Efektívne hľadanie viacerých najkratších ciest v orientovanom ohodnotenom grafe pomocou Dijkstrovho algoritmu v C# .NET aplikácii

Igor Košťál[1]

Abstrakt
Dijkstrov algoritmus je často používaný v rôznych smerovacích softvéroch, ako sú napr. aplikácie hľadajúce najkratšie cesty pri smerovaní paketov jednotlivými smerovačmi v počítačových sieťach reprezentovaných orientovanými ohodnotenými grafmi. Tiež je ho možné použiť pre hľadanie najkratších ciest v elektronických automapách. Zaujímalo nás, ako sa dajú efektívne vyhľadávať viaceré najkratšie cesty v orientovanom ohodnotenom grafe pomocou Dijkstrov algoritmu. Vytvorili sme konzolovú C# .NET aplikáciu, ktorá dokáže pomocou svojej inštančnej metódy s implementovaným Dijkstrovým algoritmom vyhľadávať viaceré najkratšie cesty v orientovanom ohodnotenom grafe sériovo, na viacerých vláknach a paralelne. V článku sa zaoberáme exekučnou efektívnosťou všetkých troch spôsobov vyhľadávania a hľadáme najefektívnejší z nich. Predpokladáme, že najefektívnejším spôsobom vyhľadávania viacerých takýchto najkratších ciest by malo byť paralelné vyhľadávanie. Experiment, ktorý sme vykonali pomocou našej konzolovej C# .NET aplikácie, túto hypotézu potvrdí alebo vyvráti.

Kľúčové slová
orientovaný ohodnotený graf, najkratšie cesty v grafe, prioritná fronta, Dijkstrov algoritmus, viacvláknové vyhľadávanie, paralelné vyhľadávanie

Abstract
Dijkstra's algorithm is often used in various routing software, such as e.g. applications finding the shortest paths in the routing of packets by particular routers in computer networks represented by edge-weighted directed graphs. It can also be used to find the shortest paths in electronic road maps. We were interested in how to find efficiently several shortest paths in an edge-weighted directed graph using Dijkstra's algorithm. We have created a console C# .NET application that can use its instance method with implemented Dijkstra's algorithm to find several shortest paths in an edge-weighted directed graph in serial, on multiple threads and in parallel. In this paper, we deal with the execution efficiency of all three ways of finding and search for the most effective of them. We assume that the most efficient way to find several such shortest paths should be a parallel finding. The experiment we performed using our console C# .NET application will confirm or refute this hypothesis.

Key words
directed weighted graph, shortest paths in a graph, priority queue, Dijkstra's algorithm, multithreaded finding, parallel finding

JEL classification
C88

[1] Ekonomická univerzita v Bratislave, Fakulta hospodárskej informatiky, Katedra aplikovanej informatiky, Dolnozemská cesta 1, 852 35 Bratislava, igor.kostal@euba.sk.

1 Úvod

Ako sme uviedli vyššie, Dijkstrov algoritmus je často používaný na hľadanie najkratších ciest v rôznych smerovacích softvéroch, preto má zmysel zaoberať sa jeho efektívnou implementáciou v metóde nejakej aplikácie, ktorá používa na ukladanie dát grafu efektívnu a flexibilnú dátovú štruktúru. My sme implementovali Dijkstrov algoritmus v inštančnej metóde našej konzolovej C# .NET aplikácie (Košťál, 2020). Inštančná metóda používa na ukladanie dát grafu prioritnú frontu, ktorá je efektívnou a flexibilnou dátovou štruktúrou vhodnou na ukladanie dát grafu. Zdrojový kód našej C# .NET aplikácie sme v rozsiahlej miere prepracovali, v súčasnosti aplikácia pracuje s inak koncipovanými dátovými vstupmi ako pôvodná aplikácia, a rozšírili pridaním pomerne rozsiahlych častí so sériovým, viacvláknovým a paralelizovaným zdrojovým kódom, ktorý umožňuje tejto C# .NET aplikácii hľadať viaceré najkratšie cesty pomocou jej inštančnej metódy sériovo, na viacerých vláknach a paralelne. Pomocou takejto upravenej a značne rozšírenej C# .NET aplikácie sme hľadali efektívny spôsob hľadania viacerých najkratších ciest v orientovanom ohodnotenom grafe. Ako sme uviedli vyššie, C# .NET aplikácia dokáže vyhľadávať viaceré cesty v takomto grafe pomocou svojho sériového, viacvláknového a paralelizovaného kódu. Okrem toho, že výsledky svojich hľadaní viacerých najkratších ciest v orientovanom ohodnotenom grafe zobrazí táto aplikácia vo svojom výstupe a zmeria exekučné časy sériového, viacvláknového a paralelného vyhľadávania, ktoré tiež zobrazí vo svojom výstupe, všetky výsledky hľadaní a ich exekučné časy tiež zapíše do logovacieho diskového súboru. Pomocou týchto zmeraných exekučných časov skúmame v experimente efektívnosť sériového, viacvláknového a paralelného hľadania viacerých najkratších ciest v orientovanom ohodnotenom grafe pomocou Dijkstrovho algoritmu. Predpokladáme, že najefektívnejším spôsobom vyhľadávania by malo byť paralelné vyhľadávanie. Vyhodnotenie experimentu potvrdí alebo vyvráti túto našu hypotézu.

V nasledujúcich kapitolách sa krátko zaoberáme Dijkstrovým algoritmom, prioritnou frontou, sériovou, viacvláknovou a paralelizovanou časťou zdrojového kódu našej C# .NET aplikácie a vyššie spomenutým experimentom.

2 Princíp fungovania Dijkstrovho algoritmu (Košťál, 2020)

Dijkstrov algoritmus (DA) sa dá použiť na hľadanie najkratšej cesty zo štartovacieho vrcholu orientovaného alebo neorientovaného grafu do jeho niektorého cieľového vrcholu, alebo na nájdenie najkratších ciest zo štartovacieho vrcholu do všetkých vrcholov grafu. Pri takomto hľadaní najkratších ciest v grafe DA postupne generuje strom najkratších ciest. Finálna verzia tohto stromu obsahuje vrcholy grafu s priradenými hodnotami najkratších ciest k nim. DA používa dve sady vrcholov, *spt_set* sadu, ktorá obsahuje vrcholy zahrnuté v strome najkratších ciest a *NOspt_set* sadu, ktorá obsahuje vrcholy, ktoré ešte nie sú zahrnuté v *spt_set* sade. DA postupne prehľadáva *NOspt_set* sadu, v každom kroku v nej hľadá vrchol, ktorý má minimálnu vzdialenosť od štartovacieho vrcholu, ak ho nájde, tak ho z *NOspt_set* sady vyberie a s jeho vzdialenosťou od štartovacieho vrcholu ho vloží do *spt_set* sady. Pri vyhľadávaní takýchto vrcholov v *NOspt_set* sade postupuje DA nasledovne:

