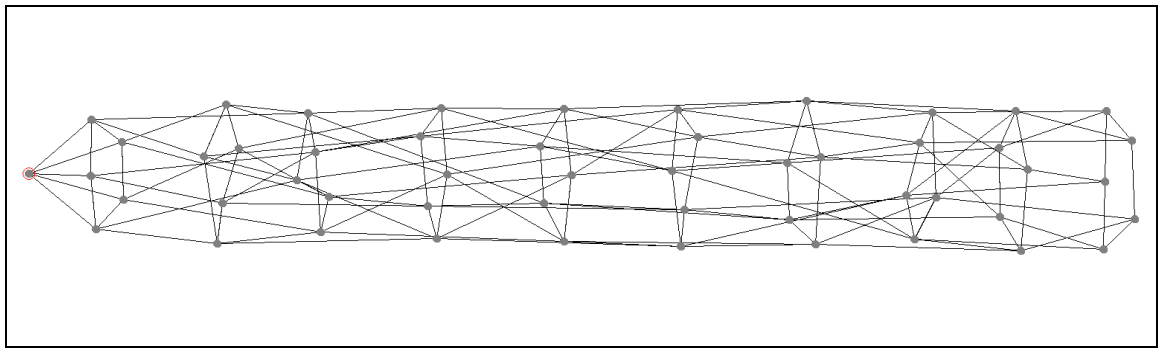
Výstupné problémy možno znázorniť do celočíselnej mriežky obdĺžnikového tvaru. Povinnými parametrami generátoru sú parametre **X** a **Y**, ktoré udávajú horizontálny a vertikálny rozmer mriežky. Vrcholom zodpovedajú body na mriežke so súradnicami $[x, y]$, $1 \leq x \leq \mathbf{X}$, $1 \leq y \leq \mathbf{Y}$. Vždy sú pridané horizontálne hrany $([x, y], [x + 1, y])$, $1 \leq x < \mathbf{X}$, $1 \leq y \leq \mathbf{Y}$. Vždy je pridaný jeden vrchol ako zdroj, prepojený s vrcholmi $[1, y]$, $1 \leq y \leq \mathbf{Y}$. Vrcholy ktorým prislúcha rovnaká súradnica x budú ďalej odkazované ako vrcholy ležiace na jednej vrstve. Každá vrstva obsahuje práve **Y** vrcholov.

Parameter **seed** je tretím povinným parametrom. Slúži ako inicializácia náhodného generátoru a jeho hodnota musí byť nepárne celé číslo uložiteľné do 32 bitového typu int. Ostatné uvedené parametre sú nepovinné a majú preto svoje implicitné hodnoty. Parameter **c** určuje spôsob prepojenia vrcholov vo vrstve. Môže nadobúdať hodnoty 'p', 'c' a 'd'. Pre hodnotu 'p' (path) sú vrcholy prepojené zhora nadol cestou, hrany sú tvaru $([x,y],[x,y+1])$ pre $1 \leq x \leq \mathbf{X}$, $1 \leq y < \mathbf{Y}$. Hodnota 'c' (cycle) prepája vrcholy do jednoduchého cyklu, kde ku hranám zodpovedajúcim prepojeniu 'p', pribudnú hrany $([x, \mathbf{Y}], [x, 1])$, $1 \leq x \leq \mathbf{X}$. Hodnota 'd' (double-cycle) prepája vrcholy ako pre 'c', ale navyše pridáva aj hrany opačné, vzniknú tak dva protichodné cykly v rámci vrstvy. Hodnota 'd' je pre tento parameter implicitnou.



O5.1.2a Nakreslenie problému pre **X** *= 2,* **Y** *= 3 a pre parameter* **c** *= 'p' vľavo a* **c** *= 'd' napravo.*

Parametre **cl** a **cm** slúžia pre definovanie maximálnej a minimálnej dĺžky prepájajúcej hrany. Prepájajúcimi hranami sú všetky hrany v rámci vrstvy. Oba parametre sú celočíselnej hodnoty a v prípade potreby sú ich hodnoty vymenené tak, aby **cm** ≤ **cl**. Prípadná výmena a celočíselnosť hodnôt platí i pre všetky ostatné parametre určujúce dĺžky hrán. Skutočná dĺžka hrany je vždy volená zo zadaného intervalu náhodne. Implicitné hodnoty pre **cm** a **cl** sú 0 a 100. Parameter **ax** udáva počet pridaných hrán pre každú vrstvu, implicitne je tento počet 0. Prepájané sú náhodné dvojice vrcholov s rovnakou hodnotou x. Parametre **am** a **al** udávajú pre pridané hrany rozsah dĺžky, predvolená hodnota je 0 až 10000.
Pre pridanie hrán medzi vrstvami slúži parameter **ix**, ktorého implicitná hodnota (a súčasne minimálna) je 1. Jeho hodnota udáva počet vrstiev na ktoré sa každá vrstva napája. Vždy sú prepojené všetky vrcholy vrstvy na zodpovedajúce vrcholy inej vrstvy. Pre hodnotu 1 sú spojené iba susedné vrstvy. Úzko súvisiaci parameter **ih**, udáva krok medzi napájanými vrstvami (má zmysel iba ak **ix** > 1). Pre dané hodnoty **ix** a **ih** sú teda všetky vrcholy $[x,y]$ každej vrstvy, prepojené na $[x + 1, y]$, $[x + 1 +$ **ih**$, y]$, ..., $[x+1+($**ix**$-1)\cdot$**ih**$, y]$. Ak je súčasne uvedený parameter **ip**, cieľové vrcholy prepojení nie sú všetky na tej istej súradnici y, ale vybrané cez permutovanú postupnosť 1 až **Y**. Dĺžky hrán medzi vrstvami sú určené parametrami **im** a **il**. Implicitné hodnoty **im** a **il** sú 1000 a 10000.



O5.1.2b Ukážka výstupu pre **X** *= 10,* **Y** *= 5,* **ix** *= 2,* **ih** *= 1 a použitý* **ip**.

Pre sťaženie úloh je možné definovanie lineárneho a kvadratického multiplikátoru, ktoré zvýšia dĺžky hrán. Parameter **in** udáva hodnotu lineárneho multiplikátoru a **is** hodnotu kvadratického multiplikátoru. Ak sú tieto multiplikátory nenulové, dĺžka hrany l medzi vrstvami x_1 a x_2 sa zvýši o hodnotu $l \cdot multiplikátor \cdot |x_2-x_1|$.

Pre problémy obsahujúce záporné hrany, je výhodné použitie generovania pomocou potenciálov. Potenciál $P:V \rightarrow R$, je zobrazenie na vrcholoch grafu. Pomocou potenciálu je definovaná redukovaná dĺžka hrany. Nech dĺžka hrany $e=(v, w)$ je l, potom redukovaná dĺžka hrany $l_d = l + P(v)–P(w)$. Takto upravený problém, kde všetky dĺžky hrán sú redukované pomocou potenciálov, neobsahuje cykly zápornej dĺžky aj v prípade, ak obsahuje záporne ohodnotené hrany. (Pôvodný problém nesmie obsahovať cyklus zápornej dĺžky. Zamýšľané použitie je prevod problému s nezáporným ohodnotením hrán, na problém, kde sa takéto ohodnotenia vyskytujú.) Pre použitie potenciálov slúži parameter **p**, pričom horná a dolná hranica potenciálu pre vrchol, je udaná parametrami **pm** a **pl**. Potenciály je možné modifikovať multiplikátormi **pn** a **ps** v závislosti od poradového čísla vrcholu. Ak je hodnota **pn** nenulová, pripočítava sa $\mathbf{pn}\cdot i$ ku hodnote potenciálu vrcholu i, ak je hodnota **ps** nenulová, pripočíta sa $\mathbf{ps}\cdot i^2$.
Umelý zdroj je možné pridať pomocou parametru **s**, tento je napojený na každý z $\mathbf{X}\cdot\mathbf{Y}+1$ vrcholov pôvodnej siete. Takýmto spôsobom sa v praxi, pri úlohách, kde nieje zaručená súvislosť grafu, zabezpečuje nájdenie nejakého riešenia. Dĺžky hrán smerujúcich z umelého zdroja, sa volia veľké v porovnaní s ostatnými hranami, zvyčajne preto, aby neovplyvnili výsledok ak má pôvodná úloha riešenie. Jedinou výnimkou je hrana pripájajúca umelý zdroj k pôvodnému zdroju, ktorej dĺžka sa volí 0. Rozsah pre dĺžky prípojných hrán sa zadáva pomocou parametrov **sl** a **sm**.

5.2 Generátor Rand

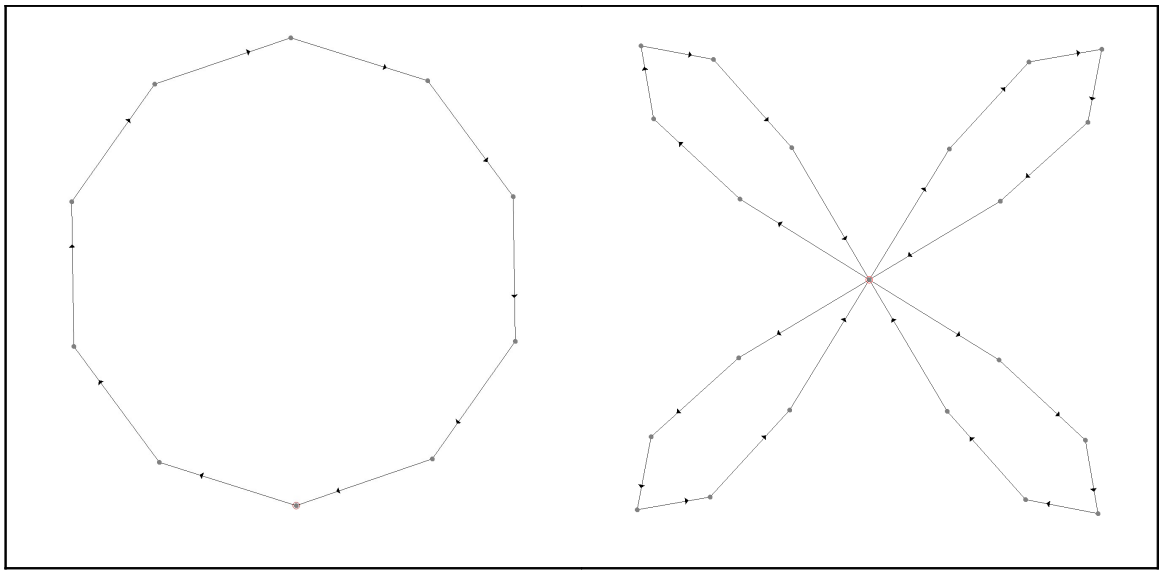
5.2.1 Pôvod generátora

Generátor Rand je druhým generátorom prebraným z práce [1], ako zástupca generátorov náhodných sietí. Náhodné siete sú široko používané pre testovanie implementácií grafových algoritmov i napriek tomu, že reálne riešené problémy zvyčajne nie sú náhodnými sieťami.

5.2.2 Parametre a vlastnosti výstupu

Vrcholy vo výstupnej sieti tohto algoritmu nie sú pomyselne rozmiestnené v žiadnom usporiadaní a sú značené prirodzenými číslami od 1. Algoritmus pracuje s tromi povinnými parametrami. Parameter **seed** má rovnaký význam ako pri generátore Grid. Parametre **n** a **m** udávajú počet vrcholov a hrán vytváraného grafu. Musí platiť $\mathbf{m} \geq \mathbf{n}$, pretože algoritmus vždy prepája vrcholy do hamiltonovskej kružnice. Dĺžka hrán na kružnici sa zadáva jediným parametrom **cl**. Počet vrcholov na kružnici sa implicitne volí **n**, je možné ho predefinovať pomocou parametra **ch**, v tom prípade je nutné zvýšiť hodnotu **m** tak, aby mal generátor dostatok hrán na prepojenie všetkých vrcholov. Parameter **ch** nesmie byť vyšší než je celkový počet vrcholov. Ak $\mathbf{ch} < \mathbf{n}$, je vytvorených viacero kružníc, pričom všetky vytvárané kružnice obsahujú vrchol s číslom 1 (zdroj, v sieti bez umelého zdroja). Ak pre poslednú kružnicu nie je dostatok nepripojených vrcholov, zostrojí sa kružnica o menšom počte vrcholov, prípadne sa pripojí posledný zostávajúci vrchol.

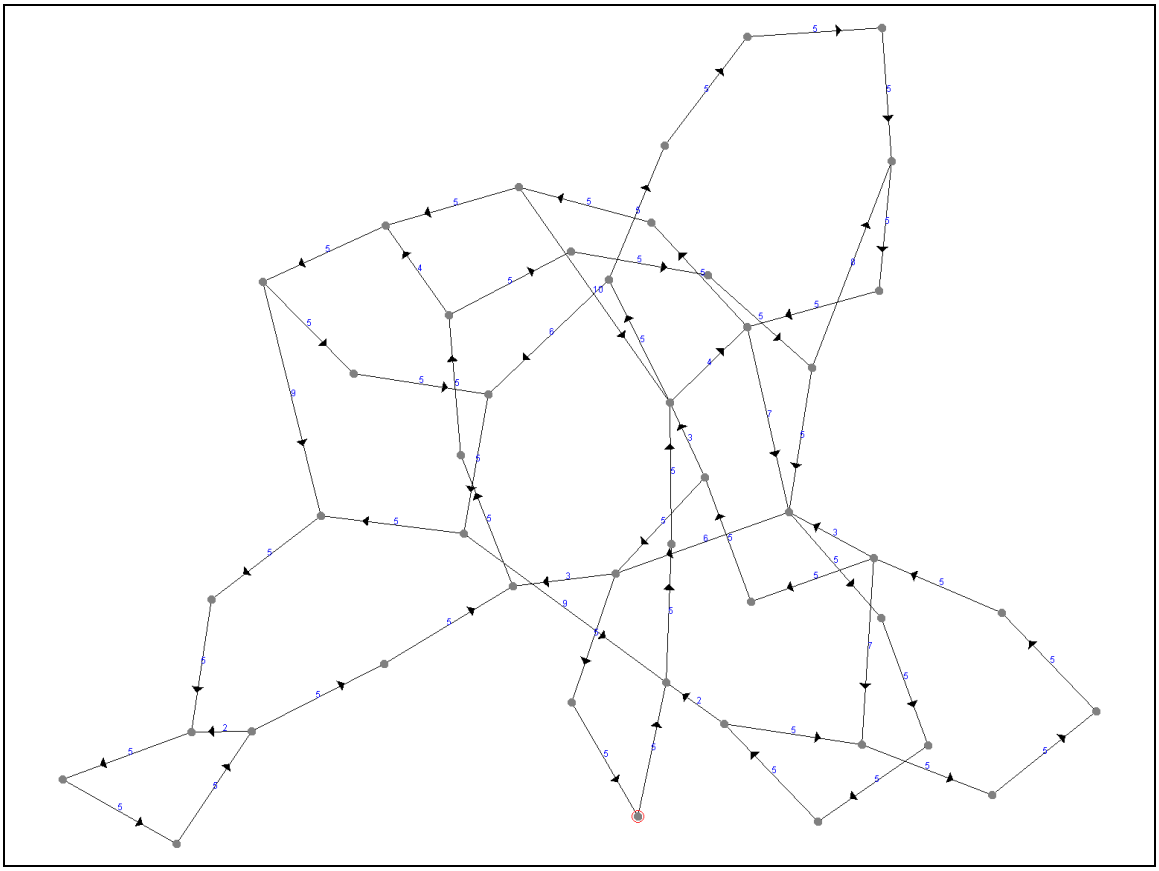
Prepájajúce kružnice zaisťujú dostupnosť všetkých vrcholov a v priemernom prípade (v kombinácií s náhodnými hranami) i silnú prepojenosť. Ak je navyše dĺžka hrán na kružniciach nízka a počet vrcholov na jednej kružnici vysoký, je pravdepodobné že sa veľká časť z nich nachádza na najkratších cestách. Výsledný strom tvorený najkratšími cestami má potom vyššiu hĺbku a takéto problémy sú pre niektoré z testovaných algoritmov ťažšie. Voľba vyššieho počtu kratších kružníc vedie ku kratším výsledným cestám a u niektorých algoritmov k výraznému zlepšeniu časov. Voľba pevnej dĺžky hrán cyklu je zámerná, pretože ak by bola volená náhodne, ako pri ostatných hranách, výsledný graf by bol takmer úplne náhodný a pre väčšinu algoritmov jednoduchý.



O5.2.2a Nakreslenie problému vľavo pre **n** *= 10,* **m** *= 10 a* **ch** *= 10,*

vpravo pre **n** *= 21,* **m** *= 24 a* **ch** *= 6.*

Zvyšný počet požadovaných hrán, ktoré požaduje parameter **m**, je tvorený náhodne pridávanými hranami. Počiatočný a koncový vrchol náhodnej hrany sa vyberá z intervalu 1 až **n**. Dĺžka hrany je volená náhodne z intervalu určeného parametrami **im** a **il**. Ku dĺžke hrany l_e kde $e=(i, j)$, $1 \leq i \leq \mathbf{n}$, $1 \leq j \leq \mathbf{n}$, je možné pripočítať hodnotu modifikovanú lineárnym a kvadratickým multiplikátorom **ln** a **ls**. Dĺžka hrany sa pre nenulovú hodnotu multiplikátoru vypočíta ako $l=l_e+l_e \cdot multiplikátor \cdot |j-i|$ pre lineárny multiplikátor a $l=l_e+l_e \cdot multiplikátor \cdot |j-i|^2$ pre kvadratický multiplikátor.



O5.2.2b Nakreslenie problému pre **n** *= 45 a* **m** *= 60.*

Generovanie dĺžok hrán upravených pomocou potenciálov je možné povoliť parametrom **p**. Význam potenciálov pri tvorbe problému je popísaný pri generátore Grid. Zhodný je taktiež význam parametrov **pl** a **pm**, udávajú hornú a dolnú hranicu potenciálu a parametrov **pn** a **ps**, ktoré určujú hodnotu lineárneho a kvadratického multiplikátoru pre potenciál. Generátor Rand navyše používa parametre **pap** a **pac**. Hodnota **pap** je celé číslo z intervalu 0 až 100 a udáva podiel alternatívnych potenciálov v % z celkového počtu vrcholov. Hodnota **pac** je ľubovolné reálne číslo. Ak je hodnota **pap** > 0, približne **pap**% vrcholov výslednej siete bude mať svoj potenciál $P'(v)=P(v)\cdot$**pac**. Implicitná hodnota pre **pap** je 0 a pre **pac** -1.

Za zdroj sa volí vždy vrchol s číslom 1, okrem výstupov s umelým zdrojom, kedy je to **n**+1. Umelý zdroj sa pridáva parametrom **s**. Prípojné hrany zdroja sú implicitne vytvárané o dĺžke 10^8, túto hodnotu možno zmeniť parametrami **sm** a **sl**, ktoré udávajú minimum a maximum dĺžky prípojnej hrany. Dĺžky prípojných hrán sú volené zo zadaného rozsahu náhodne. Výnimkou je hrana smerujúca k pôvodnému zdroju, ktorá má dĺžku vždy 0.

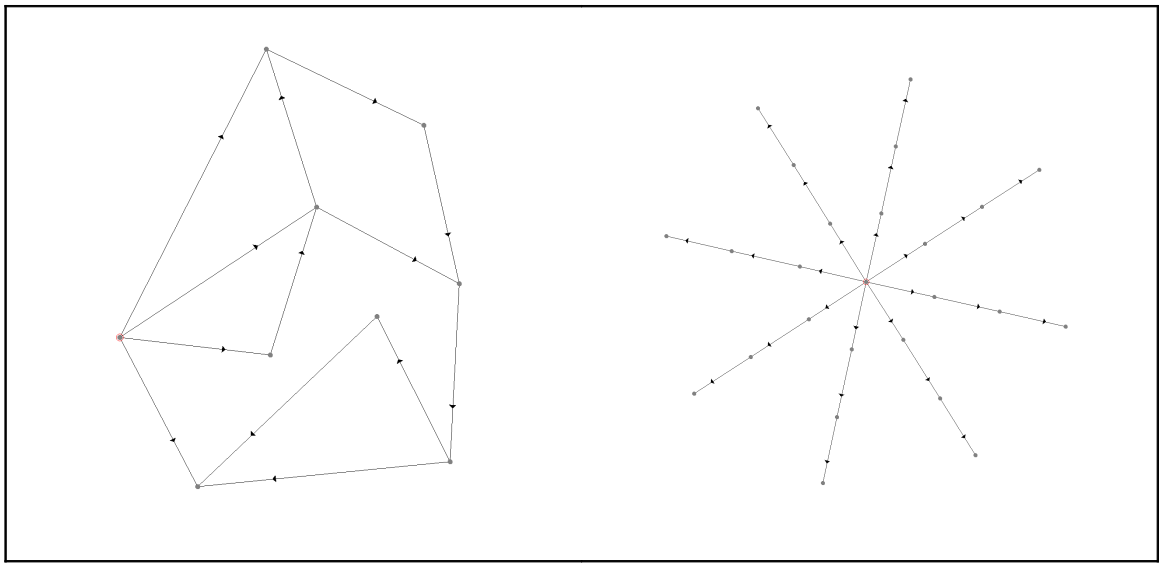
5.3 Generátor Acyc

5.3.1 Pôvod generátora

Acyc je tretí generátor, ktorý pochádza z práce [1]. Tento generátor je upravenou verziou Rand a podporuje tie isté parametre. Význam väčšiny parametrov zostáva oproti Rand nezmenený a popísané sú iba rozdiely. Význam acyklických sietí pri experimentoch s algoritmami pre hľadanie najkratších ciest nieje zanedbateľný. Existuje rad praktických problémov ktoré vyžadujú riešenie acyklických sietí alebo sietí s veľkými acyklickými podgrafmi (elektrická sieť, work flow - hľadanie kritickej cesty).

5.3.2 Parametre a vlastnosti výstupu

Význam povinných parametrov **seed**, **n** a **m** zostáva nezmenený. Podobne ako v pôvodnom Rand je vždy vytvorené prepojenie všetkých vrcholov. Bez nastavenia parametru **ch**, sú prepojené všetky vrcholy na jednej ceste začínajúcej v zdroji. Ak je uvedený **ch**, dĺžka cesty je obmedzená na maximálne **ch** vrcholov, pričom prvý je vždy zdroj. Pre **ch** < **n**-1 je teda vytvorených viacero ciest obsahujúcich **ch** vrcholov a **ch**-1 hrán. Posledná cesta môže byť kratšia. Parameter **cl** určuje dĺžku hrán na prepojujúcich cestách.



O5.3.2 Vľavo ukážka výstupu pre **n** *= 9 a* **m** *= 16. Vpravo pre* **n** *= 25,* **m** *= 24 a* **ch** *= 3.*

5.4 Generátor State

5.4.1 Pôvod generátora

Na rozdiel od ostatných použitých generátorov, State využíva reálne dáta. Jedinou premenlivou zložkou vstupu je počiatočný vrchol. Dáta pochádzajú z [43], použité súbory popisuje tabuľka T5.1a. Pôvodný zdroj je US Census Bureau – TIGER/Line®, údaje sú prevedené do formátu DIMACS [46].

O5.4.1 Zobrazuje nakreslenie pre Missouri (nieje samostatne zahrnutý v testovacej sade). Vyššia hustota (napravo) reprezentuje približnú hustotu použitých vstupov. Ilustrácia pochádza z [47].

5.4.2 Parametre a vlastnosti výstupu

Parametre generátoru sú **state** a **seed**. Hodnota **state** udáva kódom mapu ktorá sa použije. Prípustné kódy definuje tabuľka T5.1a. Ak je hodnota seed medzi 1 až *n* pre konkrétnu sieť, použije sa ako číslo počiatočného vrcholu. V prípade že hodnota prekračuje počet vrcholov, použije sa hodnota seed modulo *n* navýšená o 1.

Dĺžky hrán sú vždy kladné a nenulové, reprezentujú skutočné vzdialenosti v celých číslach. Jednotka dĺžky nie je explicitne uvedená, ale z hľadiska algoritmov nie je dôležitá. Skutočné vzdialenosti sú prepočítané na celé čísla pomocou vzorca $l(e)=\lfloor dĺžka*10+0.5 \rfloor$. Grafy neobsahujú slučky, ale môžu obsahovať násobné hrany. Všetky cesty sú obojsmerné, v rámci vstupného formátu pre orientované grafy, je preto každá hrana zdvojená jej obrátenou hranou rovnakej dĺžky, t. j. $\forall e \in E, e=(u,v) \exists f=(v,u): f \in E \wedge l(e)=l(f)$. Graf je súvislý, existuje (orientovaná) cesta medzi každou dvojicou vrcholov.

Kód	Anglický názov	Slovenský názov	Počet vrcholov	Počet hrán
CTR	Central USA	Stredná časť USA	14081816	34292496
W	Western USA	Západ USA	6262104	15248146
E	Eastern USA	Východ USA	3598623	8778114
LKS	Great Lakes	Veľké kanadské jazerá	2758119	6885658

CAL	California and Nevada	Kalifornia a Nevada	1890815	4657742
NE	Northeast USA	Severo-Východ USA	1524453	3897636
NW	Northwest USA	Severo-Západ USA	1207945	2840208
FLA	Florida	Florida	1070376	2712798
COL	Colorado	Kolorádo	435666	1057066
BAY	San Francisco Bay Area	Prístavná časť, San Francisco	321270	800172
NY	New York City	Mesto New York	264346	733846

T5.4.2 Zoznam použitých máp zoradený podľa počtu vrcholov.

Počty vrcholov a hrán sú odstupňované tak, aby bolo možné skúmanie rastu časových nárokov pre rastúce veľkosti vstupu. Do zostavy nie sú zaradené všetky štáty, ale menšie a väčšie celky. Počet hrán na najkratšej ceste z náhodne zvoleného zdroja, v týchto grafoch, dosahuje hodnoty rádovo 10^3, už pri najmenšej veľkosti vstupu (NY).

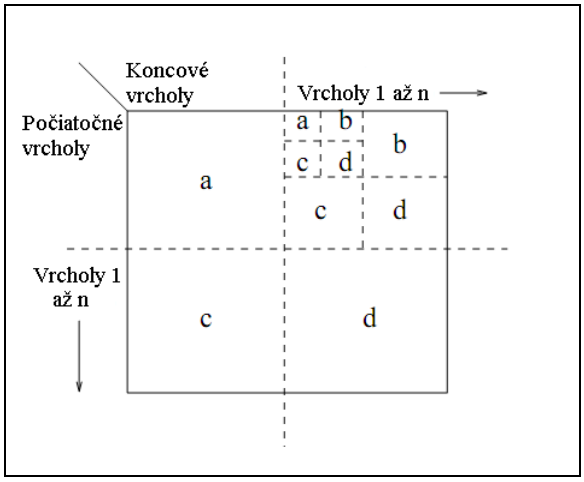
5.5 Generátor R-MAT

5.5.1 Pôvod generátora

Implementácia generátoru pochádza z balíku GTgraph [40]. Podrobný popis generátoru a sietí ktoré modeluje, je možné nájsť v [56]. Názov je skratkou „Recursive Matrix“, čo vystihuje základný princíp jeho fungovania. V závislosti od vstupných parametrov generuje bezškálové siete, ktoré aproximujú topológiu internetu, odkazov WWW, sociálnych sietí a podobných.

5.5.2 Parametre a vlastnosti výstupu

Generátor je navrhnutý zámerne tak, aby pracoval s malým počtom parametrov. Počet vrcholov grafu – **n**, by mal byť mocninou 2, aby pri rozdeľovaní matice susednosti nevznikali osamotené vrcholy. Parameter **m** udáva počet hrán grafu. Ako bude popísané ďalej, tento počet slúži iba pre prvú fázu generovania grafu, vo výslednom grafe bude počet približne dvojnásobný. Pre parameter **m** je vhodné, ak je párnou mocninou 2. Parameter **seed** udáva číslo počiatočného vrcholu, ak je vyšší než **n**, použije sa hodnota **seed** modulo **n** + 1. Minimálna dĺžka hrany je udaná parametrom **i**, maximálna dĺžka je 4 násobok hodnoty parametru **z**. Ako váhy pre generátor slúžia parametre **a**, **b**, **c** a **d**, ktorých hodnoty sú reálne čísla z intervalu 0.0 až 1.0 a zároveň **a**+**b**+**c**+**d**=1.0.

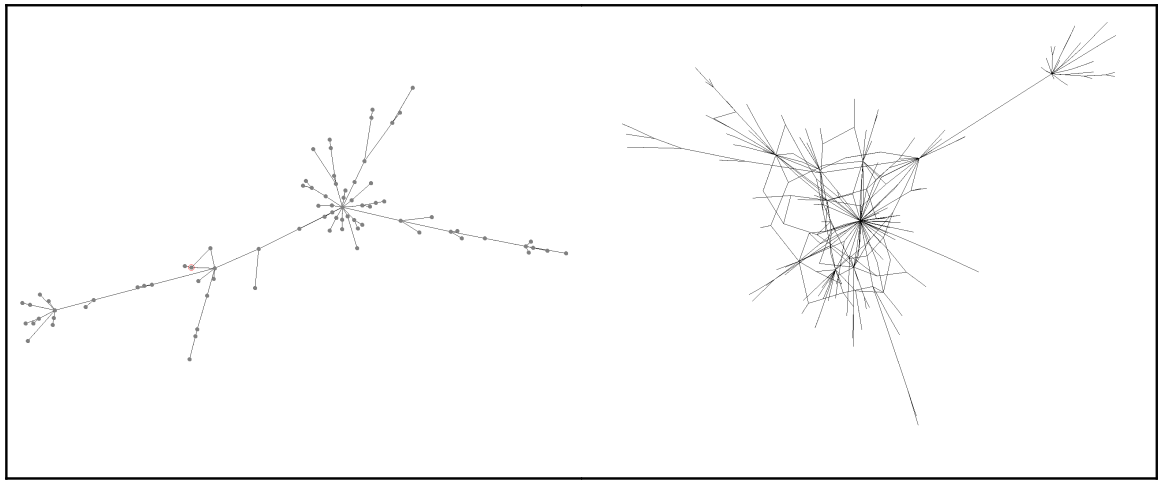


O5.5.2a Rozdelenie **m** *hrán do matice susednosti rekurzívnym delením. Ilustrácia - [56].*

Význam reálnych parametrov znázorňuje obrázok O5.5.2a, pomyselná matica susednosti pre požadovaný graf je rekurzívne delená na štvrtiny. Hrany, ktorých počet špecifikuje parameter **m**, sú rozdeľované do každej časti podľa váh **a**, **b**, **c**, **d**.

Výsledný graf nemusí byť za takýchto okolností dosiahnuteľný z počiatočného vrcholu, dokonca môže obsahovať niekoľko samostatných komponent, medzi ktorými nevedie žiadna hrana. Na výsledok generovania pomocou váh je aplikovaný nasledujúci postup:

1. Pre každú hranu $e_1=(u, v)$ s váhou $l(e_1)$ je pridaná hrana $e_2=(v, u)$ s váhou $l(e_2) = 4 \cdot l(e_1)$.
2. Pre každý vrchol u, v ktorom nezačína žiadna hrana, je náhodne vybraný vrchol v a sú pridané hrany $e_1 = (u, v)$ a $e_2 = (v, u)$, $l(e_1) = l(e_2) = 1$.
3. V grafe sú nájdené samostatné komponenty. Pre každú z nich je vybraný zástupca s najnižším číslom vrcholu. Všetci zástupcovia komponent sú pripojení ku zástupcovi prvej komponenty, ako v bode 2.



O5.5.2b Nakreslenie problému pre **n**=*80,* **m**=*40 vľavo a* **n**=**m**=2^8 *vpravo, v oboch prípadoch je* **a**=**c**=*0.05 a* **b**=**d**=*0.45.*

Pridávanie hrán bodom 2 sa, pri dodržaní odporúčania pre hodnotu **n** a dostatočne vysokej hodnote **m**, vyskytuje s nízkou pravdepodobnosťou. Vzhľadom ku zdvojeniu hrán, ktoré ho predchádza, ide vždy o osamotené vrcholy. Popísaný postup vytvára sieť, v ktorej existuje cesta medzi každými dvomi vrcholmi, spätné hrany sú však penalizované štvornásobkom dĺžky pôvodnej hrany. Väčšie komponenty a osamotené samostatné body sú pripojené obojsmernými, lacnými hranami.

Generovaný problém má simulovať komunikačnú sieť. Ceny hrán môžu napríklad reprezentovať čas, ktorý je potrebný na prenesenie najmenšieho balíka údajov (latencia komunikácie).

Východzie hodnoty parametrov pre generátor R-MAT sú **n**=**m**=2^8, **seed**=111, **i**=1, **z**=100, **a**=0.45, **b**=**c**=0.15, **d**=0.25.

5.6 Generátor Erdős–Rényi

5.6.1 Pôvod generátora

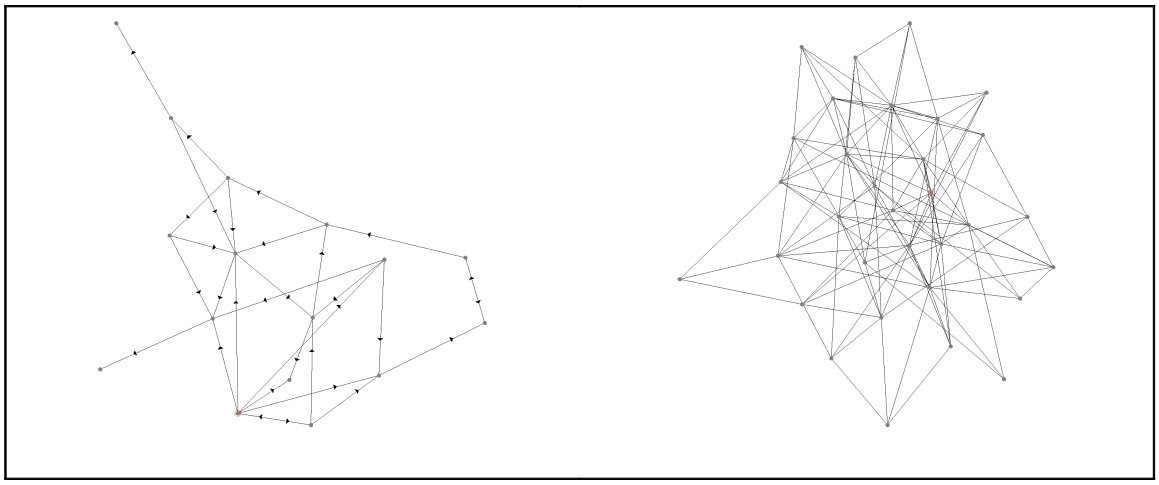
Implementácia je súčasťou sady GTgraph [40], použitý je ten istý spustiteľný súbor, ako pre R-MAT, ktorý pre nastavenie všetkých váh na ¼ a vylúčení násobných hrán, generuje výstup

podľa modelu Erdős–Rényi [56]. Náhodné grafy tohto typu sú často používané v teórii, od inštancii z reálneho sveta sa však výrazne líšia, čo je ukázané aj odlišným správaním algoritmov v experimentoch.

