FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Algoritmy pro řešení problému čínského listonoše
Student:	Matěj Razák
Vedoucí:	doc. Ing. Ivan Šimeček, Ph.D.
Studijní program:	Informatika
Studijní obor:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce zimního semestru 2020/21

Pokyny pro vypracování

1) Nastudujte problém čínského listonoše [1,2,3].
2) Analyzujte různá řešení problému čínského listonoše, např. Maďarská metoda [4], síťové toky [5].
3) Implementujte metody z bodu 2) v jazyce C++.
4) Implementované algoritmy optimalizujte pomocí paralelizace technologií OpenMP.
5) Otestujte a porovnejte implementované algoritmy.

Seznam odborné literatury

[1] Martin Groetschel, Ya-xiang Yuan: Euler, Mei-Ko Kwan, Konigsberg, and a Chinese Postman, Documenta Mathematica · Extra Volume ISMP (2012) 43–50
[2] M.K. Gordenko, S.M. Avdoshin: The Mixed Chines Postman Problem. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 4, 2017, pp. 107-122. DOI: 10.15514/ISPRAS-2017-29(4)-7
[3] M. Guan: On the Windy Postman Problem. Discrete Applied Mathematics 9 (1984), pp 41-46
[4] H.W. Kuhn: The Hungarian Method for the Assignment Problem, Naval Research Logistics Quarterly 2 (1955) 83–97
[5] J.B. Orlin, R. K. Ahuja, and Thomas L. Magnanti: Network Flows: Theory, Algorithms, and Applications, USA 1993

doc. Ing. Jan Janoušek, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 9. ledna 2019

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Algoritmy pro řešení problému čínského listonoše

Matěj Razák

Katedra teoretické informatiky
Vedoucí práce: Ing. Ivan Šimeček, Ph.D.

7. ledna 2021

Poděkování

Děkuji vedoucímu práce Ing. Ivanu Šimečkovi, Ph.D. za trpělivost a vstřícnost při psaní této práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principu při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisu. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisu, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programu, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli zpusobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelum). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným zpusobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 7. ledna 2021 . . . . . .

Odkaz na tuto práci

Razák, Matěj. *Algoritmy pro řešení problému čínského listonoše*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

Abstrakt

Tato bakalářská práce se věnuje vybraným algoritmům pro řešení problému čínského listonoše (varianta DCPP - Directed Chinese Postman Problem), tj. jedné z optimalizačních úloh z oboru teorie grafů. Úkolem této úlohy je najít nejkratší trasu listonoše. Ta vede všemi ulicemi (to jsou hrany grafu) s tím, že se nakonec musí vrátit do výchozího bodu (vrcholu grafu). Práce řeší verzi s orientovanými ohodnocenými hranami. Cílem práce je použít různé dílčí algoritmy pro jednotlivé fáze řešení celého problému. Realizace používá paralelizaci a porovnává její použití s verzí sekvenční. Pro testování kompletní implementace se používají efektivnější z možných dílčích algoritmů.

Testování s existující implementací prokázalo výrazně lepší efektivitu (až 10-ti násobné zrychlení) mého řešení.

Klíčová slova teorie grafů, problém čínského listonoše, algoritmizace, paralelizace, eulerovský cyklus

Abstract

This bachelor thesis deals with selected algorithms for solving the Problem of Chinese Postman (DCPP variant), i.e. one of the optimization problems

in the field of graph theory. The task is to find the shortest route of the postman on his way through all the streets (those are the edges of the graph), considering that he must eventually return to the starting point. The work solves the version with oriented weighted edges. The thesis aims to use various partial algorithms for individual phases of solving the whole problem. The implementation uses the parallelization and compares it with the sequential version. More efficient of possible sub-algorithms are used to test the complete implementation. Testing of the existing implementation proved that my solution is significantly more efficient (up to 10-times higher speedup).

Keywords graph theory, Chinese Postman Problem, alghorithm development, parallelism, Eulerian cycle

Obsah

Seznam obrázků

Seznam algoritmů

Seznam tabulek

Úvod

Tato práce se zaměřuje na úlohu z teorie grafů, konkrétně na řešení tzv. problému čínského listonoše (Chinese Postman Problem = CPP). Úloha má jméno po čínském matematikovi Mei-Ko Kwanovi, který v roce 1962 publikoval článek v časopise Chinese Mathematics [1]. Ve svém článku se zabýval optimalizací trasy listonoše v úloze s obecně formulovaným cílem minimalizovat délku cesty, která vede alespoň jednou přes všechny hrany a vrací se do výchozího místa.

Další praktické použití je např. řešení sítě svozu komunálního odpadu nebo úklid sněhu (příp. zimní posyp cest) městskými službami. Ulice jsou hranami grafu, křižovatky ulic jsou vrcholy (uzly). Jedná se o úlohy nalezení cesty s vynaložením nejnižších nákladů (těmi může být kvantifikovaná časová náročnost, finanční náklady na provoz – pohonné hmoty, minimalizace celkové délky trasy apod.).

Podle charakteru hran lze rozlišovat modely úlohy čínského listonoše pro orientovaný, neorientovaný a smíšený graf. Tato práce řeší CPP pro orientované grafy s kladnými celými hodnotami hran grafu.

Přípravou pro vlastní implementaci řešení CPP je kapitola 3. Samotná realizace je pak popsána v kapitole 4. Kapitola postupně obsahuje popis použitých prostředků a optimalizace 4.1, následují základní dva přístupy k řešení, sekvenční 4.2 a paralelní 4.3. Podkapitola 4.4 obsahuje popis použití a spouštění programu, včetně popisu vstupních a výstupních dat.

Kapitola 5 je věnována fázi testování. Stanovuje metodiku testování, prezentuje i vlastní testy, porovnání výsledných implementací algoritmu a test správnosti.

Kapitola 1

Cíle práce

Úvodním dílčím cílem je nastudování potřebné teorie a stavu současného bádání v odborné literatuře. Hlavním cílem práce je navržení a implementace programu pro řešení varianty problému čínského listonoše (Directed CPP), a to za použití paralelizace algoritmů. Řešení předpokládá použití jen orientovaných grafů. Program ošetřuje správnost vstupních dat (platnosti zadání). Snahou je efektivní implementace zvolených algoritmů optimalizací, nasazením paralelizace zvolených algoritmů a porovnání s jednovláknovou variantou řešení. Následuje srovnání jednotlivých použitých algoritmů z hlediska jejich efektivnosti a testování kompletní úlohy.

Kapitola 2

Analýza současného stavu řešení problému

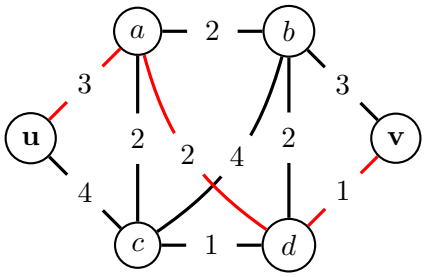
Tato kapitola obsahuje základní pojmy, vztahující se konkrétně k aplikovanému řešení daného problému (tedy k úloze DCPP). Jedná se o popis vybraných relevantních termínů teorie grafů, přehled použité terminologie. Další část kapitoly obsahuje uvedení do samotného problému čínského listonoše, včetně různých typů úloh CPP a metodiku řešení.

2.1 Základní pojmy teorie grafů

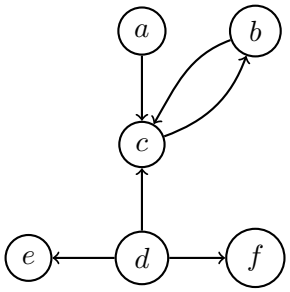
Aplikace teorie grafů v praxi pokrývá celou řadu úloh. Grafy zachycují jevy reálného světa. Jejich společným principem je zachycení struktury dané úlohy a jejích vlastností pomocí grafu. Základními prvky jsou hrany a vrcholy. Takto vypracovaný popis je prvním krokem k řešení daného problému.

Definice 1. Graf G *lze definovat jako uspořádanou dvojici neprázdné množiny vrcholů* V *(angl. vertices) a množiny hran* E *(angl. edges):* $G = (V, E)$.

V grafickém zobrazení se vrcholy zobrazují jako body a hrany jako spojnice spojující tyto body. Obrázek 2.1 znázorňuje jednu z typických úloh teorie grafů, totiž hledání nejkratší cesty (v tomto případě z bodu u do bodu v). Vrcholy jsou v tomto případě místa (např. města, městské čtvrtě, či domy, tedy $a, b, \ldots$), hrany jejich vzdálenosti (případně další vlastnosti nebo vztahy těchto spojnic, hranám jsou přiřazeny hodnoty 1, 2, 3 a 4).



Obrázek 2.1: Cesta v grafu z u do v



Obrázek 2.2: Orientovaný graf

Hrany a vrcholy, stupeň vrcholu

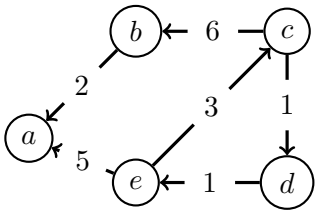
Definice 2. Orientovaná hrana *(viz obr. 2.2) má vyznačený směr průchodu z vrcholu do vrcholu.* Neorientovanou hranou *lze procházet v obou směrech. Neorientovaná hrana je množina dvou vrcholů* $\{x, y\}$*, orientovaná hrana je uspořádaná dvojice vrcholů* (x, y)*.*

Definice 3. Ohodnocená hrana *(viz obr. 2.3) – ohodnocení vyjadřuje kvantitu nebo kvalitu vztahů mezi vrcholy (např. vzdálenost nebo průchodnost).*

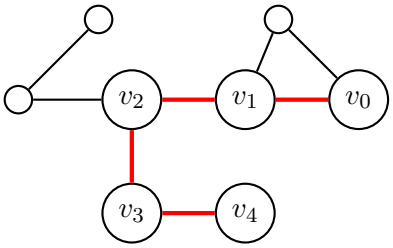
Rozlišujeme dva případy **stupně vrcholu**: v neorientovaném a orientovaném grafu.