- vyberie vrchol *u* z *NOspt_set* sady, ktorý má minimálnu vzdialenosť od štartovacieho vrcholu grafu a vloží ho do *spt_set* sady,
- aktualizuje vzdialenosti od štartovacieho vrcholu grafu všetkých priľahlých vrcholov vrchola *u*. Pri aktualizovaní tejto vzdialenosti každého priľahlého vrcholu *v* postupuje podľa nasledujúceho pseudokódu

```
if (dist[u] + graph[u, v] < dist[v])                    (1)
   dist[v] = dist[u] + graph[u, v];
```

tzn., ak suma vzdialenosti vrcholu *u* od štartovacieho vrcholu a hrany *u* - *v* grafu je menšia ako doterajšia vzdialenosť vrcholu *v* od štartovacieho vrcholu, tak vzdialenosť vrcholu *v* je aktualizovaná touto sumou. To je kľúčový krok DA.

Dôležitá je efektívna implementácia DA v metóde aplikácie a tiež voľba dátovej štruktúry pre ukladanie dát spracovávaného grafu, ktorú táto metóda používa. My sme použili pre ukladanie takýchto dát prioritnú frontu, ktorou sa krátko zaoberáme v nasledujúcej kapitole.

3 Prioritná fronta v našej C# .NET aplikácii

Prioritná fronta je dátová štruktúra položiek s kľúčmi, ktorá podporuje dve základné operácie: *vloženie* novej položky a *vymazanie položky s najväčším* alebo *najmenším kľúčom*, podľa toho, či je orientovaná na prácu s maximálnym alebo minimálnym kľúčom (Sedgewick, 1998). Okrem týchto základných operácií podporuje prioritná fronta tiež nasledujúce operácie (Sedgewick, 1998):

- vytvorenie prioritnej fronty z N daných prvkov,
- zmena priority ľubovoľne špecifikovanej položky,
- vymazanie ľubovoľne špecifikovanej položky,
- test prázdnosti fronty atď.

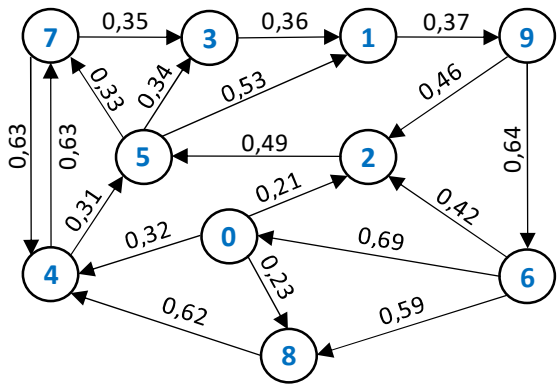
Tieto rôzne operácie, ktoré je možné s prioritnou frontou vykonávať a flexibilita pri vytváraní jej položiek ju robia veľmi populárnou a používanou v profesionálnych aplikáciách.

Metóda ***Dijkstra_PriorityQueue*** našej C# aplikácie používa prioritnú frontu, ktorá je orientovaná na prácu s minimálnym kľúčom, a ktorá je implementovaná pomocou úplného, hromadovo usporiadaného binárneho stromu uloženého v poli (Košťál, 2020). *Strom* je hromadovo usporiadaný, ak kľúč v každom uzle je menší alebo zhodný s kľúčmi všetkých uzlov potomkov (ak existujú) (Sedgewick, 1998). Ekvivalentne, kľúč v každom uzle hromadovo usporiadaného stromu je väčší alebo zhodný s kľúčom v rodičovskom uzle (Sedgewick, 1998).

Takýto postup ukladania položiek zodpovedá dátovej štruktúre *hromada* (angl. heap), ktorá je množinou uzlov s kľúčmi usporiadanými do úplného, hromadovo usporiadaného binárneho stromu reprezentovaného poľom (Sedgewick, 1998).

Uvedené implementačné princípy používa metóda *Dijkstra_PriorityQueue* našej C# .NET aplikácie pri vytváraní objektu *pq* (objekt našej triedy *PriorityQueue*) prioritnej fronty, do ktorej sú uložené dáta spracovávaného grafu, napr. z obrázku 1, touto metódou (Košťál, 2020). C# .NET aplikácia načítava vstupné dáta grafu zo vstupného diskového znakového (ASCII) súboru, napr. zo súboru *tinyEWDm.txt* (obr. 2).

Obr. 1: 10-vrcholový 19-hranový orientovaný ohodnotený graf, v ktorom hľadá metóda 'Dijkstra_PriorityQueue' sériovo, viacvláknovo a paralelne viaceré najkratšie cesty



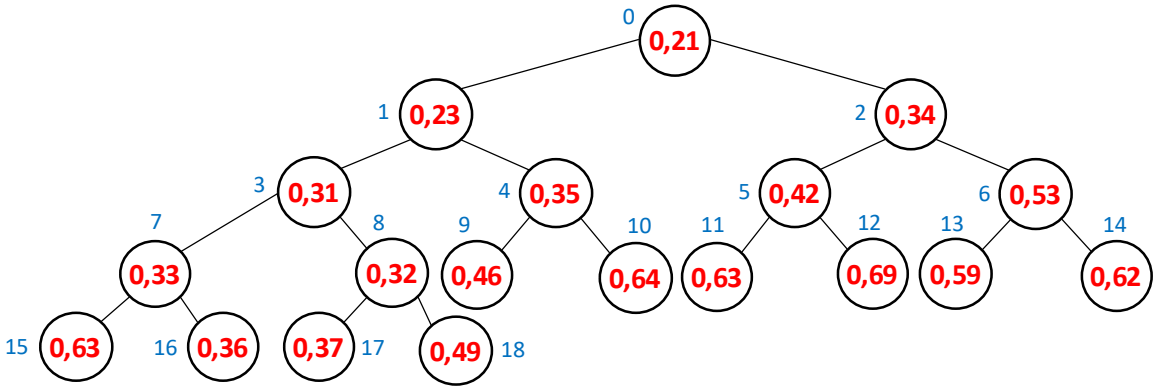
Zdroj: Vlastné spracovanie

Obr. 2: Vstupný diskový súbor 'tinyEWDm.txt', z ktorého načítava C# .NET aplikácia vstupné dáta grafu z obr. 1 (1. riadok: počet vrcholov grafu; 2. riadok: počet hrán grafu; každý ďalší riadok: zdrojový vrchol hrany, cieľový vrchol hrany, jej ohodnotenie)

```
10
19
4 7 0,63
7 4 0,63
7 3 0,35
4 5 0,31
5 7 0,33
5 3 0,34
5 1 0,53
3 1 0,36
1 9 0,37
9 2 0,46
9 6 0,64
6 2 0,42
6 0 0,69
6 8 0,59
8 4 0,62
0 4 0,32
0 8 0,23
0 2 0,21
2 5 0,49
```