***Poznámka**: Generátor ER je odlišný od Rand. Kým Rand prepája vrcholy do cesty, alebo kružnice s krátkymi hranami, aby zaistil dostupnosť vrcholov, tento generátor pridáva do náhodného grafu iba hrany, ktoré prepájajú zdroj so všetkými komponentami. Prepojenie je typicky potrebné iba v riedkych grafoch, kde je počet pridaných hrán maximálne n-1, v hustých grafoch s vysokou pravdepodobnosťou nie sú pridávané žiadne hrany.*

5.6.2 Parametre a vlastnosti výstupu

Model Erdős–Rényi, podľa autorov Paul Erdős a Alfréd Rényi, definuje dva úzko prepojené varianty. Jedným z nich je $G(n, m)$, ktorý vyberá náhodnú inštanciu z množiny všetkých inštancií náhodných grafov na n vrcholoch a m hranách. Keďže pre účely práce je už použitý generátor podľa modelu R-MAT, ktorý zahŕňa i model Erdős–Rényi, je použitý ten istý generátor. Parametre generátoru pre počet vrcholov a hrán zostávajú rovnaké – **n** a **m**. Počiatočný vrchol je zadaný rovnako, pomocou **seed**.



O5.6.2 Nakreslenie náhodného grafu pre **n***=16,* **m***=30 vľavo a* **n***=32,* **m***=128 vpravo.*

Maximálna a minimálna dĺžka hrany je zadaná parametrami **z** a **i**. Všetky hodnoty parametrov sú kladné, celé čísla. Pôvodné parametre **a**, **b**, **c**, **d** u R-MAT sú implicitne nastavené na **a**=**b**=**c**=**d**=0.25. Z postupu, ktorý popisujú body 1 až 3 generátora R-MAT je vykonaný iba bod 3, aby bola zaistená dostupnosť všetkých vrcholov zo zdroja, pre riešenie úlohy SSSP.

Štruktúra náhodných grafov nieje z praktického hľadiska príliš zaujímavá, pretože sa vyskytuje iba zriedka. Náhodné grafy majú navyše pomerne jednoduchú štruktúru z hľadiska hľadania najkratších ciest. Pretože hrany nerešpektujú žiadnu hierarchiu vrcholov a naviac smerujú zhruba s rovnakou pravdepodobnosťou do všetkých častí grafu, najkratšie cesty sú pomerne krátke a majú malý počet hrán.

5.7 Generátor SSCA#2

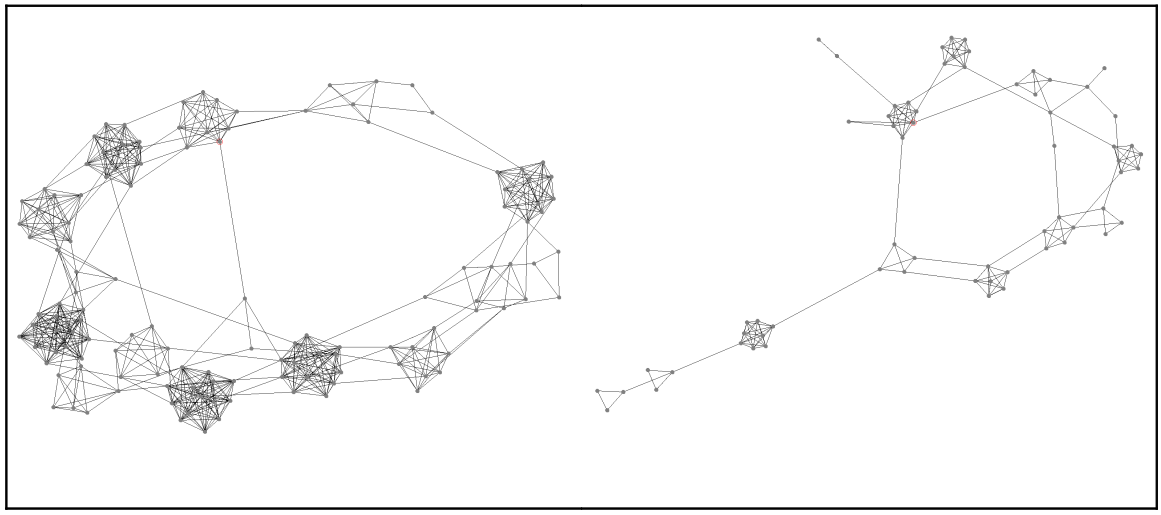
5.7.1 Pôvod generátora

Generátor SSCA#2 je súčasťou sady testov „Scalable Synthetic Compact Applications“ (SSCA), vyvinutej v rámci programu DARPA: „High Productivity Computing Systems“ (HPCS). Ide teda o benchmark určený pre superpočítače, štruktúra generovanej siete pre testy

je však zaujímavá i pre testovanie sekvenčných algoritmov. Zdrojový kód, prispôsobený pre beh na systémoch Unix/Linux, je dostupný v rámci [40]. Podrobnejší popis testov a grafovej štruktúry ktorú SSCA#2 generuje, je možné nájsť v prácach [54] a [55].

5.7.2 Parametre a vlastnosti výstupu

Výstupné siete možno v krátkosti popísať ako náhodne prepojené kliky rôznej veľkosti. Klika veľkosti k, v rámci orientovaného grafu, obsahuje k^2 hrán. Všetky hrany sú pridávané ako dvojice, takže pre (u, v) je zahrnutá aj hrana (v, u), je však možné definovanie pravdepodobnosti, že sa opačná hrana nezahrnie. Prepojenie klík je dané taktiež pravdepodobnosťou pre prepojenie každej dvojice. Dĺžky hrán sú celočíselné, volené náhodne z intervalu. Základný generátor nezaisťuje existenciu orientovanej cesty medzi zdrojom a každým vrcholom, preto je vygenerovaný graf doplnený o prídavné hrany - zo zdroja do každej komponenty.



O5.7.2 Nakreslenie grafu pre 64 vrcholov a maximálnu veľkosť kliky 8 vpravo a pre 128 vrcholov a maximálnu veľkosť kliky 16 vľavo.

Parametre algoritmu pre počet vrcholov, maximálnu veľkosť kliky a maximálnu dĺžku hrany je možné definovať súčasne, parametrom **SCALE**. Počet vrcholov a maximálna dĺžka hrany sa podľa **SCALE** určí ako $2^{\mathbf{SCALE}}$, počet vrcholov najväčšej kliky je $\lfloor 2^{(SCALE/3)} \rfloor$. Samostatne je možné určiť počet vrcholov grafu pomocou parametru **n**. Veľkosť kliky sa určí parametrom **klika** a maximálna dĺžka hrany parametrom **z**. Minimálna dĺžka hrany sa určuje parametrom **i**. Pre záporné hodnoty **i** nie sú vylúčené cykly zápornej dĺžky. Pomocou parametru **r** je možné určiť maximálny počet paralelných hrán ktoré sa vygenerujú pre tú istú dvojicu vrcholov. Všetky doteraz uvedené parametre sú celočíselné. Parametre **p** a **q** sú reálne, z intervalu 0.0 až 1.0, **p** udáva pravdepodobnosť, že bude vytvorená hrana jednosmerná, **q** určuje pravdepodobnosť prepojenia pre každú dvojicu klík. Implicitné hodnoty sú **n**=2^{16}, **klika**=40, **i**=0, **z**=2^{16}, **p**=0.2, **q**=0.5, **r**=3. Parameter **seed** slúži pre výber počiatočného vrcholu, implicitne je táto hodnota 111, ak je **n** menšie ako **seed**, použije sa **seed** modulo **n**+1.

6 Testy

Veľká časť podobných štúdií, používa ako hlavné kritérium odmerané časy spotrebované riešením. I keď táto miera nie je všeobecne platná, poskytuje možnosť priameho porovnania výsledkov rôznych platforiem (GPU a CPU), u ktorých sa diametrálne líši cena operácií. Výsledky uvedené v tejto kapitole, sú získané meraním času, ktorý spotrebujú jednotlivé implementácie riešením vstupnej úlohy tak, ako to popisuje kapitola 4.3.2. Na rovnakých vstupoch a implementáciách, ako sú použité v 6.1, testuje algoritmy i práca [1], kde sú sledované okrem času, i počty skenovaní pre každý vrchol. Porovnanie výsledkov kapitoly 6.1 s pôvodnou prácou, vedie k záveru, že sú stále aktuálne, i napriek veľkému rozdielu vo výkone použitých platforiem. Podľa meraní 6.2.2, oproti pôvodným výsledkom, došlo ku zrýchleniu implementácií o 75 až 120 násobok (v prípade DIKH o 135 násobok).

Všetky uvedené hodnoty vo výsledkoch, boli získané pomocou 25 meraní. Pre ten istý vstup sa každé meranie opakovalo 5x a bol vybraný najmenší čas. Tento postup sa opakoval pre 5 rôznych vstupov. Pre všetkých, takmer 3000 výsledných hodnôt, bola vypočítaná hodnota výberovej smerodajnej odchýlky, minimum a maximum, pre žiadny z bodov, nie je hodnota odchýlky vyššia než 60% z hodnoty príslušného priemeru. Pre viac než 96% hodnôt je odchýlka menšia než 15% z hodnoty priemeru, výsledky, kde prekračuje 15%, sú zvýraznené v tabuľkách červenou farbou. Popísané parametre výsledkov sú zachytené v grafoch OA.38a a OA.38b.

Použitie piatich výsledkov pre priemery, nie je zo štatistického hľadiska ideálne, avšak vo väčšine prípadov sú vysoké hodnoty smerodajnej odchýlky spojené s výrazne horším chovaním algoritmu na konkrétnom vstupe, keď aj minimálna hodnota o niekoľko rádov prekračuje maximálne časy najlepších implementácií. Výsledné poradie algoritmov, ale i namerané hodnoty sú navyše konzistentné, v zmysle 1. nasledovania trendov pre rastúce veľkosti vstupu, 2. opakované vyhodnotenia testov a predbežných testov, 3. porovnania výsledkov s predošlými štúdiami. Napríklad v A.17, A.18 a A19 sa výsledky prekrývajú v prvých a posledných 2 bodoch pre nasledujúce grafy, vo všetkých prípadoch ide o novú sadu výsledkov získanú pomocou iných piatich vstupov tej istej triedy – hodnoty nadväzujú s vysokou presnosťou. Vyhodnotenie experimentov v tej podobe, ako je uvedené, trvalo spolu s generovaním vstupov niekoľko týždňov čistého času. Ďalšie vyhodnocovanie, napríklad pre 10 rôznych vstupov na každý výsledok, by predĺžilo vyhodnotenie o ďalšie týždne, pravdepodobne by však neviedlo k zmene vyvodených záverov.

Popisy testov, obsiahnuté v ďalších častiach kapitoly, upresňujú vstupné parametre a upozorňujú na rôzne zistenia. Význam popisovaných parametrov v kontexte konkrétneho

generátora je nutné hľadať v predošlej kapitole. Zhrnutie záverov a odporúčania, plynúce z výsledkov testov, sú uvedené v poslednej kapitole (7).

V prípadoch, keď nie je zrejmý dôvod použitia konkrétnych hodnôt parametrov, ide vždy o experimentálne získané hodnoty, ktoré poskytujú dobrú možnosť rozlíšenia implementácií. Niektoré testy sú založené na skutočných dátach z reálneho sveta, alebo na aproximácii (State, R-MAT), iné slúžia ako syntetické testy, ktoré odhaľujú slabé a silné stránky algoritmov (PHard, Acyc-Neg). Obsiahnuté testy na náhodných sieťach (Rand, ER), umožňujú posúdenie relevantnosti teoretického skúmania algoritmov na náhodných grafoch, v kontexte výsledkov ostatných testov.

6.1 Reprodukcia testov SpLib

Testy obsiahnuté v tejto kapitole, popisuje práca [1]. Názov SpLib pochádza z názvu archívu s pôvodnými implementáciami. Zdrojové kódy a stručnú dokumentáciu ponúka na stiahnutie jeden z autorov, Andrew Goldberg, na svojich stránkach [41], taktiež sú v prílohe B.

Rozsahom testovaných implementácií a širokým záberom testov bola práca [1] v čase publikácie výnimočná. Všetky implementácie boli navyše vytvorené jednotne a používajú vstupný formát DIMACS [43], ktorý sa stal štandardom pre vstupný formát mnohých testovacích implementácií grafových algoritmov. Výsledky tejto práce sú odkazované aj v neskorších prácach, napríklad [42]. Pre zaradenie do širšieho kontextu, nadviazanie a zároveň aktualizáciu výsledkov, boli do tejto práce prebrané implementácie, generátory aj testy. Ako vidno ďalej, ani desaťročia zlepšovania hardvéru, nespravili z horších implementácií lepšie a zvyčajne ani použiteľné. Nezmenilo sa ani konečné poradie testovaných algoritmov.

Tam, kde to malo zmysel, boli testy rozšírené o väčší rozsah vstupných dát, prípadne boli zjemnené pridaním ďalších testov. Aktuálne výsledky testov 6.1.5 ukázali niektoré zaujímavé trendy pre Dijkstrov algoritmus s haldou, ktoré sa stali námetmi pre kapitolu 6.2. Všetky ostatné výsledky a plynúce závery, súhlasia so závermi pôvodnej štúdie [1]. Použité konštanty pre náhodný generátor (a teda vstupy) sú identické, pre rozširujúce merania boli náhodne vybrané nové.

6.1.1 Grid SSquare

Prvý test, založený na výstupe generátora Grid, je test na štvorcovej mriežke. V tomto teste je **X**=**Y** a počet vrcholov sa mení od 4097 po 4194305. Vyhodnotenie výsledkov vo forme grafu a tabuľky obsahuje príloha A.1. Najlepšie v tomto prípade dopadol algoritmus PAPE, podobne sa choval aj TWO_Q. Do dvojnásobku spotrebovaného času najlepšieho riešenia sa vošli ešte algoritmy THRESH, GOR a DIKBD. Rádovo horšie boli algoritmy GOR1 a BFP. Tento typ vstupu je pomerne jednoduchý, ale často využívaný, rovnako ako 6.1.3 a 6.1.4. Jednoduché mriežky sú v praxi základným typom aproximácie terénu.

6.1.2 Grid SSquare-S

Vychádza priamo z testu SSquare, do siete je však pridaný umelý zdroj, ktorý sa pripája ku každému vrcholu. Takto získané grafy majú navyše približne 30% hrán oproti vstupom predošlého testu. Ako vidno z výsledkov (A.2), pomerne malá zmena spôsobila zásadný obrat situácie. Najrýchlejšie algoritmy predošlého experimentu, PAPE a TWO_Q, nedobehli v časovom limite pre polovicu vstupov. Celkovo najlepší bol v tomto experimente DIKBD, do dvoj násobku času sa tesne zmestil iba GOR. Umelý zdroj spôsobuje problém najmä algoritmom, ktoré zlepšujú cesty opakovaným skenovaním (BFP, TWO_Q, PAPE), venujú totiž zbytočné úsilie zlepšovaniu ciest z vrcholov dosiahnutých iba cez „umelé“ hrany,

ktorých dĺžka je zámerne zvolená tak, aby sa v žiadnej výslednej ceste do vrcholu dosiahnuteľného v pôvodnej sieti nevyskytovali.

6.1.3 Grid SWide

Test na obdĺžnikovej mriežke s výrazným rozdielom v pomeroch strán. Testovaná mriežka má v každom meraní výšku **X**=16 a mení sa šírka **Y**. V tomto teste sú najrýchlejšie opäť TWO_Q a PAPE, ktoré sa s rastúcou veľkosťou vstupu k sebe navzájom približujú a zároveň sa vzďaľujú od svojich konkurentov. Žiadny algoritmus nemožno prehlásiť za nevhodný pre tento problém, nakoľko všetky implementácie sa vošli do päťnásobku času spotrebovaného najlepším algoritmom (pre najväčší rozsah úlohy). Konzistentne najhoršie časy pre všetky veľkosti vstupu dosahovala implementácia DIKH. (A.4) V prípade DIKH je vo všetkých testoch 6.1.1 až 6.1.3 pozorovateľný nepriaznivý efekt zaplnenia haldy, čo je hlavný dôvod straty na DIKBD. Pri problémoch, kde nedochádza k súčasnému dosiahnutiu veľkého počtu vrcholov si DIKH počína citeľne lepšie, ukazuje to nasledujúci test, ale i všetky testy „Hard“.

6.1.4 Grid SLong

Testuje rovnako ako SWide, na mriežke obdĺžnikového tvaru, pomery strán sú ale vymenené, v tomto prípade je pevné **Y**=16. Tento problém je pre algoritmy o niečo ťažší ako SWide. Najlepšie implementácie zostali PAPE a TWO_Q. Ostatné implementácie sa držia pohromade, s približne dvojnásobnými časmi. Ako nevhodná sa prejavila implementácia BFP, ktorá na najlepšie výsledky stráca viac, než pomerom 10^3. Približne desať násobné spomalenie taktiež zaznamenala implementácia GOR.

6.1.5 Grid PHard

Problémy PHard a NHard predstavujú ťažšie inštancie, ktoré dokáže vytvárať generátor Grid. Sieť je založená na podobnom základe ako problémy SLong, **Y**=32. Vrcholy v rámci jednej vrstvy sú prepojené do kruhu a vrstvy navzájom sú pripojené jednotkovými hranami. To čo robí tieto vstupy ťažkými, sú prídavné hrany medzi vrstvami smerujúce z nižšej vrstvy do vyššej (ako popisuje kapitola 5.1), ktoré majú väčšiu dĺžku. V konečnom dôsledku sú tak dosiahnuté vrcholy, do ktorých ale najkratšia cesta vedie cez kratšie hrany základnej mriežky. Niektoré algoritmy pri týchto vstupoch strácajú čas opakovaným prehľadávaním tých istých vrcholov. Výsledky sú uvedené v A.5 (menšie rozsahy problémov), A.6 (pôvodný rozsah).

Už pre malé rozsahy vstupov rádovo strácajú algoritmy PAPE a TWO_Q oproti ostatným riešeniam. Na pomerne malých vstupoch, v rozsahu niekoľko tisíc vrcholov, dosahujú časy merateľné v sekundách, čo je oproti najlepším implementáciám DIKBD a DIKH, horšie o viac než 10^3. Najlepšie testované algoritmy sú DIKBD a DIKH, kde DIKH konverguje k časom DIKBD pre rastúce vstupy. Doplňujúce testy, pre ešte väčšie rozsahy, uvedené v A.15, ukázali, že na veľkých inštanciách tohto typu, od 10^6 vrcholov, je najrýchlejšia implementácia DIKBM.

6.1.6 Grid NHard

Je založený na rovnakých parametroch ako PHard, výnimkou je iba rozsah pre dĺžky prídavných hrán. Hrany pridávané medzi vrstvy sú volené výlučne záporné. Výsledky pre tento test sú uvedené v A.7. Test je jedným z najťažších, problém majú aj implementácie Dijkstrovho algoritmu. Tie vsádzajú na výber vrcholu s najmenšou dosiahnutou vzdialenosťou od počiatku, čo v prípade záporných hrán nezabezpečuje jediné spracovanie každého vrcholu, ako je to pri sieťach s nezápornými hranami.

Ako vidno z výsledkov, jediná rozumná alternatíva pre tento problém je niektorá z implementácií GOR alebo GOR1. Obe varianty škálujú rovnako, GOR1 je približne o 50%

rýchlejší. Ostatné algoritmy nedokážu do časového limitu spočítať ani druhý najväčší testovaný vstup, implementácia DIKH dokonca nedokáže do časového limitu spočítať ani najmenší zo vstupov. Algoritmus TWO_Q je jediný (okrem GOR a GOR1), ktorý dopočíta výsledok aj pre najväčší rozsah, je však takmer o dva rády pomalší.

6.1.7 Rand-4

V tomto teste (popis generátoru Rand je uvedený v 5.1) je zvolené **m**=4**n**, čo v priemere znamená, že z každého vrcholu smerujú 4 hrany. Jedná sa teda o riedku sieť. Dĺžky náhodných hrán sú volené z intervalu 0 až 10^4, ako aj v ostatných testoch, okrem Rand-Len.

Výsledky sumarizuje príloha A.8, všetky implementácie sú pre tento typ problému použiteľné. Najrýchlejším algoritmom je DIKBD, dvojnásobne pomalší je DIKH. Ostatné algoritmy držia krok a sú 3 až 6 krát pomalšie ako DIKBD.

Oproti pôvodným výsledkom uvedeným v [1], je od istého rozsahu vstupu (32, resp. 64 tisíc vrcholov) možné pozorovať prudší nárast spotrebovaného času pre väčšie vstupy. Tento jav môže byť spôsobený vplyvom cache procesoru, keď sa pre menšie vstupy, vďaka nízkemu počtu hrán, zmestí celý zoznam nasledovníkov do cache pamäte. V pôvodných experimentoch bol použitý procesor SPARCstation 10 Model 41, ktorý obsahoval iba 1MB externej cache [45], oproti 6MB integrovanej cache u Core 2 Duo [39].

6.1.8 Rand 1:4

Je zástupcom testov na hustých grafoch, z teoretického maxima n^2 hrán v orientovanom grafe (ak neuvažujeme násobné hrany) je vybraná ¼. Výsledky sú zhrnuté v A.9, ako vidno, DIKH sa na týchto vstupoch vyrovná DIKBD a na väčších rozsahoch sa dostáva do vedenia. Oba algoritmy sú ale prakticky rovnocenné. Všetky algoritmy na tomto vstupe dosahujú dobré časy, v rozmedzí do 10 násobku času spotrebovaného najlepším riešením.

Z oboch testov na náhodných grafoch je vidno, že sú pomerne ľahké pre všetky testované implementácie. Testované vstupy neposkytujú ani žiadnu možnosť spoľahlivého rozlíšenia efektivity algoritmu, pretože všetky implementácie sa s rastúcim vstupom chovajú podobne. Neoptimalizovaná implementácia lepšieho algoritmu a optimalizovaná implementácia horšieho, by mohli prehodiť výsledné poradie. Dôvodom je rýchle orezanie množiny prehľadávaných vrcholov, keď je spojený takmer „každý s každým“. Prehľadávanie z nevýhodných vrcholov je tak efektívne zastavené pomerne rýchlo. (Z iného uhla pohľadu sa dá hovoriť o typicky nízkom počte hrán na najkratších cestách.)

6.1.9 Rand-Len

Tento test je príkladom toho, ako sa môže v prípade niektorých algoritmov efektivita radikálne meniť v závislosti od počtu hrán na najkratších cestách. Všetky merania sú vykonané na vstupe rovnakom ako v Rand 4, o 131072 vrcholoch a 524288 hranách. Premenlivá je iba horná hranica dĺžky náhodnej hrany. Výsledky obsahuje príloha A.10. Časy sú nanesené v lineárnej mierke.

Najlepšie implementácie v tomto teste sú DIKBD a DIKH, ktoré na rastúcu hornú hranicu zareagovali dokonca miernym skrátením doby behu. Ostatné implementácie sa chovali presne naopak a s rastúcou hornou hranicou pre dĺžku hrany, stúpal aj čas potrebný pre ich beh. Zaujímavý efekt je pozorovateľný u implementácie GOR1, ktorá pre veľmi veľké povolené dĺžky hrán zase zlepšuje svoj čas. Celkovo však GOR1 skončil 5 násobne horšie ako DIKH a ostatné implementácie svoj čas zhoršili viac ako desať násobne.

Dôvodom predlžovania najkratších ciest, čo do počtu hrán, je základný cyklus, ktorý prepája hrany jednotkovými hranami. Pri predĺžení náhodných hrán, sú tak opakované skenovania

algoritmov ako BFP či TWO_Q väčšinou zbytočné, pretože vrcholy dosiahnuté náhodnými hranami budú neskôr dosiahnuté pomocou krátkych hrán cyklu a znovu skenované.

6.1.10 Rand P

Testuje algoritmy na podobnom princípe ako Rand-Len, s tým rozdielom, že horná hranica udáva maximálnu hodnotu pre potenciál vrcholu. O generovaní problému pomocou potenciálov hovorí kapitola 5.1 a ďalšie informácie je možné nájsť v [1]. Vstupný problém má rovnako 131072 vrcholov a 524288 hrán. Výsledky v A.11 jasne ukazujú odolnosť niektorých algoritmov vzhľadom ku hodnotám potenciálov, čo je v práci [1] aj teoreticky dokázané. Pri hodnotách hornej hranice potenciálov nad 10^4 sa už začínajú naplno prejavovať negatívne dĺžky náhodných hrán, čo zapríčiňuje náhle spomalenie riešení DIKBD, DIKH a THRESH.

Pre hodnoty potenciálov do hornej hranice pre dĺžku hrany, sú najlepšie implementácie DIKH a DIKBD. Pre hodnoty nad touto hranicou je najlepšia implementácia GOR1, ktorá pracuje nad vstupným problémom pre všetky hodnoty potenciálov rovnako. V tomto teste sú experimentálne jasne identifikované algoritmy, ktoré pracujú nad topológiou grafu bez využitia momentálnej hodnoty *td* pre riadenie výpočtu.

6.1.11 Acyc Pos

Prvý test na acyklických grafoch používa vstupy s nezápornými hranami. Hrany na ceste majú jednotkovú dĺžku a náhodné hrany sú dĺžky 0 až 10^4. Pre popis generátoru Acyc je možné nahliadnuť do kapitoly 5.1.Acyc alebo do práce [1]. Výsledky testu uvedené v A.12 ukazujú, že najlepším algoritmom na veľkých vstupoch je ACC, ktorý má (dokázanú) lineárnu zložitosť. Veľmi blízko efektivite ACC, na menších vstupoch dokonca niekedy lepšie, sú algoritmy DIKBD a GOR1. Podobne sa chová DIKH. Ostatné implementácie sú približne 10 násobne pomalšie. Najhoršie algoritmy sú BFP a GOR. GOR1 má oproti GOR, zlepšený horný odhad zložitosti pre acyklické grafy, čo sa v tomto teste aj experimentálne prejavilo.

6.1.12 Acyc Neg

Pri tomto teste sú všetky dĺžky hrán negatívne. Hrany základnej cesty majú dĺžku -1, dĺžky náhodných hrán sa volia z rozsahu -10^4 až 0. Riešenie sú vlastne najdlhšie cesty v sieti, kde sú dĺžky hrán násobené -1. Výsledky (A.13) ukazujú výrazné zhoršenie výkonu takmer všetkých algoritmov. Najrýchlejším je stále ACC, podobne rýchle sú GOR a GOR1. Niektoré algoritmy nedokázali spočítať výsledok v časovom limite ani pre druhý, v prípade DIKH pre tretí, najväčší testovaný vstup. Tento test je pomerne jednoduchou, ale zároveň veľmi efektívnou pripomienkou významu odhadu najhoršej časovej zložitosti, kde GOR1 a ACC sú stále zaručene lineárne, ostatné algoritmy túto výhodu nemajú.

6.1.13 Acyc P:N

Skúma podobne ako Rand-Len algoritmy na tom istom vstupe (čo do počtu počtu vrcholov a hrán), premenlivý je podiel hrán s nezápornou a zápornou dĺžkou. Výsledky A.14 obsahujú jemnejšie delenie ako pôvodná štúdia. Okrem problémov s malým podielom záporných hrán, kde sú dobré všetky implementácie, je najrýchlejší algoritmus ACC. Časové nároky ACC dokonca s rastúcim podielom záporných hrán mierne klesajú. Rovnako sa správa aj GOR1, ktorý kopíruje ACC, pričom je približne o 25% pomalší. Zaujímavé správanie, podobné chovaniu GOR1 pri Rand-Len možno pozorovať u algoritmu GOR. Časové nároky tohto algoritmu postupne stúpajú s rastúcim podielom záporných hrán až po 60%, pre 100% podiel však klesnú a GOR je mierne rýchlejší než GOR1. Pre tento test plynie rovnaké ponaučenie ako v prípade 6.1.12, ponúka aj odpoveď na správanie GOR v predošlom teste. Zložitosť GOR totiž nie je na rozdiel od GOR1 pre tento typ vstupu zaručene lineárna, v 6.1.12 sa však táto nevýhoda neprejavila, test 6.1.13 ponúka v tomto ohľade komplexnejší pohľad.

6.2 Dijkstrov algoritmus

Pre sekvenčné riešenie sietí s nezápornými hranami, sú stále najpoužívanejšie varianty Dijkstrovho algoritmu. Veľmi známa a zároveň pomerne efektívna implementácia využíva haldu pre hľadanie vrcholu s najmenšou hodnotou *td*. Táto časť testov je venovaná najmä implementácií DIKH na triede problémov Grid P Hard z 6.1.5 a podobných.

6.2.1 Grid P Hard D

Test efektívnejších implementácií Dijkstrovho algoritmu na kompletnej škále veľkostí vstupu pre triedu problémov Grid P Hard (A.15). Pre najmenšie vstupy je badateľný vplyv cache. Pre malé veľkosti vstupu sú najrýchlejšie implementácie DIKBM a DIKBD, ktoré sú stále najrýchlejšie pre veľké vstupy, tam sú už ale prakticky rovnako rýchle aj implementácie DIKR a DIKH. Verzia s Fibonacciho haldou (DIKF) je dvojnásobne pomalšia a verzia s rozsahovými priehradkami (DIKBA) zaostáva s trojnásobnými časmi. Pre tento, ale v zásade aj celú sériu testov časti 6.2, je hlavným ťažiskom zhodnotenie implementácií vnútorných štruktúr pre výber vrcholu s najmenšou hodnotou *td*, zvyšok kódu (a i priebeh výpočtu pre fixný vstup) je totiž úplne identický.

6.2.2 Zrýchlenie Grid P Hard

Vstupy pre tento test sú identické s testom 6.1.5, preto by nemalo byť potrebné ďalšie vyhodnotenie, avšak kvôli povahe skúmanej anomálie, boli testy vykonané aj na ďalších dvoch PC a pre istotu, bola znovu skompilovaná aj testovaná implementácia DIKH. Porovnanie nových výsledkov s pôvodnými, spočíva vo vyhodnotení identických vstupov (rovnaký seed a parametre generátora). Získané časy sú následne prepočítané na pomer zrýchlenia súčasnej platformy (A.16) oproti tej, ktorá bola použitá v [1]. Ak sú dáta pôvodnej štúdie správne, implementácia DIKH, ktorá zaostávala približne o 65% za najlepším DIKBD, je v súčasnosti takmer na tej istej úrovni. Rozdiel časov pre najväčšie vstupy je približne 4%.

6.2.3 Grid P Hard(er) D

Pre širšie porovnanie implementácií DIKH a DIKBD boli zostavené obmeny DIKH pre rôzne *k* haldy, pôvodný DIKH používa 4-regulárnu haldu, do testu sú pridané ešte DIKH2, DIKH8 a DIKH16. Ako vidno z výsledkov A.17, A.18 a A19, žiadna z verzií nieje rýchlejšia ako DIKBD, avšak v teste pre A.20 boli mierne upravené parametre testu a situácia je presne opačná. Zároveň je možné usúdiť, že implementácia DIKH4 – halda so stupňom 4, je najlepšia spomedzi testovaných háld pre tento prípad. Pre test Grid P Harder D boli zmenené parametre **y** = 16, **cl** = 1000, **ax** = 256, **ix** = 10 a **ih** = 2 oproti **y** = 32, **cl** = 1, **ax** = 64 a **ix**=**ih**=5. Zároveň boli zvýšené hodnoty **x**, aby vstupy obsahovali rovnaký počet vrcholov pri zmenenom **y**. Vstupná mriežka je teda oproti Grid P Hard užšia a dlhšia, obsahuje viacej náhodných hrán, ktoré sú navyše ohodnocované väčším rozsahom hodnôt a prepájajú vzdialenejšie vrstvy. Všetky implementácie na týchto vstupoch sú pomalšie (vstupy sú väčšie počtom hrán), DIKBD je však pomalší oproti všetkým verziám DIKH približne o 7% až 14%.

6.3 Cestné siete

Hľadanie najkratších ciest v cestných sieťach, testuje algoritmy azda na ich prvotnej doméne problémov. Výsledky nemožno v žiadnom prípade považovať priamo za výsledky na reálnych cestách, nakoľko u nich môžu platiť dopravné obmedzenia, ktoré by mohli zmeniť zásadným spôsobom topológiu. Testované vstupy sú i tak zaujímavé, už na najmenších vstupoch sa vyskytujú najkratšie cesty obsahujúce až 10^3 hrán, čo spôsobuje problémy napríklad pre BFP.

6.3.1 State Group 2

V tomto teste je použitá „druhá skupina“ algoritmov, ktorá obsahuje štvoricu najvhodnejších podľa predbežného testovania. Podrobnejší rozbor a viacej testov obsahuje [42, 47], odkiaľ pochádza tiež prvotný námet pre výber algoritmov. Pre každý zo vstupov je vykonaných 5 rôznych testov, líšia sa výberom počiatočného vrcholu. Stabilne najlepšou je implementácia DIKBA; DIKBD a THRESH sú približne rovnocenné, pričom THRESH získava najme na menších vstupoch a stráca na väčších (môže ísť o neoptimálne zvolené konštanty, na ktorých je vyváženie algoritmu založené). Vo väčšine testov je posledná implementácia TWO_Q, všetky implementácie sú však približne rovnocenné a žiadna nezaostáva za najlepšou o viac ako 100%. Výsledky testu sú obsiahnuté v A.21. Implementácia CUDASSSP sa ukazuje ako nevýhodná, je to spôsobené práve počtom hrán najkratších ciest.