Definice 4. *V neorientovaném grafu platí, že* stupeň vrcholu *je počet hran spojující daný vrchol s jinými vrcholy:*
$deg(u) = |\{e \in E | u \in e\}|$

Definice 5. *V orientovaném grafu se rozlišuje* vstupní stupeň *vrcholu* $deg^+(u)$ *a* výstupní stupeň *vrcholu* $deg^-(u)$*:*
vstupní stupeň $deg^+(u) = |\{e \in E | \exists v \in V : e = (v, u)\}|$
výstupní stupeň $deg^-(u) = |\{e \in E | \exists v \in V : e = (u, v)\}|$



Obrázek 2.3: Orientovaný ohodnocený graf s pěti vrcholy



Obrázek 2.4: Cesta v grafu

Definice 6. Sled *je posloupnost vrcholů a hran* $(v_0, e_1, v_1, e_2, \dots e_n, v_n)$ *v grafu* G *taková, že* $e_i = \{v_i, v_i + 1\} \in E(G)$.

Definice 7. *Pokud se v posloupnosti (ve sledu) neopakují hrany, tak ji nazveme* tah *a pokud se neopakují ani vrcholy, tak ji nazveme* cesta *(viz obr. 2.4).*

Definice 8. Uzavřený tah *je tah, který začíná i končí ve stejném vrcholu.*

Definice 9. Délka cesty *je součet ohodnocení hran v dané cestě.*

Definice 10. Nejkratší uv-cesta *je cesta, která má minimální délku ze všech* uv*-cest.*

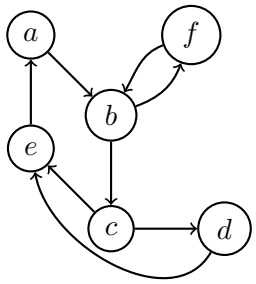
Výše uvedené definice jsou převzaty z [2, přednáška č. 1, 2].

Typy grafů

Definice 11. Orientovaný graf *je takový, v němž každá hrana je orientovaná (jeden z vrcholů je výchozí, druhý je koncový). Hraně je možné přiřadit hodnotu, pak se jedná o graf hranově ohodnocený.* Neorientovaný graf *má všechny hrany neorientované (nerozlišuje výchozí a koncové vrcholy).*

Definice 12. Graf G *je* souvislý, *jestliže v něm pro každé jeho dva vrcholy* u, v *existuje cesta. Jinak je* G nesouvislý.

Pro **orientované** grafy rozlišujeme grafy slabě a silně souvislé.



Obrázek 2.5: Silně souvislý graf

Definice 13. *Orientovaný graf $G = (V, E)$ nazveme* silně souvislý *(viz obr. 2.5), pokud pro každé dva vrcholy $u, v \in V$ existuje v G orientovaná cesta z u do v a současně orientovaná cesta z v do u.*

Definice 14. *Orientovaný graf je* slabě souvislý*, pokud zrušením orientace hran dostaneme souvislý neorientovaný graf.*

Definice 15. Graf $G = (V, E)$ *je* bipartitní*, pokud lze jeho množinu vrcholů V rozdělit na dvě části X a Y tak, že pro každou hranu $\{u, v\} \in E$ platí $u \in X$ a $v \in Y$ nebo naopak. Částem X a Y se říká partity grafu G.*

Eulerovský graf

Jeho podstatou je *kresba jedním tahem*, tedy zadání: nakreslete daný graf jedním uzavřeným tahem bez zvednutí tužky z papíru (žádná hrana se neobtahuje dvakrát). Tento tah začíná i končí ve stejném vrcholu.

Definice 16. Tah *se nazývá* eulerovský*, když prochází všechny hrany grafu právě jednou. Uzavřenému eulerovskému tahu se říká* eulerovský cyklus*.*

Definice 17. Graf *je* eulerovský*, když v něm existuje eulerovský cyklus.*

Věta 1. *Eulerovský graf je takový souvislý neorientovaný graf, který má všechny vrcholy sudého stupně.*

Důkaz. Důkaz viz [3, s. 48] □

Věta 2. *Orientovaný graf G je eulerovský, právě tehdy když G je silně souvislý a pro všechny vrcholy v platí $deg^+(v) = deg^-(v)$.*

Důkaz. Důkaz viz [4, s. 3]. □

Výše uvedené základní pojmy jsou převzaty z textů [2], [3], [5], [6].

2.2 Problém čínského listonoše

Historie

Chinese Postman Problem ([7], [8]), problém čínského listonoše (správně ovšem čínský problém listonoše) formuloval čínský vědec v oblasti matematického programování Mei-Ko Kwan (nar. 1934 v Šanghaji) ve svém článku v roce 1962 [1]. Jedná se o optimalizační úlohu.

Kwan se specializoval na problém plánování cest jako generalizaci problému eulerovského tahu s aplikací v oblasti dopravního plánování (např. určení časově nejkratší trasy vozového parku sněžných pluhů v daném městě, posyp cest, svoz komunálního odpadu apod.). Ve zmíněném článku se zabýval optimalizací tratě listonoše v úloze s obecně formulovaným cílem minimalizovat délku cesty, která vede alespoň jednou přes všechny hrany a nakonec se vrátit do výchozího vrcholu.

Jedná se o úlohu z teorie grafů. Ulice jsou hranami grafu a křižovatky jsou vrcholy grafu. Podle charakteru hran rozlišujeme úlohu pro orientovaný a neorientovaný graf. Obcházený obvod je souvislý ohodnocený graf (hrany jsou v tomto případě ulice ohodnocené délkou). V případě orientovaného grafu je graf silně souvislý.

Typy úloh

Původní nejjednodušší úloha byla časem rozvedena na další varianty CPP [9, 10].

Typy úloh CPP rozlišujeme podle typu grafu, kterým je úloha popsaná.

Výchozí úlohou CPP je neorientovaná úloha UCPP (Undirected CPP) – graf obsahuje pouze neorientované hrany. Orientovaná úloha DCPP (Directed CPP) - zadaný graf obsahuje pouze orientované hrany.

Smíšená úloha MCPP (Mixed CPP) – graf obsahuje jak orientované, tak i neorientované hrany.

Venkovský neboli příměstský RPP (Rural Postman Problem) – hrany grafu jsou rozděleny na povinné a nepovinné, tzn. součástí sledu musí být povinně pouze povinné hrany, nepovinné hrany pak mohou, ale nemusí.

Asymetrický WPP (Windy Postman Problem) [11] – hrany grafu jsou oceněné rozdílně v závislosti na jejich orientaci.

Asymetrický venkovský CPP (WRCPP) – tato úloha je kombinací WRCPP a RPP. Dále existují další úlohy, které jsou kombinací jiných.

Tato bakalářská práce se zabývá úlohou **orientovaný CPP (DCPP)**.

Metodika řešení

Zadání DCPP v termínech teorie grafů: V **silně souvislém orientovaném grafu** (představujícím listonošův okrsek) nalézt eulerovský cyklus. Pokud v zadaném grafu eulerovský cyklus neexistuje, znásobit dle potřeby některé hrany (listonoš prochází některé ulice vícekrát) tak, aby cyklus vzniknul a zároveň aby součet ohodnocení hran (délky ulic) byl minimální.

Pro popis algoritmu DCPP [12] zavádíme $\delta(v) = deg^-(v) - deg^+(v)$.
Jestliže $\delta(v) = 0$, potom v je vyvážený vrchol. Jinak je nevyvážený.
Graf je eulerovský, když každý jeho vrchol je vyvážený.
D^+ je množina vrcholů s kladnou δ $D^+ = \{v|\delta(v) > 0\}$
D^- je množina vrcholů se zápornou δ $D^- = \{v|\delta(v) < 0\}$

Eulerovský cyklus grafu je optimální trasa listonoše, protože každou hranu prochází právě jednou.
Jestliže graf není eulerovský, musíme ho na eulerovský převést přidáváním cest:

- vyhledat nejkratší cesty mezi všemi vrcholy v D^- a D^+ (vždy z vrcholů z D^- do vrcholů z D^+)
- aplikovat přiřazovací problém tak, aby součet ohodnocených hran přidávaných cest byl minimální (obecně je třeba k dodatečných cest, kde $k = \sum_{v \in D^-} \delta(v)$)
- přidat hrany výsledných cest do grafu

Věta 3. *Pro silně souvislý graf $G = (V, E)$ platí, že $\sum_{v \in D^-} \delta(v)) = O(|E|) = O(|V|^2)$.*

Důkaz. S přidáním hrany do grafu se suma může zvětšit pouze o 1 (jelikož se pouze jednomu vrcholu zvětší deg^- o 1). Tedy je ohraničena počtem hran.

V úplném grafu je $(|V|^2 - |V|)$ hran. To že suma může být kvadratická vzhledem k počtu vrcholů lze ukázat na následujícím příkladu.

Máme dva eulerovské grafy (tj. silně souvislé se všemi vrcholy vyváženými) se stejným počtem vrcholů n. Přidají se hrany s každého vrcholu jednoho grafu do každého vrcholu druhého grafu. Poté se ještě přidá jedna hrana opačným směrem, aby se zachovala silná souvislost. Výsledný graf (spojení dvou eulerovských grafu tímto způsobem) má $2n$ vrcholů a $\sum_{v \in D^-} \delta(v)) = n^2 - 1$. □

Vždy se všechny vrcholy v přiřazovacím problému musí spárovat, jinak by některé zůstaly nevyvážené. V implementaci to znamená, že vstupní matice do přiřazovacího problému (matice nejkratších vzdáleností mezi vrcholy) je vždy čtvercová.

Výsledkem bude graf se všemi vrcholy vyváženými, tedy eulerovský graf. V něm pak hledáme eulerovský cyklus (v našem případě Hierholzerovým algoritmem, viz kap. 3.4.1).