Zdroj: Vlastné spracovanie

Obr. 3: Reprezentácia prioritnej fronty pomocou úplného hromadovo usporiadaného binárneho stromu uloženého v poli 'arr' (inštančná premenná objektu prioritnej fronty 'pq'). Ak je rodič uzla na pozícii 'i' v tomto poli, tak ľavý potomok je na pozícii '2i + 1' a pravý na pozícii '2i + 2' tohto poľa



Zdroj: Vlastné spracovanie

Obr. 4: Prioritná fronta obsahujúca dáta spracovávaného grafu (z obr. 1) metódou 'Dijkstra_PriorityQueue', uložená v inštančnej premennej 'arr' objektu 'pq'

pq

	Node.data = 0	Node.priority = 0,21
arr[0]		
arr[1]	0	0.23
arr[2]	5	0,34
arr[3]	4	0,31
arr[4]	7	0,35
arr[5]	6	0,42
arr[6]	5	0,53
arr[7]	5	0,33
arr[8]	0	0,32
arr[9]	9	0,46
arr[10]	9	0,64
arr[11]	4	0,63
arr[12]	6	0,69
arr[13]	6	0,59
arr[14]	8	0,62
arr[15]	7	0,63
arr[16]	3	0,36
arr[17]	1	0,37
arr[18]	2	0,49

count = 19

Zdroj: Vlastné spracovanie

4 C# .NET aplikácia hľadajúca viaceré najkratšie cesty v grafe sériovo, viacvláknovo a paralelne pomocou Dijkstrovho algoritmu

Naša konzolová C# .NET aplikácia bola vytvorená v programovacom jazyku C# vo vývojom prostredí Microsoft Visual Studio Enterprise 2019. Pomocou svojich sériových, viacvláknových a paralelizovaných častí kódu, v ktorých sú implementované volania inštančnej metódy *Dijkstra_PriorityQueue* objektu C# .NET aplikácie, dokáže táto aplikácia vyhľadať viaceré najkratšie cesty vložené používateľom v orientovanom ohodnotenom grafe, ktorého dáta sú uložené vo vstupnom ASCII diskovom súbore, napr. v súbore *tinyEWDm.txt* (obr. 2). Ten tvorí dátový vstup aplikácie. Inštančná metóda *Dijkstra_PriorityQueue* ukladá dáta ňou spracovávaného grafu do objektu prioritnej fronty *pq*, ktorého inštančné premenné *count* a pole *arr* budú obsahovať dáta zobrazené na obr. 4, ak bol vstupným diskovým súborom C# .NET aplikácie súbor *tinyEWDm.txt* (obr. 2). Okrem vyhľadania viacerých najkratších ciest v orientovanom ohodnotenom grafe a zobrazenia výsledkov vyhľadávania v svojom výstupe, aplikácia zmeria exekučné časy všetkých svojich vyhľadávacích operácií a spolu s výsledkami vyhľadávaní viacerých najkratších ciest zapíše tieto zmerané exekučné časy do logovacieho diskového súboru *ShortestPaths.txt*. C# .NET aplikácia vo svojich výsledkoch nezobrazuje len samotné vyhľadané najkratšie cesty, ale aj dĺžky podciest jednotlivých vyhľadaných najkratších

ciest. C# .NET aplikácia obsahuje nasledovné 2 triedy s nasledovnými členmi (uvedené sú len dôležité členy tried):

- *GraphPQ*, ktorá má nasledovné členy:
  - členské premenné *numbVerticesPQ* a *numbEdgesPQ* pre uloženie počtu vrcholov a počtu hrán spracovávaného grafu, členskú premennú 2-rozmerné pole *edgesWeights*, ktoré slúži na prechodné uloženie dát grafu načítaných zo vstupného diskového ASCII súboru pred ich načítaním do prioritnej fronty inštančnou metódou *Dijkstra_PriorityQueue*
  - členskú metódu *CreateInputArr*, ktorá po zavolaní číta po riadkoch dáta zo vstupného diskového ASCII súboru, napr. zo súboru *tinyEWDm.txt* (obr. 2) a zapisuje ich do 2-rozmerného poľa *edgesWeights*, z ktorého tieto dáta číta inštančná metóda *Dijkstra_PriorityQueue* a vkladá ich do prioritnej fronty
  - vnorenú triedu *PriorityQueue*, ktorá obsahuje:
    - vnorenú triedu *Node*, ktorej objekty reprezentujú jeden uzol prioritnej fronty. Objekty tejto triedy tvoria prvky poľa *arr*, ktoré je členskou premennou triedy *PriorityQueue*. Trieda *Node* obsahuje členské premenné *data* a *priority*, v ktorých sú uložené dáta jedného uzla prioritnej fronty, ktorý je ako objekt triedy *Node* uložený v prvkoch poľa *arr*, ktoré je inštančnou premennou objektu *pq* (objekt triedy *PriorityQueue*) vytvoreného inštančnou metódou *Dijkstra_PriorityQueue*. Medzi ďalšie členy triedy *Node* patria operátorové funkcie, ktoré slúžia na porovnávanie jednotlivých uzlov prioritnej fronty podľa ich priorít.
    - členskú premennú pole *arr*, ktoré je typu *Node*, čiže prvkami tohto poľa, keď je konštruktorom *PriorityQueue* vytvorené, budú objekty triedy *Node* reprezentujúce uzly prioritnej fronty.
    - členskú premennú *count* slúžiacu pre ukladanie počtu objektov triedy *Node* v prioritnej fronte
    - členské metódy *Enqueue* a *Dequeue_min* slúžiace objektu prioritnej fronty na usporiadané vkladanie nových uzlov (objektov triedy *Node*) do prioritnej fronty a vyberanie čísla vrcholu s minimálnou vzdialenosťou od štartovacieho vrcholu spracovávaného grafu.
    - členské metódy *getParentIndex*, *getLeftChildIndex* a *getRight_ChildIndex*, ktoré po ich zavolaní vypočítajú a vrátia indexy rodičovského uzla, ľavého a pravého detského uzla pri usporiadanom vkladaní nových uzlov (objektov triedy *Node*) do prioritnej fronty a pri oprave usporiadania tejto fronty zhora alebo zdola členskými metódami *siftDown* a *siftUp* tejto triedy *PriorityQueue*.
  - členskú metódu ***Dijkstra_PriorityQueue***, ktorá hľadá pomocou Dijkstrovho algoritmu najkratšie cesty z používateľom vloženého štartovacieho vrcholu do všetkých vrcholov orientovaného grafu, ktorého dáta metóda ukladá do a neskôr číta z inštančnej premennej *arr* objektu prioritnej fronty *pq* (pre vstupný dátový súbor C# .NET aplikácie *tinyEWDm.txt* (obr. 2) bude inštančná premenná *arr* obsahovať dáta zobrazené na obr. 4).
  - členskú metóda *Relax*, ktorú volá členská metóda *Dijkstra_PriorityQueue* a ktorá aktualizuje vzdialenosti od štartovacieho vrcholu grafu všetkých priľahlých vrcholov spracovávaného vrcholu *u* metódou *Dijkstra_PriorityQueue* (Košťál, 2020).
  - ďalšie členské metódy, ktoré slúžia na zobrazenie na výstupe C# .NET aplikácie a na zapísanie do výstupného logovacieho diskového súboru