6.4 Testy R-MAT, SSCA#2, ER

6.4.1 R-MAT 1:1

Význam vstupov R-MAT (ako popisuje aj 5.5) spočíva v aproximácií reálnych sietí podobných internetu, citáciám, sociálnym skupinám a podobným. Názov testu obsahuje „1:1“ ako naznačenie symetrie vstupu. Váhy pre generátor sú zvolené **a**=**c**=0.05, **b**=**d**=0.45. Počet základných hrán je navyše zhodný s počtom vrcholov, výsledné siete obsahujú približne troj násobok hrán ku počtu vrcholov. Ukážky vstupov pre toto rozloženie váh, obsahuje kapitola 5.5. Vrcholy sú rozdelené na dve množiny (prvých $n/2$ a druhých $n/2$ podľa čísla vrcholu), pričom 90% hrán smeruje medzi týmito množinami. Graf je teda takmer bipartitný, prepojeniam vrcholov v rámci jednej množiny je venovaných iba zvyšných 10% hrán.

Výsledky (A.22) na tomto type problému ukazujú, že najrýchlejšie časy dosahujú práve jednoduchšie implementácie BFP, TWO_Q a PAPE, ktoré skúmajú vrcholy v závislosti od zaradenia do fronty. Pomalšie sú implementácie DIKBD, THRESH, GOR, ktoré venujú isté výpočetné úsilie výberu najvhodnejšieho vrcholu na preskúmanie.

Celkovo je však možné považovať tento typ vstupu za ľahký, pretože žiadny z algoritmov nespotrebuje viac než štvor násobok času najlepšieho riešenia.

6.4.2 R-MAT r

Podľa [56], reálne siete je možné najlepšie aproximovať pomocou hodnôt parametrov **a**=0.45, **b**=**c**=0.15 a **d**=0.25. Použité vstupy obsahujú podobný (o málo nižší) počet hrán ku počtu vrcholov ako vstupy 6.4.1, zistené časy sú však naopak vyššie. Ako najlepšie implementácie sa stále javia TWO_Q a PAPE, v tomto prípade je však rovnako dobrý aj DIKBD. Do päť násobku času najlepšej implementácie sa zmestia všetky testované, najhoršie s viac než 4 násobným odstupom skončil GOR1. Hoci je tento test náročnejší než 6.4.1, celkovo sa javí (rovnako ako i predbežné testy s inými kombináciami parametrov) ako pomerne jednoduchý, všetky dobré implementácie predošlých testov dokážu vyriešiť vstupy tohto generátora v rozumnom čase.

6.4.3 SSCA#2 base (s)

V tomto teste sú použité základné nastavenia generátora (popísané v 5.7). Použité hodnoty **SCALE** sú 13 až 18, čo znamená vstupy o veľkosti 2^{13} až 2^{18} vrcholov. Pre test „SSCA#2 base s” (s = kratšie hrany) boli dĺžky hrán iba 1, 2 a 3, pre zachytenie tendencie CUDASSSP boli pridané vstupy o rozsahoch 2^{19} a 2^{20} vrcholov, maximálna veľkosť kliky bola tiež obmedzená na 32 vrcholov. V základnom teste (A.24) sa ukázali ako najlepšie implementácie DIKH,

DIKBD a TWO_Q, čo je nezvyklá kombinácia. Implementácia pre GPU, je pre tento typ vstupu nevhodná, dosahuje časy takmer 25x horšie ako DIKH. V teste zo skrátenými hranami (A.25), pracujú rýchlejšie všetky algoritmy. Rozdiel medzi najrýchlejšou a najpomalšou implementáciou je takmer všade do dvojnásobku času, nemá teda zmysel uprednostnenie žiadnej z implementácií. Algoritmus CUDASSSP prejavuje tendenciu dosahovať lepšie časy, na veľkých vstupoch, pre najväčšie 2 veľkosti je o 8% rýchlejší než druhý najlepší BFP a TWO_Q.

6.4.4 SSCA#2 snow (L)

Oba testy používajú rovnakú triedu vstupov, ktorá je podstatne odlišná od základnej verzie. Maximálna veľkosť kliky je obmedzená na 16 vrcholov, všetky hrany sú ale jednosmerné. Pravdepodobnosť prepojenia klík je nízka: **q**=5%. Dĺžky hrán sú volené z intervalu 1 až 10. Prvý test – SSCA#2 snow, porovnáva správanie algoritmov pre rôzne veľkosti vstupu (A.26). Najlepšie implementácie sú TWO_Q a BFP. CUDASSSP je pri menších vstupoch neefektívna, ale približne od 2^{16} vrcholov je najrýchlejšou. V druhom teste (SSCA#2 snow L, A.27) je veľkosť vstupu fixná, počet vrcholov je 2^{19}, mení sa maximálna povolená dĺžka hrany. V tomto prípade je najrýchlejšia implementácia všade CUDASSSP, tesne ju však nasleduje TWO_Q na CPU. Správanie CUDASSSP kopíruje BFP, ktorý je väčšinou o 30% pomalší. Pre rastúce dĺžky hrán je pozorovateľný mierny nárast časov pre všetky algoritmy, výraznejší prepad je iba u THRESH a DIKH, čo je veľmi odlišné správanie od 6.1.9, kde práve implementácie BFP a TWO_Q reagovali na predlžovanie hrán nepriaznivo a DIKH reagoval miernym zlepšením.

6.4.5 ER 4 a ¼

Testy ER sú doplňujúce ku Rand 4 (6.1.7) a Rand 1:4 (6.1.8), testujú rovnaké rozsahy vstupov (čo do počtu vrcholov a hrán), vstupy sú však úplne náhodné grafy. Výsledky testov pre ER ¼ (A.29) ukazujú, že výhoda DIKBD a DIKH na skutočne náhodných grafoch, nie je až tak veľká ako v 6.1.8. Tak isto aj najhoršia implementácia TWO_Q, ktorá zaostávala na Rand 1:4 so 7 násobným časom, je v ER ¼ iba o 80% pomalšia za prvým pre najväčšie vstupy – DIKH. V prípade testu ER 4, je situácia ešte priaznivejšia pre ostatné implementácie, keď najrýchlejšími sú PAPE, TWO_Q a BFP. Pôvodne najrýchlejšie DIKH a DIKBD si v týchto testoch o niečo pohoršili. Z celkového pohľadu sú však náhodné grafy ER ľahšími problémami než Rand, ani najpomalšie implementácie neprekračujú 3 násobok času spotrebovaného najlepším riešením.

6.5 Testy realizované na GPU (CUDASSSP)

6.5.1 CUDA Rand ¼

Test 6.1.8 (A.9), doplnený o jeden vstup dvojnásobnej veľkosti maximálneho vstupu pôvodného testu, ukazuje správanie GPU implementácie na hustom náhodnom grafe. Pre možnosť porovnania s nasledujúcimi testami 6.5.2 a 6.5.3 je hodnota **ll**=100 (oproti implicitným 10^4 u 6.1.8). Znížením hornej hranice dĺžky hrany, sa tak problém stáva o niečo ľahším pre implementáciu BFP; DIKBD a DIKH na túto zmenu nie sú citlivé (6.1.9). Všetky zahrnuté CPU implementácie (BFP, DIKBD a DIKH) sú v tomto teste rovnocenné, algoritmus CUDASSSP je v tomto teste 10 násobne pomalší ako BFP. Tento výsledok je z hľadiska bližšieho pohľadu prekvapivý, pretože hľadať najkratšie cesty v hustých náhodných grafoch, ktorých ohodnotenie je z nízkeho rozsahu (napríklad použitých 100), je výrazne jednoduchšie, než riešenie tej istej úlohy pri riedkych grafoch (napríklad 6.5.2 oproti 6.5.1 pre DIKBD) obsahujúcich podobný počet hrán. Dôvodom neuspokojivého výkonu je horšie rozdelenie

práce medzi výpočetné jednotky. Skúmaných vrcholov je pomerne málo, zoznam nasledovníkov vrcholu je však dlhý, niektoré výpočetné jednotky preto strávia dlhý čas prechádzaním hrán jedného vrcholu (for cyklus kernelu), kým ostatné nie sú vyťažené. Pre optimálne využitie GPU by v tomto prípade bolo lepšie buď, rozdelenie hrán pre jeden vrchol viacerým vláknam, alebo spustenie vlákna pre každú hranu.

6.5.2 CUDA Rand 6 (S)

Výsledky dvojice testov A.31 a A.32 predchádzajú testu 6.5.3. Trieda vstupov pre test je zhodná s 6.1.7 s rozdielom pomeru počtu hrán k počtu vrcholov, ktorý je 6. Biele miesta vo výsledkoch sú v tomto prípade všetky jednak z dôvodu úspory času, ale najmä pre zvýraznenie podstatnej časti testu – správanie implementácie na GPU oproti najlepším CPU implementáciám. Pre oba prípady sú všetky nastavenia zhodné, jediným rozdielom je horná hranica pre dĺžku náhodnej hrany. Pre CUDA Rand 6 je **ll**=100, pre druhý test je **ll**=5. Účelom tohto testu je zachytenie správania vo vzťahu k rastúcej veľkosti vstupu. Pre vyššiu hodnotu **ll** kopíruje CUDASSSP časy DIKBD, pre kratšie hrany je konzistentne 3,3x rýchlejšia.

6.5.3 CUDA Rand/ER Len

Test 6.1.9 je ďalšou ukážkou toho, ako sa mení náročnosť rôznych algoritmov, keď sú nútené hľadať najkratšie cesty s vyšším počtom vrcholov a hrán. Implementácia CUDASSSP je na takéto vstupy zvlášť citlivá, pretože pri jednom kroku (spustenia kernelov) sú preberané všetky vrcholy, ale dĺžky skúmaných ciest za vždy zvýšia iba o 1, dochádza preto k veľkému počtu krokov a tým aj k nárastu potrebného času (A.33). Ako vyplýva z popisu generátora Rand, tvorený graf nie je náhodný – vždy je pridávaná cesta s hranami jednotkovej dĺžky, ktorá spája všetky vrcholy. Predĺženie náhodných hrán preto spôsobuje, že sa do najkratších ciest zapájajú stále dlhšie úseky tejto krátkej cesty. O tom, že takéto správanie typicky u náhodných grafov nenastáva, svedčí aj výsledok (A.34). Vstupné grafy v oboch testoch obsahujú 2^{20} vrcholov a päť násobný počet hrán. V druhom prípade je približne 3x rýchlejšia implementácia na GPU oproti najlepšej CPU implementácií (BFP), DIKBD a GOR1 sú približne o 50% pomalšie. V prvom teste je od hornej hranice dĺžky hrany 100, najlepšia implementácia DIKBD, implementácie BFP a CUDASSSP sú v najhorších prípadoch až 20x pomalšie.

6.5.4 CUDA Rand P

Test pre triedu Rand P, je podobný s testom 6.1.10, vstupný graf je však podstatne väčší. Počet vrcholov je 2^{20} a počet hrán je päť násobkom počtu vrcholov. V tomto teste nieje premenlivá veľkosť vstupu, ale horná hranica pre potenciál vrcholu. Ohodnotenia hrán sú následne pomocou vygenerovaných potenciálov redukované. Pôvodná sieť neobsahuje záporné hrany a preto je vylúčená existencia cyklu zápornej dĺžky. Ako je ukázané v práci [1], výpočet niektorých algoritmov je na takýchto sieťach vždy rovnaký, bez ohľadu na voľbu potenciálov. Algoritmus BFP, ktorý je v istom ohľade sekvenčnou verziou CUDASSSP, je jedným z takýchto algoritmov. Ako ukazuje výsledok (A.36), implementácia na GPU sa správa podobne, pričom je patrične rýchlejšia oproti BFP: 3,5 násobne. Pre nízke hodnoty potenciálu je výskyt záporných hrán pomerne nízky, do hodnoty 10^4 (maximálna dĺžka náhodných hrán) je výkonnosť lepšia u implementácie DIKBD, s nástupom veľkého podielu záporných hrán sa však situácia mení – varianty Dijkstrovho prístupu sú nútené ku častým opakovaniam skenovania vrcholov a výhodnejšie sa stávajú implementácie BFP a CUDASSSP, ktorých vlastné časy sa pre všetky vstupy líšia iba zanedbateľne.

6.5.5 CUDA R-MAT 1:1

Pre identické parametre vstupu, ako v 6.4.1, s výnimkou inicializácie náhodného generátoru

sú porovnané dve najlepšie a jedna najhoršia implementácia proti CUDASSSP. Z dobrého správania BFP, bolo možné predpokladať dobré správanie tejto implementácie. Výsledné priemery (A.36) dosiahnutých časov sú síce najlepšie z testovaných algoritmov, sú však zaťažené väčším rozptylom nameraných hodnôt. Zistené rozdiely nie sú príliš veľké (do 30%). Dôvodom nie celkom ideálneho správania CUDASSSP, sú vrcholy vysokých stupňov, ktoré tieto grafy obsahujú, čo znovu vedie k nerovnomernému vyťaženiu výpočetných jednotiek.

6.5.6 CUDA - ostatné výsledky

Implementácia CUDASSSP je zahrnutá okrem tejto kapitoly, v testoch SSCA#2 (6.4) a State (6.3). Pre všetky testy na triede Grid, je táto implementácia výrazne horšia ako ostatné (podľa predbežných testov), je to spôsobené štruktúrou samotných problémov, kde sú najkratšie cesty do veľkého množstva vrcholov, čo do počtu vrcholov a hrán, veľmi dlhé (pre zhoršenie času postačuje aj jediná taká cesta). Najlepší výsledok (nie však lepší než na CPU) je dosiahnutý na triede Grid S Wide, uvedený v A.37, v tomto prípade sú časy približne dvojnásobné oproti ostatným riešeniam na CPU.

7 Záver

Výsledky experimentov ukázali zaujímavé tendencie algoritmov. Pri mnohých vstupoch sú lepšie algoritmy s horším odhadom hornej časovej zložitosti (napríklad 6.1.1, 6.1.3-4, 6.4 a 6.5). Prepracované dátové štruktúry s lepšou teoretickou cenou operácií nie sú zaručenou výhodou (DIKF v 6.2.1). Štruktúra vstupných dát má väčší dopad než rozsah (napríklad 6.1.5 proti 6.1.7). Zmena niektorého z parametrov vstupu môže aj nemusí mať vplyv na výsledný čas (6.1.10, 6.1.13, 6.5.3), v závislosti od algoritmu. Žiadny z algoritmov nie je najlepší (a dokonca ani použiteľný) na všetkých typoch vstupu.

Pre prípad, keď sú potrebné rýchle a stručné odpovede, slúžia informácie uvedené v časti 7.1, v každom prípade je prínosné vytvorenie vlastného testu, alebo aspoň podrobnejšie preskúmanie výsledkov podobných vstupov v kapitole 6 a prílohe A. Výsledky je nutné posudzovať s ohľadom na nízky počet meraní, rozdiely v rádoch %, ale i násobky nemusia byť smerodajné (i keď sa ukazujú ako reprodukovateľné). Malé rozdiely vstupu môžu tiež spôsobiť zmenu poradia (6.4.5, 6.2.3, 6.1.9, 6.1.10 a 6.1.3).

Pre získanie inšpirácie do ďalšej práce, je možné zvážiť námety v časti 7.2. V každom prípade bude vítané využitie kontaktu peter@way-to-play.eu, pre výmenu nápadov, alebo poskytnutie ďalších informácií.

7.1 Odporúčané implementácie

Pre prípad sekvenčného riešenia jedným vláknom, je možné rozdelenie na dva prípady. Ak vstupný problém neobsahuje záporné hrany, alebo je ich podiel nízky, implementácie Dijkstrovho algoritmu sú zvyčajne dobre použiteľné. V niektorých prípadoch je haldová implementácia nevhodná (najmä ak je dostupný každý vrchol pomocou umelého zdroja, vtedy dôjde k zaplneniu haldy hneď na začiatku výpočtu). Všeobecne je možné odporučiť viacúrovňové priehradky (DIKBD). Pre grafy so zápornými hranami je robustný algoritmus Goldberg-Radzik (GOR1). Vždy je pre maximálny výkon potrebné zvážiť i ostatné implementácie. V prípade štvorcových mriežok s nezápornými hranami, sú výhodné jednoduchšie implementácie ako TWO_Q alebo BFP. Toto platí aj pre iné vstupy s jednoduchou štruktúrou, ako napríklad riedke náhodné grafy (6.4.5).

Ak je požadované využitie viacerých jadier CPU, najlepším prístupom je v súčasnosti súbežné riešenie viacerými rôznymi algoritmami. Vstupné zadanie môže byť uložené v spoločnej nechránenej pamäti, pretože sa počas výpočtu nemení, každý algoritmus beží v samostatnom vlákne s vlastnými pracovnými dátami. Týmto odpadajú náklady na synchronizáciu, ktoré na

CPU sťažujú efektívnu implementáciu. Prvé vlákno ktoré dopočíta výsledok, zastaví ostatné. Tento prístup je vhodný aj pre vstupy bez vopred známej štruktúry, keď nie je jasné, ktorý algoritmus bude najvhodnejší.

V prípade dostupnosti mnoho jadrovej výpočetnej jednotky, ako napríklad GPU, alebo niektorý z produktov odkazovaných v kapitole 4, je potrebné podrobné zváženie ponúkaných možností synchronizácie, organizácie pamäte a predpokladaných parametrov vstupného problému. Pre riedke náhodné grafy s nezápornými ohodnoteniami hrán, je implementácia CUDASSSP rýchlejšia než CPU implementácie, už na GPU strednej triedy. Porovnateľný výkon s CPU implementáciami dosahuje aj na niektorých ďalších triedach problémov. Vo všeobecnosti nastávajú problémy na vstupoch s vysokými stupňami vrcholov, alebo veľkými dĺžkami výsledných ciest (vzhľadom k počtu vrcholov hrán). Dobré vstupy pre riešenie na GPU pomocou algoritmu Harish – Narayanan sú tie, v ktorých je rozloženie hrán na vrcholy približne rovnomerné, výsledné cesty sú krátke a počet vrcholov vstupu je dostatočne veľký, aby bolo možné efektívne využitie výkonu stoviek procesorov. Algoritmus je v istom smere podobný BFP, preto je možné ako jednoduchú pomôcku najprv odskúšanie BFP – ak poskytuje dobrý výkon, GPU implementácia má šancu na dobré výsledky.

V každom prípade je veľmi dôležité zváženie špecifických vlastností vstupu. Pre acyklické siete je vhodný špecializovaný algoritmus ACC, dobrý výkon dosahuje aj GOR1. V prípade veľmi obmedzenej množiny hodnôt ohodnotenia hrán, je lepšia prispôsobená priehradková implementácia, než všeobecné verzie, ktoré boli testované (s prispôsobeným počtom priehradiek). Využitie rôznych heuristík, keď sú napríklad dostupné polohy vrcholov, môže výrazne urýchliť výpočet [20, 22, 30]. Niektoré algoritmy a dátové štruktúry, ponúkajú tiež možnosť ladenia výkonu pomocou parametrov (*k* pri halde, *t* pri THRESH). Jednoduché zlepšenia, ako uchovávanie indexu prvej a poslednej neprázdnej priehradky v DIKB_ algoritmoch, alebo adaptívne správanie (napríklad striedanie prístupu DIKH a DIKBD počas výpočtu) sa taktiež ukazujú ako výhodné [11, 48 - 52].

7.2 Námety pre ďalšiu prácu

Podľa osobných skúseností, v každej práci väčšieho rozsahu je nutné prijatie kompromisov, zvolenie niektorých ciest a zanechanie iných, táto práca nie je výnimkou.

Pre vykonanie praktických experimentov je nutná príprava väčšieho množstva vstupov, spúšťanie výpočtov, spracovanie výsledkov. Pri prvých experimentoch v predošlom období som experimentoval s rôznymi prístupmi – skripty pre shell, generovanie databázových skriptov, ručné spracovanie. Hľadanie hotových riešení v tejto oblasti, neprinieslo požadovaný výsledok. Existuje síce veľká podpora funkčného testovania, záťažového testovania a rôzne balíky pre vizualizáciu algoritmov, žiadny nájdený software však nevyhovoval všetkým potrebám experimentálnej algoritmiky. Pre účel práce bol vytvorený čiastočne funkčný prototyp prostredia, ktoré dokáže prepojiť výstupy rôznych generátorov s implementáciami, načítať výsledky po dobehnutí riešenia a spracovať dáta. Zdrojové súbory so stručným komentárom sú priložené na DVD. Scenáre testov, vstupy a výsledky sa spracúvajú na podobnom princípe, ako spustiteľné programy – vstupné súbory sú postupne transformované na výstupné v niekoľkých krokoch, aktualizácia odvodených súborov sa vykonáva iba ak došlo ku zmene zdrojových. Vytvorenie kompletného prostredia, ktoré umožňuje pridávanie generátorov, implementácií, spracúva namerané dáta pomocou štatistických balíkov a generuje výstupy do grafov a tabuliek by mohlo byť námetom pre ďalšiu prácu, možno i softvérový projekt.

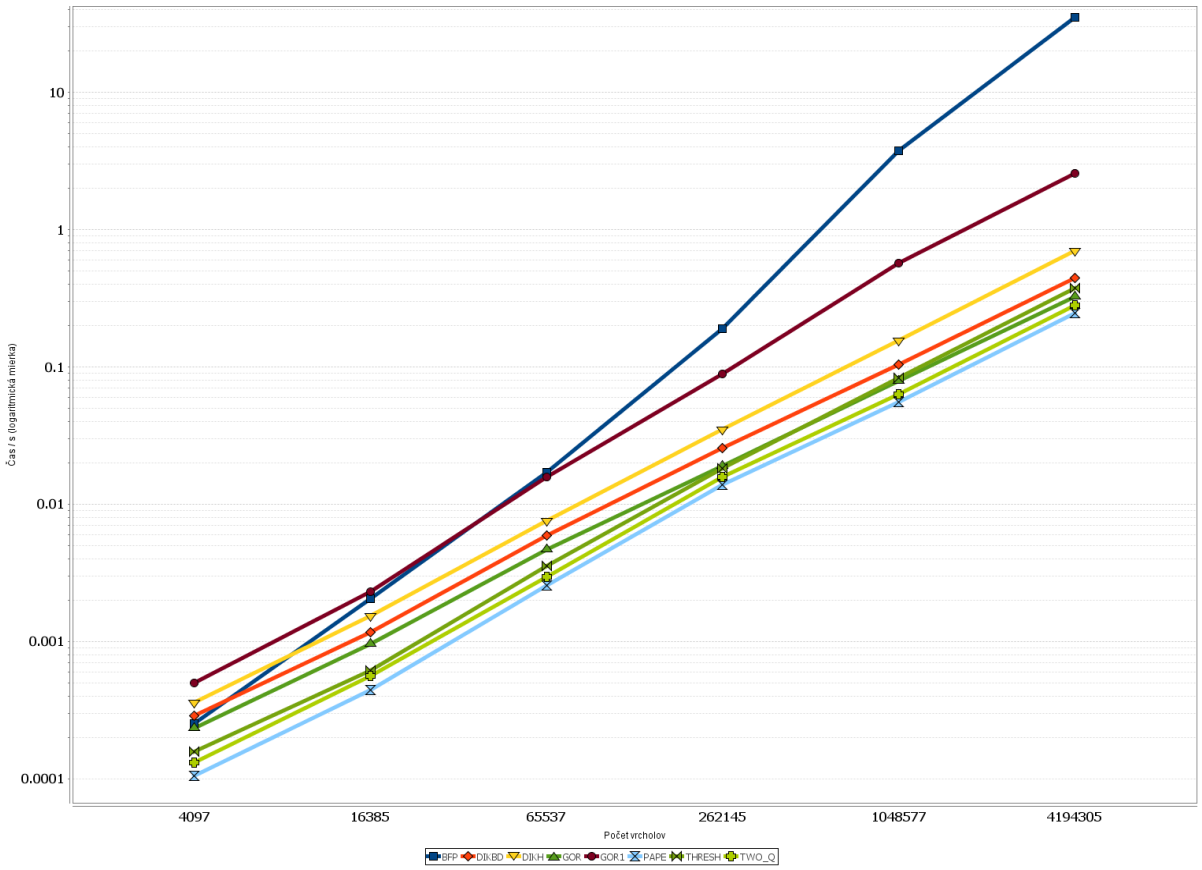
Počas štúdia materiálov a zdrojových kódov, je často možné nájsť námety na zlepšenie. V

prípade GPU implementácie CUDASSSP, je otvorená veľmi dobrá príležitosť. Niektoré slabé stránky algoritmu ktoré odhaľujú testy, je možné odstrániť. Novšie verzie CUDA, prípadne ďalšie platformy, ktoré budú dostupné v blízkej budúcnosti, umožňujú i využitie synchronizácie (alebo aspoň funkcií, ktoré zamedzujú nejednoznačným výsledkom), je preto pravdepodobné, že pri podrobnejšom experimentovaní s rôznymi prístupmi, je možné dosiahnutie rádových zlepšení oproti testovanej verzii (hrubý výkon GPU je o dva rády vyšší než u CPU). Práca [33] navyše ukazuje (na prípade riešenia APSP pomocou aplikácie CUDASSSP pre každý vrchol), že aktuálne riešenie je obmedzené pamäťovou priepustnosťou grafickej karty a nie výkonom samotného GPU.

Z hľadiska rozsahu a presnosti výsledkov, je vždy možné niečo zlepšiť. Pridanie ďalších implementácií, generátorov, zväčšenie počtu vzorkov pre experimenty, zväčšenie rozsahu a jemnosti odstupňovania vstupov, sú na prvý pohľad až príliš jednoduché vylepšenia. Každý nový experiment, i keď len mierne modifikovaného algoritmu, však môže viesť k zaujímavým zisteniam a vytvoreniu efektívnejších implementácií. Zaujímavými zdrojmi sú v tomto smere [48, 51 – 53].

A Príloha - Výsledky meraní

A.1 Grid SSquare

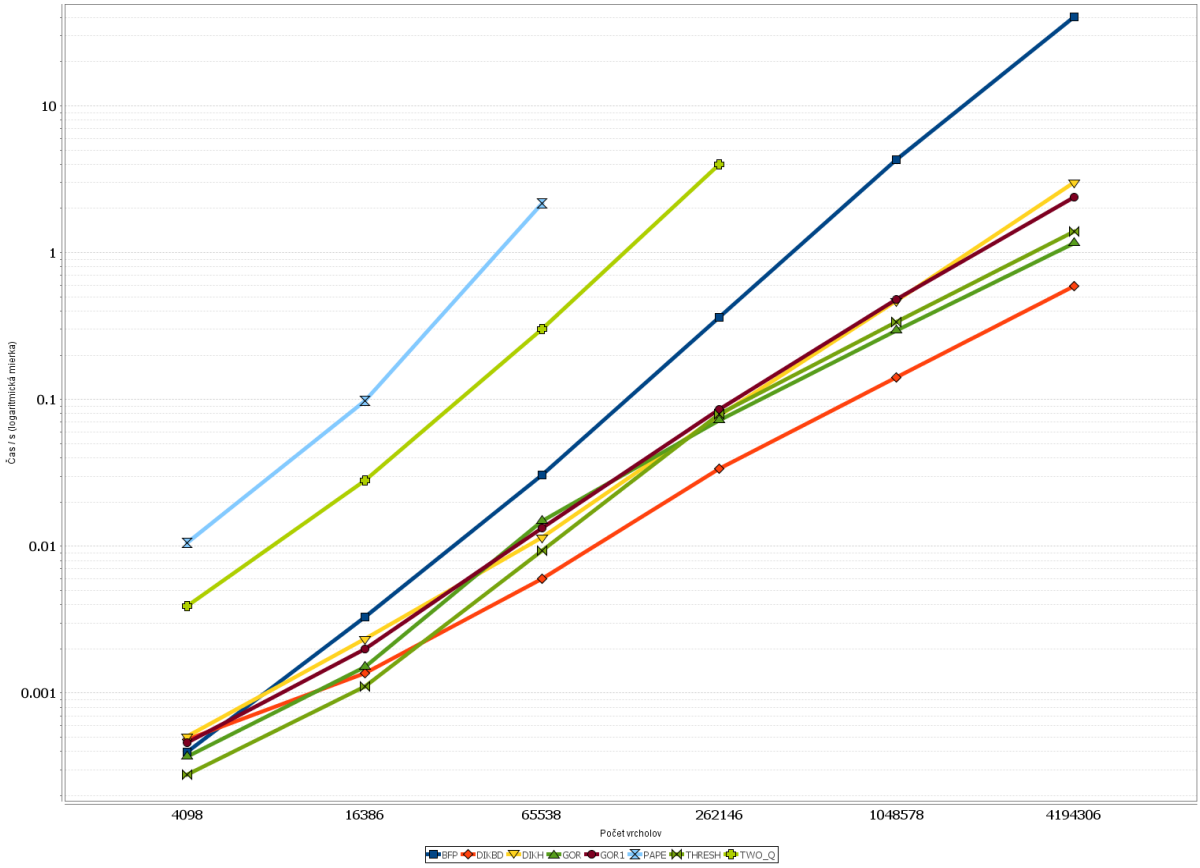


OA.1 Štvorcová mriežka **X=Y**.

počet vrcholov / hrán	BFP	DIKBD	DIKH	GOR	GOR1	PAPE	THRESH	TWO_Q
4097 / 12288	0.0003	0.0003	0.0004	0.0002	0.0005	0.0001	0.0002	0.0001
16385 / 49152	0.0021	0.0012	0.0015	0.0010	0.0023	0.0004	0.0006	0.0006
65537 / 196608	0.0171	0.0059	0.0076	0.0047	0.0157	0.0025	0.0036	0.0029
262145 / 786432	0.1892	0.0255	0.0348	0.0188	0.0888	0.0138	0.0183	0.0157
1048577 / 3145728	3.7505	0.1038	0.1560	0.0783	0.5709	0.0556	0.0834	0.0631
4194305 / 12582912	34.9954	0.4446	0.7038	0.3237	2.5680	0.2458	0.3753	0.2800

TA.1 Štvorcová mriežka **X=Y**.

A.2 Grid SSquare-S

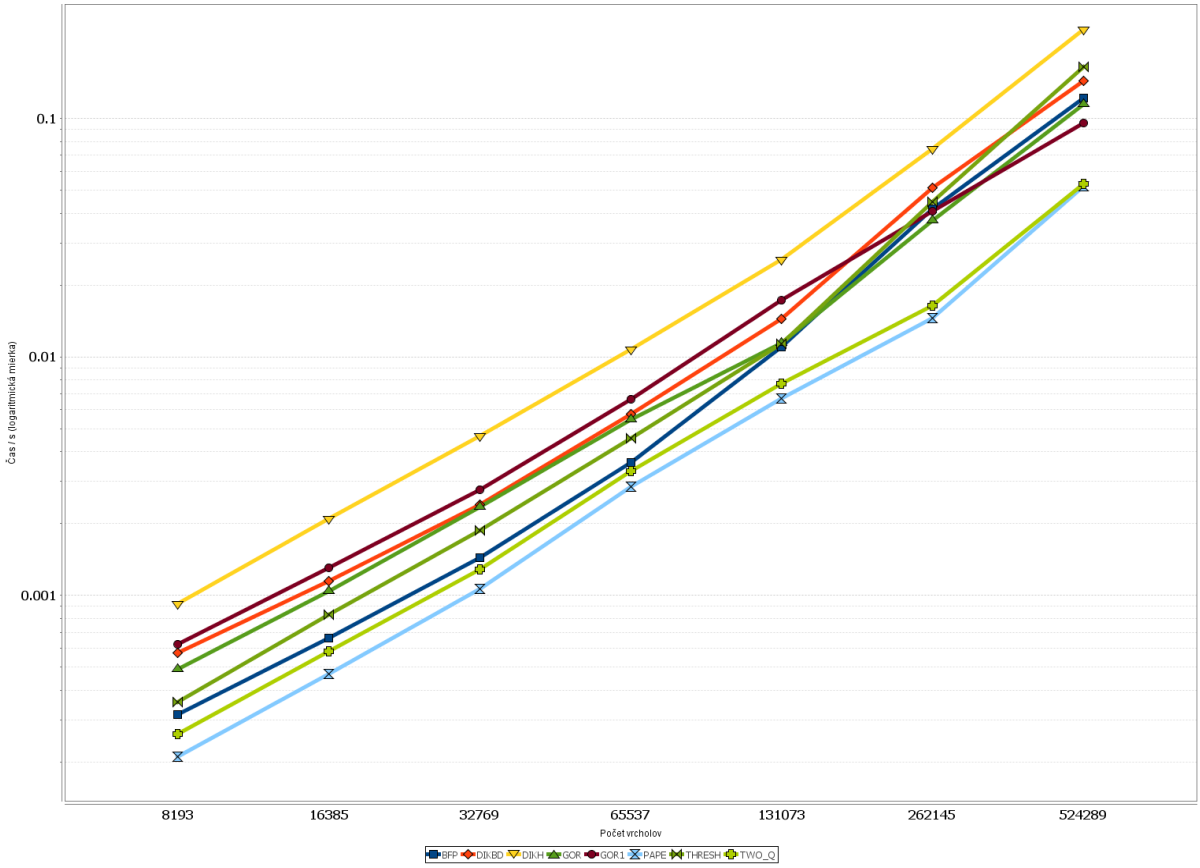


OA.2 Štvorcová mriežka s umelým zdrojom.

počet vrcholov / hrán	BFP	DIKBD	DIKH	GOR	GOR1	PAPE	THRESH	TWO_Q
4098 / 16385	0.0004	0.0005	0.0005	0.0004	0.0005	0.0105	0.0003	0.0039
16386 / 65537	0.0033	0.0014	0.0023	0.0015	0.0020	0.0974	0.0011	0.0281
65538 / 262145	0.0306	0.0060	0.0115	0.0148	0.0133	2.1551	0.0093	0.3019
262146 / 1048577	0.3609	0.0338	0.0775	0.0721	0.0858		0.0794	3.9894
1048578 / 4194305	4.3163	0.1405	0.4697	0.2944	0.4797		0.3388	
4194306 / 16777217	40.3529	0.5936	3.0158	1.1653	2.3821		1.3959	

TA.2 Štvorcová mriežka s umelým zdrojom.