Zadání úlohy [12]:

vypočti: δ, D^-, D^+, d ($d_{i,j}$ je délka nejkratší cesty z vrcholu i do j)

pro f minimalizuj $\sum d_{i,j}\, f_{i,j}$ ($f_{i,j}$ je kolikrát přidáme do grafu cestu z vrcholu i do j)

přičemž:

- $f_{i,j} \in \mathbb{N}$
- $f_{i,j} \geq 0$
- $\sum_{j \in D^+} f_{i,j} = -\delta(i)$
- $\sum_{i \in D^-} f_{i,j} = \delta(j)$

Kapitola 3

Řešení problému čínského listonoše (metody a algoritmy)

DCPP je optimalizační úloha. Při řešení problému DCPP je třeba postupně vyřešit několik dílčích úkolů, na které lze aplikovat různé metody a algoritmy.

Tato kapitola se věnuje vybraným metodám a algoritmům řešení DCPP, tedy těm které byly použity v implementaci (viz kap. 4).

Jedná se o následující algoritmy:

- Hledání nejkratší cesty mezi vrcholy (Dijkstrův algoritmus, Floydův-Warshallův algoritmus) (viz kap. 3.1),
- algoritmy řešící přiřazovací problém (Maďarská metoda) (viz kap. 3.2),
- toky v sítích (Fordův-Fulkersonův algoritmus, Párování v bipartitním ohodnoceném grafu) (viz kap. 3.3),
- hledání eulerovského cyklu (Hierholzerův algoritmus) (viz kap. 3.4),
- tvorba silně souvislého grafu (modifikovaný Tarjanův algoritmus) (viz kap. 3.5).

Pro jednotlivé algoritmy (resp. metody) je uvedeno:

- Princip algoritmu, základní vlastnosti,
- základní věty a odkaz na důkaz korektnosti,
- pseudokód (resp. popis kroků),
- údaj o jejich časové složitosti.

3.1 Hledání nejkratších cest

K nalezení optimální trasy listonoše je třeba vytvořit eulerovský graf. Jedním z prvních kroků jeho vytvoření je nalezení nejkratších cest mezi nevyváženými vrcholy (z vrcholů se zápornou δ do vrcholů s kladnou δ) ohodnoceného orientovaného grafu.

Vybrané cesty budou v dalším kroku přidány do grafu. Pro řešení jsou použity zvolené algoritmy, konkrétně Floydův-Warshallův algoritmus (hledání nejkratší cest v grafu mezi všemi dvojicemi vrcholů grafu) a Dijkstrův algoritmus (hledání nejkratší cest z daného vrcholu grafu do všech ostatních vrcholů grafu).

3.1.1 Dijkstrův algoritmus

Jde o jeden z nejznámějších algoritmů pro hledání nejkratších cest. Navrhl ho Edsger Dijkstra v roce 1959 [13].

Dijsktrův algoritmus [14, s. 146-148], [2, přednáška č. 12] slouží k nalezení nejkratší cesty v grafu. Je konečný, protože v každé iteraci algoritmu se do množiny navštívených vrcholů přidá právě jeden vrchol, který se pak znova nenavštíví. To znamená, že průchodů cyklem je nanejvýš tolik, kolik má graf vrcholů. Používá se pro graf s nezáporně ohodnocenými hranami.

Algoritmus 1: Dijkstrův algoritmus

Vstup: Graf G, vstupní vrchol v_0
Výstup: Pole vzdáleností ($dist$), pole předchůdců ($prev$)

```
1  dist[v0] ← 0;
2  prev[v0] ← −1;
3  for each vertex v in graph G do
4  |   if v ≠ v0 then
5  |   |   dist[v] ← infinity;
6  |   |   prev[v] ← −1;
7  |   end
8  end
9  create priority queue Q;
10 Q.add_with_priority(v0,dist[v0]);
11 while Q is not empty do
12 |   u ← Q.extract_min();
13 |   for each neighbor v of u do
14 |   |   alt ← dist[u] + length(u, v);
15 |   |   if alt < dist[v] then
16 |   |   |   dist[v] ← alt;
17 |   |   |   prev[v] ← u;
18 |   |   |   Q.decrease_priority(v,alt);
19 |   |   end
20 |   end
21 end
22 return dist, prev
```

Důkaz korektnosti lze najít v [15, s. 117-119].

Algoritmus řeší obecnější problém:

Pro zadaný ohodnocený (orientovaný) graf G a počáteční vrchol v_0, nalezni vzdálenosti všech vrcholů grafu G od vrcholu v_0.

Principem tohoto algoritmu je prohledávání grafu do šířky (BFS). Vlna se však šíří na základě vzdálenosti od výchozího vrcholu. Vlna tedy zpracovává jen vrcholy, ke kterým již byla nalezena nejkratší cesta.

Složitost [14, s. 148-149]: uložíme-li všechna ohodnocení do pole (pro sekvenční vyhledávání), algoritmus poběží v čase $O(|V|^2)$. Pokud místo pole použijeme binární haldu, poběží v čase $O((|V| + |E|) \log |V|)$, případně s Fibonacciho haldou v $O(|E| + |V| \log |V|)$.

3.1.2 Floydův-Warshallův algoritmus

Floydův-Warshallův algoritmus ([14, s. 154-155], [3, kap. 16]) zpracovává orientovaný i neorientovaný graf. Graf může obsahovat i hrany záporné délky (nikoliv záporné cykly). Algoritmus porovnává cesty v grafu mezi všemi dvojicemi vrcholů. Pracuje tak, že postupně vylepšuje odhad na nejkratší cestu. Končí odhadem optimálním.

Algoritmus 2: Floydův-Warshallův algoritmus

Vstup: Graf G
Výstup: Matice nejkraších cest ($dist$), matice následníků ($next$)

```
1  dist[i][j] = 0 if i = j;
2  dist[i][j] = length(i, j) if edge between v and u exists;
3  dist[i][j] = infinity otherwise;
4  next[i][j] = j;
5  for k ← 0 to |V| − 1 do
6  |   for i ← 0 to |V| − 1 do
7  |   |   for j ← 0 to |V| − 1 do
8  |   |   |   if dist[i][j] > dist[i][k] + dist[k][j] then
9  |   |   |   |   dist[i][j] ← dist[i][k] + dist[k][j];
10 |   |   |   |   next[i][j] ← next[i][k];
11 |   |   |   end
12 |   |   end
13 |   end
14 end
15 return dist, next
```

Důkaz korektnosti lze najít v [16, s. 147-150].

Princip porovnávání:
dist(A,B) = min(dist(A,B),dist(A,C) + dist(C,B))

Popis algoritmu: Na vstupu je matice ohodnocení hran. Pokud mezi dvěma vrcholy vede hrana, matice obsahuje v daném místě tuto délku. Na diagonále matice jsou samé nuly a na ostatních pozicích bez hrany je nekonečno. V každém kroku algoritmu se tato matice přepočítá tak, aby zobrazovala vzdálenost všech dvojic vrcholů pomocí postupně se zvětšující množiny vhodných

prostředníků. S malou modifikací algoritmu lze rekonstruovat cesty mezi všemi vrcholy.

Složitost[14, s. 155]: asymptotická složitost je kubická, tedy $O(n^3)$.

3.2 Přiřazovací problém

Dalším krokem řešení DCPP je nalezení optimálního párování z vypočtených nejkratších cest. Přidáním cest tohoto párování do grafu se získá eulerovský graf. Optimální párování lze nalézt pomocí přiřazovacího problému (angl. Assignment problem) [17].

Přiřazovací problém je základním optimalizačním problémem. V praxi se jedná např. o problém přiřazení určitého počtu pracovníků na množinu úkolů (činností). Libovolný pracovník může být přiřazen na libovolný úkol. To je vázáno na určitý náklad (cenu práce), který se může lišit podle konkrétního přiřazení pracovník – úkol. Požaduje se obsazení co největšího počtu úkolů při přiřazování maximálně jednoho pracovníka na každý úkol. A to tak aby celkové náklady (celková hodnota) přiřazení byly minimální.

Jedná se tedy o následující úkol z teorie grafů: hledání párování daného rozsahu v ohodnoceném bipartitním grafu, kdy součet ohodnocení je minimální.

Jestliže je počet pracovníků a úkolů roven, jedná se o vyvážené (angl. balanced) přiřazení. V takovém případě mají obě části bipartitního grafu stejný počet vrcholů. V opačném případě jde o nevyvážené přiřazení.

Zadání úlohy:

Minimalizuj výraz $\sum_{i=1}^{n}\sum_{j=1}^{n} c_{i,j}\, x_{i,j}$, kde $c_{i,j}$ je cena spojená s přiřazením i-tého pracovníka k j-tému úkolu a pro proměnnou $x_{i,j}$ platí: $x_{i,j} = 1$, jestliže je i-tý pracovník přiřazen k j-tému úkolu a $x_{i,j} = 0$, jestliže není.

Přičemž platí:

- $\sum_{j=1}^{n} x_{i,j} = 1, \forall i \in \{1, \ldots, n\}$,
- $\sum_{i=1}^{n} x_{i,j} = 1, \forall j \in \{1, \ldots, n\}$.

Naivní algoritmus znamená procházet všechna přiřazení a počítat náklady každého z nich. Pro n pracovníků a n úkolů dostáváme $n!$ různých přiřazení.

Existuje ale mnoho různých algoritmů, které úlohu řeší v polynomiálním čase vzhledem k n. Jedním z nich je i **maďarská metoda** (pro řešení vyváženého přiřazovacího problému) [18].

3.2.1 Maďarská metoda

Maďarská metoda kombinuje prostředky lineárního programování a teorie grafů. Je nazvána podle prací maďarských vědců J. Egerváryho a D. Koeniga, jejichž práci využil a zpracoval H.W. Kuhn. Představil ji ve své stati [18] a nazval maďarskou metodou.

Původní Kuhnův problém řešil přiřazení množiny n lidí k množině n pracovních míst, přičemž stav pracovníka byl 0 nebo 1 (přiřazen či nepřiřazen podle toho, zda má pro danou práci kvalifikaci). Zadání je uspořádáno do matice, řádky představují pracovníky a sloupce zaměstnání. Úkolem je pak přiřadit pracovníky co největšímu počtu pracovních míst (na která jsou kvalifikovaní), každý pracovník může vykonávat jen jednu pozici. V našem případě je použitá metoda minimalizační (např. minimalizace celkových nákladů).