ShortestPaths.txt touto aplikáciou nájdených viacerých najkratších ciest a všetkých nájdených najkratších ciest začínajúcich v štartovacích vrcholoch viacerých zadaných hľadaných ciest a končiacich v ostatných vrcholoch orientovaného ohodnoteného grafu.

- triedu *Program* - v jej statickej metóde *Main* sú načítané vstupy používateľa do lokálneho poľa reťazcov a podľa počtu ním vložených požadovaných hľadaných najkratších ciest je vytvorený zodpovedajúci počet objektov triedy *GraphPQ*. Potom sú inštančnou metódou *CreateInputArr* týchto objektov načítané vstupné dáta grafu zo vstupného diskového súboru C# .NET aplikácie, napr. zo súboru *tinyEWDm.txt* (obr. 2), do inštančnej premennej objektu triedy *GraphPQ*, do 2-rozmerného dynamického poľa *edgesWeights*. Odtiaľ tieto dáta, po svojom zavolaní, číta inštančná metóda *Dijkstra_PriorityQueue* objektu triedy *GraphPQ* a ukladá ich do objektu prioritnej fronty *pq*. V tomto objekte prioritnej fronty má inštančná metóda *Dijkstra_PriorityQueue* uložený spracovávaný graf, v ktorom hľadá všetky najkratšie cesty zo zadaného štartovacieho vrcholu, odovzdaného do jej parametra, do všetkých ostatných vrcholov prehľadávaného grafu. Výsledky svojho hľadania zapíše inštančná metóda *Dijkstra_PriorityQueue* do dvoch výstupných polí *distance* a *previous*.
  Metóda *Main* obsahuje sekcie zdrojového kódu pre sériové, viacvláknové a paralelizované vyhľadávanie viacerých najkratších ciest s implementovanými volaniami inštančnej metódy *Dijkstra_PriorityQueue* pre viaceré počty viacerých najkratších hľadaných ciest. Takéto tri sekcie sú vytvorené pre nasledovné počty zadaných viacerých hľadaných najkratších ciest: 2, 3, 4, 5, 6, 7, 8, 9 a 10. Výsledky každého vyhľadávania viacerých hľadaných najkratších ciest, vrátane exekučných časov týchto vyhľadávaní, C# .NET aplikácia zapíše do svojho výstupu a do výstupného logovacieho diskového súboru *ShortestPaths.txt*.

Obr. 5: Zdrojový kód členskej metódy 'Dijkstra_PriorityQueue' triedy 'GraphPQ'

```
public void Dijkstra_PriorityQueue(ref double[] distance, ref int[] previous, int sV) {
  distance[sV] = 0;

  //vytvorenie objektu 'pq' triedy 'PriorityQueue', cize objektu prioritnej fronty
  PriorityQueue<int> pq = new PriorityQueue<int>();

  //vlozenie vsetkych vrcholov grafu (jeho data su ulozene v poli 'edgesWeights') aj s hodnotami hran
  //do prioritnej fronty 'pq' (vytvaranie uzlov prioritnej fronty)
  for (int i = 0; i < numbEdgesPQ; i++)
        pq.Enqueue(Convert.ToInt32(edgesWeights[i][0]), edgesWeights[i][2]);

  while (!pq.Empty()) //kym nie je prioritna fronta 'pq' prazdna
  {
    //z prioritnej fronty 'pq' vyberieme vrchol s minimalnou vzdialenostou od startov. vrcholu grafu 'sV'
    //(uzol prioritnej fronty s najnizsou prioritou)
    int u = pq.Dequeue_min();

    //pre kazdy prilahly vrchol 'v' ('edgesWeights[i][1]') spracovavaneho vrcholu 'u' ('edgesWeights[i][0]')
    for (int i = 0; i < numbEdgesPQ; i++)
          if (edgesWeights[i][0] == u)
    //sa metoda 'Relax' pokusi aktualizovat pomocou pseudokodu (1) jeho vzdialenost od startovacieho
    //vrcholu grafu 'sV'
    Relax(u, Convert.ToInt32(edgesWeights[i][1]), ref distance, ref previous, ref pq, edgesWeights[i][2]);
  }
}
```

Zdroj: Vlastné spracovanie inšpirované (Oumghar, 2015)

Obr. 6: Sekcia zdrojového kódu statickej metódy 'Main', ktorá pomocou inštančnej metódy 'Dijkstra_PriorityQueue' hľadá sériovo 4 najkratšie cesty v orientovanom ohodnot. grafe

```
else if ((split_inputs.Length / 2) == 4)  //ak pouzivatel vlozil 4 najkratsie cesty  {  . . .
  //vytvorenie a inicializacia 4 objektov 'g_pqX' triedy 'GraphPQ'
  GraphPQ g_pq1 = new GraphPQ();
  GraphPQ g_pq2 = new GraphPQ();
  GraphPQ g_pq3 = new GraphPQ();
  GraphPQ g_pq4 = new GraphPQ();
  . . .
  stopWatch.Reset(); //"vynulovanie" meraca casu
  stopWatch.Start(); //spustenie merania exekucneho casu
  //seriove hladanie 4 najkratsich ciest pomocou instancnej metody 'Dijkstra_PriorityQueue'
  g_pq1.Dijkstra_PriorityQueue(ref shortestDsts1, ref parents1, Convert.ToInt32(split_inputs[0]));
  g_pq2.Dijkstra_PriorityQueue(ref shortestDsts2, ref parents2, Convert.ToInt32(split_inputs[2]));
  g_pq3.Dijkstra_PriorityQueue(ref shortestDsts3, ref parents3, Convert.ToInt32(split_inputs[4]));
  g_pq4.Dijkstra_PriorityQueue(ref shortestDsts4, ref parents4, Convert.ToInt32(split_inputs[6]));

  stopWatch.Stop(); //zastavenie merania exekucneho casu
  TimeSpan ts_PQS = stopWatch.Elapsed; //ziskanie zmeranej dlzky casoveho intervalu
  string strIntervalStopWatch_PQS = ts_PQS.ToString();
  double msIntervalStopWatch_PQS = ts_PQS.TotalMilliseconds; . . .  }
```