A.3 Grid SWide

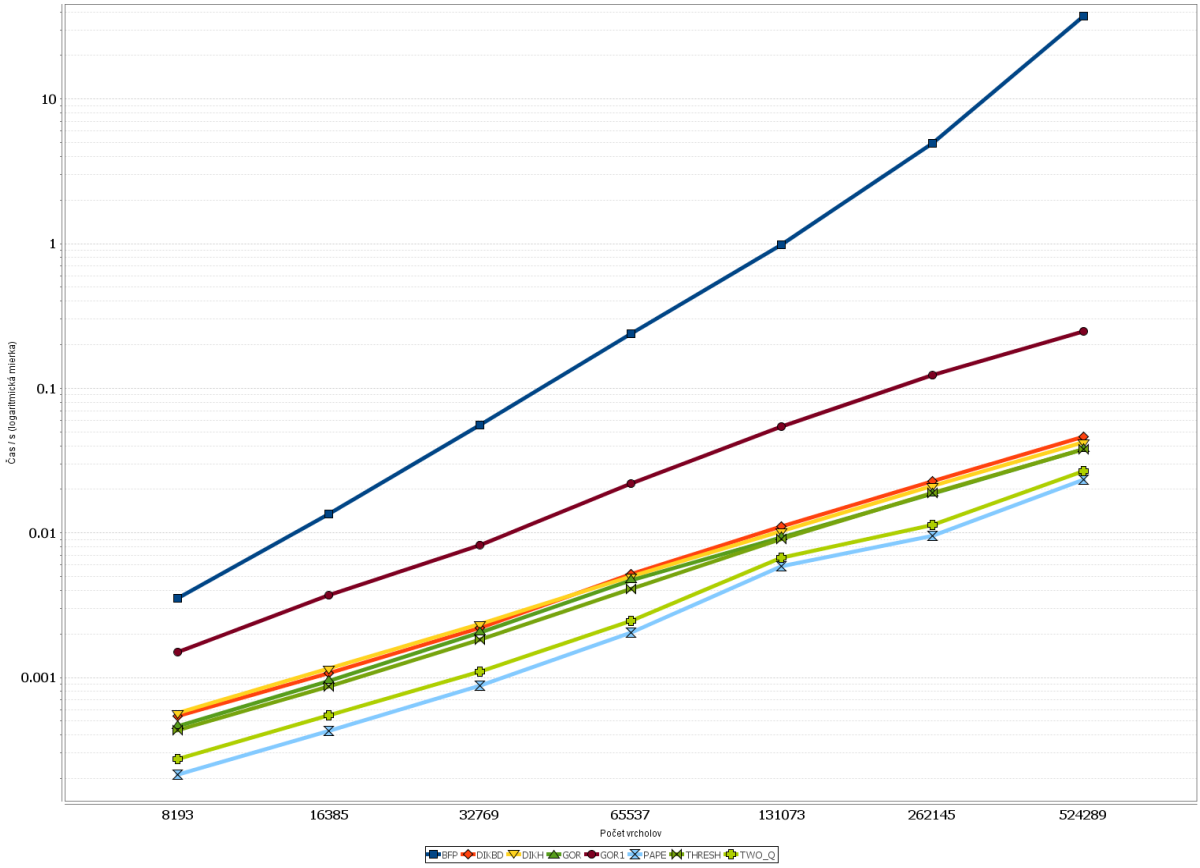


OA.3 Obdĺžniková mriežka, **X** *= 16.*

počet vrcholov / hrán	BFP	DIKBD	DIKH	GOR	GOR1	PAPE	THRESH	TWO_Q
8193 / 24576	0.0003	0.0006	0.0009	0.0005	0.0006	0.0002	0.0004	0.0003
16385 / 49152	0.0007	0.0011	0.0021	0.0010	0.0013	0.0005	0.0008	0.0006
32769 / 98304	0.0014	0.0024	0.0047	0.0023	0.0028	0.0011	0.0019	0.0013
65537 / 196608	0.0036	0.0058	0.0108	0.0055	0.0066	0.0029	0.0046	0.0033
131073 / 393216	0.0110	0.0144	0.0255	0.0114	0.0172	0.0067	0.0113	0.0077
262145 / 786432	0.0418	0.0513	0.0747	0.0373	0.0408	0.0145	0.0446	0.0164
524289 / 1572864	0.1218	0.1435	0.2356	0.1144	0.0956	0.0519	0.1638	0.0530

TA.3 Obdĺžniková mriežka, **X** *= 16.*

A.4 Grid SLong

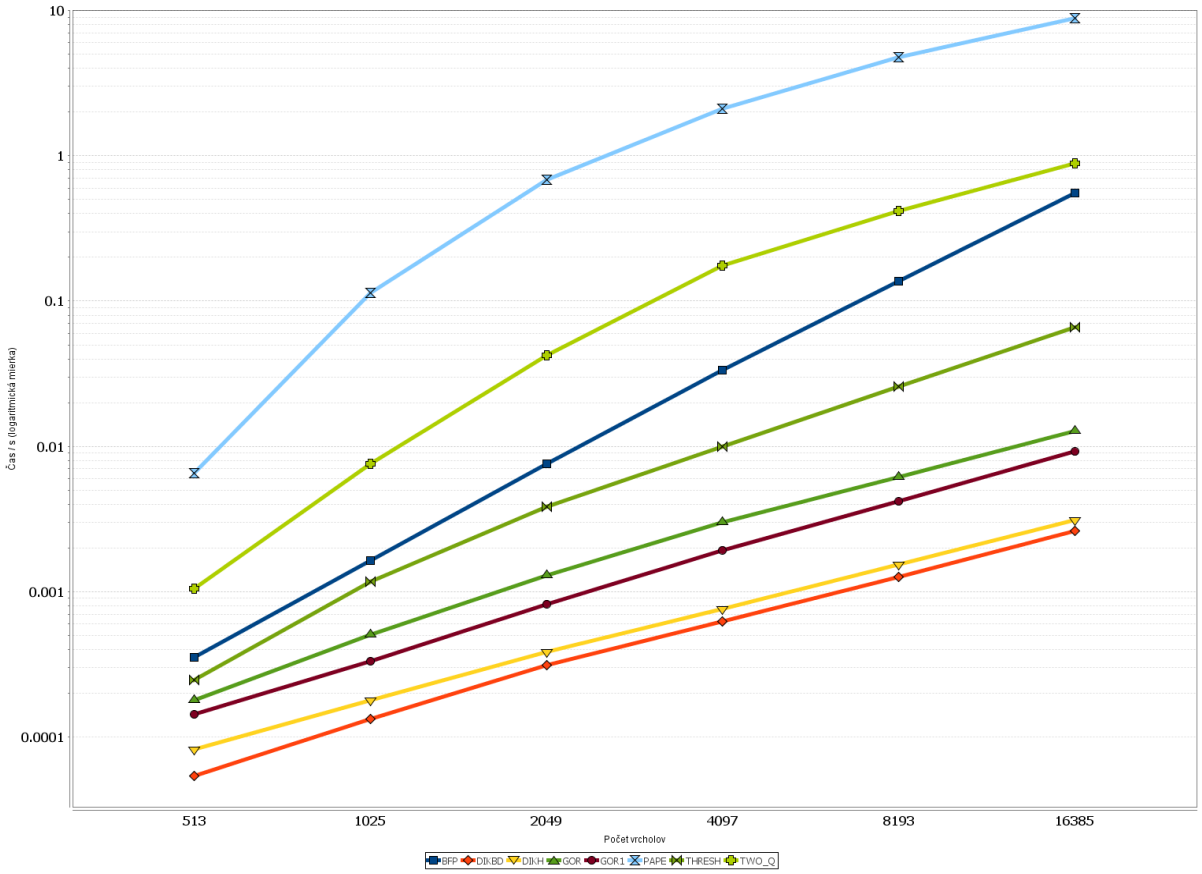


OA.4 Obdĺžniková mriežka, **Y** *= 16.*

počet vrcholov / hrán	BFP	DIKBD	DIKH	GOR	GOR1	PAPE	THRESH	TWO_Q
8193 / 24576	0.0035	0.0005	0.0006	0.0005	0.0015	0.0002	0.0004	0.0003
16385 / 49152	0.0135	0.0011	0.0012	0.0009	0.0037	0.0004	0.0009	0.0005
32769 / 98304	0.0556	0.0022	0.0023	0.0021	0.0083	0.0009	0.0018	0.0011
65537 / 196608	0.2383	0.0052	0.0050	0.0047	0.0219	0.0020	0.0041	0.0025
131073 / 393216	0.9842	0.0111	0.0103	0.0093	0.0544	0.0058	0.0091	0.0067
262145 / 786432	4.9501	0.0227	0.0211	0.0187	0.1226	0.0096	0.0188	0.0114
524289 / 1572864	37.4273	0.0460	0.0425	0.0380	0.2478	0.0232	0.0377	0.0267

TA.4 Obdĺžniková mriežka, **Y** *= 16.*

A.5 Grid PHard (malé vstupy)

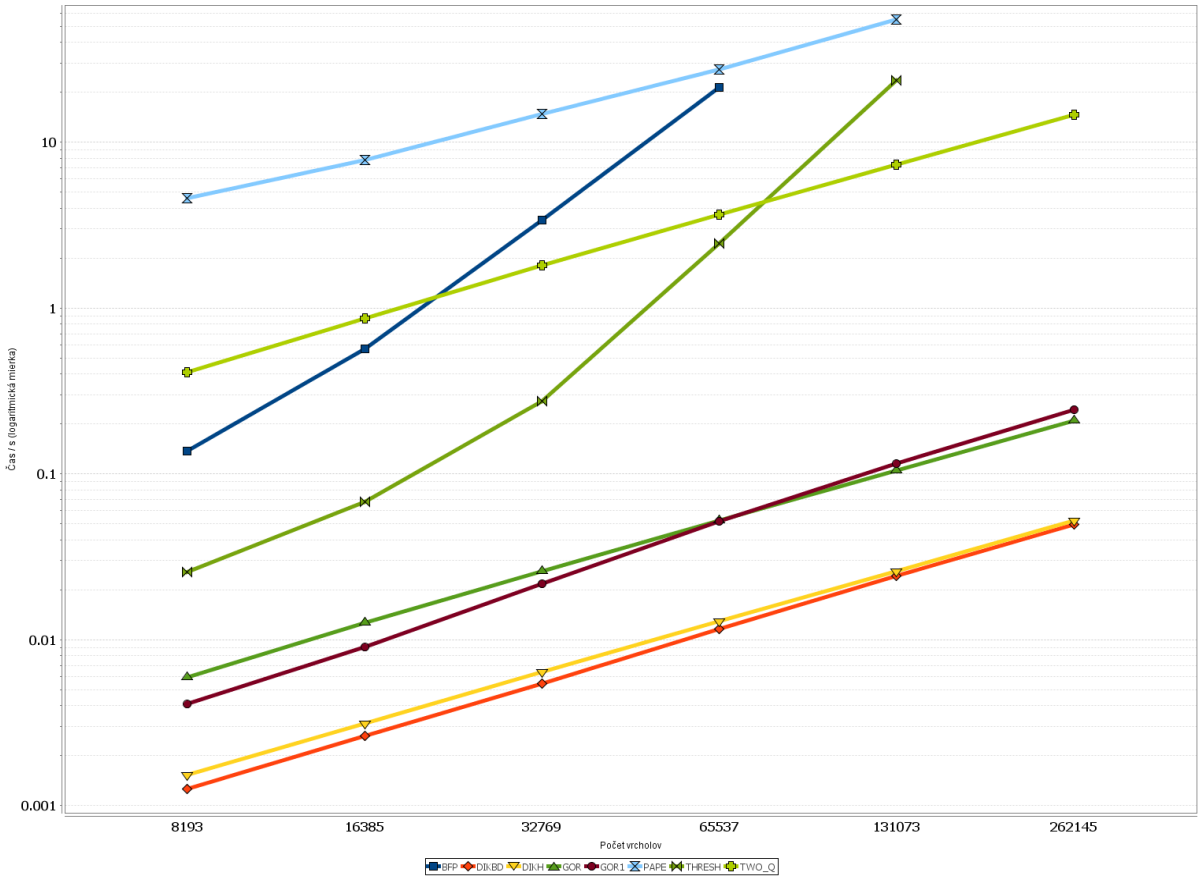


OA.5 Grid PHard pre malé rozsahy vstupov, obdĺžniková mriežka s náhodnými hranami.

počet vrcholov / hrán	BFP	DIKBD	DIKH	GOR	GOR1	PAPE	THRESH	TWO_Q
513 / 2528	0.0004	0.0001	0.0001	0.0002	0.0001	0.0065	0.0002	0.0010
1025 / 6464	0.0016	0.0001	0.0002	0.0005	0.0003	0.1142	0.0012	0.0076
2049 / 14656	0.0076	0.0003	0.0004	0.0013	0.0008	0.6810	0.0038	0.0421
4097 / 31040	0.0335	0.0006	0.0008	0.0030	0.0019	2.0989	0.0100	0.1753
8193 / 63808	0.1376	0.0013	0.0015	0.0061	0.0042	4.7543	0.0259	0.4152
16385 / 129344	0.5535	0.0026	0.0031	0.0127	0.0092	8.8040	0.0659	0.8859

TA.5 Grid PHard pre malé rozsahy vstupov, obdĺžniková mriežka s náhodnými hranami.

A.6 Grid PHard

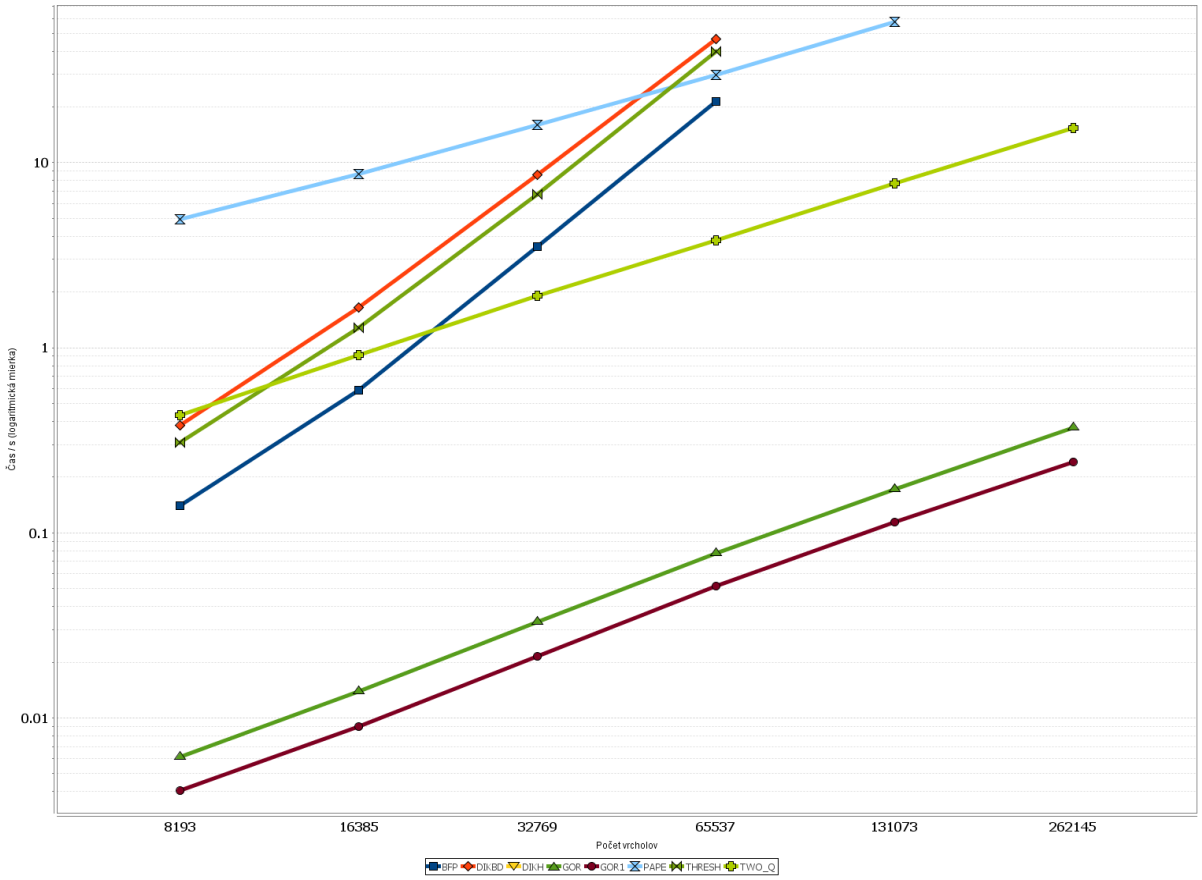


OA.6 Grid PHard, obdĺžniková mriežka s náhodnými hranami.

počet vrcholov / hrán	BFP	DIKBD	DIKH	GOR	GOR1	PAPE	THRESH	TWO_Q
8193 / 63808	0.1373	0.0013	0.0015	0.0059	0.0041	4.5945	0.0256	0.4113
16385 / 129344	0.5675	0.0026	0.0031	0.0127	0.0090	7.8505	0.0680	0.8649
32769 / 260416	3.3999	0.0055	0.0064	0.0259	0.0217	14.8041	0.2742	1.8123
65537 / 522560	21.3644	0.0116	0.0129	0.0525	0.0518	27.4055	2.4628	3.6409
131073 / 1046848		0.0244	0.0258	0.1051	0.1154	54.7348	23.7130	7.3534
262145 / 2095424		0.0495	0.0521	0.2095	0.2446			14.6527

TA.6 Grid PHard, obdĺžniková mriežka s náhodnými hranami.

A.7 Grid NHard

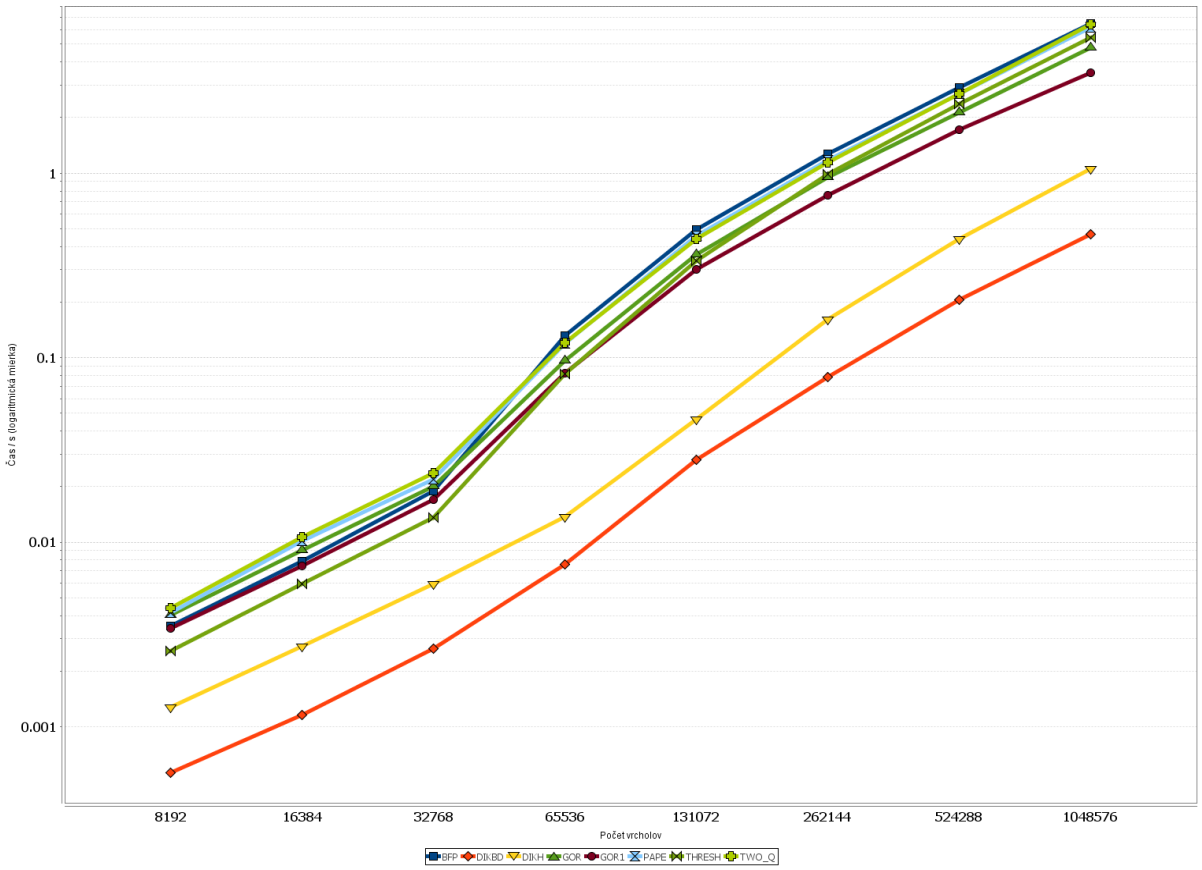


OA.7 Grid NHard, všetky náhodné hrany sú záporné.

počet vrcholov / hrán	BFP	DIKBD	DIKH	GOR	GOR1	PAPE	THRESH	TWO_Q
8193 / 63808	0.1396	0.3792		0.0061	0.0040	4.9649	0.3066	0.4304
16385 / 129344	0.5914	1.6479		0.0139	0.0090	8.6463	1.2820	0.9092
32769 / 260416	3.5370	8.6392		0.0330	0.0215	16.0773	6.7812	1.9020
65537 / 522560	21.4293	46.5626		0.0772	0.0515	29.7929	39.9974	3.8136
131073 / 1046848				0.1726	0.1141	57.8746		7.7013
262145 / 2095424				0.3687	0.2419			15.3479

TA.7 Grid NHard, všetky náhodné hrany sú záporné.

A.8 Rand 4

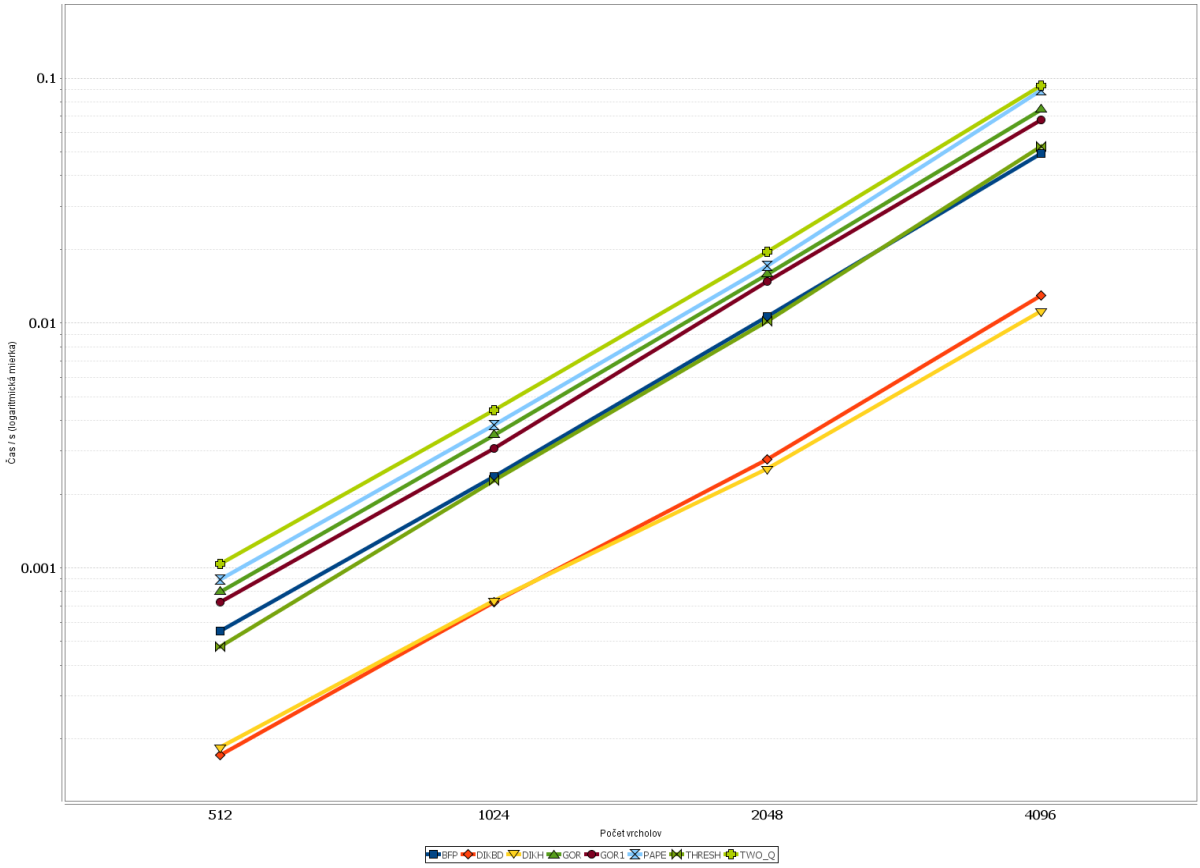


OA.8 Rand 4, **m**=*4***n**, náhodný graf s krátkou kružnicou.

počet vrcholov / hrán	BFP	DIKBD	DIKH	GOR	GOR1	PAPE	THRESH	TWO_Q
8192 / 32768	0.0035	0.0006	0.0013	0.0040	0.0034	0.0041	0.0026	0.0044
16384 / 65536	0.0079	0.0012	0.0027	0.0090	0.0075	0.0102	0.0060	0.0106
32768 / 131072	0.0189	0.0027	0.0059	0.0200	0.0170	0.0218	0.0136	0.0238
65536 / 262144	0.1313	0.0076	0.0138	0.0960	0.0825	0.1187	0.0815	0.1202
131072 / 524288	0.4959	0.0280	0.0464	0.3603	0.3014	0.4573	0.3362	0.4398
262144 / 1048576	1.2761	0.0788	0.1613	0.9517	0.7584	1.1755	0.9871	1.1426
524288 / 2097152	2.8971	0.2061	0.4408	2.1380	1.7177	2.7026	2.3818	2.6903
1048576 / 4194304	6.4805	0.4674	1.0578	4.7746	3.4953	6.1294	5.4398	6.4054

TA.8 Rand 4, **m**=*4***n**, náhodný graf s krátkou kružnicou.

A.9 Rand 1:4

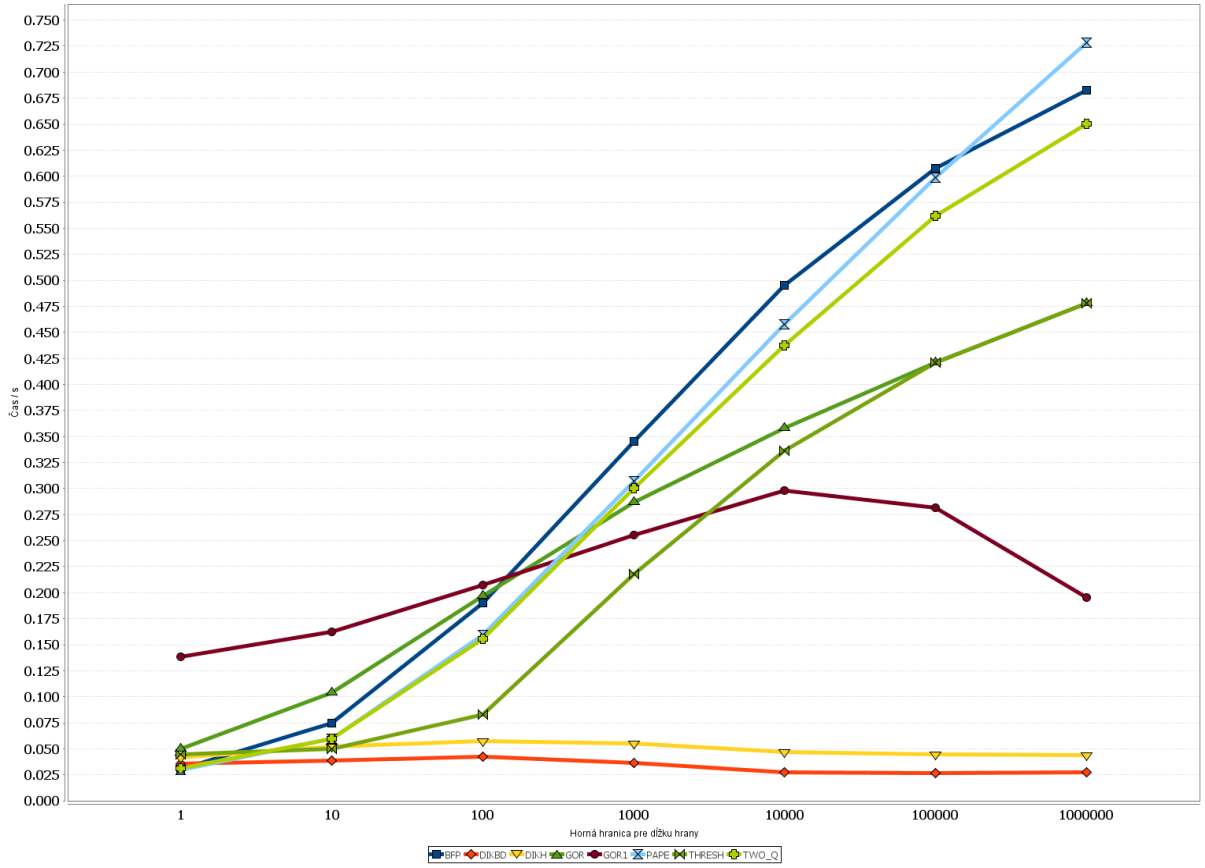


OA.9 Rand 1:4, $\mathbf{m}=\mathbf{n}^2/4$*,hustý náhodný graf s jednotkovou kružnicou.*

počet vrcholov / hrán	BFP	DIKBD	DIKH	GOR	GOR1	PAPE	THRESH	TWO_Q
512 / 65536	0.0006	0.0002	0.0002	0.0008	0.0007	0.0009	0.0005	0.0010
1024 / 262144	0.0024	0.0007	0.0007	0.0035	0.0031	0.0038	0.0023	0.0044
2048 / 1048576	0.0106	0.0028	0.0025	0.0157	0.0148	0.0172	0.0102	0.0196
4096 / 4194304	0.0492	0.0130	0.0112	0.0744	0.0675	0.0891	0.0527	0.0931

TA.9 Rand 1:4, $\mathbf{m}=\mathbf{n}^2/4$*,hustý náhodný graf s jednotkovou kružnicou.*

A.10 Rand-Len

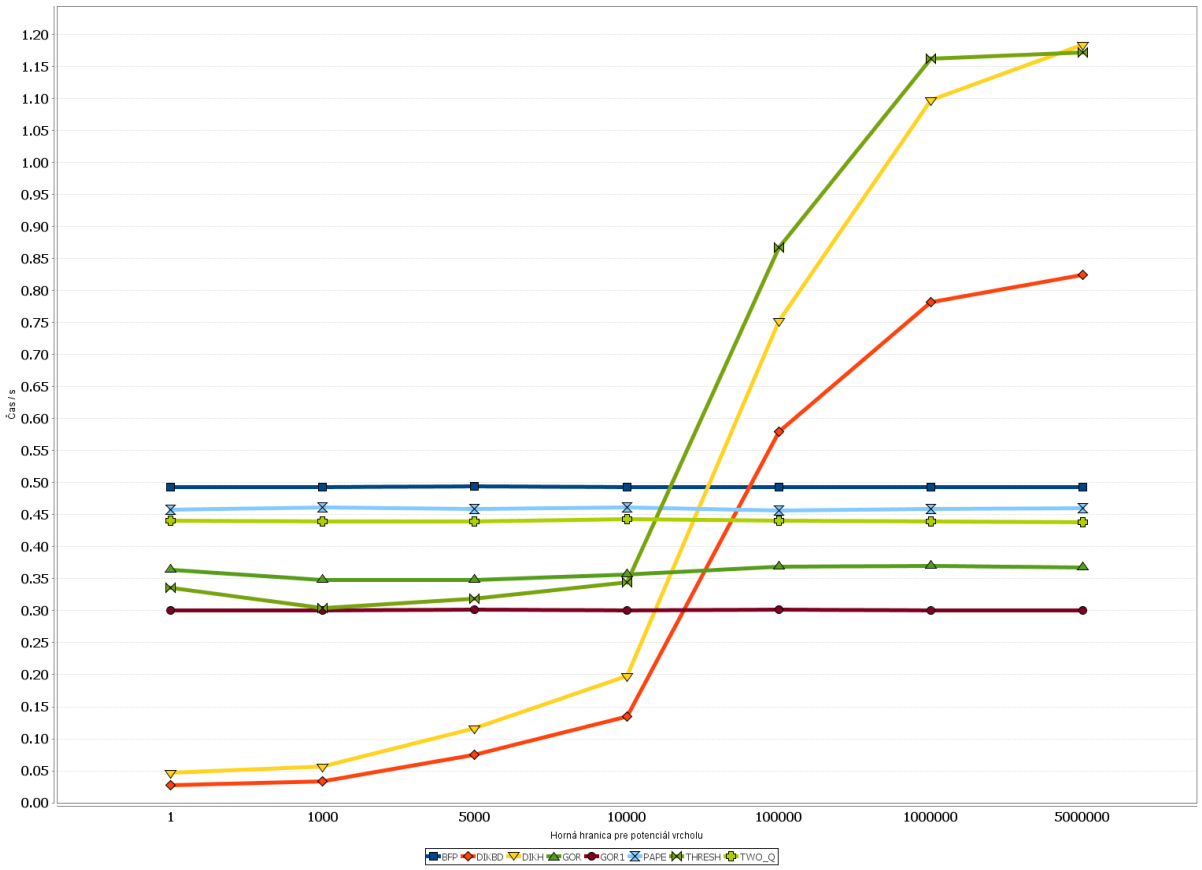


OA.10 Výsledky Rand-Len, **n***=131072,* **m***=524288.*

Maximum dĺžky hrany	BFP	DIKBD	DIKH	GOR	GOR1	PAPE	THRESH	TWO_Q
1	0.0307	0.0356	0.0420	0.0500	0.1387	0.0298	0.0444	0.0311
10	0.0743	0.0390	0.0519	0.1038	0.1622	0.0594	0.0496	0.0594
100	0.1902	0.0426	0.0575	0.1971	0.2072	0.1596	0.0832	0.1558
1000	0.3455	0.0362	0.0549	0.2867	0.2555	0.3071	0.2180	0.3007
10000	0.4949	0.0278	0.0468	0.3581	0.2983	0.4575	0.3364	0.4377
100000	0.6078	0.0269	0.0443	0.4213	0.2817	0.5989	0.4208	0.5617
1000000	0.6830	0.0273	0.0441	0.4783	0.1953	0.7285	0.4781	0.6506