Popis jádra metody (pro čtvercovou matici):

Vstupní matice se skládá z prvků, prvek v i-tém řádku a j-tém sloupci značí např. náklady (cenu práce) j-tého úkolu i-tým pracovníkem.

Algoritmus 3: Maďarská metoda

Vstup: Matice hodnot (např. nejkratší vzdálenosti)
Výstup: Optimální párování této matice

1 Od každého prvku odečti nejmenší prvek příslušejícího řádku;
2 Od každého prvku odečti nejmenší prvek příslušejícího sloupce;
3 Označme všechny nuly v matici použitím některého minimálního počtu horizontálních a vertikálních čar;
4 **Test optimality**: jestliže minimální počet označených čar je n, pak optimální přiřazení je možné. Když je čar méně než n, nenašli jsme optimální přiřazení a pokračujeme dalším krokem;
5 Urči nejmenší hodnotu, která není pokryta žádnou čarou. Odečti tuto hodnotu od všech nepokrytých řádků, a pak ji přičti ke všem pokrytým sloupcům. Vrať se na krok 3.

3.2.2 Munkresova modifikace maďarské metody

James Munkres rozvedl do detailu původní maďarskou metodu [19]. Přidal algoritmické řešení hledání minimálního počtu čar, které pokrývají všechny nuly a maximální množiny nezávislých nul. Viz algoritmus 4.

Algoritmus 4: Munkresova maďarská metoda

Vstup: Matice hodnot
Výstup: Optimální párování v matici A

```
 1 for each row r of A do
 2 |  r ← r− minimum element in row;
 3 end
 4 for each row c of A do
 5 |  c ← c− minimum element in column;
 6 end
 7 while there exists a zero Z with no starred zero in its row and column do
 8 |  Star Z;
 9 end
10 lines ← 0;
11 for every column c of A with a starred zero do
12 |  Cover c;
13 |  lines ← lines + 1;
14 end
15 repeat
16 |  while there exists an uncovered zero Z do
17 |  |  Prime Z;
   |  |  /* označíme tuto nulu                                          */
18 |  |  if there exists a starred zero Z* in the row r of Z then
19 |  |  |  Cover row r of Z*;
20 |  |  |  Uncover column c of Z*;
21 |  |  end
22 |  |  else
23 |  |  |  Unprime Z;
   |  |  |  /* zrušíme označení                                         */
24 |  |  |  Star Z;
25 |  |  |  while there exists another already starred zero Z* in the column of Z do
26 |  |  |  |  Unstar Z*;
27 |  |  |  |  Z ← Primed zero in the row of Z*;
28 |  |  |  |  Unprime Z and star Z;
29 |  |  |  end
30 |  |  |  lines ← lines + 1;
31 |  |  |  Erase previous covering;
32 |  |  |  for every column c of A with a starred zero do
33 |  |  |  |  Cover c;
34 |  |  |  end
35 |  |  end
36 |  end
37 |  if lines < number of columns then
38 |  |  Find h = minimum uncovered element of A;
39 |  |  for each covered row r of A do
40 |  |  |  r ← r + h;
41 |  |  end
42 |  |  for each uncovered column c of A do
43 |  |  |  c ← c − h;
44 |  |  end
45 |  end
46 until lines = number of columns;
```

Důkaz korektnosti lze najít v [19].

Složitost: původní Munkresův algoritmus má složitost $O(n^4)$. V článku [20] je prezentováno vylepšení algoritmu na $O(n^3)$. Viz také dále v kapitole implementace 4.2.10.

3.3 Toky v sítích

Přiřazovací problém lze převést na problém hledání maximálního toku v sítích (angl. Network Flow) ([16], [14, kap. 14]), konkrétně na nalezení maximálního toku s minimální cenou v bipartitním ohodnoceném grafu (angl. Max-Flow Min-Cost in Bipartite Weighted Graph).

3.3.1 Teorie

V této kapitole jsou použity definice z [3].

Definice 18. Sítí *nazveme čtveřici* (G, z, s, c)*, kde* $G = (V, E)$ *je orientovaný graf,* z *a* s *dva různé vrcholy grafu* G*(říkáme jim zdroj a stok) a kapacita* $c : E \to \mathbb{R}_0^+$ *splňující*

- *pro každou hranu* $e \in E$ *platí* $0 \geq f(e) \geq c(e)$*,*
- *pro každý vrchol* $u \in V$ *mimo zdroj a stok platí* $\sum_{(x,u)\in E} f(x,u) = \sum_{(u,y)\in E} f(u,y)$*.*

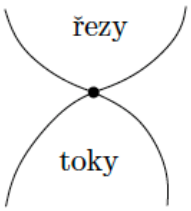
Definice 19. Velikost toku *je* $w(f) = \sum_{(z,x)\in E} f(z,x) - \sum_{(x,z)\in E} f(x,z)$*.*

Definice 20. Řezem *mezi zdrojem* z *a stokem* s *v síti* (G, z, s, c) *nazveme množinu hran* $R \supseteq E(G)$ *takovou, že v síti* (G', z, s, c) *neexistuje žádná orientovaná cesta ze zdroje do stoku, kde* $G' = (V(G), E(G) \setminus R)$*.*

Definice 21. Kapacita řezu *je* $c(R) = \sum_{e\in R} c(e)$*.*

Věta 4 (Hlavní věta o tocích)**.** *Pro každou síť se velikost maximálního toku rovná kapacitě minimálního řezu.*

$$\max_{f \text{ tok}} w(f) = \min_{R \text{ řez}} c(R)$$



Obrázek 3.1: Věta o tocích

Zdroj: [3]

Důkaz. Důkaz této věty lze najít v [3, kap. 8, s. 13-15,17]. □

Definice 22. Cesta *v síti je* nasycená *(vzhledem k danému toku f), pokud pro nějakou hranu $e_i = (v_{i-1}, v_i)$ orientovanou po směru je $f(e_i) = c(e_i)$ nebo pro nějakou hranu $e_i = (v_i, v_{i-1})$ orientovanou proti je $f(e_i) = 0$.*
Cestě, která není nasycená, budeme říkat nenasycená.
Nenasycené cestě ze zdroje do stoku také často říkáme (hlavně v popisu algoritmů) zlepšující cesta.
Tok *nazveme* nasycený*, když každá cesta ze zdroje do stoku je nasycená.* Nasycená cesta *je tedy taková, v níž se tok nedá zvětšit.*

Věta 5. *Tok je maximální, právě když je nasycený. Pro každý maximální tok f existuje řez R takový, že $w(f) = c(R)$.*

Důkaz. Důkaz této věty lze najít v [3, kap. 8, s. 15]. □

3.3.2 Fordův-Fulkersonův algoritmus

Fordův-Fulkersonův algoritmus [14] zkoumá, jaký maximální tok může danou sítí procházet. Hledá zlepšující cestu a vylepší ji. To se opakuje, dokud nějaká nenasycená cesta existuje. Výsledkem je maximální tok.

Algoritmus 5: Fordův-Fulkersonův algoritmus

Vstup: Graf G s kapacitou toku c, zdrojem s a stokem t
Výstup: Maximální tok f z s do t

```
1  for each edge (u,v) ∈ G do
2  |  f(u,v) = 0
3  end
4  while there exists path p from s to t in current network G_f do
5  |  c_f(p) = min{c_f(u,v) : (u,v) is in p};
6  |  for each edge (u,v) in p do
7  |  |  if (u,v) ∈ G then
8  |  |  |  f(u,v) = f(u,v) + c_f(p);
9  |  |  end
10 |  |  else
11 |  |  |  f(v,u) = f(v,u) − c_f(p);
12 |  |  end
13 |  end
14 end
```

V případě, kdy se zlepšující cesta hledá pomocí algoritmu hledání do šířky (BFS), nazývá se algoritmus **Edmondsův-Karpův** [21].

Složitost [22]: Edmondsův-Karpův algoritmus má složitost $O(|V||E|^2)$.

3.3.3 Párování v bipartitním ohodnoceném grafu

Následný algoritmus je modifikací algoritmu Hopcroftova-Karpova [23]. Hledá v bipartitním ohodnoceném grafu maximální párování s nejmenší cenou.

Nejprve se vytvoří potřebný graf, ve kterém se hledá maximální tok s minimální cenou [24, 25]. Nechť je bipartitní graf $G = (X \cup Y, E)$, kde X je partita vrcholů z D^- a Y je partita z D^+. Všechny hrany E mají orientaci z X do Y. Ohodnocení hran v G odpovídá délce nejkratší cesty mezi danými vrcholy. Tento graf G je rozšířen na síť přidáním vrcholů s (zdroj) a t (stok). Zdroj se spojí hranami s vrcholy v partitě X se směrem od zdroje. Stok se spojí hranami s vrcholy v partitě Y směrem do stoku. Těmto nově přidaným hranám je dáno ohodnocení 0. Takto sestrojený graf je vstupem do následujícího algoritmu.

Algoritmus 6: Algoritmus párování v bipartitním ohodnoceném grafu

Vstup: Bipartitní grap G s kapacitou toku c, zdrojem s a stokem t
Výstup: Optimální párování M

```
1  M ← ∅;
2  while there exists path p from s to t in residual network G_f do
3  |  find path p with minimal weight;
4  |  delete first and last edge of path p from G_f ();
5  |  for each edge (u, v) ∉ p do
6  |  |  c(u, v) ← c(u, v) + d(u) − d(v);
   |  |  /* d(v) je vypočtená délka nejkratší cesty z s do v
   |  |     */
7  |  end
8  |  for All remaining edges (u, v) ∈ p do
9  |  |  (u, v) ← (v, u)// edges are inverted
10 |  |  c(v, u) ← 0;
11 |  end
   |  /* update with edges in p with direction from Y to X  */
12 |  Update matching M;
13 end
```

Důkaz korektnosti lze najít v [24], [25].