Zdroj: Vlastné spracovanie

Obr. 7: Sekcia zdrojového kódu statickej metódy 'Main', ktorá pomocou inštančnej metódy 'Dijkstra_PriorityQueue' hľadá na viacerých vláknach 4 najkratšie cesty v orientovanom ohodnotenom grafe

```
else if ((split_inputs.Length / 2) == 4)  //ak pouzivatel vlozil 4 najkratsie cesty  {  . . .
  //vytvorenie delegate 'callerPQM1' (objektu) delegatoveho typu 'methodDeleg_PQ', ktory ukazuje na
  //instancnu metodu 'g_pq1.Dijkstra_PriorityQueue'
  methodDeleg_PQ callerPQM1 = new methodDeleg_PQ(g_pq1.Dijkstra_PriorityQueue);
  methodDeleg_PQ callerPQM2 = new methodDeleg_PQ(g_pq2.Dijkstra_PriorityQueue);
  methodDeleg_PQ callerPQM3 = new methodDeleg_PQ(g_pq3.Dijkstra_PriorityQueue);
  methodDeleg_PQ callerPQM4 = new methodDeleg_PQ(g_pq4.Dijkstra_PriorityQueue);

  stopWatch.Reset();
  stopWatch.Start(); //spustenie merania exekucneho casu
  //spustenie asynchronneho vykonavania instancnej metody 'g_pq1.Dijkstra_PriorityQueue' na
  //sekundarnom vlakne pomocou volania instanc. metody 'BeginInvoke' delegata (objektu) 'callerPQM1'
  IAsyncResult asyncSg_pq1Dijk_PQ = callerPQM1.BeginInvoke(ref shortestDsts1, ref parents1,
                                                           Convert.ToInt32(split_inputs[0]), null, null);
  IAsyncResult asyncSg_pq2Dijk_PQ = callerPQM2.BeginInvoke(ref shortestDsts2, ref parents2,
                                                           Convert.ToInt32(split_inputs[2]), null, null);
  IAsyncResult asyncSg_pq3Dijk_PQ = callerPQM3.BeginInvoke(ref shortestDsts3, ref parents3,
                                                           Convert.ToInt32(split_inputs[4]), null, null);
  IAsyncResult asyncSg_pq4Dijk_PQ = callerPQM4.BeginInvoke(ref shortestDsts4, ref parents4,
                                                           Convert.ToInt32(split_inputs[6]), null, null);

  //ukoncenie asynchronneho vykonavania instancnej metody 'g_pq1.Dijkstra_PriorityQueue' na
  //sekundarnom vlakne pomocou volania instanc. metody 'EndInvoke' delegata (objektu) 'callerPQM1'
  callerPQM1.EndInvoke(ref shortestDsts1, ref parents1, asyncSg_pq1Dijk_PQ);
  callerPQM2.EndInvoke(ref shortestDsts2, ref parents2, asyncSg_pq2Dijk_PQ);
  callerPQM3.EndInvoke(ref shortestDsts3, ref parents3, asyncSg_pq3Dijk_PQ);
  callerPQM4.EndInvoke(ref shortestDsts4, ref parents4, asyncSg_pq4Dijk_PQ);

  stopWatch.Stop(); //zastavenie merania exekucneho casu
  TimeSpan ts_PQM = stopWatch.Elapsed;
  string strIntervalStopWatch_PQM = ts_PQM.ToString();
  double msIntervalStopWatch_PQM = ts_PQM.TotalMilliseconds;  . . .  }
```

Zdroj: Vlastné spracovanie

Obr. 8: Sekcia zdrojového kódu statickej metódy 'Main', ktorá pomocou inštančnej metódy 'Dijkstra_PriorityQueue' hľadá paralelne 4 najkratšie cesty v orientov. ohodnotenom grafe

```
else if ((split_inputs.Length / 2) == 4)  //ak pouzivatel vlozil 4 najkratsie cesty
{ . . .
  stopWatch.Reset();
  stopWatch.Start(); //spustenie merania exekucneho casu

  // paralelne vykonanie 4 uloh
  Parallel.Invoke(
    () =>
    {
      g_pq1.Dijkstra_PriorityQueue(ref shortestDsts1, ref parents1, Convert.ToInt32(split_inputs[0]));
    }, //zatvorenie 1. 'Action' delegata
    () =>
    {
      g_pq2.Dijkstra_PriorityQueue(ref shortestDsts2, ref parents2, Convert.ToInt32(split_inputs[2]));
    }, //zatvorenie 2. 'Action' delegata
    () =>
    {
      g_pq3.Dijkstra_PriorityQueue(ref shortestDsts3, ref parents3, Convert.ToInt32(split_inputs[4]));
    }, //zatvorenie 3. 'Action' delegata
    () =>
    {
      g_pq4.Dijkstra_PriorityQueue(ref shortestDsts4, ref parents4, Convert.ToInt32(split_inputs[6]));
    } //zatvorenie 4. 'Action' delegata
); //zatvorenie 'Parallel.Invoke' metody

  stopWatch.Stop(); //zastavenie merania exekucneho casu
  TimeSpan ts_PQP = stopWatch.Elapsed;
  string strIntervalStopWatch_PQP = ts_PQP.ToString();
  double msIntervalStopWatch_PQP = ts_PQP.TotalMilliseconds;
  . . .
}
```