TA.10 Výsledky Rand-Len, **n***=131072,* **m***=524288.*

A.11 Rand P

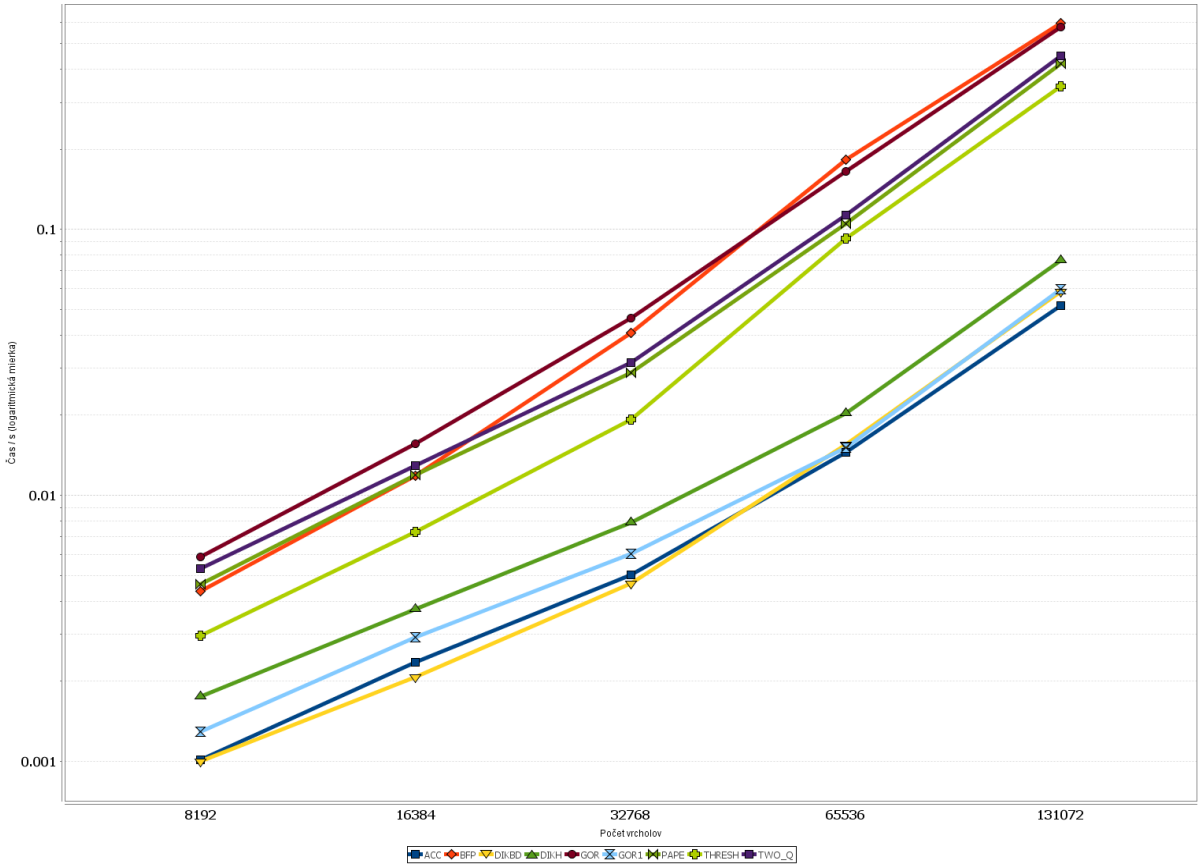


OA.11 Výsledky Rand P, **n***=131072,* **m***=524288.*

Horná hranica P	BFP	DIKBD	DIKH	GOR	GOR1	PAPE	THRESH	TWO_Q
1	0.4930	0.0278	0.0466	0.3633	0.3006	0.4582	0.3356	0.4408
1000	0.4929	0.0338	0.0568	0.3475	0.3003	0.4616	0.3040	0.4397
5000	0.4938	0.0751	0.1162	0.3485	0.3013	0.4589	0.3193	0.4397
10000	0.4935	0.1347	0.1977	0.3563	0.3002	0.4609	0.3449	0.4425
100000	0.4929	0.5798	0.7526	0.3684	0.3021	0.4570	0.8666	0.4400
1000000	0.4929	0.7820	1.0973	0.3701	0.3000	0.4594	1.1624	0.4392
5000000	0.4936	0.8244	1.1842	0.3678	0.3006	0.4598	1.1713	0.4379

TA.11 Výsledky Rand P, **n***=131072,* **m***=524288.*

A.12 Acyc Pos

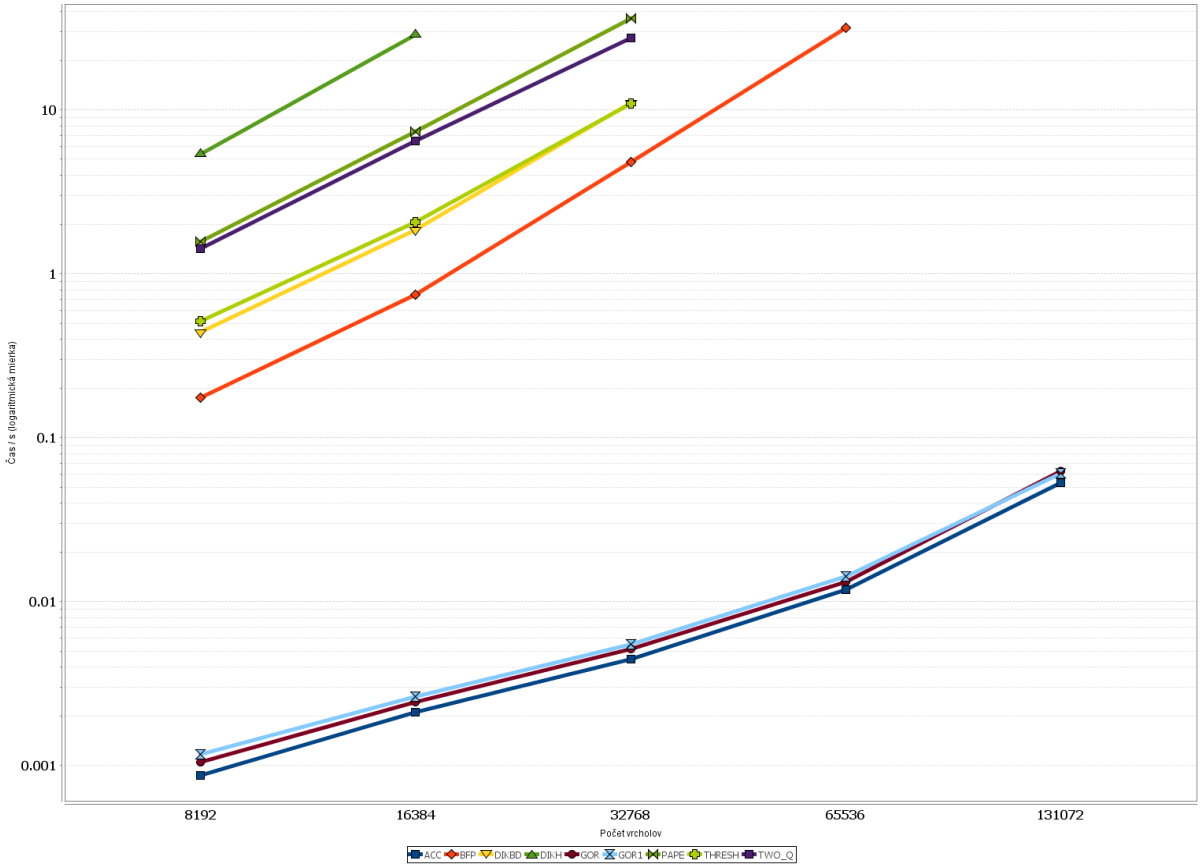


OA.12 Výsledky Acyc Pos. Acyklické siete s nezápornými hranami.

počet vrcholov / hrán	ACC	BFP	DIKBD	DIKH	GOR	GOR1	PAPE	THRESH	TWO_Q
8192 / 131072	0.0010	0.0044	0.0010	0.0017	0.0059	0.0013	0.0046	0.0030	0.0053
16384 / 262144	0.0024	0.0118	0.0021	0.0037	0.0156	0.0029	0.0119	0.0073	0.0129
32768 / 524288	0.0050	0.0407	0.0047	0.0079	0.0462	0.0060	0.0290	0.0192	0.0315
65536 / 1048576	0.0145	0.1823	0.0155	0.0203	0.1649	0.0151	0.1050	0.0923	0.1128
131072 / 2097152	0.0518	0.5957	0.0584	0.0766	0.5761	0.0597	0.4192	0.3450	0.4475

TA.12 Výsledky Acyc Pos. Acyklické siete s nezápornými hranami.

A.13 Acyc Neg

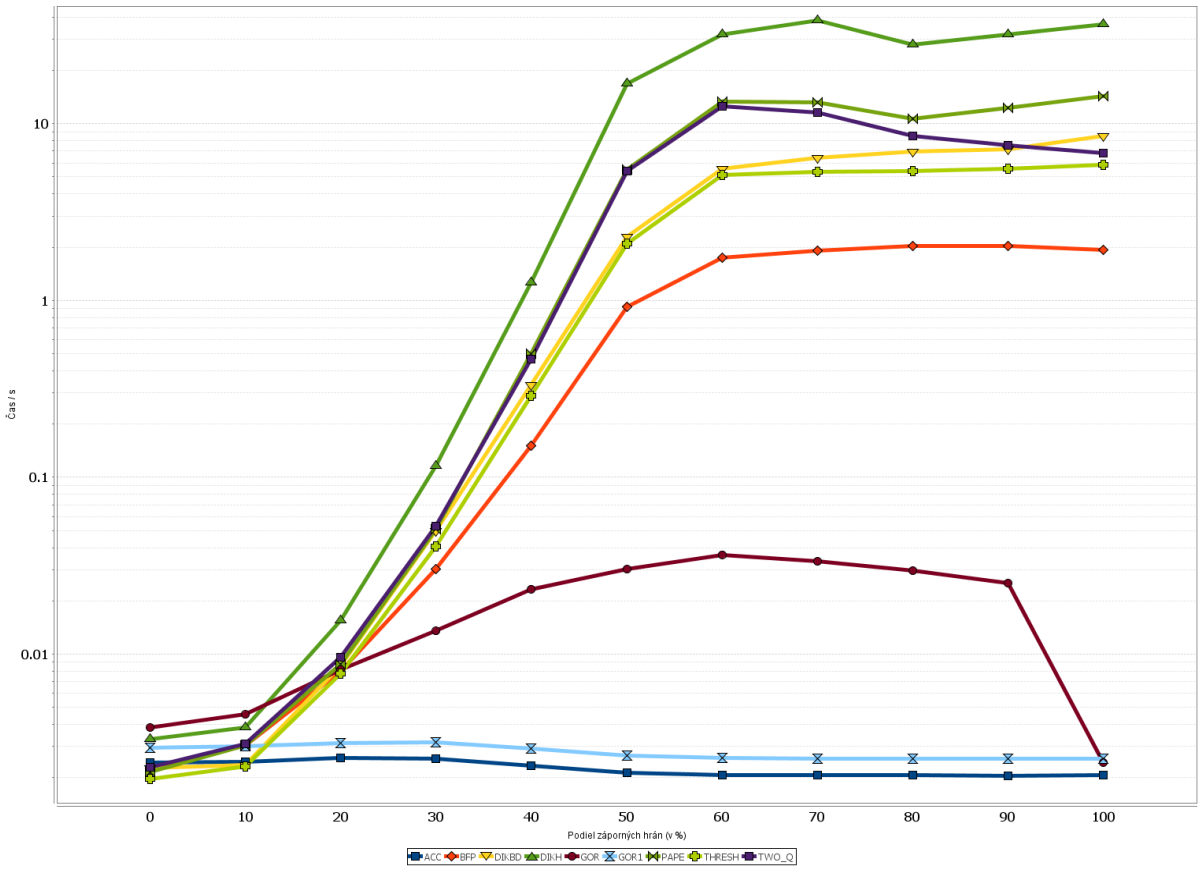


OA.13 Výsledky Acyc Neg. Acyklické siete s negatívnymi hranami.

počet vrcholov / hrán	ACC	BFP	DIKBD	DIKH	GOR	GOR1	PAPE	THRESH	TWO_Q
8192 / 131072	0.0009	0.1759	0.4406	5.3778	0.0010	0.0012	1.5805	0.5159	1.4268
16384 / 262144	0.0021	0.7430	1.8611	28.7456	0.0024	0.0026	7.3727	2.0733	6.4345
32768 / 524288	0.0045	4.8219	11.0929		0.0051	0.0055	36.3329	10.9446	27.4964
65536 / 1048576	0.0118	31.7578			0.0132	0.0143			
131072 / 2097152	0.0532				0.0626	0.0608			

TA.13 Výsledky Acyc Neg. Acyklické siete s negatívnymi hranami.

A.14 Acyc P:N

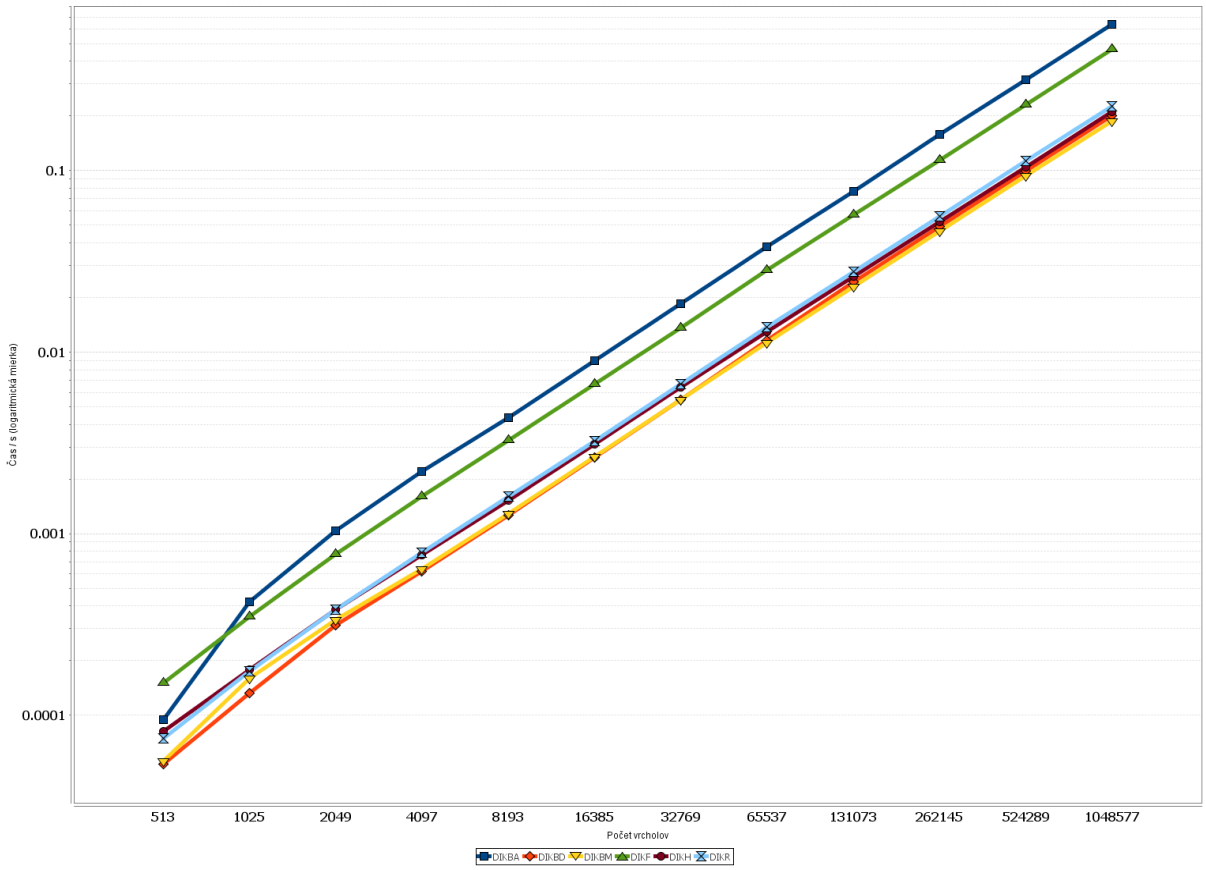


OA.14 Výsledky Acyc P:N, **n***=16384,* **m***=262144, rôzny pomer kladných a záporných ohodnotení.*

f (%)	ACC	BFP	DIKBD	DIKH	GOR	GOR1	PAPE	THRESH	TWO_Q
0	0.0024	0.0022	0.0023	0.0033	0.0038	0.0030	0.0021	0.0020	0.0023
10	0.0025	0.0031	0.0024	0.0038	0.0046	0.0030	0.0030	0.0023	0.0031
20	0.0026	0.0080	0.0086	0.0154	0.0081	0.0032	0.0088	0.0077	0.0096
30	0.0026	0.0302	0.0498	0.1153	0.0135	0.0032	0.0507	0.0408	0.0532
40	0.0023	0.1512	0.3317	1.2588	0.0233	0.0029	0.4982	0.2901	0.4664
50	0.0021	0.9248	2.3089	16.8966	0.0302	0.0027	5.5187	2.0926	5.3796
60	0.0021	1.7394	5.5427	32.0696	0.0365	0.0026	13.2776	5.1304	12.6049
70	0.0021	1.9213	6.4421	38.2749	0.0334	0.0026	13.1966	5.3574	11.5353
80	0.0021	2.0398	6.9512	28.0126	0.0296	0.0026	10.6469	5.3734	8.5584
90	0.0021	2.0393	7.1958	31.9483	0.0252	0.0026	12.3304	5.5604	7.5832
100	0.0021	1.9329	8.5393	36.6131	0.0024	0.0026	14.2949	5.8259	6.8128

TA.14 Výsledky Acyc P:N, **n***=16384,* **m***=262144, rôzny pomer kladných a záporných ohodnotení.*

A.15 Grid P Hard D

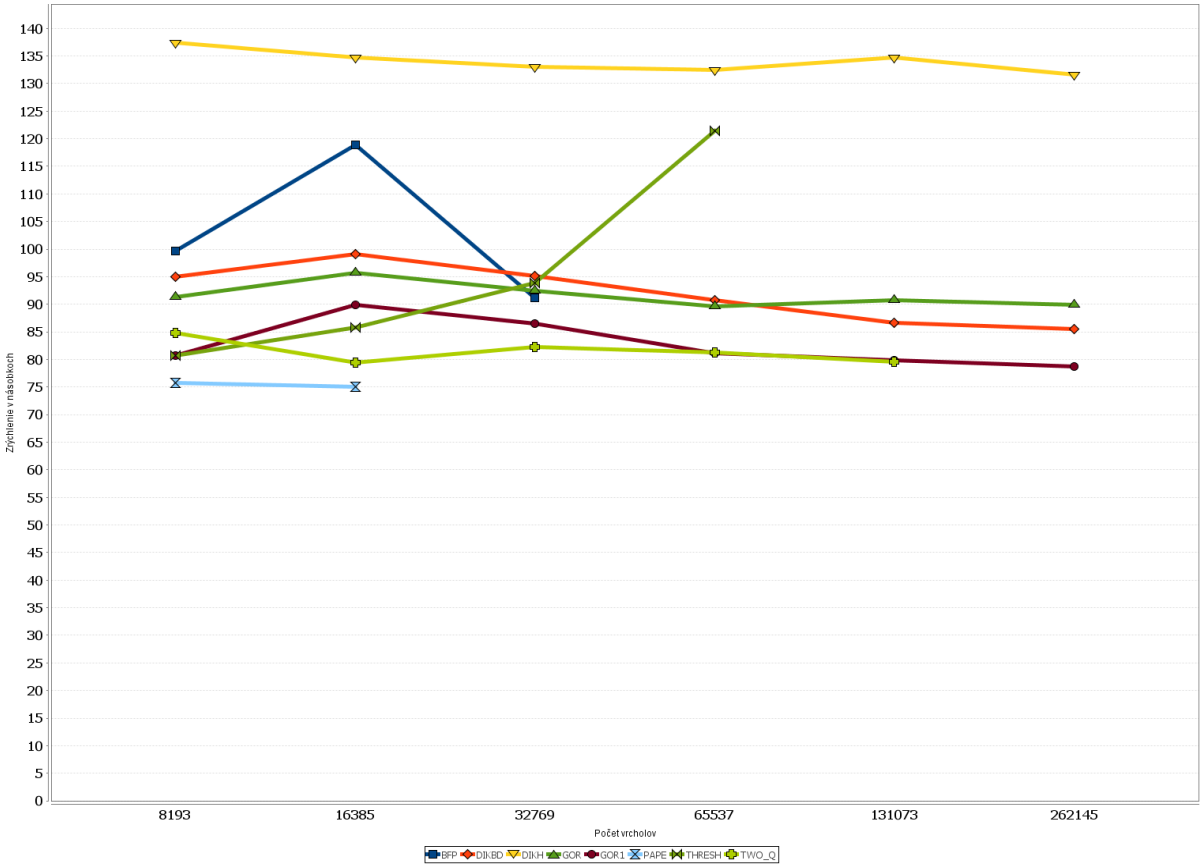


OA.15 Výsledky Grid P Hard pre varianty Dijkstrovho algoritmu

počet vrcholov / hrán	DIKBA	DIKBD	DIKBM	DIKF	DIKH	DIKR
513 / 2528	0.0001	0.0001	0.0001	0.0002	0.0001	0.0001
1025 / 6464	0.0004	0.0001	0.0002	0.0003	0.0002	0.0002
2049 / 14656	0.0010	0.0003	0.0003	0.0008	0.0004	0.0004
4097 / 31040	0.0022	0.0006	0.0006	0.0016	0.0008	0.0008
8193 / 63808	0.0044	0.0013	0.0013	0.0033	0.0015	0.0016
16385 / 129344	0.0090	0.0026	0.0027	0.0067	0.0031	0.0032
32769 / 260416	0.0185	0.0055	0.0055	0.0136	0.0064	0.0067
65537 / 522560	0.0380	0.0115	0.0112	0.0283	0.0129	0.0137
131073 / 1046848	0.0770	0.0244	0.0229	0.0568	0.0260	0.0277
262145 / 2095424	0.1581	0.0492	0.0464	0.1144	0.0523	0.0558
524289 / 4192576	0.3173	0.0994	0.0937	0.2311	0.1046	0.1127
1048577 / 8386880	0.6409	0.2001	0.1877	0.4670	0.2097	0.2260

TA.15 Výsledky Grid P Hard pre varianty Dijkstrovho algoritmu

A.16 Zrýchlenie DIKH na Grid P Hard

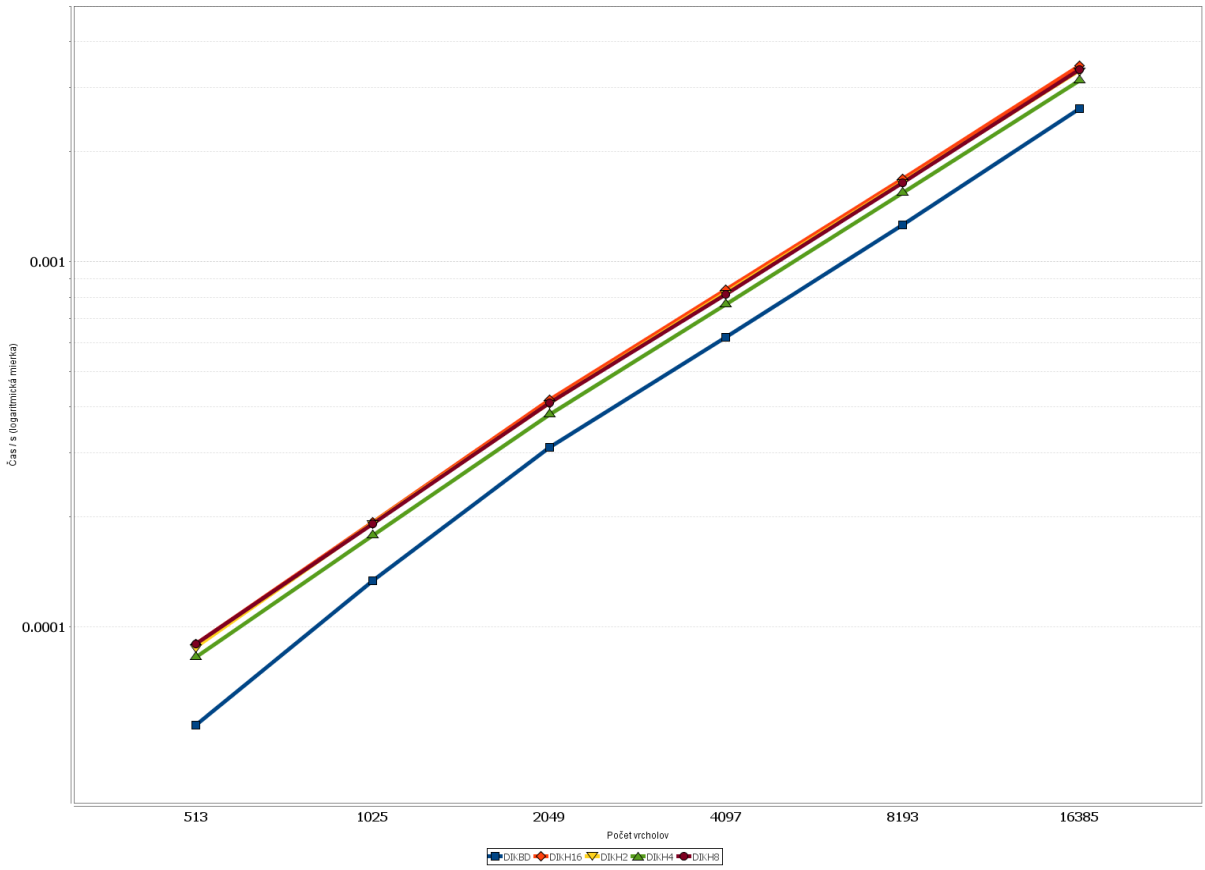


OA.16 Zrýchlenie implementácií oproti pôvodným testom.

počet vrcholov / hrán	BFP	DIKBD	DIKH	GOR	GOR1	PAPE	THRESH	TWO_Q
8193 / 63808	99.6135	94.9326	137.4463	91.3230	80.7335	75.7534	80.7606	84.8093
16385 / 129344	118.8765	99.1284	134.6968	95.7407	89.8797	75.0783	85.7700	79.3845
32769 / 260416	91.1720	95.1613	133.0823	92.4254	86.4845		93.8377	82.3228
65537 / 522560		90.6843	132.4509	89.6594	81.1360		121.3919	81.2015
131073 / 1046848		86.6096	134.7220	90.7623	79.8424			79.5133
262145 / 2095424		85.4710	131.5872	89.8288	78.7080			

TA.16 Zrýchlenie implementácií oproti pôvodným testom.

A.17 Grid P Hard D mini

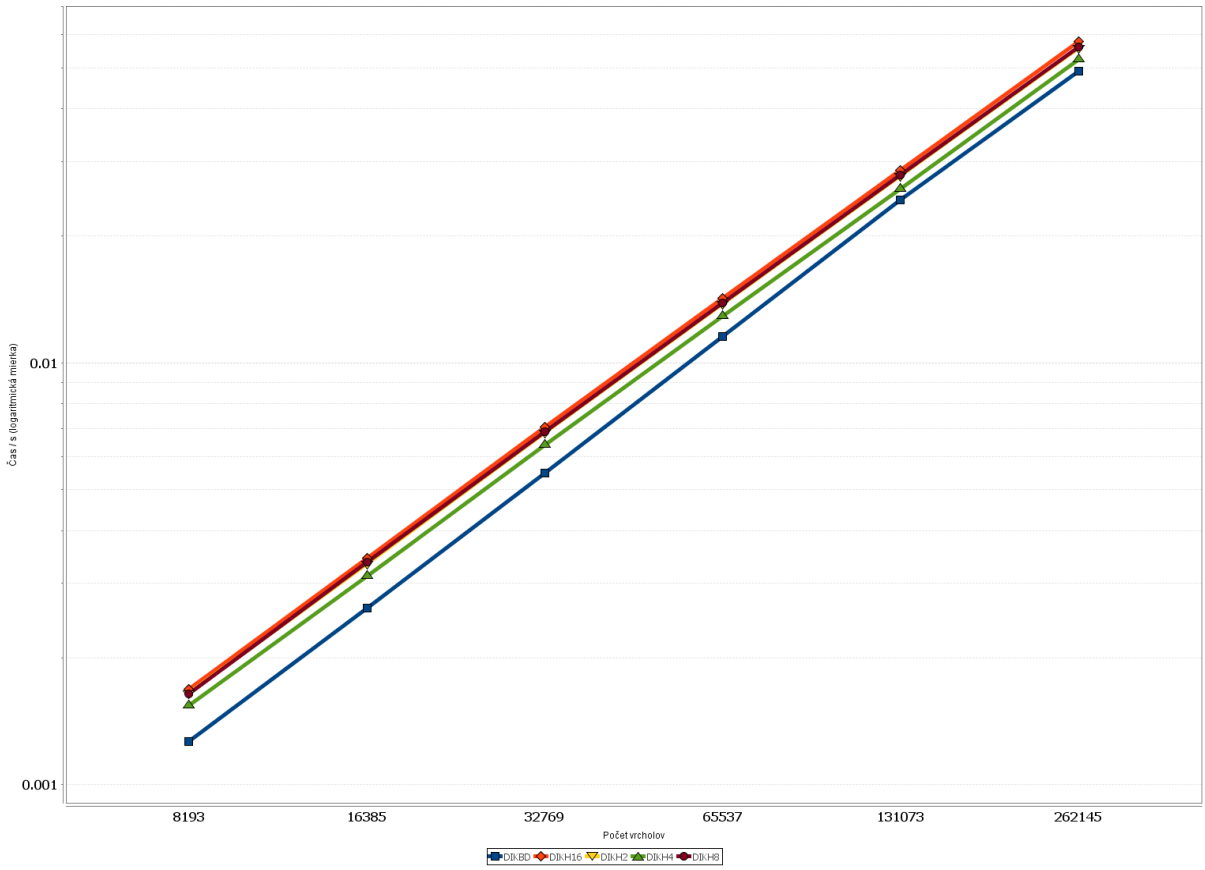


OA.17 Varianty DIKH a porovnanie s DIKBD na Grid P Hard, najmenšie vstupy.

počet vrcholov / hrán	DIKBD	DIKH16	DIKH2	DIKH4	DIKH8
513 / 2528	0.0001	0.0001	0.0001	0.0001	0.0001
1025 / 6464	0.0001	0.0002	0.0002	0.0002	0.0002
2049 / 14656	0.0003	0.0004	0.0004	0.0004	0.0004
4097 / 31040	0.0006	0.0008	0.0008	0.0008	0.0008
8193 / 63808	0.0013	0.0017	0.0017	0.0015	0.0016
16385 / 129344	0.0026	0.0034	0.0033	0.0031	0.0034

TA.17 Varianty DIKH a porovnanie s DIKBD na Grid P Hard, najmenšie vstupy.