Složitost [25]: tento algoritmus má za použití Dijkstrova algoritmu (verze s fibonacciho haldou) složitost $O(|V|^2 \log |V| + |V||E|)$.

3.4 Hledání eulerovského tahu

K nalezení eulerovského tahu využívám Hierholzerův algoritmus.

3.4.1 Hierholzerův algoritmus

Hierholzerův algoritmus [3] přijímá na vstupu eulerovský graf. Začíná z libovolného vrcholu nalezením uzavřeného tahu. Ten se vždy může najít díky sudým stupňům všech vrcholů (případně u orientovaných grafů díky vyváženosti všech vrcholů). V další části se pak hledá vrchol, který stále má dosud neprošlou hranu. Pokud se takový vrchol najde, opakuje se z tohoto vrcholu první část algoritmu. Nalezený tah z tohoto vrcholu se připojí do aktuálně kontrolovaného tahu k danému vrcholu. Takto se pokračuje, dokud nějaký vrchol v aktuálním tahu má neprošlé hrany.

Algoritmus 7: Hierholzerův algoritmus

Vstup: Eulerovský graf G
Výstup: Eulerovký cyklus

```
1  u is any vertex of G;
2  HEAD and TAIL are stacks;
3  HEAD ← u TAIL ← ∅;
4  while HEAD ≠ ∅ do
5  |  if deg⁻(u) > 0 then
6  |  |  let (u, v) be unvisited edge;
7  |  |  HEAD.push(v);
8  |  |  delete edge (u, v);
9  |  |  decrease deg⁻(u);
10 |  |  u ← v;
11 |  end
12 |  else
13 |  |  add u to TAIL;
14 |  |  u ← HEAD.top();
15 |  |  HEAD.pop();
16 |  end
17 end
18 TAIL is Euler tour;
```

Obecný důkaz korektnosti Hierholzerova algoritmu lze najít v [3, kap. 20, s. 48-49].

Složitost [3, kap. 20, s. 49]: každou hranu projde algoritmus právě jednou, tedy pokud dokážeme hrany mazat v konstantním čase $O(1)$ jeho složitost je $O(m)$, kde m je počet hran. Mažeme totiž vždy poslední hranu v poli.

3.5 Modifikovaný Tarjanův algoritmus

Definice 23. Silně souvislá komponenta *je maximální množina vrcholů orientovaného grafu taková, že mezi každými dvěma vrcholy existuje sled.*

Tarjanův algoritmus vyhledává v orientovaném grafu silně souvislé komponenty [14, kap. 5.10].

Algoritmus je založen na prohledávání do hloubky (DFS). Vrcholy při procházení indexujeme podle pořadí nalezení. Při návratu z rekurze každému vrcholu přiřadíme nejnižší index, na jaký lze dosáhnout. Po skončení algoritmu všechny vrcholy, které mají stejný index, patří do stejné silně souvislé komponenty.

Na vytvoření silně souvislého grafu se využívá modifikovaný Tarjanův algoritmus [26]. Při detekci nové silně souvislé komponenty se přidá hrana, vedoucí z této komponenty do vrcholu s indexem nižším než je index zkoumaného vrcholu. Tím se tato komponenta silně souvisle připojí ke zbytku grafu.

Složitost: Takto modifikovaný Tarjanův algoritmus má složitost $O(|E|)$. U mé implementace, kde vstupní graf algoritmu má $(|V|-1)$ hran, je složitost $O(|V|)$ (viz článek [26]).

Kapitola 4

Implementace DCPP

Tato kapitola popisuje moji implementaci řešení DCPP a výběr použitých technologií. Odlišuji sekvenční a paralelní provedení.

4.1 Použité prostředky

Popisuji využívaný programovací jazyk implementace a prostředky používané při paralelizaci řešení.

Programovací jazyk implementace

Jako programovací jazyk jsem zvolil jazyk C++. Je to jazyk velmi rozšířený a mám v něm největší zkušenosti. Umožňuje nízko úrovňovou optimalizaci, která je vhodná pro tyto typy úloh. Pro tento jazyk existuje také mnoho vhodných knihoven, které využívám pro ulehčení implementace.

Knihovny Boost

Boost jsou volně přístupné knihovny, které podporují celou sadu úloh a struktur v programovacím jazyce C++.

Ve třídách *CDijkstra* a *CBipartiteMatching* používám strukturu *fibonacci_heap* jako prioritní frontu při řešení Dijkstrova algoritmu. Dosahuji tak lepší časové složitosti než bez *fibonacci_heap*.

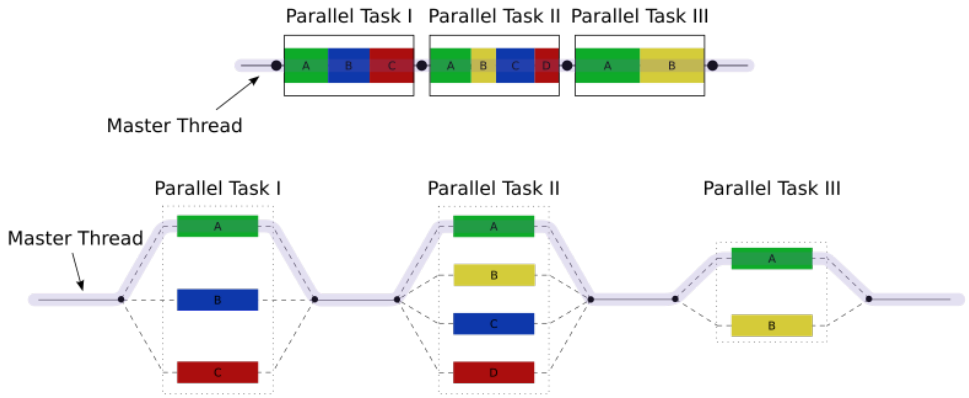
Paralelní provedení

Jednou z hlavních náplní mé implementace je využití paralelizace [27]. To znamená možnost využití více vláken najednou na vhodných místech algoritmů. Důsledkem je (ideálně) výrazné zrychlení výsledného algoritmu.

Na paralelní provedení využívám knihovnu OpenMP [28], [29]. S OpenMP se velmi dobře pracuje a výsledky jsou přehledné. OpenMP je vysoko-úrovňové API pro programování vícevláknových aplikací nad virtuálně sdílenou pamětí.

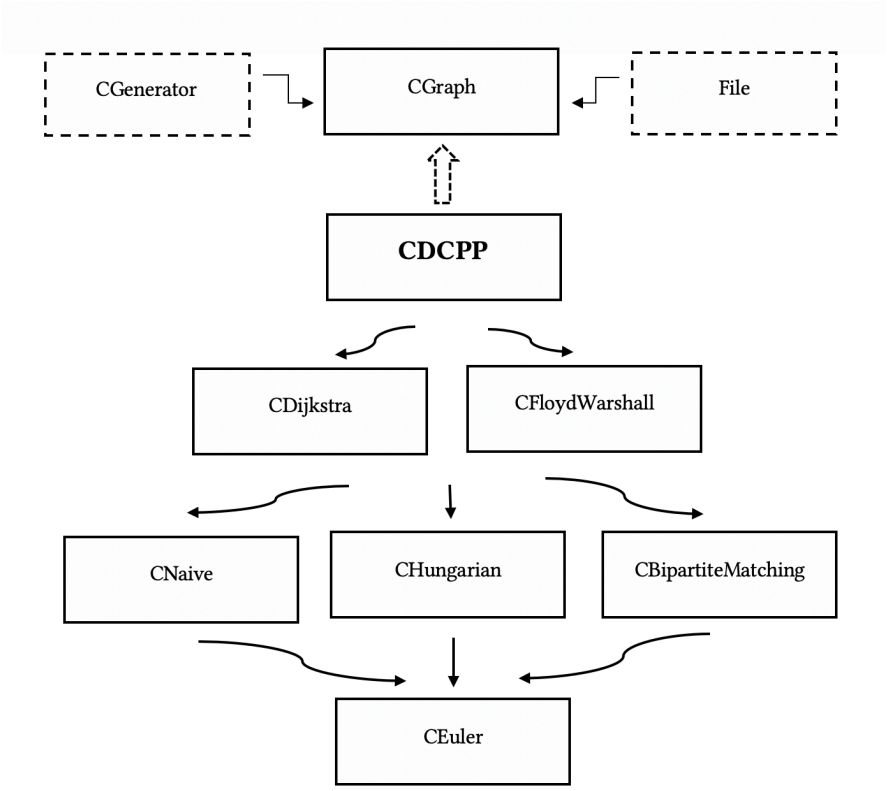
V určitých částech programu jsou pomocí fork-join vytvářena (viz obr. 4.1), prováděna a ukončována vlákna. V ostatních částech je pouze hlavní vlákno.

OpenMP API se skládá z direktiv (např. *parallel*), proměnných (např. *OMP_NUM_TREADS*) a operací (např. *omp_set_num_threads*).



Obrázek 4.1: Fork-join schéma

Zdroj:`https://en.wikipedia.org/wiki/Fork%E2%80%93join_model`



Obrázek 4.2: Schéma řešení DCPP

4.2 Sekvenční struktura řešení

Rozebírám zde jednotlivé části sekvenčního provedení implementace. Každá podkapitola se zabývá jednou třídou, která reprezentuje jednotlivý algoritmus či komponentu. U každé třídy popisuji zásadní metody, atributy a rozhodnutí při implementaci třídy. V některých případech uvádím také složitost výpočtu v dané třídě.

Na obrázku 4.2 je znázorněno hrubé schéma mého řešení DCPP a možné průchody (šipky) jednotlivými komponentami řešení.

4.2.1 Class CDCPP

Seznam atributů:

m_graph Ukazatel na instanci třídy *CGraph*, která drží aktuální graf,

m_path informace o tom, který algoritmus na hledání nejkratší cesty se má při výpočtu použít,

m_assign informace o tom, který algoritmus na přiřazovací problém se má při výpočtu použít.