Zdroj: Vlastné spracovanie

Obr. 9: Príklad vstupu (tučným písmom) a výstupu C# .NET aplikácie (zobrazená je iba jeho časť), ktorá hľadala 4 najkratšie cesty v orientovanom ohodnotenom grafe s 10 vrcholmi a 19 hranami (vstupné dáta grafu načítané zo súboru 'tinyEWDm.txt' (obr. 2)) z obr. 1

```
Choose input file: 10vEWDG.txt (insert: 1), tinyEWD.txt (insert: 2), tinyEWDm.txt (insert: 3),
mediumEWD.txt (insert: 4)  3
The adjacency list loaded from the 'tinyEWDm.txt' file.

Insert 1 source and 1 destination vertex or several source and destination vertices separated by ',' (e.g.
'sv1 dv1, sv2 dv2, sv3 dv3'; '5 2, 4 6, 7 1' for 3 source and 3 destination vertices) for MULTIPLE finding:
5 2, 4 6, 7 1, 2 4

The graph in which will be found the shortest paths in serial, on multiple threads and in parallel has '10'
vertices and '19' edges.

== Dijkstra's Shortest Paths computed by the 'Dijkstra_PriorityQueue' method in Serial ==
** Dijkstra's Shortest Paths computed by the 'Dijkstra_PriorityQueue' method **
from -> to      Distance Path
-------------------------------------------------------------
The shortest path from the source vertex (5) to the destination vertex (2)
                         0,53 0,37 0,46
5 -> 2          1,36     5 -> 1 -> 9 -> 2
-----------------------------------
All the shortest paths from the source vertex (5) to other vertices
                         0,53 0,37 0,64 0,69
5 -> 0          2,23     5 -> 1 -> 9 -> 6 -> 0

                         0,53
5 -> 1          0,53     5 -> 1

                         0,53 0,37 0,46
5 -> 2          1,36     5 -> 1 -> 9 -> 2

                         0,34
5 -> 3          0,34     5 -> 3
          . . .

** Dijkstra's Shortest Paths computed by the 'Dijkstra_PriorityQueue' method **
from -> to      Distance Path
-------------------------------------------------------------
The shortest path from the source vertex (4) to the destination vertex (6)
                         0,31 0,53 0,37 0,64
4 -> 6          1,85     4 -> 5 -> 1 -> 9 -> 6
-----------------------------------
All the shortest paths from the source vertex (4) to other vertices
                         0,31 0,53 0,37 0,64 0,69
4 -> 0          2,54     4 -> 5 -> 1 -> 9 -> 6 -> 0

                         0,31 0,53
4 -> 1          0,84     4 -> 5 -> 1

                         0,31 0,53 0,37 0,46
4 -> 2          1,67     4 -> 5 -> 1 -> 9 -> 2
        . . .
Execution time of serial finding 4 shortest paths: 00:00:00.0017756, 1,7756 ms
```

Zdroj: Vlastné spracovanie

5 Experiment, jeho výsledky, ich krátka analýza

Účelom experimentu je potvrdiť alebo vyvrátiť našu hypotézu: predpokladáme, že zo sériového, viacvláknového a paralelného hľadania viacerých najkratších ciest v orientovanom ohodnotenom grafe pomocou inštančnej metódy *Dijkstra_PriorityQueue* našej C# .NET aplikácie, ktorá má implementovaný Dijkstrov algoritmus, je najefektívnejším spôsobom vyhľadávania paralelné vyhľadávanie.

Experiment sme vykonali pomocou našej C# .NET aplikácie, ktorá na vstupe načítala vstupné dáta postupne dvoch orientovaných ohodnotených grafov z nasledujúcich vstupných ASCII diskových súborov:

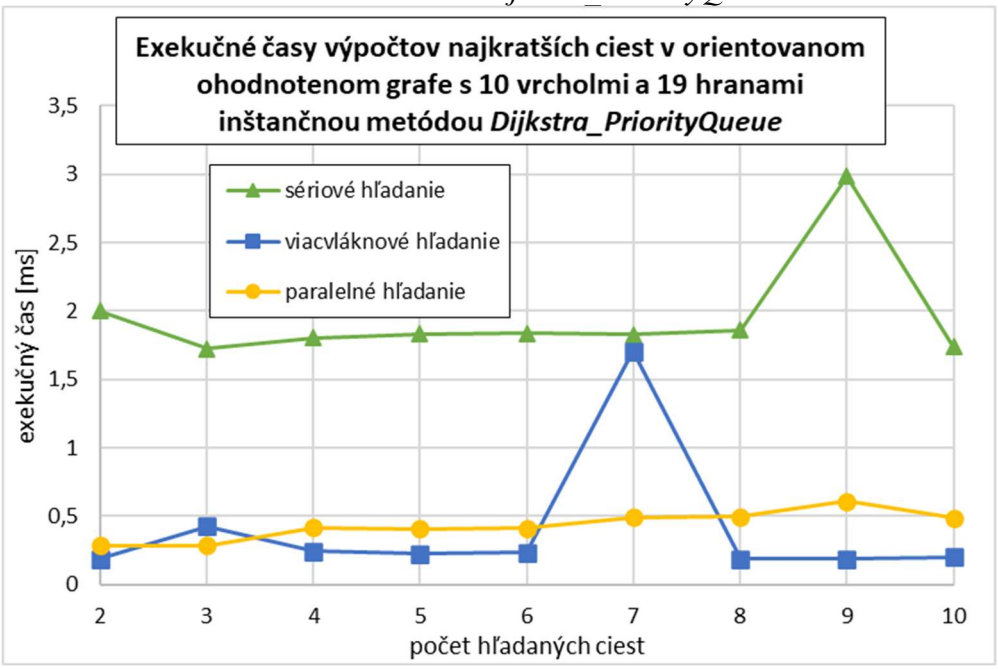
- *tinyEWDm.txt* (obr. 2) - súbor obsahuje dáta 10-vrcholového 19-hranového orientovaného ohodnoteného grafu
- *mediumEWD.txt* (Sedgewick, 2018) - súbor obsahuje dáta 250-vrcholového 2546-hranového orientovaného ohodnoteného grafu

Po načítaní vstupných dát grafu z každého z týchto vstupných diskových súborov našou C# .NET aplikáciou sme spustili vyhľadávanie postupne 2, 3, 4, 5, 6, 7, 8, 9 a 10 najkratších ciest v každom z týchto grafov. Postupne sme hľadali nasledujúce rovnaké sady najkratších ciest v každom z týchto grafov (napr. vstupné dáta '*v1 v2, v3 v4*' reprezentujú 2 hľadané najkratšie cesty, prvú z vrcholu *v1* do vrcholu *v2* a druhú najkratšiu cestu z vrcholu *v3* do vrcholu *v4*)