A.18 Grid P Hard D medium

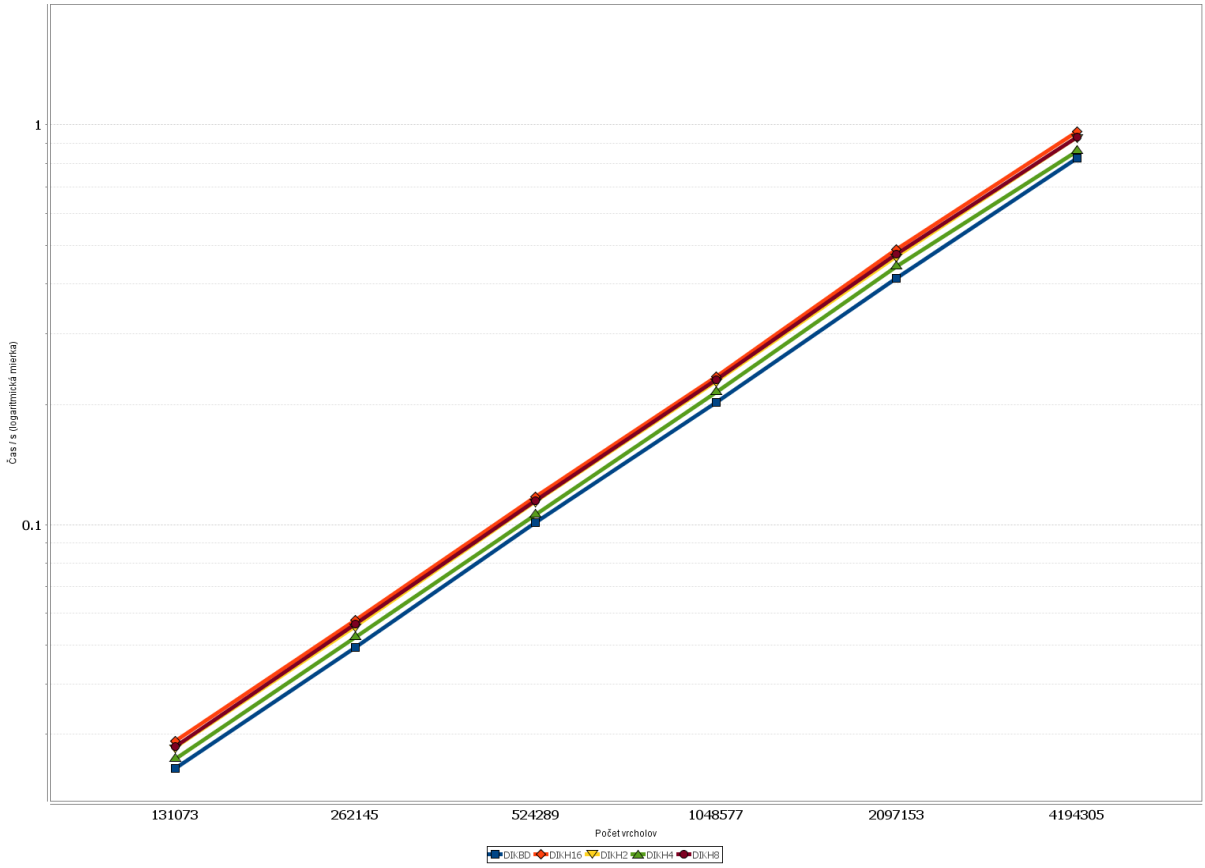


OA.18 Varianty DIKH a porovnanie s DIKBD na Grid P Hard, stredné vstupy.

počet vrcholov / hrán	DIKBD	DIKH16	DIKH2	DIKH4	DIKH8
8193 / 63808	0.0013	0.0017	0.0016	0.0015	0.0016
16385 / 129344	0.0026	0.0034	0.0033	0.0031	0.0034
32769 / 260416	0.0055	0.0070	0.0068	0.0064	0.0069
65537 / 522560	0.0115	0.0142	0.0138	0.0129	0.0139
131073 / 1046848	0.0244	0.0286	0.0277	0.0259	0.0279
262145 / 2095424	0.0493	0.0577	0.0558	0.0524	0.0562

TA.18 Varianty DIKH a porovnanie s DIKBD na Grid P Hard, stredné vstupy.

A.19 Grid P Hard D maxi

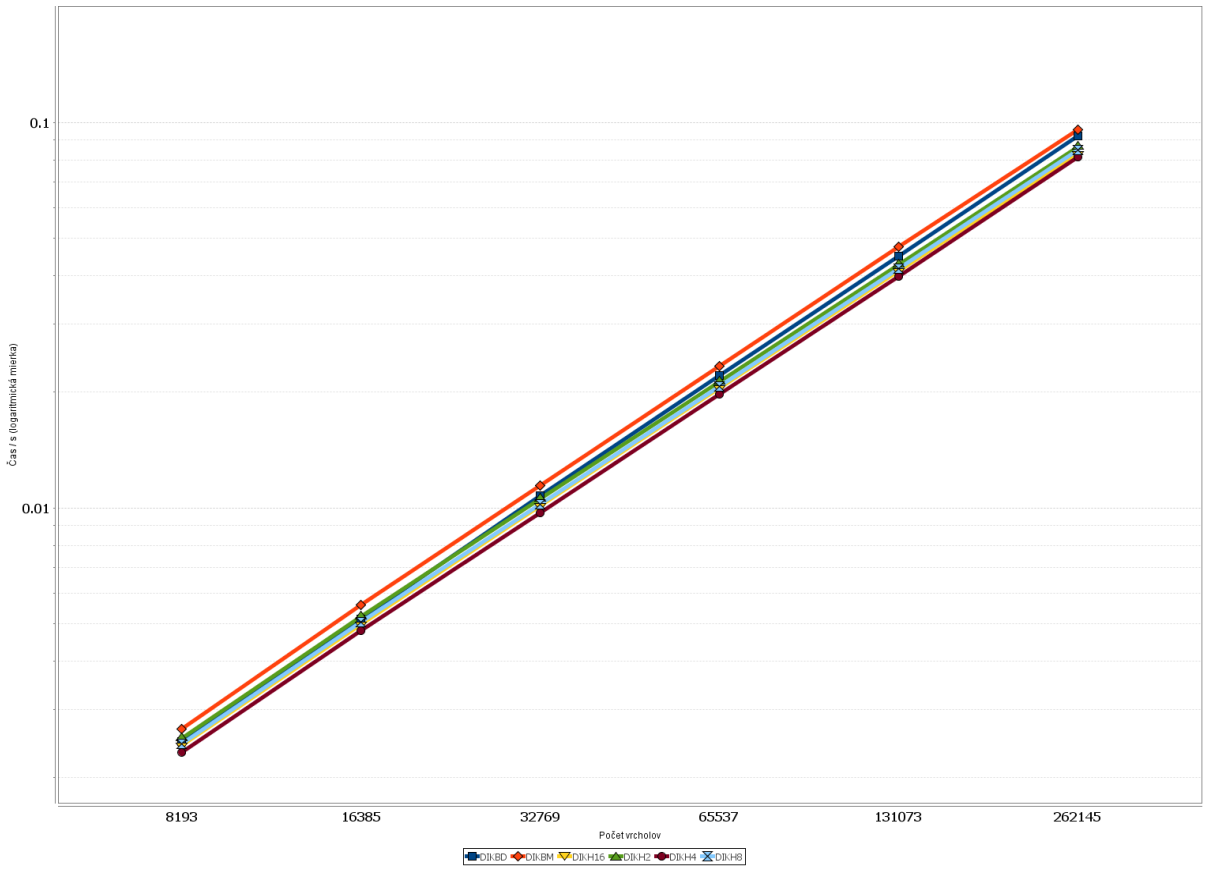


OA.19 Varianty DIKH a porovnanie s DIKBD na Grid P Hard, najväčšie vstupy.

počet vrcholov / hrán	DIKBD	DIKH16	DIKH2	DIKH4	DIKH8
131073 / 1046848	0.0245	0.0287	0.0277	0.0259	0.0279
262145 / 2095424	0.0493	0.0576	0.0558	0.0523	0.0564
524289 / 4192576	0.1011	0.1175	0.1145	0.1058	0.1147
1048577 / 8386880	0.2027	0.2343	0.2281	0.2147	0.2306
2097153 / 16775488	0.4128	0.4875	0.4686	0.4416	0.4751
4194305 / 33552704	0.8275	0.9608	0.9335	0.8612	0.9319

TA.19 Varianty DIKH a porovnanie s DIKBD na Grid P Hard, najväčšie vstupy.

A.20 Grid P Harder D medium

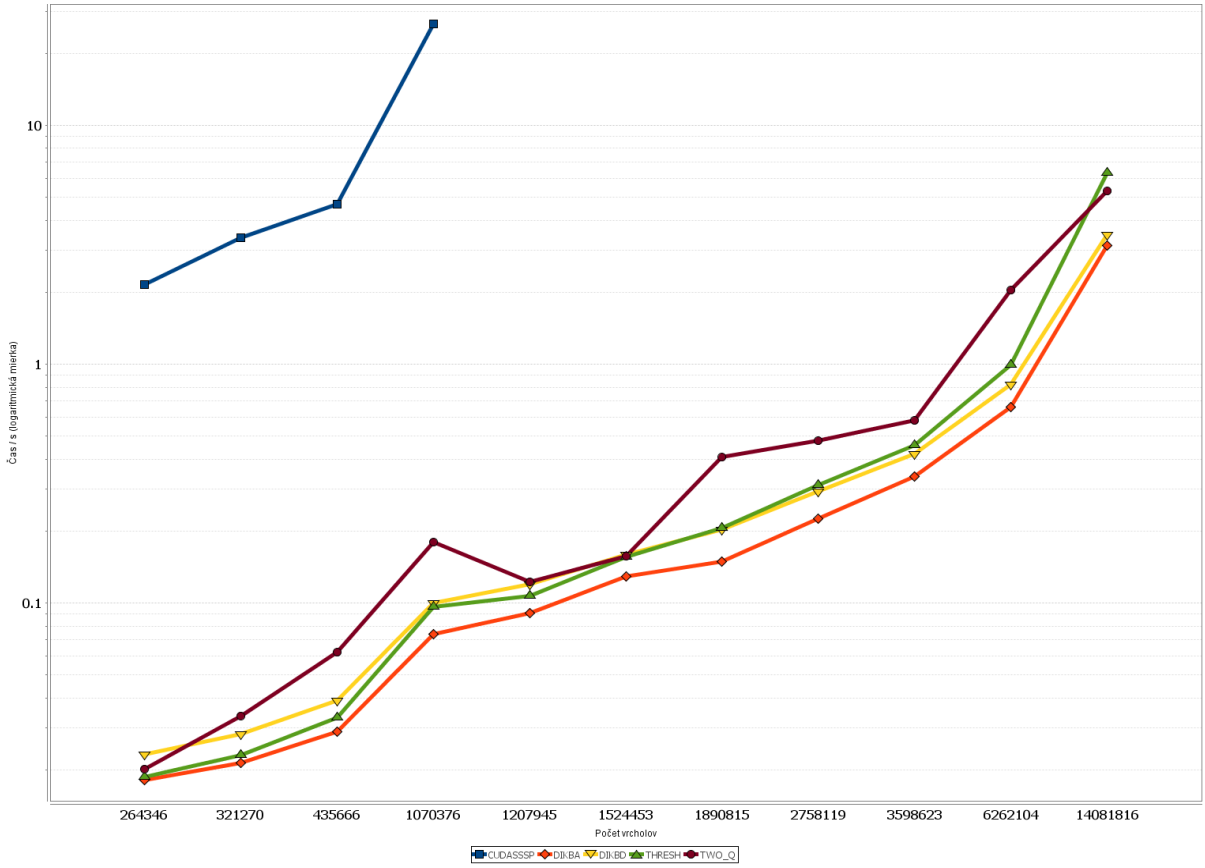


TA.20 Varianty DIKH a porovnanie s DIKBD na upravenom Grid P Hard.

počet vrcholov / hrán	DIKBD	DIKBM	DIKH16	DIKH2	DIKH4	DIKH8
8193 / 219600	0.0025	0.0027	0.0024	0.0025	0.0023	0.0024
16385 / 440784	0.0052	0.0056	0.0050	0.0052	0.0048	0.0051
32769 / 883152	0.0108	0.0114	0.0101	0.0105	0.0097	0.0102
65537 / 1767888	0.0220	0.0233	0.0204	0.0213	0.0197	0.0207
131073 / 3537360	0.0451	0.0476	0.0413	0.0428	0.0398	0.0418
262145 / 7076304	0.0925	0.0957	0.0830	0.0865	0.0813	0.0847

OA.20 Varianty DIKH a porovnanie s DIKBD na upravenom Grid P Hard.

A.21 State Group 2

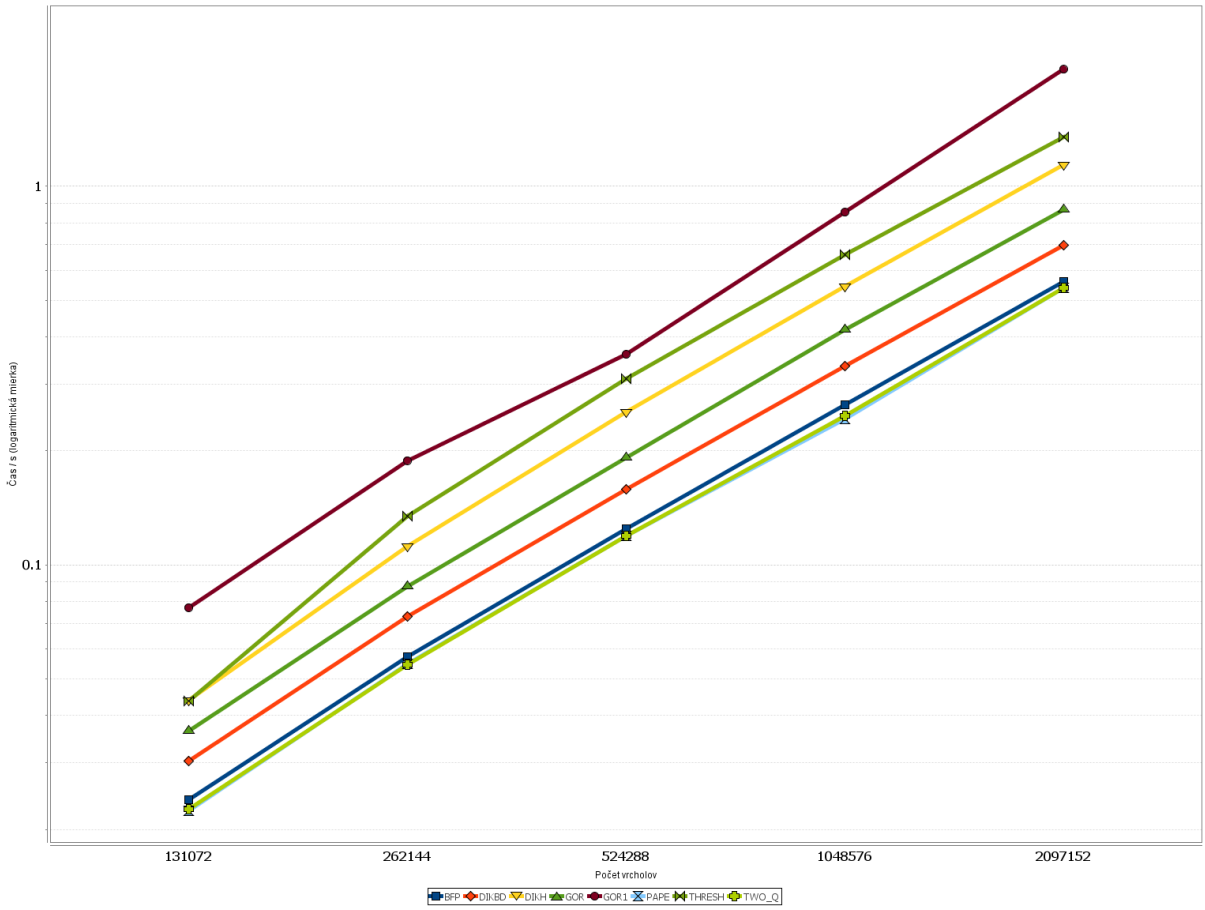


OA.21 Testy na cestných sieťach štátov a regiónov USA.

počet vrcholov / hrán	CUDASSSP	DIKBA	DIKBD	THRESH	TWO_Q
264346 / 733846 (NY)	2.1449	0.0182	0.0232	0.0187	0.0202
321270 / 800172 (BAY)	3.3921	0.0215	0.0284	0.0232	0.0337
435666 / 1057066 (COL)	4.6577	0.0290	0.0390	0.0332	0.0621
1070376 / 2712798 (FLA)	26.6124	0.0742	0.0998	0.0960	0.1797
1207945 / 2840208 (NW)		0.0909	0.1195	0.1073	0.1227
1524453 / 3897636 (NE)		0.1288	0.1592	0.1564	0.1569
1890815 / 4657742 (CAL)		0.1493	0.2036	0.2066	0.4077
2758119 / 6885658 (LKS)		0.2247	0.2924	0.3117	0.4772
3598623 / 8778114 (E)		0.3393	0.4219	0.4563	0.5800
6262104 / 15248146 (W)		0.6612	0.8238	0.9950	2.0354
14081816 / 34292496 (CTR)		3.1309	3.4908	6.3257	5.2980

TA.21 Testy na cestných sieťach štátov a regiónov USA.

A.22 R-MAT 1:1

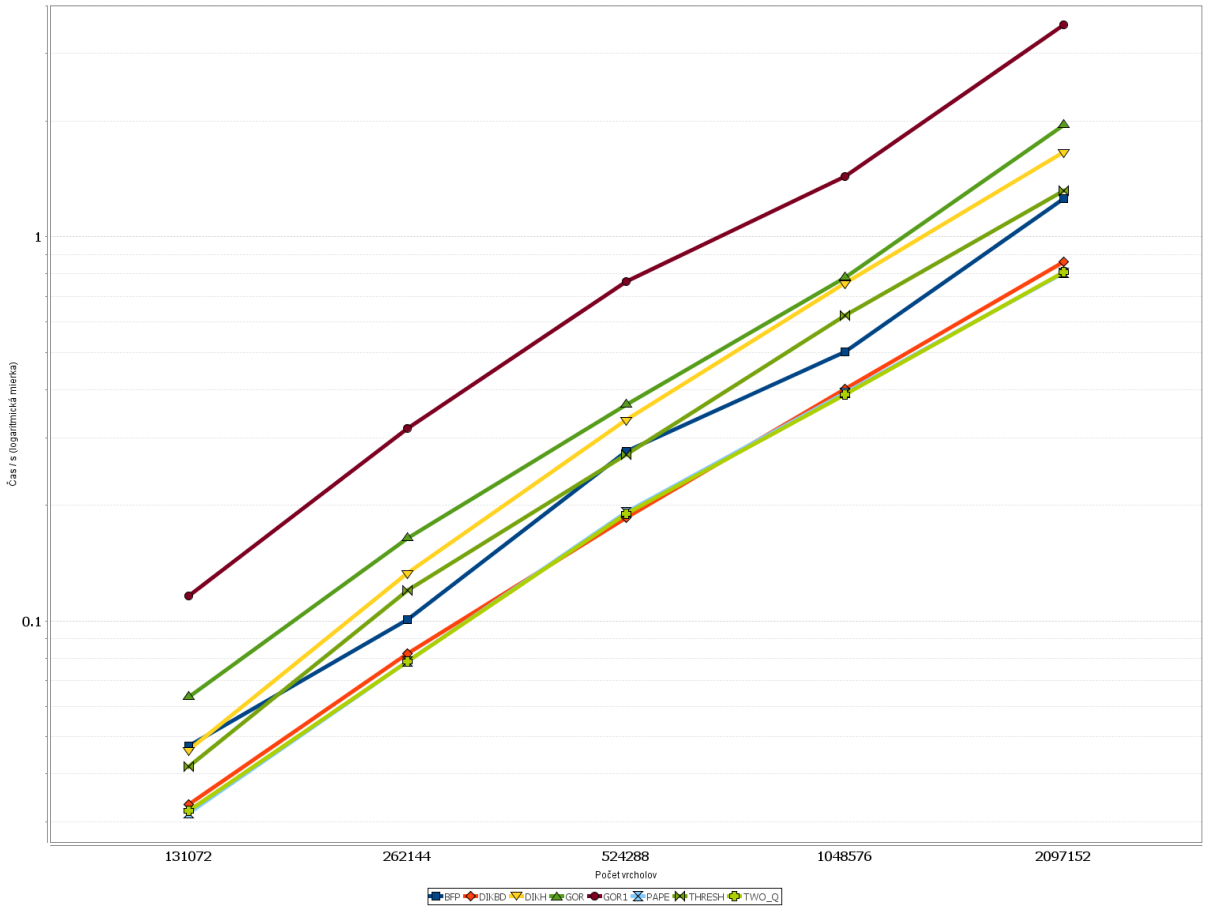


OA.22 Test na bezškálovej sieti, **a**=**c**=*0.05,* **b**=**d**=*0.45.*

vrcholy / hrany	BFP	DIKBD	DIKH	GOR	GOR1	PAPE	THRESH	TWO_Q
131072 / 351390	0.0240	0.0304	0.0439	0.0364	0.0771	0.0224	0.0438	0.0226
262144 / 704558	0.0571	0.0732	0.1122	0.0876	0.1886	0.0549	0.1345	0.0545
524288 / 1414374	0.1245	0.1583	0.2535	0.1915	0.3596	0.1192	0.3093	0.1189
1048576 / 2836636	0.2650	0.3347	0.5432	0.4176	0.8535	0.2431	0.6590	0.2479
2097152 / 5696084	0.5589	0.6971	1.1401	0.8649	2.0335	0.5375	1.3478	0.5393

TA.22 Test na bezškálovej sieti, **a**=**c**=*0.05,* **b**=**d**=*0.45.*

A.23 R-MAT r

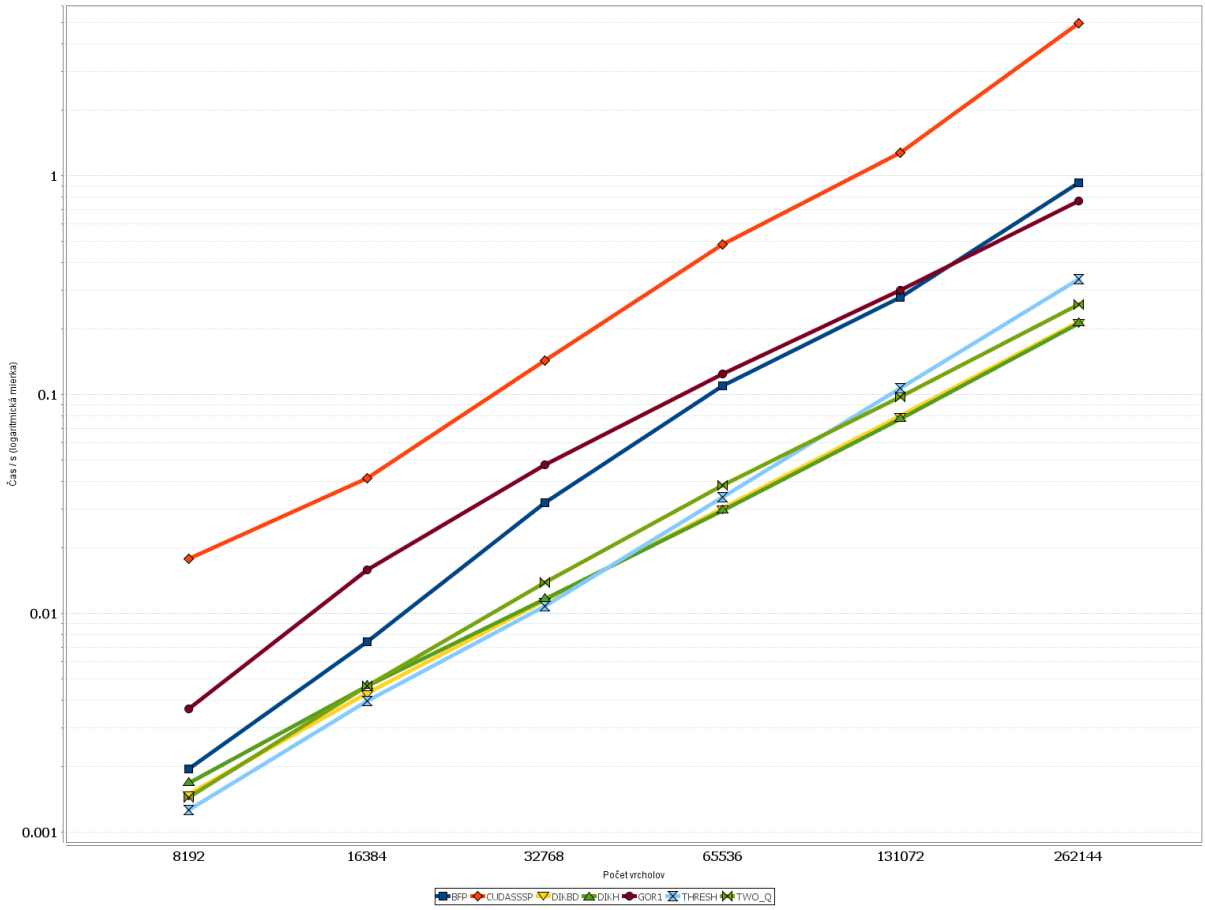


OA.23 Test na bezškálovej sieti, **a***=0.45,* **b**=**c**=*0.15,* **d**=*0.25.*

vrcholy / hrany	BFP	DIKBD	DIKH	GOR	GOR1	PAPE	THRESH	TWO_Q
131072 / 342024	0.0474	0.0333	0.0462	0.0634	0.1162	0.0316	0.0418	0.0320
262144 / 686158	0.1009	0.0824	0.1331	0.1638	0.3176	0.0786	0.1199	0.0785
524288 / 1377662	0.2764	0.1859	0.3339	0.3652	0.7646	0.1918	0.2709	0.1898
1048576 / 2765826	0.5008	0.4016	0.7562	0.7814	1.4345	0.3921	0.6251	0.3882
2097152 / 5550722	1.2557	0.8595	1.6601	1.9502	3.5635	0.8049	1.3199	0.8104

TA.23 Test na bezškálovej sieti, **a***=0.45,* **b**=**c**=*0.15,* **d**=*0.25.*

A.24 SSCA#2 base

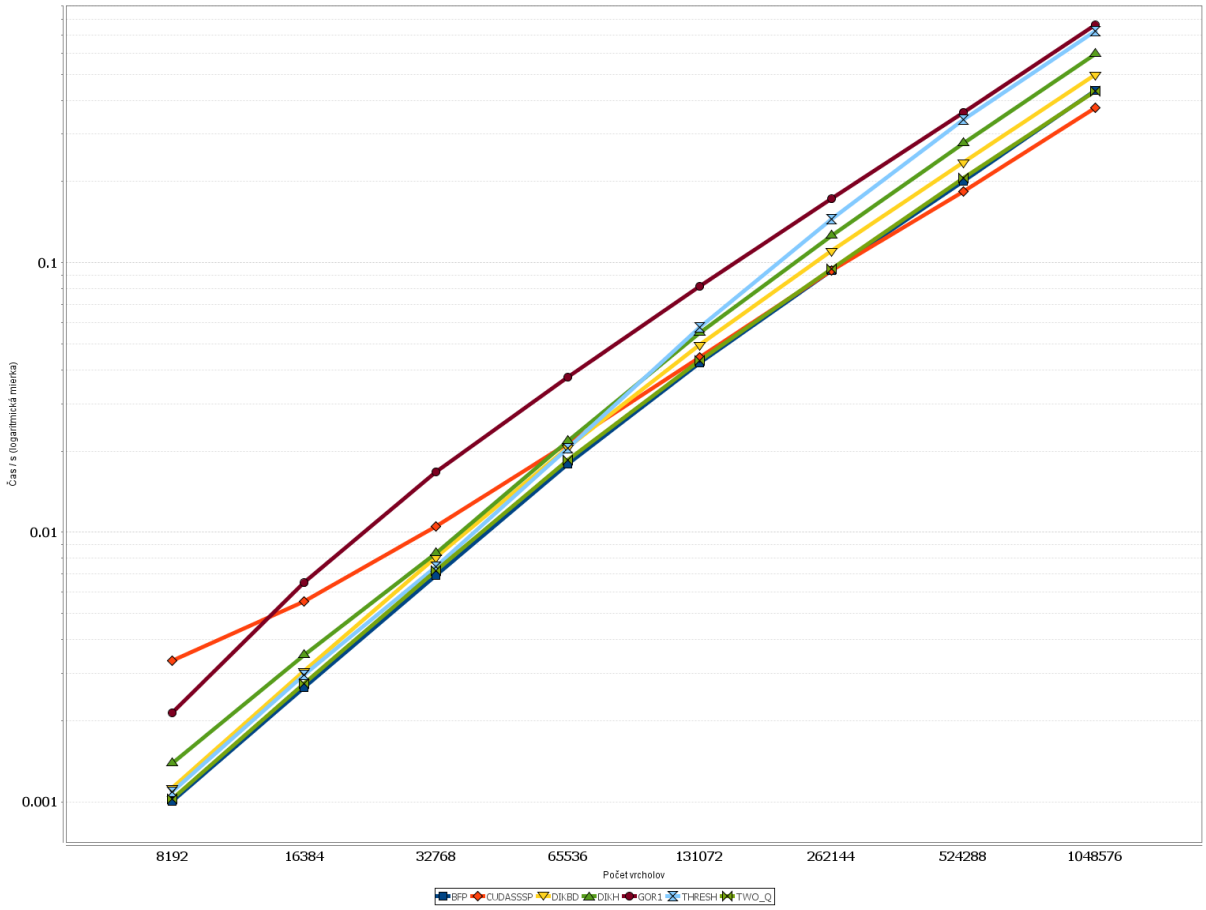


OA.24 Základný graf SSCA#2.

vrcholy / hrany	BFP	CUDASSSP	DIKBD	DIKH	GOR1	THRESH	TWO_Q
8192 / 198018	0.0020	0.0178	0.0015	0.0017	0.0037	0.0013	0.0014
16384 / 499080	0.0074	0.0413	0.0043	0.0047	0.0158	0.0040	0.0047
32768 / 1297958	0.0321	0.1428	0.0114	0.0116	0.0478	0.0108	0.0138
65536 / 3242958	0.1099	0.4836	0.0301	0.0296	0.1243	0.0340	0.0385
131072 / 8109856	0.2789	1.2763	0.0798	0.0774	0.2996	0.1072	0.0974
262144 / 20796766	0.9306	4.9643	0.2156	0.2117	0.7653	0.3355	0.2576

TA.24 Základný graf SSCA#2.

A.25 SSCA#2 base s

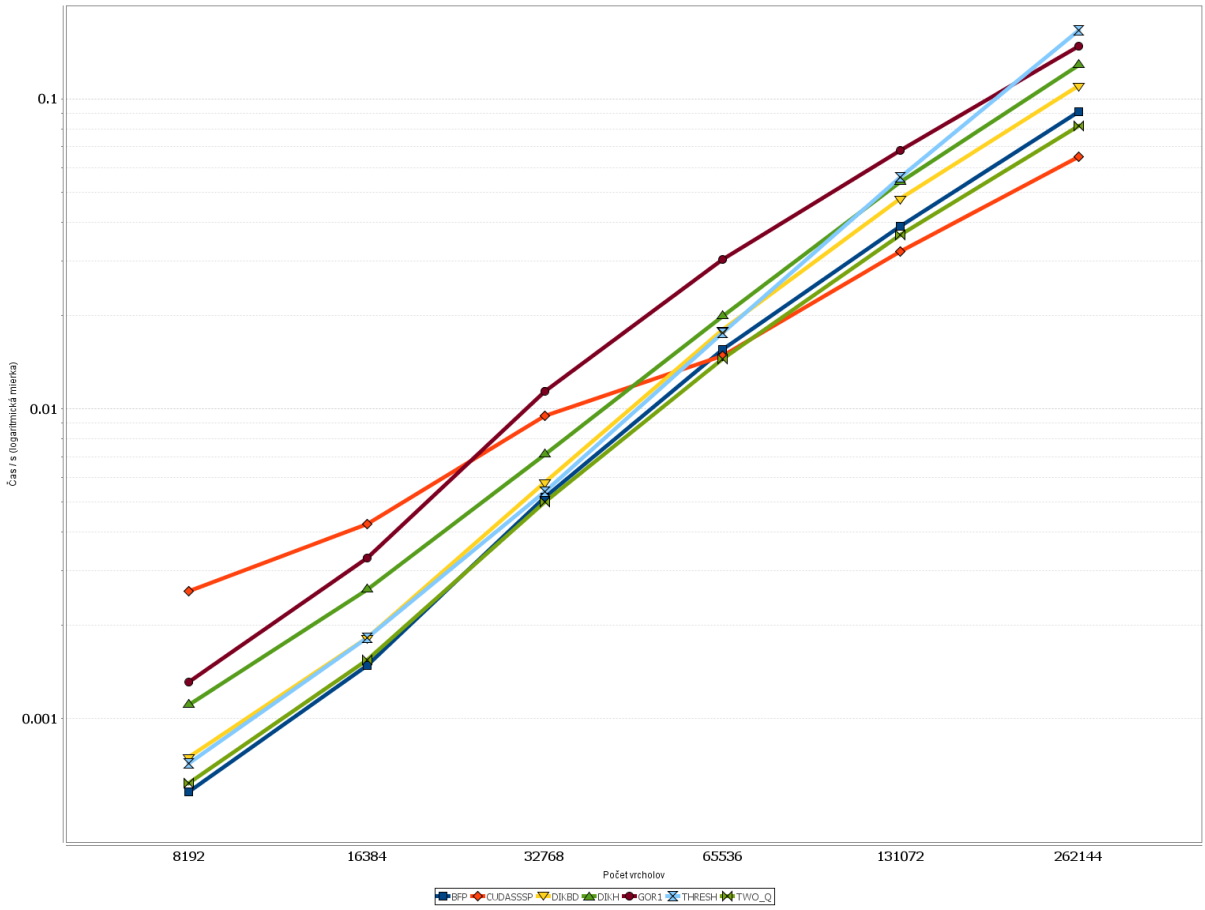


OA.25 Základný graf SSCA#2 s krátkymi hranami.

vrcholy / hrany	BFP	CUDASSSP	DIKBD	DIKH	GOR1	THRESH	TWO_Q
8192 / 227633	0.0010	0.0033	0.0011	0.0014	0.0021	0.0011	0.0010
16384 / 449235	0.0026	0.0056	0.0031	0.0035	0.0065	0.0029	0.0028
32768 / 905164	0.0069	0.0105	0.0081	0.0084	0.0168	0.0075	0.0072
65536 / 1803536	0.0179	0.0213	0.0207	0.0218	0.0376	0.0205	0.0185
131072 / 3586892	0.0424	0.0447	0.0495	0.0550	0.0816	0.0576	0.0433
262144 / 7147872	0.0935	0.0933	0.1104	0.1260	0.1723	0.1446	0.0951
524288 / 14323170	0.2003	0.1838	0.2359	0.2776	0.3616	0.3386	0.2059
1048576 / 28621484	0.4353	0.3754	0.4971	0.5934	0.7606	0.7214	0.4336

TA.25 Základný graf SSCA#2 s krátkymi hranami.