Seznam metod:

Calculate Spustí výpočet a vrátí seznam vrcholů tvořící eulerovský cyklus a délku tohoto cyklu,

SetAlgorithms nastaví jaké třídy se mají při výpočtu využívat,

PrintResult vypíše výsledek z metody *Calculate* na terminál.

Hlavní třída řídící celý algoritmus DCPP. Konstruktor přijme jako argument vstupní graf, ze kterého se vytvoří pracovní hluboká kopie. Dále pomocí metody *SetAlgoritmus* se nastaví jaké třídy využívá při svém výpočtu. Samotný výpočet se spustí metodou *Calculate*. V případě zadání eulerovského grafu se hned zavolá metoda *Calculate* třídy *CEuler*, který výsledný eulerovský cyklus najde.

V jiném případě se nejdříve použije třída *CPathFinding*. Tato třída vypočítá nejkratší vzdálenosti potřebných vrcholů a uloží je do třídy *CInputAssignment*.

Posléze se třída *CInputAssignment* pošle instanci třídy *CAssignment*, která ve struktuře držící třídou *CInputAssignment* najde optimální párování.

Pomocí metody *FindPathFromIndex* třídy *CPathFinding* získáme výčet hran nejkratší cesty mezi vrcholy odpovídající indexům nalezeného optimálního párování. Tyto hrany se následně přidají do grafu.

Po těchto úpravách již máme požadovaný eulerovský graf a tedy v něm můžeme pomocí třídy *CEuler* najít výsledný eulerovský cyklus.

Metoda *Calculate* vrátí uspořádaný seznam hran tvořící eulerovský cyklus a celkovou délku tohoto cyklu. Délka se získá jako součet ohodnocení všech hran ve výsledném grafu.

Asymptotická složitost výpočtu v této třídě závisí na složitosti relevantních tříd, které mají složitost nejhorší. Pokuď ignoruji třídu *CNaive*, asymptotická složitost je $O(|E|^3)$, jelikož třídy řešící přiřazovací problém mají horší asymptotickou složitost než třídy řešící hledání nejkratších cest či hledání eulerova cyklu.

4.2.2 Class CInputAssignment

Seznam atributů:

m_size Na základě podtřídy drží velikost matice *m_matrix* či počet vrcholů v grafu *m_graph*,

m_matrix matice nejkratších vzdáleností,

m_graph *m_matrix* uložená jako bipartitní graf a vrcholy zdroj a stok,

Seznam metod:

AddElement Přidá prvek do připravené struktury.

Tato třída má dvě podtřídy *CHoldMatrix* a *CHoldNetwork*. Když třída *CDCPP* používá k výpočtu třídu *CHungarian* či *CNaive* a vstupem je matice, využije se podtřída *CHoldMatrix*. Při použití třídy *CBipartiteMatching* a grafu jako vstup se využije podtřída *CHoldNetwork*.

4.2.3 Class CGraph

Seznam atributů:

m_numVertex Udává počet vrcholů,

m_totalWeight aktuální součet všech ohodnocení v grafu,

m_Degree pole držící údaje o rozdílu vstupního a výstupního stupně každého vrcholu,

m_Graph pole struktur *vector* držící všechny hrany v grafu.

Seznam metod:

NewGraph Vytvoří nový graf ze souboru,

AddEdge přidá do grafu určitou hranu,

PrintGraph vypíše graf.

Třída *CGraph* drží informace o grafu. Hrana je reprezentována pomocí struktury *struct Edge* s atributy *name* a *weight*. Hrany jednotlivých vrcholů jsou pak uloženy jako pole pomocí struktur *vector*.

Vzhledem k tomu že nevíme, zda bude zadaný graf řídký, uložení grafu jako matice sousednosti by nemuselo být časově ani datově výhodné. *Vector* nám přináší konstantní složitost pro vkládání hran i pro přístup k prvkům.

Nový graf vznikne dvěma způsoby. První je načtení ze souboru pomocí metody *NewGraph*. Druhý je nastavení počtu vrcholů metodou *SetNumVertex* a postupné přidávání hran metodou *AddEdge*. Tento druhý způsob využívá třída *CGenerator* při generování grafu.

Při zadání grafu ze souboru je v této třídě prováděna kontrola správnosti zadaného grafu. Tedy že zadaný graf je silně souvislý s kladně ohodnocenými hranami.

4.2.4 Class CGenerator

Seznam atributů:

m_numVertex Počet vrcholů ve výsledném grafu,

m_numEdge počet hran ve výsledném grafu,

m_matrix matice sousednosti aktuálního grafu,

m_maxWeight maximální ohodnocení hran.

Seznam metod:

SetGenerator Nastaví se počet vrcholů a hran následně vygenerovaného grafu,

GetGraph vytvoří se náhodný silně souvislý graf s parametry zadanými v SetGenerator,

CreateSpanningTree vytvoří se náhodný strom,

CreateStrongCon ke stromu ze *CreateSpanningTree* se přidají hrany tak, aby graf tvořil silně souvislý graf,

CreatingEdge do silně souvislého grafu se přidá požadovaný počet náhodných hran.

Jedna z možností získání grafu pro tento program je využití třídy *CGenerator*. *CGenerator* je schopný vytvořit jakýkoli silně souvislý orientovaný graf až na izomorfismy. Počet vrcholů a hran je zadán uživatelem.

Nejprve se vytvoří kostra grafu postupným přidáváním vrcholů (pomocí hrany) k již existující množině vrcholů.

Na vytvoření silně souvislého grafu se využívá modifikovaný Tarjanův algoritmus.

V poslední fázi se přidávají hrany až do počtu zadaného uživatelem. Tyto hrany jsou v náhodně vybírány ze struktury *vector* obsahující dosud nevyužité hrany. V případě, že je tento počet hran nižší než počet hran současného grafu, nepřidávají se žádné další hrany. Stejně když je počet hran zadaný větší než možný, vytvoří se úplný graf.

Vytvářený graf ukládám postupně do instance třídy *CGraph* pomocí metody *AddEdge*. Zároveň se tvoří matice sousednosti pro přehled, které hrany již graf obsahuje. Tato informace je potřebná při přidávání hran v poslední fázi.

4.2.5 Class CPathFinding

Seznam atributů:

m_graph Ukazatel na instanci třídy *CGraph* držící aktuální graf,

m_distMatrix matice délek nejkratších cest mezi nevyváženými vrcholy,

m_pathMatrix matice držící údaje o vrcholech v cestách v *m_distMatrix*,

m_outputIndexRow vektor vrcholů ze kterých se počítá nejkratší vzdálenosti, tj. se zápornou deltou,

m_outputIndexCol vektor vrcholů do kterých se počítá nejkratší vzdálenosti, tj. s kladnou deltou.

Seznam metod:

FindPathFromIndex Přiřadí k dvěma indexům odpovídající vrcholy z grafu a nejkratší cestu mezi nimi,

FillOutputIndex naplní se vektory *m_oddIndexRow* a *m_oddIndexCol*.

Abstraktní třída jejíž podtřídy řeší problém nalezení nejkratších cest mezi nevyváženými vrcholy.

V metodě *FillOutputIndex* se vloží do atributů *m_outputIndexRow* a *m_outputIndexCol* názvy vrcholů s nenulovou δ. Ve vektoru *m_outputIndexRow* se každý vrchol v se zápornou δ opakuje $|\delta(v)|$-krát. To samé v *m_outputIndexCol* s vrcholy s kladnou δ.

Metoda *FindPathFromIndex* vrátí cestu a její délku z vrcholu odpovídající indexu z *m_outputIndexRow* do vrcholu odpovídající indexu z *m_outputIndexCol*.

4.2.6 Subclass CDijkstra

Seznam atributů:

m_startPoints Pole vrcholů ze kterých se vypočítávají nejkratší vzdálenosti (vrcholy se zápornou δ).

Seznam metod:

Calculate Spustí výpočet a výsledkem je matice nejkratších vzdáleností mezi vrcholy se zápornou δ a kladnou δ,

CreateOutputStructure posílá naměřené hodnoty do třídy *CInputAssignment*.

Tato třída řeší problém nalezení nejkratších cest za pomoci Dijkstrova algoritmu.

Nejdříve se do atributu *m_startPoints* vloží seznam vrcholů, ze kterých se hledají nejkratší cesty (tj. vrcholy množiny D^-). *Bool* pole *endVertices* drží informace o vrcholech množiny D^+ a slouží k dřívějšímu ukončení Dijkstrova algoritmu.

Následně se pro každý vrchol z *m_startPoints* spouští Dijkstrův algoritmus. Algoritmus se ukončuje, jakmile se navštívily všechny vrcholy z pole *endVertices*. V samotném algoritmu využívám jako prioritní frontu fibonacciho haldu pro zlepšení asymptotické složitosti.

V konečném kroku se metodou *CreateOutputStructure* za využití metody *AddElement* třídy *CInputAssignment* vytvoří pomocí atributů *m_outputIndexRow*, *m_outputIndexCol* a naměřených dat vstupní struktura pro následující krok, kterým je přiřazovací problém.

Inicializace *m_startPoints* a *endVertices* má časovou složitost $O(|V|)$. Dijkstrův algoritmus za použití fibonacciho haldy má časovou složitost $O(|E| + |V|log|V|)$. Dijkstrův algoritmus je spuštěn $|D^-|$-krát, což je $O(|V|)$. Vytvoření výsledné struktury z naměřených vzdáleností má složitost $O(|\sum_{v \in D^-} \delta(v)|^2) = O(|E|^2)$.

Celkově má tedy tento výpočet složitost $O(|V|+|D^-||E|+|D^-||V|log|V|+|E|^2) = O(|V|^2log|V|+|E|^2)$. Pro nejhorší možnost je $O(|V|^4)$ v případě, kdy je $\sum_{v \in D^-} \delta(v)$ kvadratická vzhledem k počtu vrcholů. Avšak pro řídké grafy může být blíže k $O(|V|^2log|V|)$. Navíc při menší početnosti vrcholů v D^- je reálná časová složitost výrazně menší.