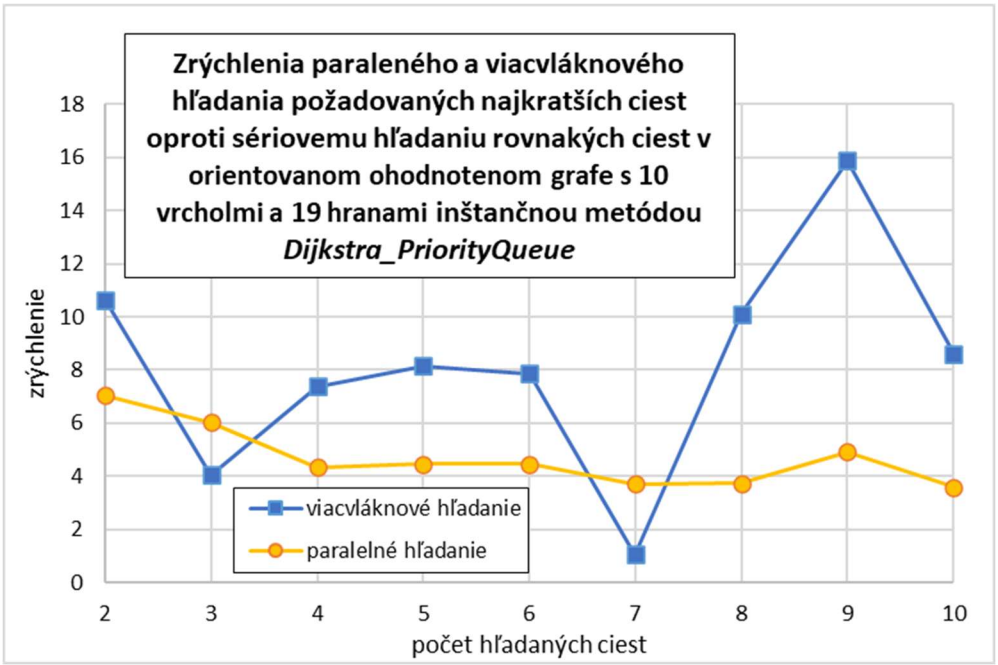
- 5 2, 4 6, (2 najkratšie cesty)
- 5 2, 4 6, 7 1 (3 najkratšie cesty)
- 5 2, 4 6, 7 1, 2 4 (4 najkratšie cesty)
- 5 2, 4 6, 7 1, 2 4, 1 5 (5 najkratších ciest)
- 5 2, 4 6, 7 1, 2 4, 1 5, 6 1 (6 najkratších ciest)
- 5 2, 4 6, 7 1, 2 4, 1 5, 6 1, 3 1 (7 najkratších ciest)
- 5 2, 4 6, 7 1, 2 4, 1 5, 6 1, 3 1, 7 2 (8 najkratších ciest)
- 5 2, 4 6, 7 1, 2 4, 1 5, 6 1, 3 1, 7 2, 3 5 (9 najkratších ciest)
- 5 2, 4 6, 7 1, 2 4, 1 5, 6 1, 3 1, 7 2, 3 5, 3 7 (10 najkratších ciest)

Pre každú z týchto 10 vstupných sád najkratších ciest vykonala C# .NET aplikácia v oboch zo vstupu načítaných grafoch sériové, viacvláknové a paralelné vyhľadávanie zadaného počtu najkratších ciest, pričom pri každom vyhľadávaní tiež zmerala a do svojho výstupu a logovacieho súboru *ShortestPaths.txt* zapísala exekučné časy týchto vyhľadávaní spolu s vyhľadanými najkratšími cestami. Exekučné časy jednotlivých vyhľadávaní viacerých najkratších ciest v dvoch orientovaných ohodnotených grafoch a zrýchlenia viacvláknových a paralelných vyhľadávaní so zobrazené v nasledujúcich grafoch.

Obr. 10: Exekučné časy výpočtov najkratších ciest v orientovanom ohodnotenom grafe s 10 vrcholmi a 19 hranami (vstupné dáta grafu načítané zo súboru 'tinyEWDm.txt' (obr. 2)) a zrýchlenia paralelného a viacvláknového hľadania požadovaných najkratších ciest inštančnou metódou 'Dijkstra_PriorityQueue'