A.26 SSCA#2 snow

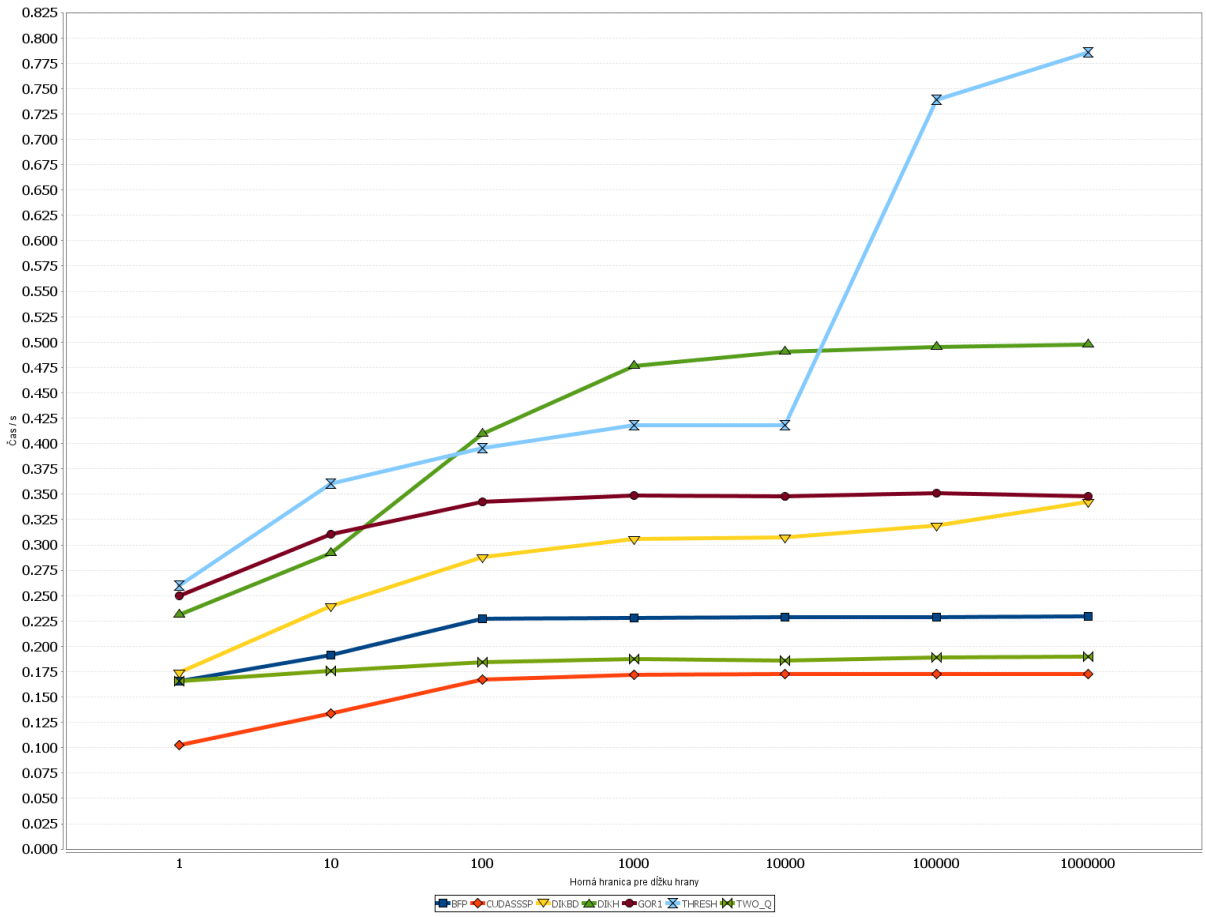


OA.26 Graf SSCA#2 s malými klikami.

vrcholy / hrany	BFP	CUDASSSP	DIKBD	DIKH	GOR1	THRESH	TWO_Q
8192 / 104659	0.0006	0.0026	0.0008	0.0011	0.0013	0.0007	0.0006
16384 / 210733	0.0015	0.0042	0.0018	0.0026	0.0033	0.0018	0.0015
32768 / 420113	0.0052	0.0095	0.0058	0.0071	0.0114	0.0054	0.0050
65536 / 836816	0.0156	0.0148	0.0180	0.0199	0.0303	0.0176	0.0145
131072 / 1665444	0.0388	0.0322	0.0476	0.0540	0.0682	0.0558	0.0363
262144 / 3339021	0.0910	0.0649	0.1101	0.1285	0.1478	0.1658	0.0819

TA.26 Graf SSCA#2 s malými klikami.

A.27 SSCA#2 snow L

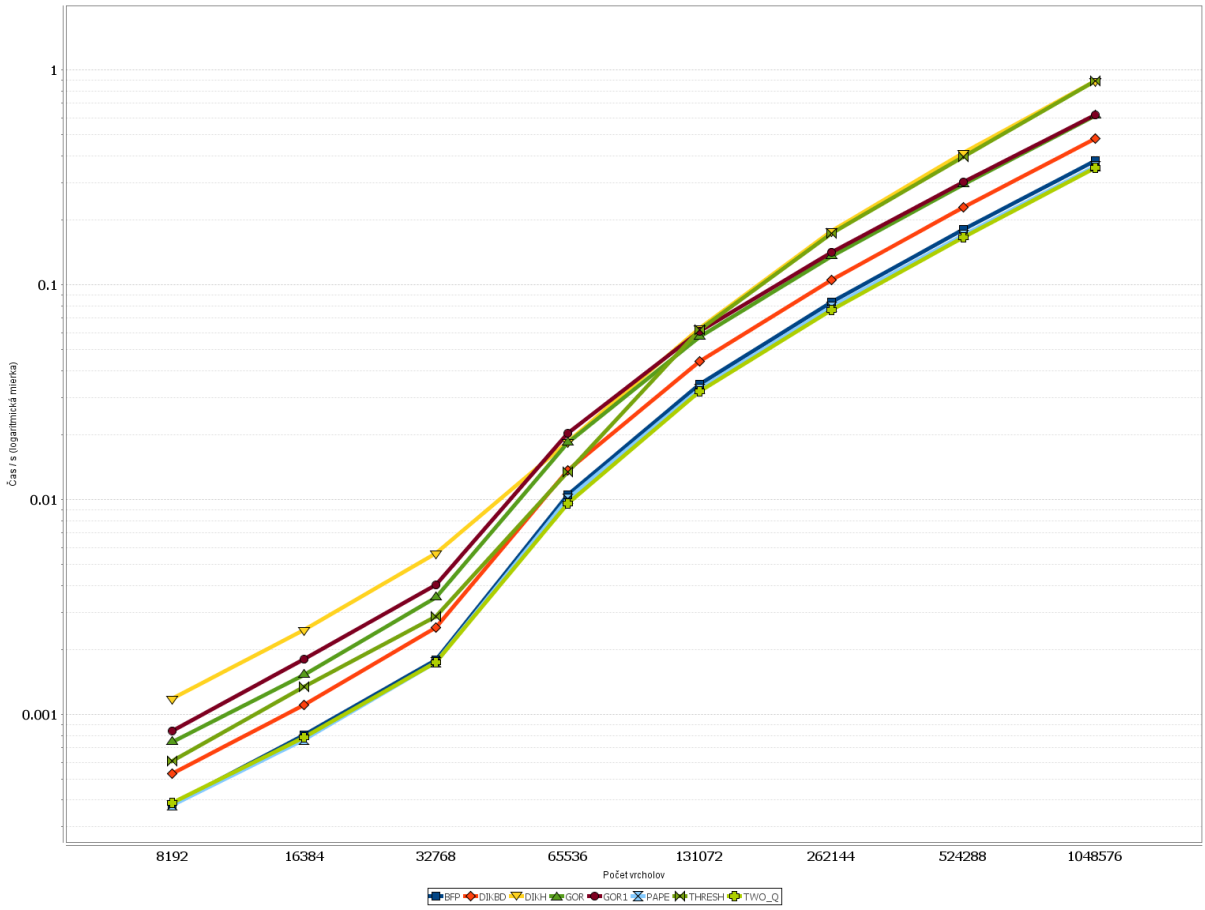


OA.27 Graf SSCA#2 s malými klikami pre rôzne dĺžky hrán, $\mathbf{n}=2^{20}$, $\mathbf{m}=5\cdot 2^{20}$.

Maximum dĺžky hrany	BFP	CUDASSSP	DIKBD	DIKH	GOR1	THRESH	TWO_Q
1	0.1659	0.1029	0.1738	0.2312	0.2500	0.2601	0.1659
10	0.1911	0.1338	0.2399	0.2920	0.3106	0.3608	0.1759
100	0.2276	0.1669	0.2883	0.4091	0.3423	0.3956	0.1847
1000	0.2278	0.1720	0.3061	0.4766	0.3487	0.4181	0.1872
10000	0.2285	0.1724	0.3074	0.4903	0.3475	0.4177	0.1862
100000	0.2288	0.1725	0.3191	0.4955	0.3508	0.7393	0.1887
1000000	0.2292	0.1724	0.3422	0.4974	0.3476	0.7858	0.1897

TA.27 Graf SSCA#2 s malými klikami pre rôzne dĺžky hrán, $\mathbf{n}=2^{20}$, $\mathbf{m}=5\cdot 2^{20}$.

A.28 ER 4

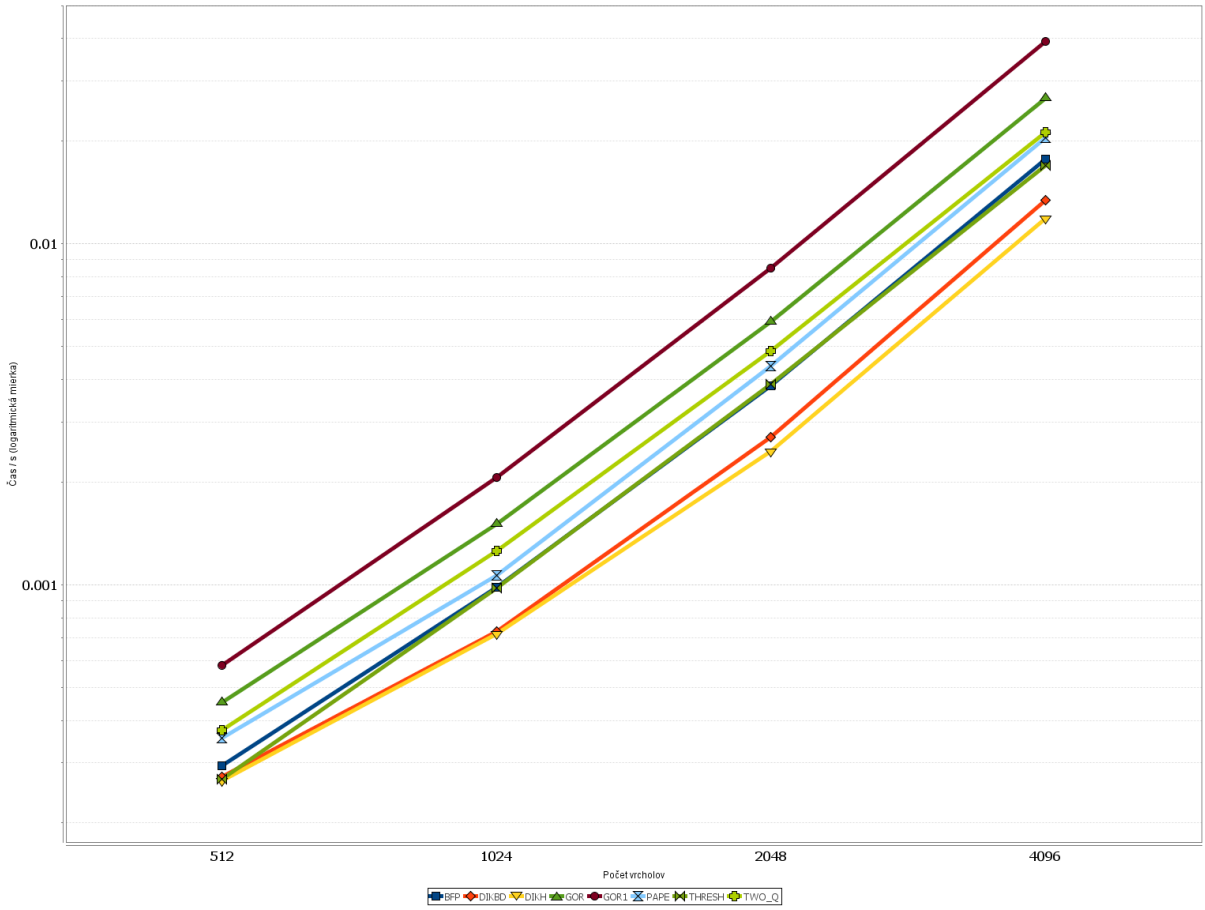


OA.28 Riedky náhodný graf, **m**=*4***n**.

vrcholy / hrany	BFP	DIKBD	DIKH	GOR	GOR1	PAPE	THRESH	TWO_Q
8192 / 33371	0.0004	0.0005	0.0012	0.0007	0.0008	0.0004	0.0006	0.0004
16384 / 66907	0.0008	0.0011	0.0025	0.0015	0.0018	0.0008	0.0013	0.0008
32768 / 134008	0.0018	0.0025	0.0056	0.0035	0.0040	0.0018	0.0029	0.0017
65536 / 268659	0.0106	0.0137	0.0186	0.0184	0.0204	0.0101	0.0135	0.0096
131072 / 538663	0.0345	0.0439	0.0629	0.0571	0.0610	0.0327	0.0621	0.0321
262144 / 1080101	0.0833	0.1054	0.1784	0.1363	0.1425	0.0788	0.1735	0.0766
524288 / 2166219	0.1814	0.2292	0.4134	0.2937	0.3014	0.1706	0.3946	0.1660
1048576 / 4346726	0.3796	0.4817	0.8899	0.6135	0.6166	0.3565	0.8932	0.3498

TA.28 Riedky náhodný graf, **m**=*4***n**.

A.29 ER ¼

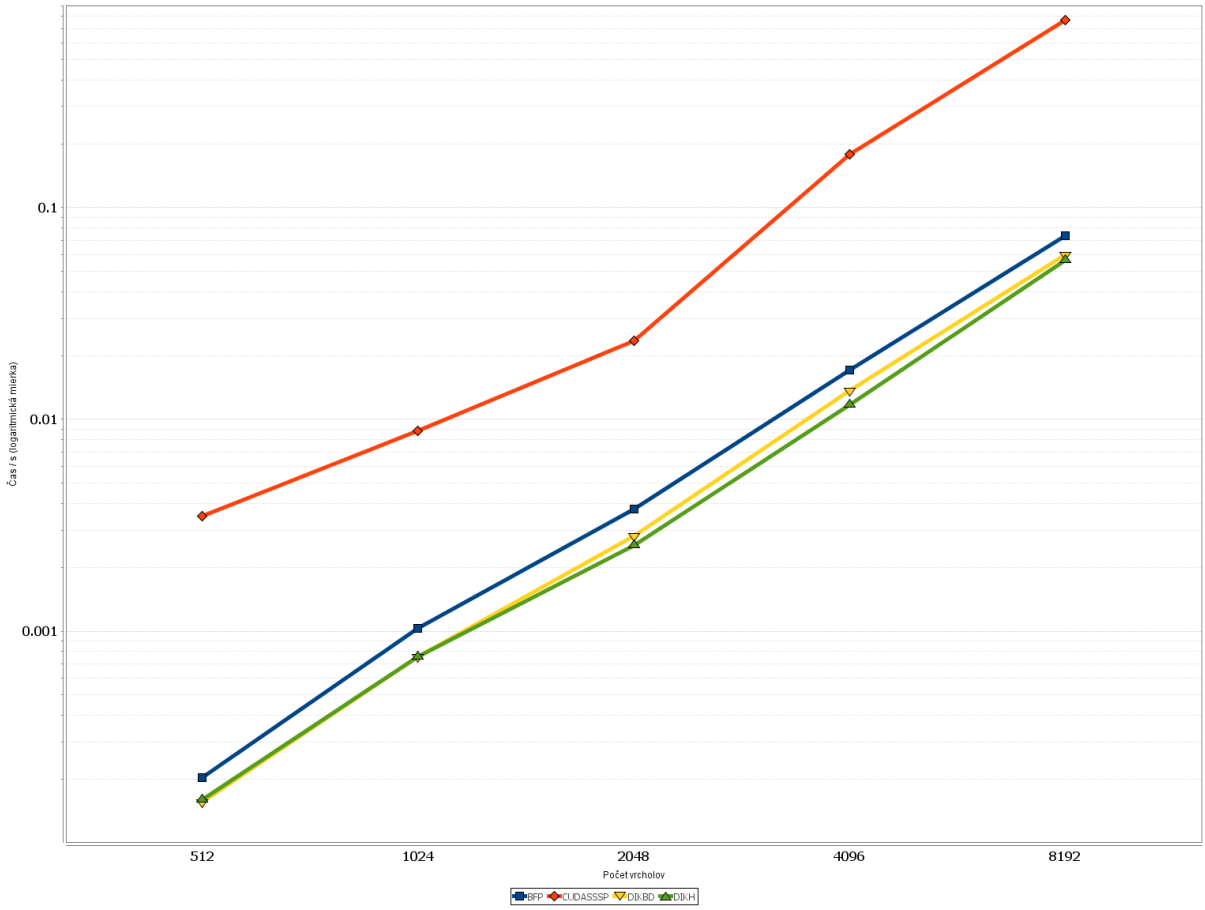


OA.29 Hustý náhodný graf, $\mathbf{m}=\mathbf{n}^2/4$.

vrcholy / hrany	BFP	DIKBD	DIKH	GOR	GOR1	PAPE	THRESH	TWO_Q
512 / 65536	0.0003	0.0003	0.0003	0.0005	0.0006	0.0004	0.0003	0.0004
1024 / 262144	0.0010	0.0007	0.0007	0.0015	0.0021	0.0011	0.0010	0.0013
2048 / 1048576	0.0038	0.0027	0.0025	0.0059	0.0085	0.0044	0.0039	0.0048
4096 / 4194304	0.0177	0.0134	0.0119	0.0268	0.0392	0.0205	0.0169	0.0212

TA.29 Hustý náhodný graf, $\mathbf{m}=\mathbf{n}^2/4$.

A.30 CUDA Rand ¼

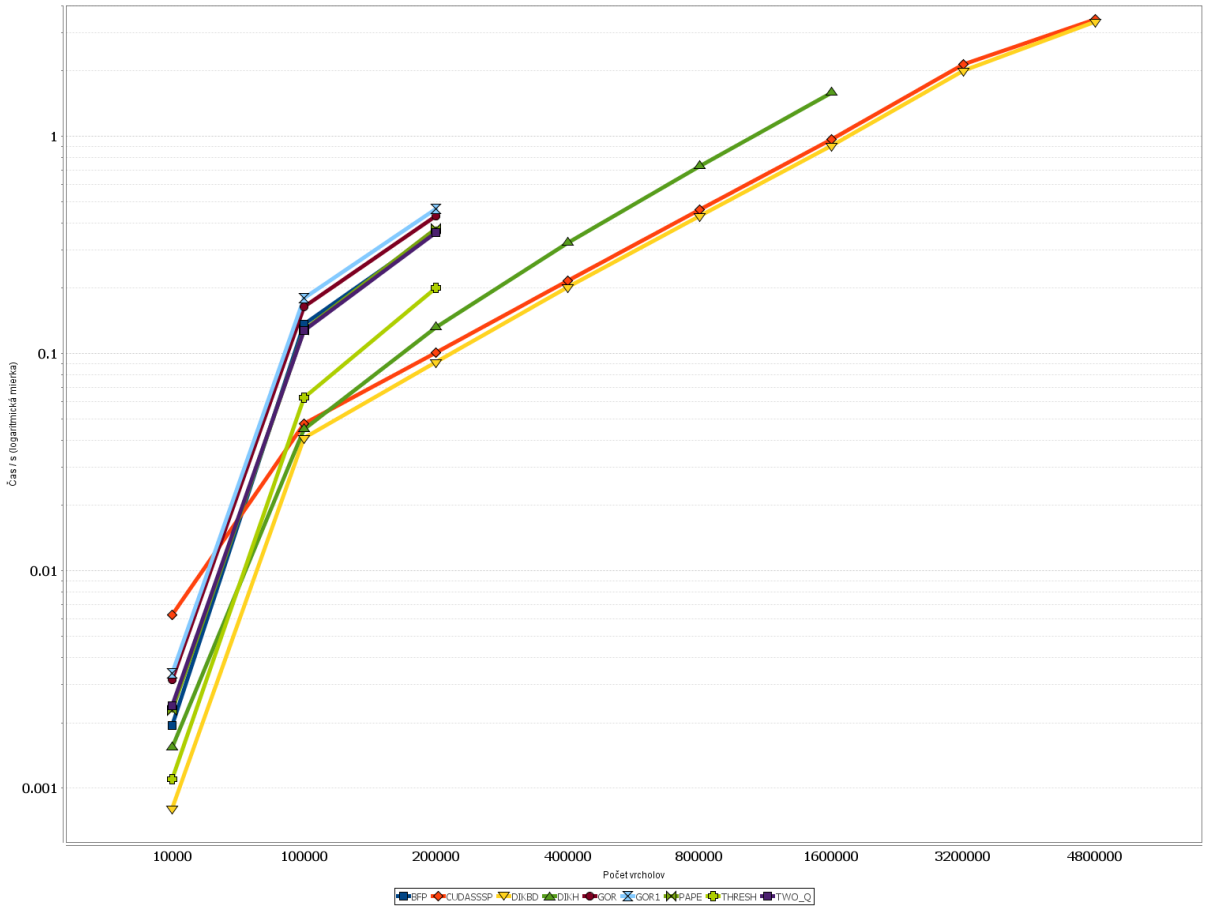


OA.30 Hustý náhodný graf, $\mathbf{m}=\mathbf{n}^2/4$.

vrcholy / hrany	BFP	CUDASSSP	DIKBD	DIKH
512 / 65536	0.0002	0.0035	0.0002	0.0002
1024 / 262144	0.0010	0.0088	0.0008	0.0008
2048 / 1048576	0.0038	0.0234	0.0028	0.0025
4096 / 4194304	0.0171	0.1789	0.0137	0.0117
8192 / 16777216	0.0738	0.7675	0.0599	0.0565

TA.30 Hustý náhodný graf, $\mathbf{m}=\mathbf{n}^2/4$.

A.31 CUDA Rand 6

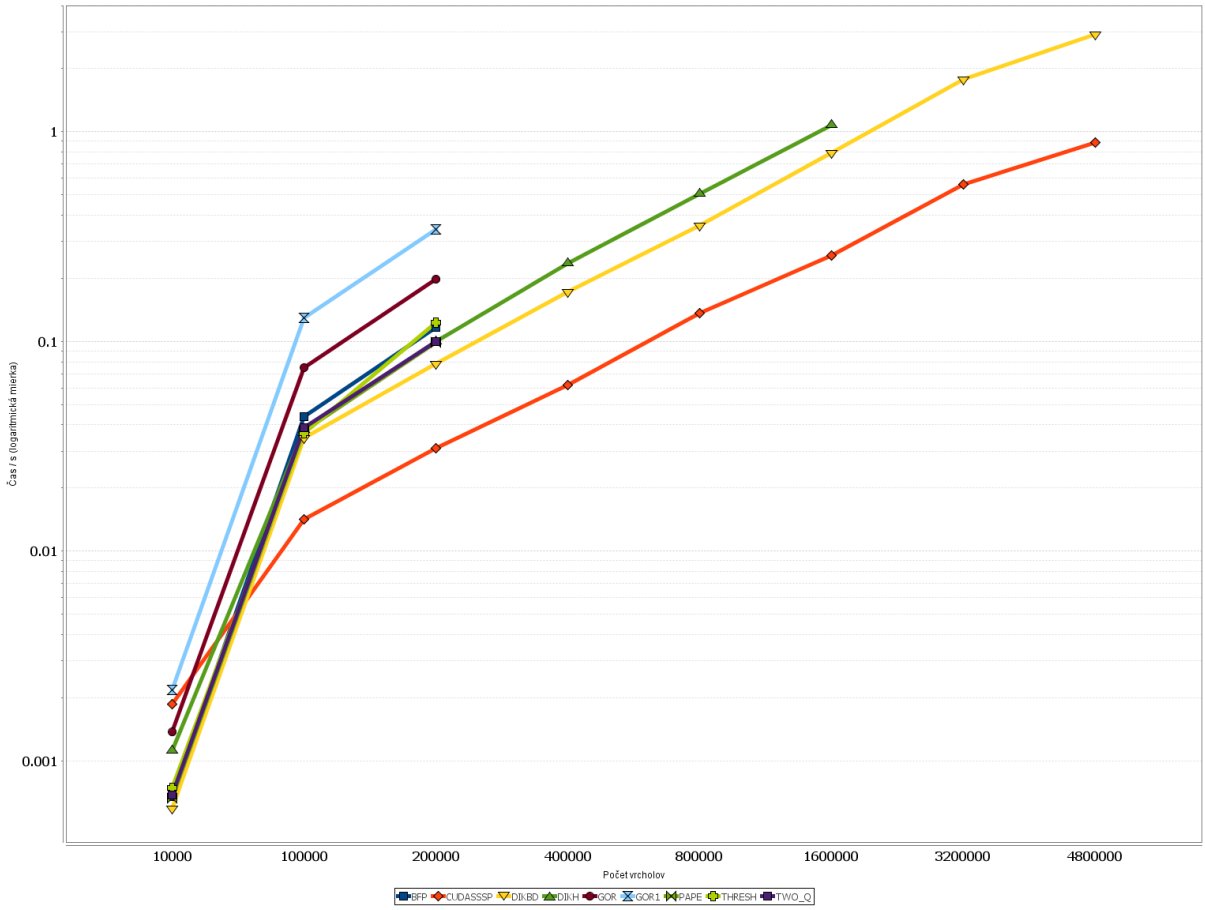


OA.31 Riedky náhodný graf, **m**=*6***n**, **ll** = *100.*

vrcholy / hrany	BFP	CUDASSSP	DIKBD	DIKH	GOR	GOR1	PAPE	THRESH	TWO_Q
10000 / 60000	0.0019	0.0063	0.0008	0.0015	0.0031	0.0034	0.0023	0.0011	0.0024
100000 / 600000	0.1367	0.0477	0.0408	0.0448	0.1646	0.1795	0.1285	0.0626	0.1282
200000 / 1200000	0.3629	0.1009	0.0912	0.1315	0.4283	0.4642	0.3767	0.1999	0.3594
400000 / 2400000		0.2159	0.2031	0.3231					
800000 / 4800000		0.4581	0.4305	0.7298					
1600000 / 9600000		0.9666	0.9065	1.5896					
3200000 / 19200000		2.1456	2.0004						
4800000 / 28800000		3.4455	3.3716						

TA.31 Riedky náhodný graf, **m**=*6***n**, **ll** = *100.*

A.32 CUDA Rand 6 S

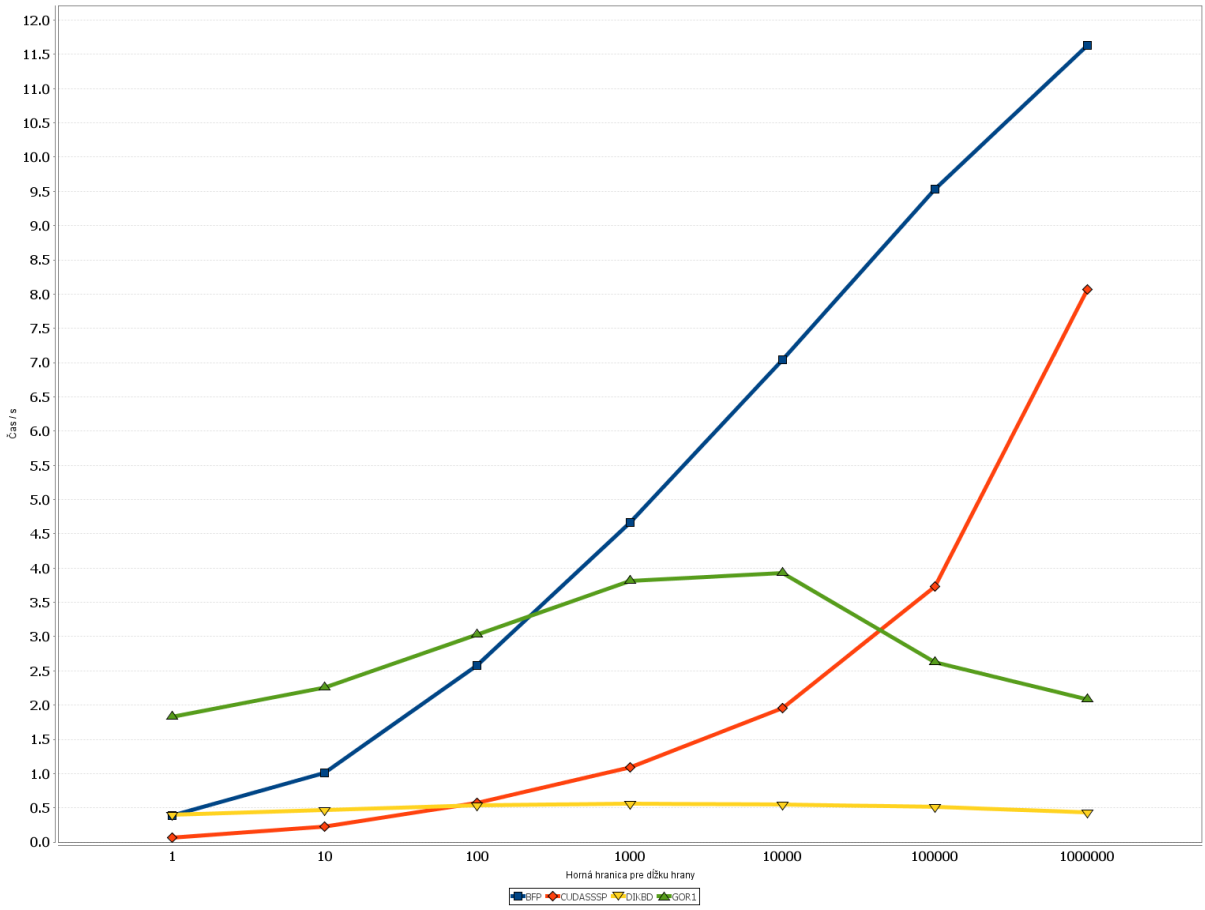


OA.32 Riedky náhodný graf, **m**=*6***n**, **ll** = *5.*

vrcholy / hrany	BFP	CUDASSSP	DIKBD	DIKH	GOR	GOR1	PAPE	THRESH	TWO_Q
10000 / 60000	0.0007	0.0019	0.0006	0.0011	0.0014	0.0022	0.0007	0.0007	0.0007
100000 / 600000	0.0436	0.0142	0.0346	0.0379	0.0751	0.1292	0.0374	0.0366	0.0388
200000 / 1200000	0.1172	0.0309	0.0782	0.0995	0.1980	0.3426	0.0992	0.1230	0.0998
400000 / 2400000		0.0619	0.1720	0.2347					
800000 / 4800000		0.1366	0.3586	0.5032					
1600000 / 9600000		0.2558	0.7907	1.0751					
3200000 / 19200000		0.5600	1.7752						
4800000 / 28800000		0.8889	2.9260						

TA.32 Riedky náhodný graf, **m**=*6***n**, **ll** = *5.*

A.33 CUDA Rand Len

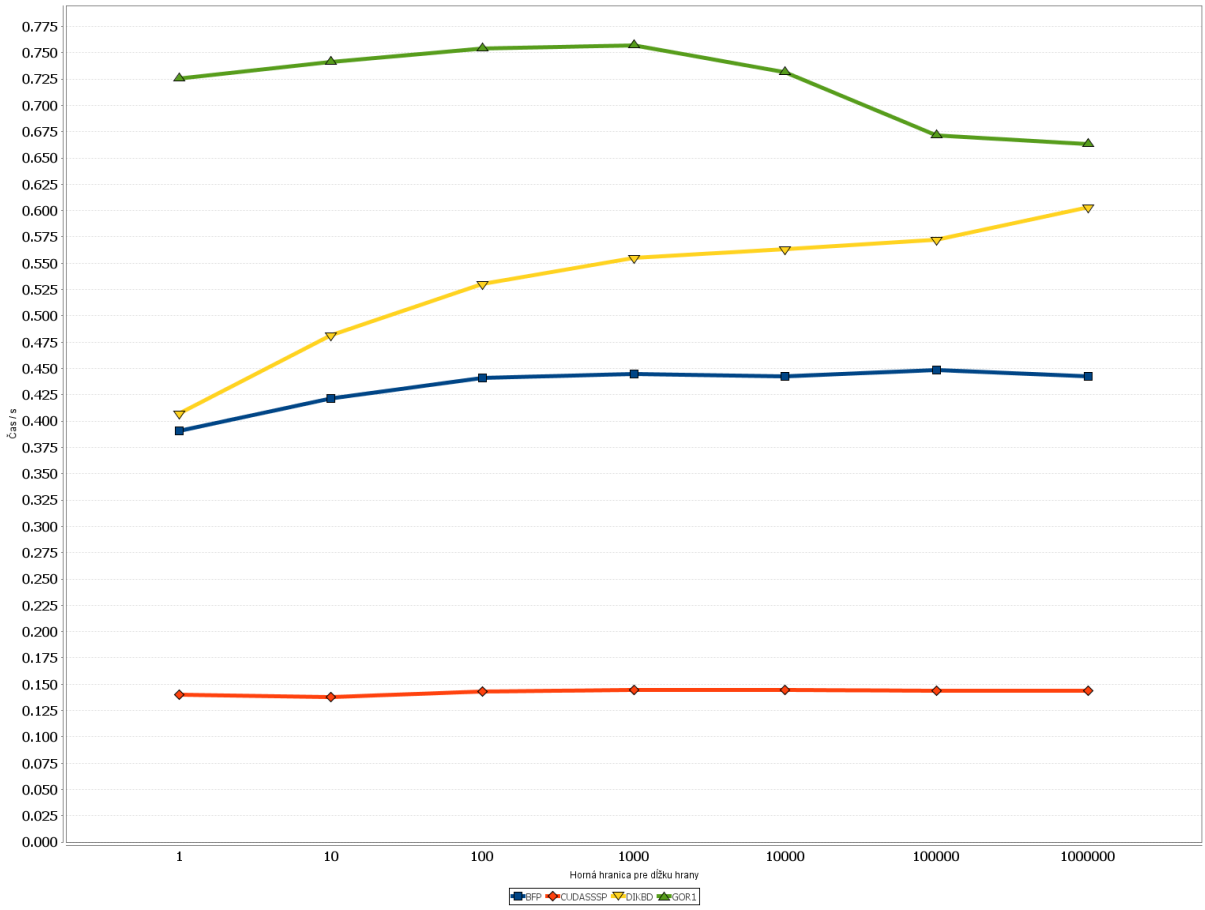


OA.33 Riedky náhodný (Rand) graf, **m**=5**n**, **n**=2^{20}.