4.2.7 Subclass CFloydWarshall

Seznam metod:

Calculate Spustí výpočet a výsledkem je matice nejkratších vzdáleností mezi vrcholy se zápornou δ a kladnou δ,

Algorithm samotný výpočet Floydova-Warshallova algoritmu s pamatováním následujícího vrcholu v každé cestě,

CreateOutputStructure posílá naměřené hodnoty do třídy *CInputAssignment*.

Tato třída řeší problém nalezení nejkratších cest za pomoci Floydova-Warshallova algoritmu.

Tento algoritmus musí počítat nejkratší vzdálenosti mezi všemi vrcholy. Nestačí jenom mezi nevyváženými vrcholy (vrcholy s nenulovou δ), protože cesty mohou obsahovat i ostatní vrcholy a princip tohoto algoritmu vyžaduje i vzdálenosti mezi těmito ostatními vrcholy.

Navíc do atributu *m_pathMatrix* ukládám informaci na pozdější rekonstrukci nalezené cesty. Na konci výpočtu každá položka ukazuje na následovníka v dané cestě.

V první fázi výpočtu se připraví matice vzdáleností *m_distMatrix* a již zmíněná matice *m_pathMatrix*. Poté proběhne samotný Floydův-Warshallův algoritmus.

V konečném kroku se metodou *CreateOutputStructure* vyplní struktura ve třídě *CInputAssignment* naměřenými hodnotami za pomoci atributů *m_oddIndexRow* a *m_oddIndexCol*.

První fáze výpočtu (inicializace) má asymptotickou složitost $O(|V|^2)$. Samotný algoritmus má složitost $O(|V|^3)$. Poslední krok vytvoření výsledné struktury z naměřených hodnot má složitost stejně jako u třídy *CDijkstra* $O(|\sum_{v \in D^-} \delta(v)|^2) = O(|E|^2)$. Dohromady má tento výpočet složitost $O(|V|^2 + |V|^3 + |E|^2) = O(|V|^3 + |E|^2)$.

4.2.8 Class CAssignment

Abstraktní třída jejíž podtřídy řeší přiřazovací problém. Jedná se konkrétně o podtřídy *CNaive*, *CHungarian*, *CBipartiteMatching*.

4.2.9 Subclass CNaive

Seznam atributů:

m_matrix Vstupní matice nejkratších vzdáleností,

m_bestScore momentální nejlepší součet přiřazení,

m_bestPairs momentální nejlepší párování.

Seznam metod:

Calculate Spouští samotný naivní rekurzivní algoritmus.

Třída *CNaive* řeší přiřazovací problém naivním rekurzivním způsobem. Jedná se stromovou rekurzi. Tudíž strukturu volání lze brát jako rozvětvený strom.

Algoritmus postupně zkouší všechny možné permutace. Což dává asymptotickou složitost $O(n!)$. V každém volání se kopíruje pole. To trvá $O(n)$. Dohromady nám tento algoritmus dává $O(n * n!)$

Ukončující podmínka má dvě části. První je, když se naplní celé pole. V tomto případě se zkontroluje globální současné řešení, a jestliže je lokální řešení lepší, tak se globální přepíše lokálním. Druhá část podmínky je, když je lokální řešení již v průběhu výpočtu horší. Tak se současné řešení ukončí.

4.2.10 Subclass CHungarian

Seznam atributů:

m_matrix Vstupní matice nejkratších vzdáleností,

m_mask matice držící informaci o typu prvku na dané pozici,

m_col pole obsahující informaci, zdali je daný sloupec kryt,

m_row pole obsahující informaci, zdali je daný řádek kryt,

m_min pole nejmenších hodnot v řádcích matice,

m_ptr pole s ukazateli na nejmenší hodnoty v řádcích matice,

m_rowAdjust pole úprav řádků matice z metody *AdjustValues*,

m_colAdjust pole úprav slouců matice z metody *AdjustValues*.

Seznam metod:

Calculate Spouští a řídí posloupnost kroků algoritmu,

SubtractMin odčítá od řádků a sloupců minimum odpovídajícího řádku či sloupce,

SetMask označuje dosud nepokryté nuly,

CountStarredZeros součet počtu označených nul,

FindZero obsahuje *while* cyklus, který probíhá, dokud je v matici nepokrytá nula,

FoundNoncovZero výpočet v případě nalezení nepokryté nuly,

AugmentingPath provádí se, když na řádku nepokryté nuly není označená nula,

AdjustValues do atributů *m_rowAdjust* a *m_colAdjust* se ukládají pozdější modifikace matice.

Tato třída implementuje Munkresovu maďarskou metodu (viz kap. 3.2.2) s modifikací dávající celkovou asymptotickou složitost $O(|\sum_{v \in D^-} \delta(v)|^3) = O(|E|^3)$, kde $|E|$ je počet hran ve vstupním grafu výpočtu DCPP.

Výpočet přijímá matici nejkratších vzdáleností z předchozího výpočtu a vrací *list* optimálního párování.

Původní Mukresova implementace má asymtotickou složitost $O(n^4)$. Ta je způsobena výpočtem modifikace matice probíhajícího v metodě *AdjustValues*. Tento výpočet má složitost $O(n^2)$ a může proběhnout nejvíce n^2-krát. Jak je vidět z článku [20], zrychlení na $O(n^3)$ lze dosáhnout uložením potřebných úprav do atributů *m_rowAdjust* a *m_colAdjust*, což lze provést v lineárním čase. Aktuální úpravy do matice pak provádět jenom při nalezení nové „star" nuly, což se stane nejvíce n-krát.

Pro hledání nepokrytých nul v čase $O(n)$ nám slouží pole *m_min* a *m_ptr*.

4.2.11 Subclass CBipartiteMatching

Seznam atributů:

m_graph Vstupní bipartitní graf.

Seznam metod:

Calculate Spouští samotný výpočet,

Recalibrating přepočítání hrany, aby nevznikly žádné záporné,

UpdateMatching aktualizuje aktuální párování po nalezení každé zlepšující cesty.

Třída *CBipartiteMatching* implementuje algoritmus hledání maximálního toku s minimální cenou v bipartitním grafu. (viz kap. 3.3.3)

Metoda *Calculate* dostane bipartitní graf $G = (X \cup Y, F)$ s hranami reprezentující nejkratší vzdálenosti mezi danými vrcholy a vrací strukturu *list* obsahující optimálního párování.

Nejprve se přidá do grafu zdroj a stok. Přidají se hrany vedené ze zdroje do partity X. A dále hrany vedené do stoku ze všech vrcholů partity Y. Těmto nově přidaným hranám se dá ohodnocení nula.

V následujících odstavcích je $|E|$ počet hran a $|V|$ počet vrcholů vstupního grafu výpočtu DCPP.

Počet vrcholů ve vstupním bipartitním grafu této třídy je $\sum_{v \in D^-} \delta(v) * 2$. A to se rovná $O(|E|)$.

Samotný výpočet začne hledáním zlepšující cesty Dijkstrovým algoritmem s využitím fibonacciho haldy. To zajistí nalezení cesty s nejmenší cenou. Tato část má pro náš úplný bipartitní graf složitost $O(|E|^2 + |E|log|E|)$.

Dále se přepočítává ceny hran $c(u, v)$ pomocí vzorečku $c(u, v) = c(u, v) + d(u) - d(v)$, kde $d(v)$ je nejkratší vzdálenost ze zdroje do vrcholu v, která je vypočítaná Dijkstrovým algoritmem v předchozím kroku. Toto přepočítání nám zajistí absenci záporných cen. Tato část má složitost $O(|E|^2)$, jelikož se prochází každá hrana právě jednou.

V konečném kroku musíme invertovat jednotlivé hrany ze zlepšující cesty, vymazat využité hrany ze zdroje a do stoku, aktualizovat momentální optimální párování. To vše lze zvládnout v $O(|E|)$. V Dijkstrově algoritmu se pamatuje nejen předchozí vrcholy ve výsledné cestě, ale i odkazy na hrany, ze kterých se cestovalo z předchůdce. To umožní najít hrany, které musíme invertovat, v konstantním čase. Mazání hran provádíme ve struktuře *vector* nahrazením dané hrany posledním prvkem, který následně vymažeme. To lze také zvládnout v konstantním čase.

Tyto kroky výpočtu se provádějí ($\sum_{v \in D^-} \delta(v)$)-krát.

Celková složitost algoritmu je pak $O(|E|^2 log|E| + |E|^3 + |E|^3 + |E|^2) = O(|E|^3)$.

4.2.12 Class CEuler

Seznam metod:

Calculate Spustí Hierholzerův algoritmus na hledání eulerovského cyklu v zadaném již eulerovském grafu a vrátí *list* výsledných vrcholů.

Třída hledá eulerovský cyklus pomocí Hierholzerova algoritmu (viz kap. 3.4.1). Kvůli předchozím výpočtům v programu je přijatý graf vždy eulerovský.

Každá hrana se navštíví právě jednou a hned po navštívení se maže. Vzhledem k tomu, že hrany jsou uloženy ve struktuře *vector* a vždy se přistupuje k poslednímu prvku, jednotlivé mazání trvá konstantní čas.

Celkový výpočet má tedy asymptotickou složitost $O(|E|)$.

4.3 Paralelní struktura řešení

V této kapitole rozebírám provedení algoritmů, které jsem paralelizoval.

Jsou to algoritmy v následujících třídách:

- *CDijkstra*
- *CFloydWarshall*
- *CHungarian*
- *CBipartiteMatching*

Třídu *CEuler* jsem neparalelizoval vzhledem k tomu, že má složitost $O(|E|)$ a v konečném výsledku bude časově zanedbatelná. Dále jsem neparalelizoval třídu *CNaive*, protože ji využívám pouze jako ukázku neefektivního algoritmu.