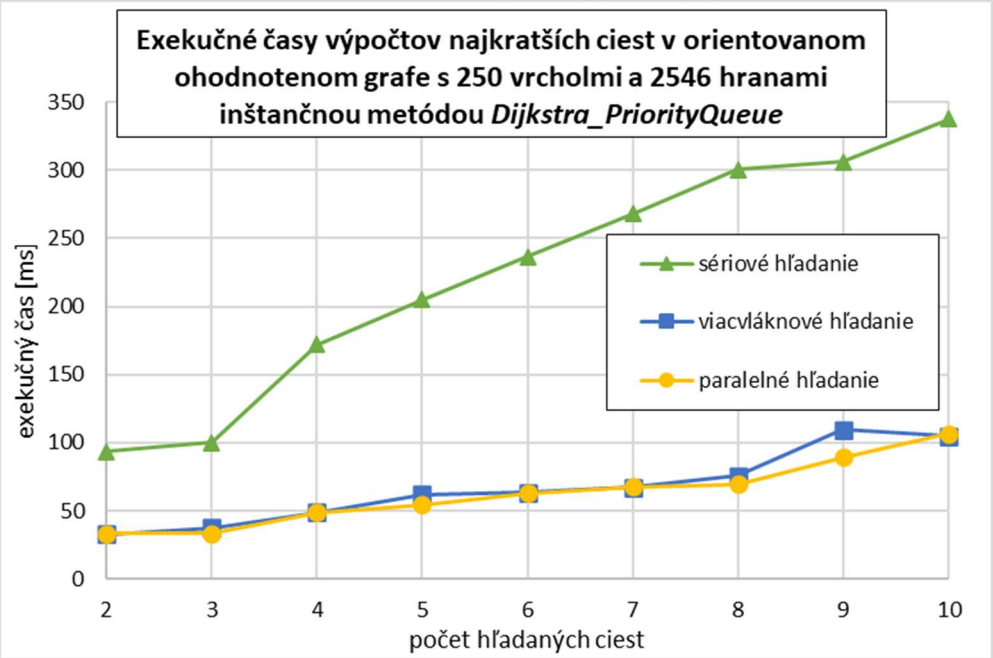


Zdroj: Vlastné spracovanie

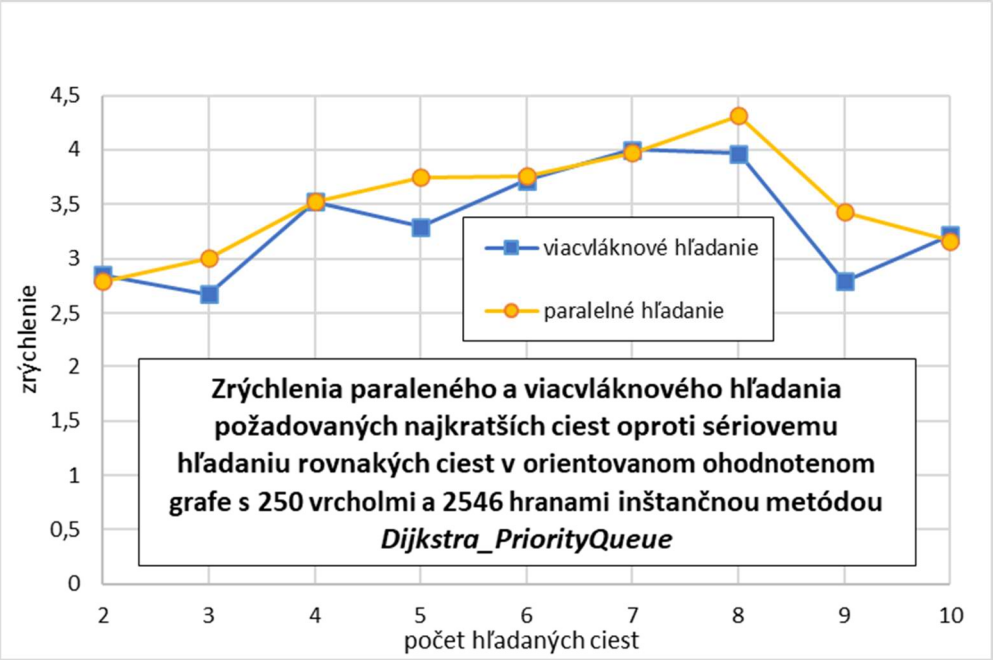


Zdroj: Vlastné spracovanie

Obr. 11: Exekučné časy výpočtov najkratších ciest v orientovanom ohodnotenom grafe s 250 vrcholmi a 2546 hranami (vstupné dáta grafu načítané zo súboru 'mediumEWD.txt') a zrýchlenia paralelného a viacvláknového hľadania požadovaných najkratších ciest inštančnou metódou 'Dijkstra_PriorityQueue'



Zdroj: Vlastné spracovanie



Zdroj: Vlastné spracovanie

Krátka analýza výsledkov experimentu. Z porovnania exekučných časov sériového, viacvláknového a paralelného hľadania viacerých najkratších ciest v orientovanom ohodnotenom grafe s 10 vrcholmi a 19 hranami (vstupné dáta grafu načítané zo súboru '*tinyEWDm.txt*' (obr. 2)) je zrejmé, že vyššiu exekučnú efektívnosť má viacvláknové hľadanie. O málo menšiu exekučnú efektívnosť, maximálny rozdiel od viacvláknového hľadania je 0,4184 ms, má paralelné hľadanie. Vo väčšine hľadaní dosahovalo viacvláknové hľadanie

oproti sériovému hľadaniu väčšie zrýchlenie, až do 15,875, ako paralelné hľadanie oproti sériovému hľadaniu.

Z porovnania exekučných časov sériového, viacvláknového a paralelného hľadania viacerých najkratších ciest v orientovanom ohodnotenom grafe s 250 vrcholmi a 2546 hranami (vstupné dáta grafu načítané zo súboru '*mediumEWD.txt*') je zrejmé, že v drvivej väčšine hľadaní má veľmi tesne vyššiu exekučnú efektívnosť paralelné hľadanie. Okrem troch hľadaní dosahovalo paralelné hľadanie oproti sériovému hľadaniu veľmi tesne väčšie zrýchlenie, až do 4,311, ako viacvláknové hľadanie oproti sériovému hľadaniu.

Z porovnania exekučných časov sériového, viacvláknového a paralelného hľadania viacerých najkratších ciest v malom orientovanom ohodnotenom grafe (10-vrcholový 19-hranový graf) a exekučných časov rovnakých operácií vo veľkom grafe (250-vrcholový 2546-hranový graf) môžeme povedať, že pri veľkom grafe sa naša hypotéza, že paralelné hľadanie je exekučne najefektívnejšie, aj keď veľmi tesne, potvrdila. Avšak, pri hľadaní viacerých najkratších ciest v malom orientovanom ohodnotenom grafe sa naša hypotéza nepotvrdila, pretože exekučne efektívnejšie, aj keď len o málo, bolo viacvláknové hľadanie.

6 Záver

Z krátkej analýzy výsledkov experimentu je zrejmé, že naša hypotéza, že paralelné hľadanie je exekučne najefektívnejšie, sa potvrdila len pri hľadaní viacerých najkratších ciest vo veľkom orientovanom ohodnotenom grafe (250-vrcholový 2546-hranový graf). Pri hľadaní viacerých najkratších ciest v malom orientovanom ohodnotenom grafe (10-vrcholový 19-hranový graf) sa naša hypotéza nepotvrdila, pretože exekučne efektívnejšie, aj keď len o málo, bolo viacvláknové hľadanie.

Z porovnania exekučných časov sériového, viacvláknového a paralelného hľadania viacerých najkratších ciest v malom orientovanom ohodnotenom grafe (10-vrcholový 19-hranový graf) a exekučných časov rovnakých operácií vo veľkom grafe (250-vrcholový 2546-hranový graf) a z porovnania zrýchlení paralelného a viacvláknového hľadania oproti sériovému hľadaniu v tomto a malom a v tomto veľkom grafe, sa dá vyvodiť predpoklad, že pri hľadaní viacerých najkratších ciest v orientovanom ohodnotenom grafe väčšom ako náš veľký graf je možné, že paralelné hľadanie by bolo exekučne efektívnejšie s väčším rozdielom oproti viacvláknovému hľadaniu ako pri hľadaní týchto ciest v našom veľkom grafe.

Literatúra

[1] Košťál, I. (2020). *Vplyv dátovej štruktúry použitej pre ukladanie dát grafu na exekučnú efektívnosť metód hľadajúcich v ňom najkratšie cesty pomocou Dijkstrovho algoritmu*. AIESA 2020 medzinárodná vedecká konferencia. AIESA - Budovanie spoločnosti založenej na vedomostiach. Bratislava: Letra Edu.

[2] Niemann, T. (1999). *Sorting and Searching Algorithms*. epaperpress.com.

[3] Oumghar, K. (2015, December 22). *Graphs and Dijkstra's Algorithm (C#)*. Retrieved August 30, 2020, from https://simpledevcode.wordpress.com/2015/12/22/graphs-and-dijkstras-algorithm-c/.

[4] Sedgewick, R. (1998). *Algorithms in C parts 1-4. Fundamentals, data structures, sorting, searching*. Addison-Wesley Publishing Company, Inc.

[5] Sedgewick, R. (2018). *Algorithms, 4th Edition*. Retrieved January 12, 2021, from https://algs4.cs.princeton.edu/home/