Maximum dĺžky hrany	BFP	CUDASSSP	DIKBD	GOR1
1	0.3839	0.0686	0.4028	1.8309
10	1.0091	0.2242	0.4699	2.2528
100	2.5744	0.5770	0.5363	3.0233
1000	4.6666	1.0896	0.5583	3.8143
10000	7.0466	1.9583	0.5479	3.9284
100000	9.5321	3.7266	0.5184	2.6296
1000000	11.6321	8.0639	0.4363	2.0858

TA.33 Riedky náhodný (Rand) graf, **m**=5**n**, **n**=2^{20}.

A.34 CUDA ER Len

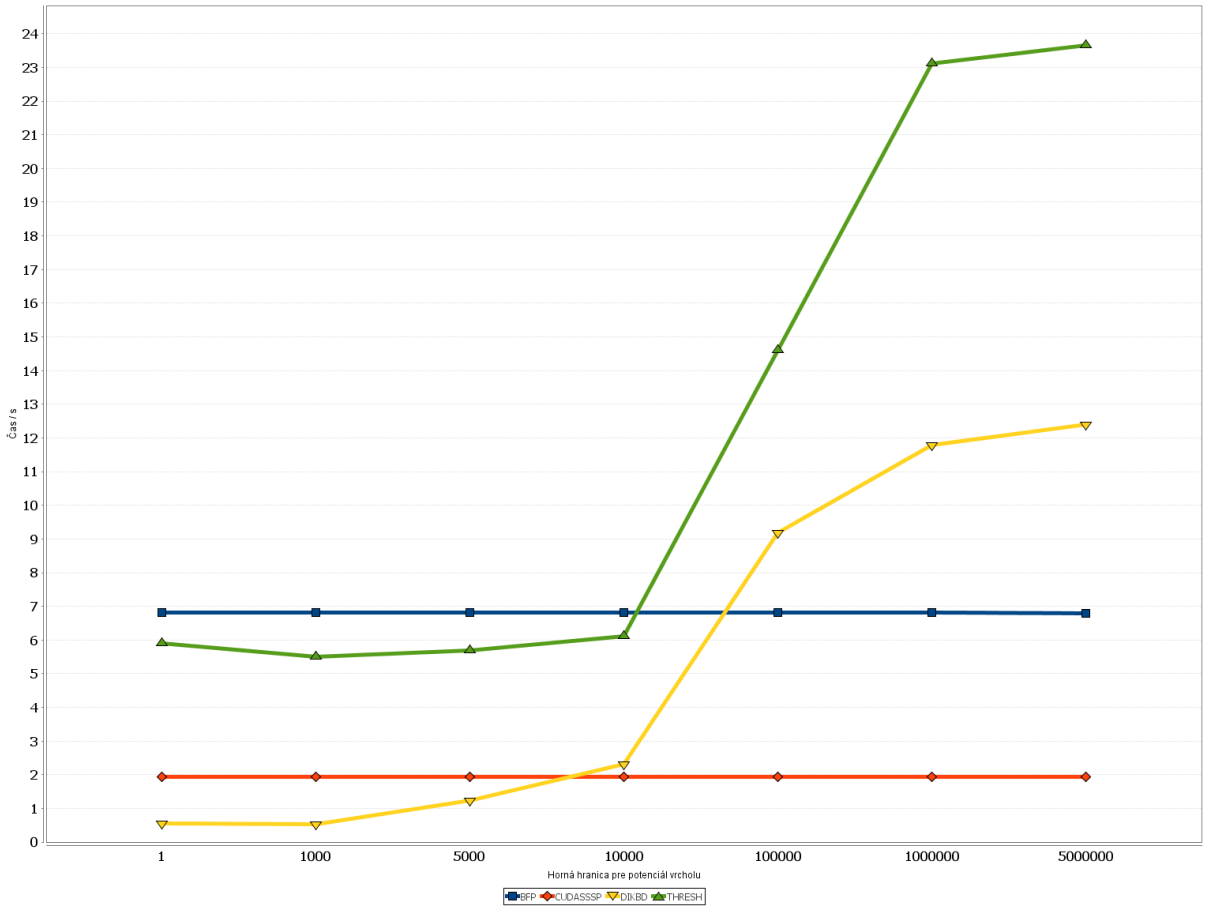


OA.34 Riedky náhodný (ER) graf, **m**=5**n**, **n**=2^{20}.

Maximum dĺžky hrany	BFP	CUDASSSP	DIKBD	GOR1
1	0.3910	0.1400	0.4070	0.7252
10	0.4213	0.1379	0.4817	0.7414
100	0.4408	0.1433	0.5303	0.7538
1000	0.4446	0.1444	0.5554	0.7570
10000	0.4426	0.1444	0.5630	0.7311
100000	0.4486	0.1439	0.5724	0.6712
1000000	0.4422	0.1439	0.6034	0.6633

TA.34 Riedky náhodný (ER) graf, **m**=5**n**, **n**=2^{20}.

A.35 CUDA Rand P

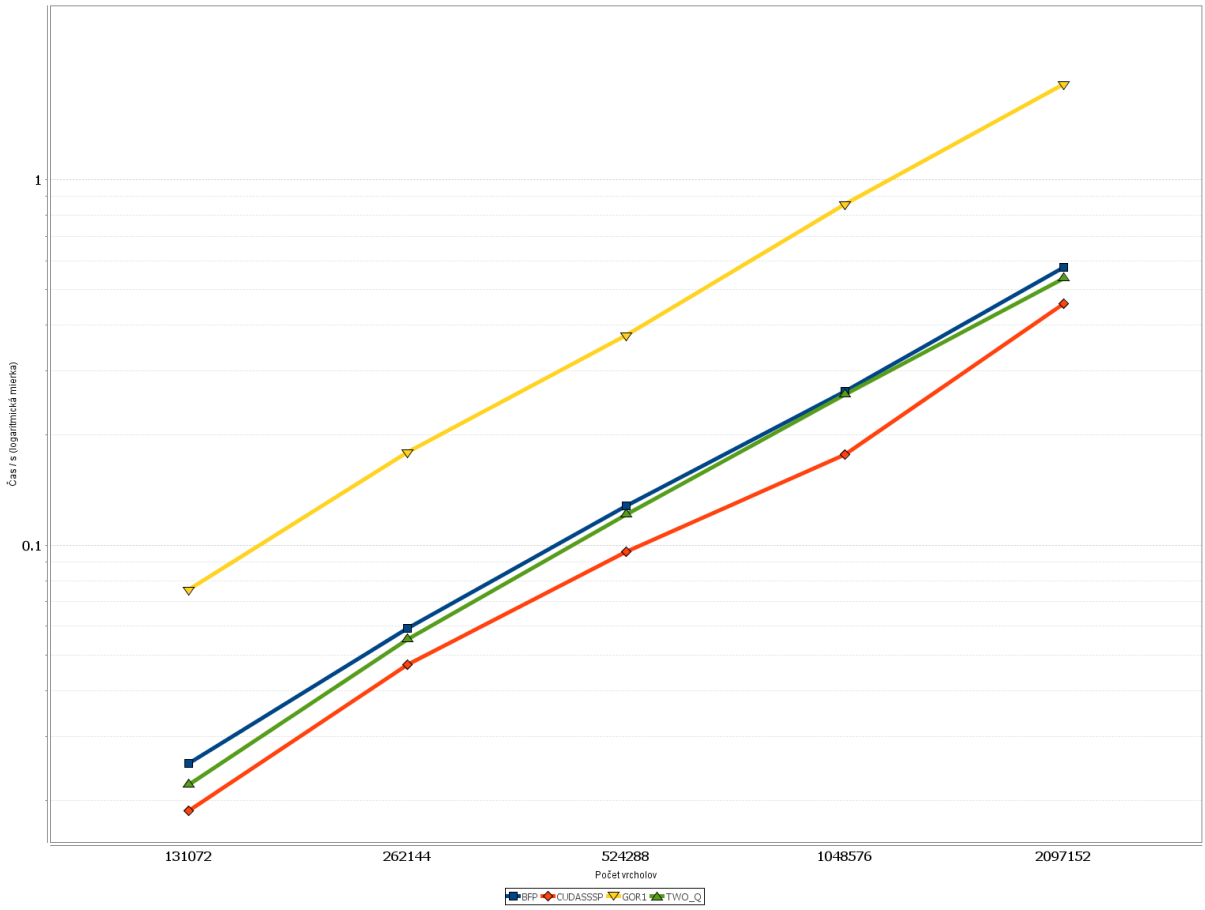


OA.35 Riedky náhodný (Rand) graf, **m**=5**n**, **n**=2^{20}.

Horná hranica P	BFP	CUDASSSP	DIKBD	THRESH
1	6.8158	1.9314	0.5476	5.8998
1000	6.8121	1.9351	0.5274	5.5106
5000	6.8186	1.9366	1.2314	5.6965
10000	6.8119	1.9462	2.3157	6.1124
100000	6.8142	1.9377	9.1837	14.6011
1000000	6.8234	1.9302	11.7764	23.1181
5000000	6.8036	1.9320	12.4030	23.6568

TA.35 Riedky náhodný (Rand) graf, **m**=5**n**, **n**=2^{20}.

A.36 CUDA R-MAT 1:1

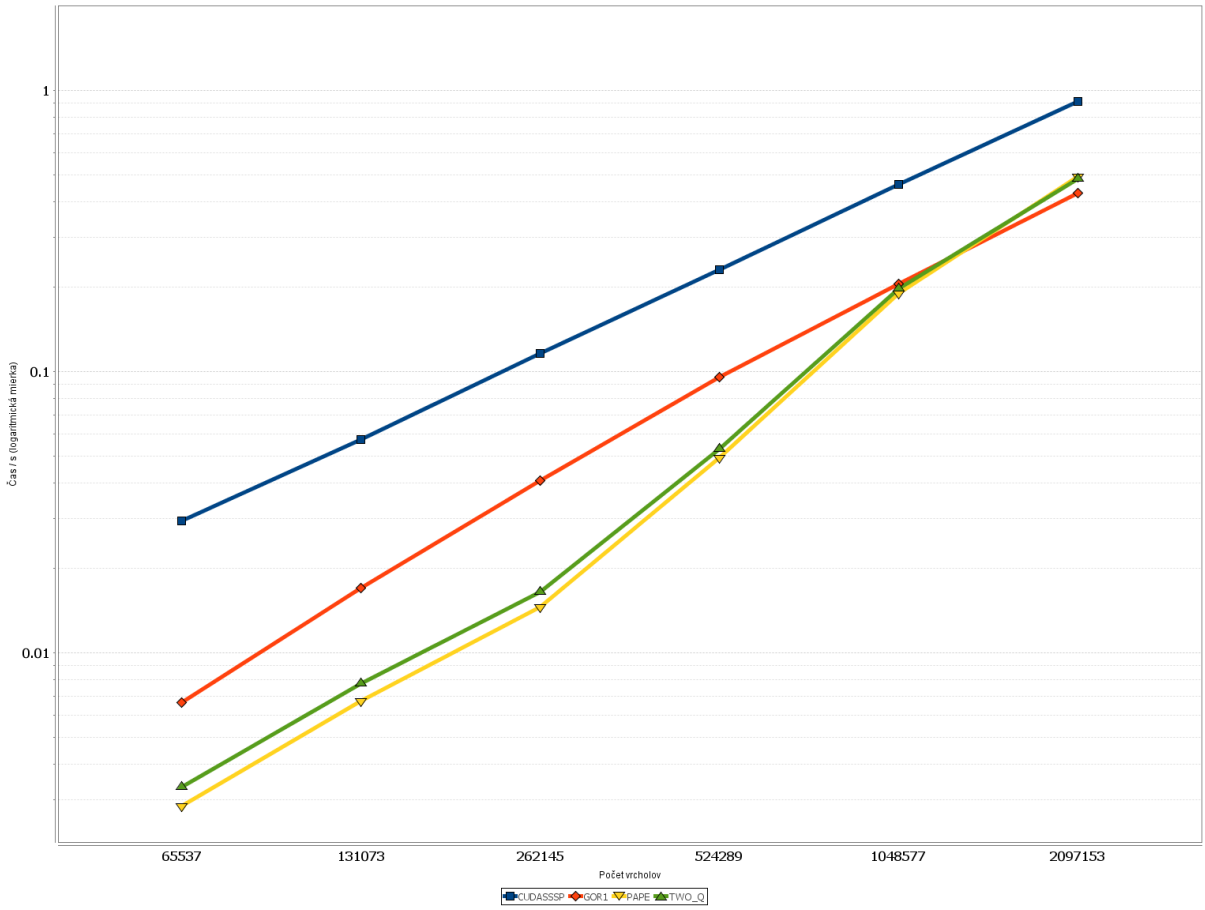


OA.36 Test na bezškálovej sieti, **a**=**c**=*0.05,* **b**=**d**=*0.45.*

vrcholy / hrany	BFP	CUDASSSP	GOR1	TWO_Q
131072 / 351390	0.0252	0.0188	0.0757	0.0221
262144 / 704558	0.0591	0.0470	0.1792	0.0553
524288 / 1414374	0.1279	0.0957	0.3746	0.1212
1048576 / 2836636	0.2639	0.1774	0.8566	0.2586
2097152 / 5696084	0.5754	0.4573	1.8321	0.5367

TA.36 Test na bezškálovej sieti, **a**=**c**=*0.05,* **b**=**d**=*0.45.*

A.37 CUDA Grid S Wide

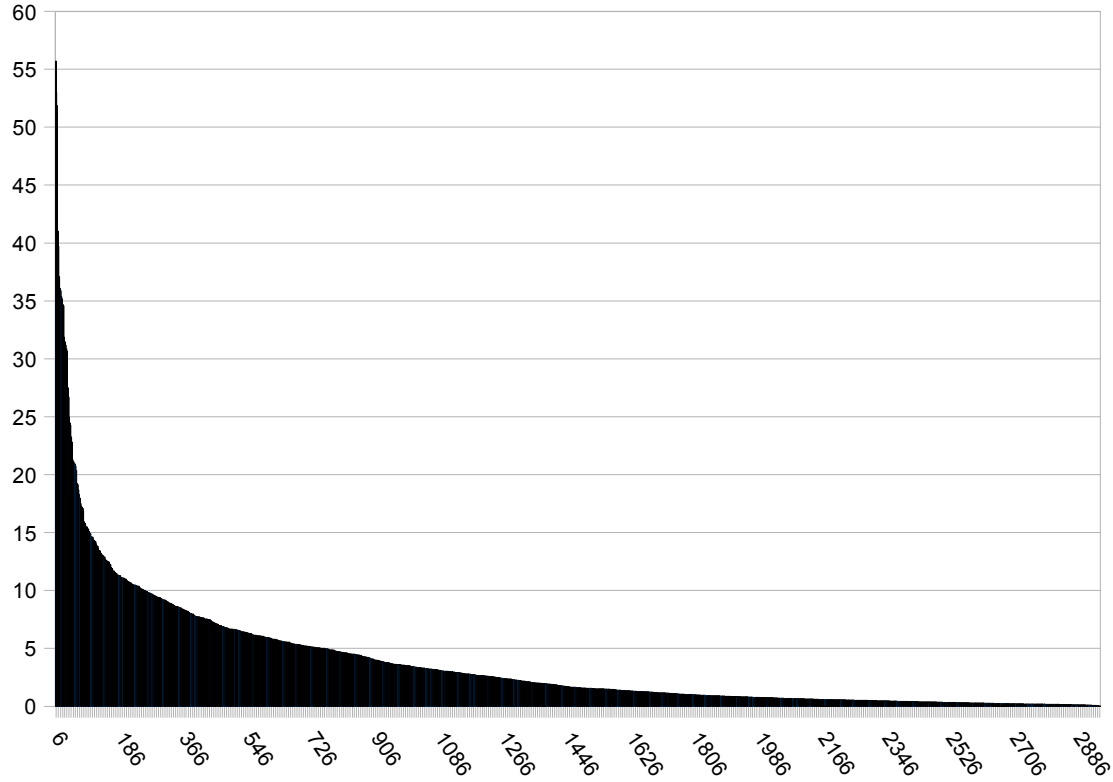


OA.37 Test na širokej štvorcovej mriežke, **X***=16.*

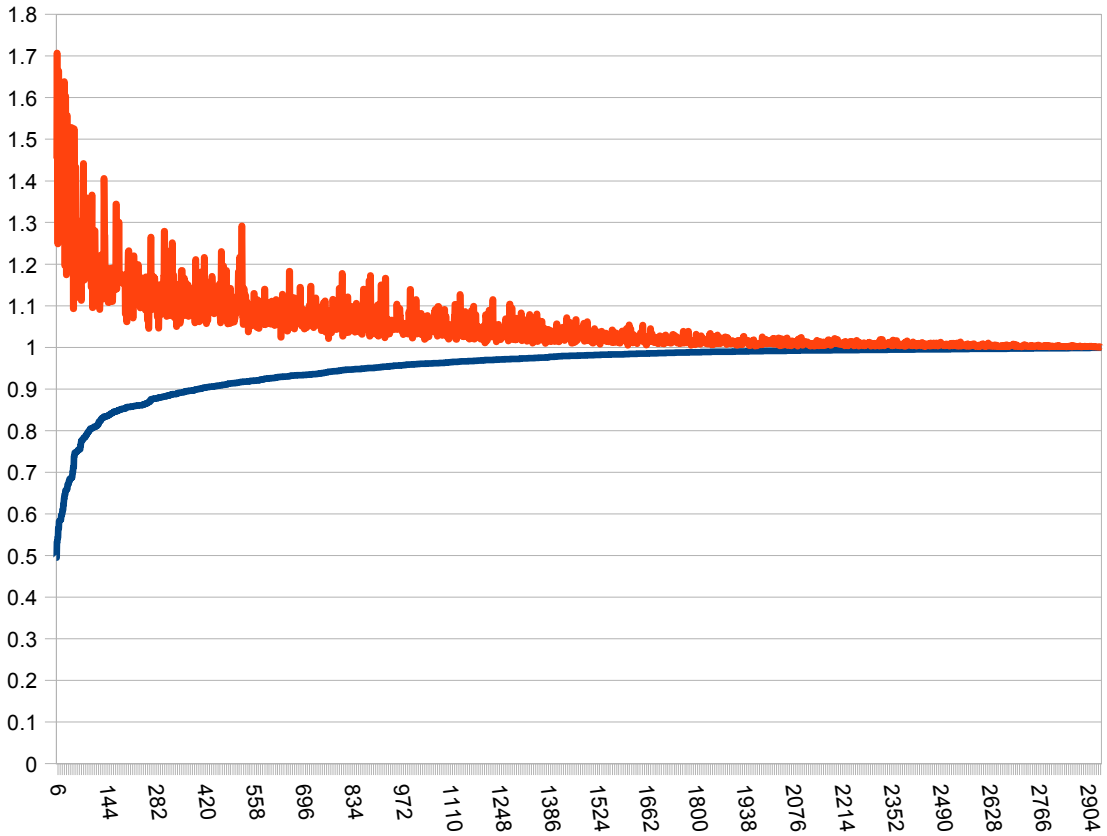
vrcholy / hrany	CUDASSSP	GOR1	PAPE	TWO_Q
65537 / 196608	0.0294	0.0067	0.0028	0.0033
131073 / 393216	0.0570	0.0170	0.0067	0.0078
262145 / 786432	0.1161	0.0408	0.0145	0.0165
524289 / 1572864	0.2294	0.0954	0.0494	0.0531
1048577 / 3145728	0.4607	0.2044	0.1899	0.1965
2097153 / 6291456	0.9144	0.4305	0.4942	0.4848

TA.37 Test na širokej štvorcovej mriežke, **X***=16.*

A.38 Vlastnosti výsledkov



OA.38a Pre všetky merania bola vypočítaná výberová smerodajná odchýlka, v grafe sú nanesené zistené hodnoty vyjadrené ako percentuálny podiel vzhľadom ku priemeru príslušného merania. Hodnoty sú zatriedené od najvyššej.



OA.38b Hodnoty minima a maxima pre výsledky meraní. Hodnoty sú vyjadrené ako podiel zo zistenej priemernej hodnoty a zoradené podľa minima. U veľkej časti výsledkov sú hodnoty minima a maxima blízke priemeru.

B Príloha – Obsah DVD

Priložené DVD okrem textu práce, vo formáte PDF, obsahuje:

- **src** – zdrojové súbory programov použitých v práci, originálne archívy prevzatého software DIMACS-ch9.tar.gz, GTgraph.tar.gz, splib.tar
  - **src/java** – vlastné testovacie prostredie implementované v jazyku Java
  - **src/windows** – upravené projekty testovaných algoritmov a niektorých generátorov pre Visual Studio C++ 2008 Express Edition.
  - **src/linux** – upravené implementácie generátorov kompilovateľné pod linuxom (pre cygwin)
- **merania** – experimentálne získané hodnoty
  - **merania/platforma** – merania súvisiace s meraním času na testovacích systémoch
  - **merania/<*ostatné adresáre*>** - merania príslušiace jednotlivým testom z kapitoly 5
- **výsledky** – adresár s výsledkami v tabuľkách a grafoch
  - **výsledky/css** – štýl pre html tabuľky výsledkov

C Príloha - Vývojárske zdroje

Pri vypracovaní štúdie bol použitý nasledujúci software.

- Eclipse IDE - pre vytvorenie testovacieho prostredia a spúšťanie testov. [37]
- Java SE – pre beh testovacieho prostredia [34]
- JFreeChart – pre tvorbu grafov v prílohe A (www.jfree.org/jfreechart)
- Visual Studio C++ 2008 Express Edition [38] – kompilácia implementácií
- SPLIB – implementácie pre CPU a generátory Rand, Grid a Acyc [1], splib.tar (B)
- GTGraph – implementácie R-MAT, SSCA#2 a ER [40], GTgraph.tar.gz (B)
- DIMACS 9th challenge – mapy, implementácie [43], DIMACS-ch9.tar.gz (B)
- CygWin – pre kompiláciu a beh generátorov GTGraph [35]
- Open Office – spracovanie textu a dát, grafy v texte (www.openoffice.org)
- Implementácia poskytnutá autormi [15] – implementácia SSSP na GPU
- CUDA SDK – kompilácia CUDASSSP [36, 58]

Zoznam citovanej literatúry

[1] B. V. Cherkassky, A. V. Goldberg, T. Radzik (1996): Shortest paths algorithms: theory and experimental evaluation, Math. Programming 73, 129 – 174.

[2] U. Meyer (2003): Average-case complexity of single-source shortest-paths algorithms: lower and upper bounds. Journal of Algorithms 48 , 91 – 134.

[3] B. M. E. Moret (2002): Towards a discipline of experimental algorithmics. DIMACS series in Discrete Math. and Theoret. Computer Science 59, 197 – 213.

[4] Microsoft Corporation (2005): Game Timing and Multicore Processors, MSDN [online] http://msdn.microsoft.com/en-us/library/bb173458.aspx

[5] Intel Corporation (2009): RDTSC - Read Time-Stamp Counter. [online] http://www.intel.com/software/products/documentation/vlin/mergedprojects/analyzer_ec/mergedprojects/reference_olh/mergedProjects/instructions/instruct32_hh/vc275.htm

[6] NTSC color television transmission, US patent 4,661,840 (2009) [online] http://xrint.com/patents/us/4661840

[7] Intel Corporation (2004): IA-PC HPET (High Precision Event Timers) Specification, p. 9 [online] http://www.intel.com/hardwaredesign/hpetspec_1.pdf

[8] Anandtech.com (2009): Know your Processor Cores: Codename Cheatsheet. [online] http://www.anandtech.com/cpuchipsets/showdoc.aspx?i=2118

[9] M. Abrash (1997), Michael Abrash's Graphics Programming Black Book, special ed., The Coriolis Group Inc., p. 6

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest (2001): Introduction to Algorithms, Second Edition, MIT Press, ISBN: 0-262-03293-7

[11] Ravindra K. Ahuja, Kurt Mehlhorn, James Orlin, Robert E. Tarjan (1990): Faster algorithms for the shortest path problem, Journal of the ACM (JACM) Volume 37 , Issue 2 (April 1990) p. 213 – 223 ISSN:0004-5411

[12] U. Pape (1974): Implementation and efficiency of Moore-algorithms for the shortest route problem, Math. Programming, Volume 7, Number 1 / December

[13] Stefano Pallottino: Shortest-path methods (1983): Complexity, interrelations and new propositions, Networks Volume 14 Issue 2, p. 257 – 267

[14] F. Glover, R. Glover, D. Klingman: Computational study of an improved shortest path algorithm, Networks Volume 14 Issue 1, p. 25 – 36

[15] Pawan Harish, P. Narayanan (2007): Accelerating Large Graph Algorithms on the GPU Using CUDA, High Performance Computing – HiPC 2007, p. 197-208

[16] U. Meyer, P. Sanders (1998): Delta-Stepping: A Parallel Single Source Shortest Path Algorithm, Proceedings of the 6th Annual European Symposium on Algorithms, p. 393 – 404

[17] Ulrich Meyer (2001): Heaps Are Better than Buckets: Parallel Shortest Paths on Unbalanced Graphs, Lecture Notes in Computer Science, Springer Berlin / Heidelberg, ISSN 0302-9743

[18] Xingzhi Wen, Uzi Vishkin (2007): PRAM-on-chip: first commitment to silicon, ACM Symposium on Parallel Algorithms and Architectures – Proc. of the 19th annual ACM symposium on Parallel algorithms and architectures, p. 301 – 302

[19] Rolf H. Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, Thomas Willhalm (2006): Partitioning graphs to speedup Dijkstra's algorithm, Journal of Experimental Algorithmics (JEA) Volume 11, ISSN:1084-6654

[20] Dorothea Wagner, Thomas Willhalm (2003): Geometric speed-up techniques for finding shortest paths in large sparse graphs, 11th European Symposium on Algorithms, volume 2832 of LNCS

[21] A. Crauser, K. Mehlhorn, U. Meyer (1998): A parallelization of Dijkstra's shortest path algorithm, Proc. 23rd MFCS'98, Lecture Notes in Computer Science

[22] PE Hart, NJ Nilsson, B Raphael (1968): A Formal Basis for the Heuristic Determination of Minimum Cost Paths, Systems Science and Cybernetics, IEEE Transactions on, Vol. 4, No. 2. p. 100-107

[23] Hans Berliner (1979): The B* tree search algorithm: A best-first proof procedure, Artificial Intelligence journal 12/1, p. 23–40

[24] Lenie Sint, Dennis de Champeaux (1977): An Improved Bidirectional Heuristic Search Algorithm, Journal of the ACM (JACM), Volume 24, Issue 2, p. 177 – 191

[25] Dennis de Champeaux (1983): Bidirectional Heuristic Search Again, Journal of the ACM (JACM), Volume 30, Issue 1, p. 22 – 32

[26] A. Stentz (1994): Optimal and efficient path planning for partially-knownenvironments, 1994 IEEE International Conference on Robotics and Automation, p. 3310-3317

[27] D. Joyal, P. Galecki, S. Giacalone (2006): RFC4750 OSPF Version 2 Management Information Base, [online] http://tools.ietf.org/html/rfc4750

[28] Floyd, W. Robert (1962): Algorithm 97: Shortest Path, Communications of the ACM

[29] Donald B. Johnson (1977): Efficient Algorithms for Shortest Paths in Sparse Networks, Journal of the ACM (JACM), Volume 24, Issue 1, p. 1 – 13

[30] Anthony Stentz (1995): The Focussed D* Algorithm for Real-Time Replanning, Proceedings of the International Joint Conference on Artificial Intelligence [online] http://www.frc.ri.cmu.edu/projects/mars/publications/focussed_dstar_ijcai95.ps.gz

[31] A. Yahja, S. Singh, A. Stentz (1998): Recent Results in Path Planning for Mobile

Robots Operating in Vast Outdoor Environments, Symposium on Image, Speech, Signal Processing and Robotics

[32] Dorothea Wagner, Thomas Willhalm, Christos Zaroliagis (2005): Geometric containers for efficient shortest-path computation, Journal of Experimental Algorithmics (JEA) Volume 10, Article 1.3

[33] A. Buluc, J. R. Gilberta, C. Budaka (2008): Gaussian Elimination Based Algorithms on the (GPU), The 5th International Workshop on Parallel Matrix Algorithms and Applications(PMAA'08), Neuchâtel, Switzerland [online] http://gauss.cs.ucsb.edu/publication/parco2008.pdf

[34] Sun microsystems: Java SE Documentation at a Glance [online] http://java.sun.com/javase/reference/index.jsp

[35] Cygwin Documentation [online] http://cygwin.com

[36] NVIDIA Corporation: CUDA Documentation [online] http://www.nvidia.com/object/cuda_develop.html

[37] Eclipse.org: Eclipse Downloads [online] http://www.eclipse.org/downloads/

[38] Microsoft Corporation: Visual Studio C++ 2008 Epress Edition [online] http://www.microsoft.com/express/vc/

[39] Intel Corporation (2009): Intel Core 2 Duo Processor E8000 and E7000 Series [online] http://download.intel.com/design/processor/datashts/318732.pdf

[40] David A. Bader, Kamesh Madduri (2006): GTgraph: A Synthetic Graph Generator Suite [online] http://www.cc.gatech.edu/~kamesh/GTgraph/gen.pdf

[41] Andrew Goldberg (2002): Andrew Goldberg's Network Optimization Library [online] http://avglab.com/andrew/soft.html

[42] F. Benjamin Zhan (2001): Three Fastest Shortest Path Algorithms on Real Road Networks:Data Structures and Procedures, Journal of Geographic Information and Decision Analysis, vol.1, no.1, p. 69-82

[43] Center for Discrete Mathematics & Theoretical Computer Science (2005): 9th DIMACS Implementation Challenge - Shortest Paths [online] http://www.dis.uniroma1.it/~challenge9/download.shtml

[44] Beman Dawes, David Abrahams (1998-2005), Rene Rivera (2004-2007): Boost 1.38.0 Library Documentation [online] http://www.boost.org/doc/libs/1_38_0/libs/graph/doc/table_of_contents.html

[45] Brian Cirulnick (2007): Sun SparcStation 10 / SparcServer 10 / Axil 311, http://obsolyte.com [online] http://obsolyte.com/sun_ss10/

[46] Center for Discrete Mathematics & Theoretical Computer Science (2005): File formats [online] http://www.dis.uniroma1.it/~challenge9/format.shtml

[47] F. Benjamin Zhan, Charles E. Noon (1998): Shortest path algorithms: An evaluation using real road networks, Transportation Science [online] http://www.txstate.edu/~fz01/reprints/zhan_1998_ts.pdf

[48] A. V. Goldberg (2001): A Simple Shortest Path Algorithm with Linear Average Time, Springer Lecture Notes in Computer Science LNCS 2161, p. 230-241

[49] A. V. Goldberg (2001): Shortest Path Algorithms: Engineering Aspects, Lecture

Notes in Computer Science [online] www.avglab.com/andrew/pub/isaac01.ps

[50] Goldberg, A. V., & Silverstein, C. (1995): Implementations of Dijkstra's algorithm based on multi-level buckets. [online] http://www.avglab.com/andrew/pub/neci-tr-95-187.ps

[51] Boris V. Cherkassky, Andrew V. Goldberg, Craig Silverstein (1997): Buckets, heaps, lists, and monotone priority queues, Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms, p. 83 – 92, ISBN:0-89871-390-0

[52] Andrew V. Goldberg (2001): A Practical Shortest Path Algorithm with Linear Expected Time, [online] http://www.avglab.com/andrew/pub/sp-lin.ps

[53] Andrew V. Goldberg (2001): Heuristic Improvement Of The Bellman-Ford Algorithm [online] http://ftp.cs.stanford.edu/cs/theory/goldberg/stan-cs-93-1464.ps.Z

[54] D.A. Bader, K. Madduri (2005): Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors, Lecture Notes in Computer Science, 3769:465-476 [online] www.cc.gatech.edu/~bader/papers/SSCA2.html

[55] John R. Gilbert, S.Reinhardt, V. Shah (2006): High-performance graph algorithms from parallel sparse matrices [online] http://www.interactivesupercomputing.com/downloads/pgraph.pdf

[56] D. Chakrabarti, Y. Zhan, C. Faloutsos (2004): R-MAT: A recursive model for graph mining [online] http://www.cs.cmu.edu/~christos/PUBLICATIONS/siam04.pdf

[57] Dennis Schmitz, (2004): Ilustrácia Dual Core CPU [online] http://en.wikipedia.org/wiki/File:Dual_Core_Generic.svg

[58] NVIDIA Corporation (2008): CUDA Programming guide verzia 2.1

[59] Tom R. Halfhill (2008): Parallel Processing With CUDA, MICROPROCESSOR REPORT, www.MPRonline.com [online] http://www.nvidia.com/docs/IO/55972/220401_Reprint.pdf

[60] K. Madduri, D.A. Bader, J.W. Berry, and J.R. Crobak (2006): Parallel Shortest Path Algorithms for Solving Large-Scale Instances, 9th DIMACS Implementation Challenge -- The Shortest Path Problem [online] http://www.cc.gatech.edu/~bader/papers/ShortestPaths-ALENEX2007.pdf

[61] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, Katherine Yelick (2006): The potential of the cell processor for scientific computing, Proceedings of the 3rd conference on Computing frontiers, p 9-20.

[62] NVIDIA Corporation (2009): Tesla many core parralel supercomputing [online] http://www.nvidia.com/object/tesla_computing_solutions.html

[63] IBM (2009): The Cell architecture, [online] http://domino.research.ibm.com/comm/research.nsf/pages/r.arch.innovation.html

[64] Larry Seiler a kolektív autorov (2008): Larrabee: a many-core x86 architecture for visual computing, ACM Trans. Graph., Vol. 27, No. 3. (August 2008), p. 1-15.

[65] Khronos OpenCL Working Group (2009): The OpenCL specification [online] http://www.khronos.org/registry/cl/specs/opencl-1.0.33.pdf