4.3.1 Třída CDijkstra

Ve třídě *CDijkstra* má velkou časovou složitost smyčka, která provádí jednotlivé výpočty Dijkstrova algoritmu. My však nemusíme tyto výpočty provádět sekvenčně. Na tuto smyčku proto využijeme datový paralelismus pomocí direktiv *parallel* a *for*. Jelikož Dijkstrův algoritmus počítáme pouze do některých vrcholů, může výpočet probíhat různě dlouhé doby. Proto využijeme dynamické přidělení iterací pomocí *schedule(dynamic)*.

Bylo by také možné paralelizovat jednotlivé výpočty Dijkstrova algoritmu pomocí paralelizace prohledávání hran zkoumaného vrcholu. To by se však mohlo uplatnilo pouze v případě, kdy máme větší počet vláken než kolikrát počítáme Dijkstrův algoritmus. To však nastane jenom když $|D^-|$ je menší než počet vláken, v takovém případě paralelizace není tak užitečná.

Dále bylo možné paralelizovat konečné vytváření výstupní struktury. Avšak ukázalo se, že pro testované velikosti vstupů je tato paralelizace časově nevýhodná. Vysvětlení může být větší režie paralelizace spojená s využitím *gcc* optimalizace *-O3*.

4.3.2 Třída CFloydWarshall

Ve třídě má společně s konečným vytvářením výstupní struktury největší časovou složitost samotný Floydův-Warshallův algoritmus. Způsobují to tři do sebe vnořené cykly. Zde se tedy nabízí využít datový paralelizmus. Nemůžeme však paralelizovat vnější *for* cyklus, jelikož jednotlivé iterace závisí na předchozí.

Dva vnitřní cykly již však paralelizovat můžeme. Je to umožněno tím, že každé vlákno nahlíží jenom na vlastní řádek matice a na k-tý řádek, který se v momentální iterace nemění. Tudíž nedochází k žádnému konfliktu.

Na paralelizaci tedy použijeme direktivy *parallel* a *for* vložené před druhý cyklus. Jelikož se v cyklech provádí konstantní operace, může využít statické přidělení iterací.

Stejně jako u třídy *CDijkstra* paralelizace výstupní struktury byla nevýhodná.

4.3.3 Třída CHungarian

Jednotlivé kroky algoritmu v třídě *CHungarian* se musejí provádět v sekvenčním pořadí. Můžeme tedy paralelizovat algoritmus pouze v rámci jednotlivých kroků (metod).

Ve všech případech, kde můžeme využít paralelizaci, se jedná o procházení maticí či polem s konstantním výpočtem. Využíváme tedy direktivy *parallel* a *for* se statickým přidělením iterací.

V některých případech však nelze paralelizovat vzhledem k potřebě sekvenčního provedení (např. v metodě *FindZero*, kde v cyklu hledáme nepokryté nuly), tedy v případech kdy výsledek následující iterace závisí na předchozí.

4.3.4 Třída CBipartiteMatching

Ve třídě *CBipartiteMatching* má největší časovou složitost smyčka, která v každé iteraci hledá zlepšující cestu, přepočítává hodnot hran a aktualizuje momentální párování.

Aktualizace párování má pouze lineární časovou složitost. Hledání zlepšující cesty a přepočítávání hodnot hran mají však kvadratickou časovou složitost.

Na přepočítávání hodnot hran lze využít datovou paralelizaci za použití direktiv *parallel* a *for*. Kvůli velkému rozdílu množství hran vedoucí z různých partit, přepočítávání hran se rozdělí na základě z jaké partity hrany vedou. To umožní využití paralelizaci statické přidělení iterací.

Hledání zlepšující cesty je řešeno Dijkstrovým algoritmem. Algoritmus by bylo možné paralelizovat pomocí paralelizace prohledávání hran zkoumaného vrcholu. Ukázalo se však, že tato paralelizace algoritmus značně zpomalila. Důvodem může být velká paralelní režie nutné kritické sekce při aktualizování haldy.

4.4 Použití programu

Výsledkem kompilace zdrojového kódu je program DCPP. Vstupní graf se načítá ze souboru či se vytvoří generátorem s danými parametry. Výstup (tj. řešení) je zobrazen na terminálu.

4.4.1 Vstupní data

Vstupní graf musí být silně souvislý orientovaný s kladným ohodnocením hran. Jsou dvě možnosti zadání grafu.

Může se zadat do souboru. První řádek souboru obsahuje číslo udávající počet vrcholů v grafu. Dále následují řádky s jednotlivými hranami ve formátu <vrchol A> <vrchol B> <ohodnocení> oddělené mezerami. Názvy vrcholů jsou zadávány čísly od 0 do $(n-1)$, kde n je počet vrcholů z prvního řádku. Program kontroluje správnost vstupu (silnou souvislost a kladné ohodnocení hran).

Nebo se může nechat vygenerovat náhodný silně souvislý graf na základě počtu vrcholů a hran. Ty se zadávají jako argumenty při spouštění programu.

4.4.2 Výstupní data

Výsledek je zobrazen v terminálovém okně. Na první řádce je seznam vrcholů tvořící výsledný eulerovský cyklus. Na druhém řádku je celková délka cyklu.

4.4.3 Spuštění programu

Pokud spouštíme program pomocí grafu uloženém v souboru, tak se program spouští s jedním argumentem udávající jméno daného souboru.

./DCPP <soubor s grafem>

V případě použití generátoru grafu se program spouští se dvěma argumenty. První je počet vrcholů a druhý je počet hran. Při zadání hran v jiném než možném rozsahu se zvolí odpovídající maximální či minimální možný počet.

./DCPP <počet vrcholů> <počet hran>

Na zvolení používaných algoritmů v jednotlivých částech programu a nastavení počtu vláken u paralelizovaných částí programu se použije soubor *config*. Na prvním řádku se udává počet vláken. Na druhém číslo odpovídající třídě řešící část programu hledání nejkratších cest (1 - *CFloydWarshall*, 2 - *CDijkstra*). Na třetím řádku číslo odpovídající třídě řešící přiřazovací problém (1 - *CHungarian*, 2 - *CBipartiteMatching*, 3 - *CNaive*).

Výchozí nastavení *config* souboru je použití *CDijkstra* a *CHungarian*, s jedním vláknem.

V programu není naimplementována automatická volba použitých algoritmů. Program je chápán jako testovací, a tedy se předpokládá, že uživatel si sám algoritmy zvolí.

Kapitola 5

Testování

Kapitola se věnuje testování implementace popsané v kapitole 4. Uvádí stručně použitý hardware a software, popisuje metody testování a použité metriky, způsob měření výkonnosti algoritmů, včetně porovnání v rámci optimalizace. Popisuje výsledky testů jednotlivých tříd algoritmů, jejich porovnání, testování kompletní implementace a závěrečné vyhodnocení testování.

5.1 Hardware a software

Testování výkonnosti implementací jednotlivých algoritmů a jejich porovnání bylo prováděno na školním svazku STAR s konfigurací: dva procesory Intel Xeon E5-2630 v4 @ 2.20GHz a 64GB RAM [30].

STAR zpracovává úlohy postupně z fronty úloh, tak je zajištěna nezávislost dané úlohy na jiných procesech. Opakovaná měření tak dávají výsledky s minimálními odchylkami.

Pro kompilaci programů byl použit překladač *g++* ve verzi 4.8.5. s přepínači *-std=c++11 -Wall -pedantic -O3 -fopenmp*.

5.2 Způsob testování

Pro generování vstupních grafů se používá generátor náhodných silně souvislých grafů, třída *CGenerator* (viz kap. 4.2.4). Pro generování náhodných čísel v maticích (pro testování tříd *CHungarian*, *CBipartiteMatching* a *CNaive*) se používá standardní knihovna *random*.

Měření výsledných časů se provádí pomocí knihovny *Chromo* a jejich uložení do textových souborů. Každý prezentovaný naměřený čas je vypočítán jako průměr z deseti dílčích měření.

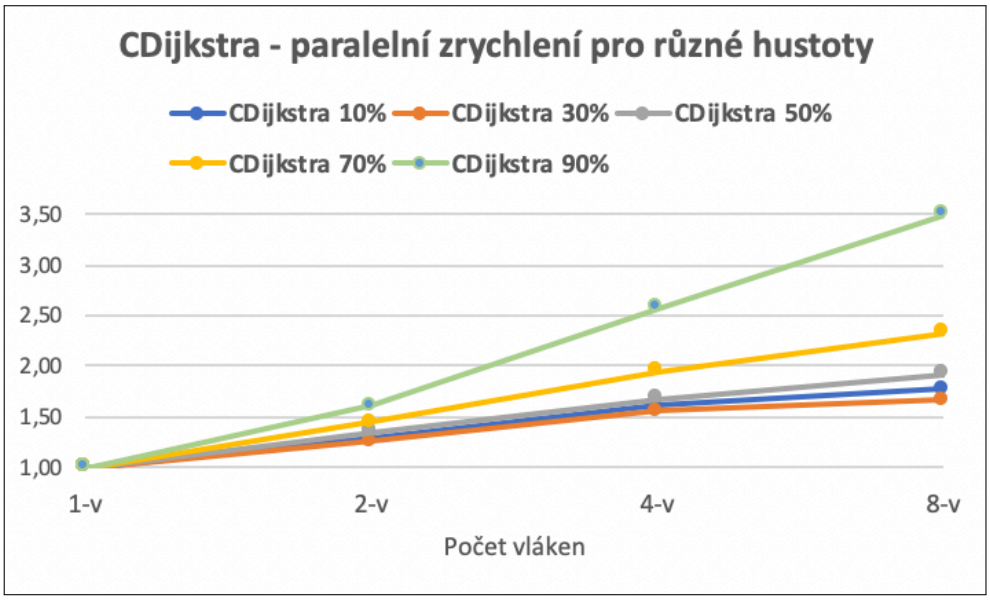
Testování lze rozdělit do čtyř částí:

- testování jednotlivých tříd s porovnáním sekvenčního provedení s paralelním,
- testování se záměrem porovnat třídy řešící stejný problém,
- testování kompletní třídy *CDCPP*,
- testování porovnáním *CDCPP* s existujícím řešením.

5.3 Testování třídy CDijkstra

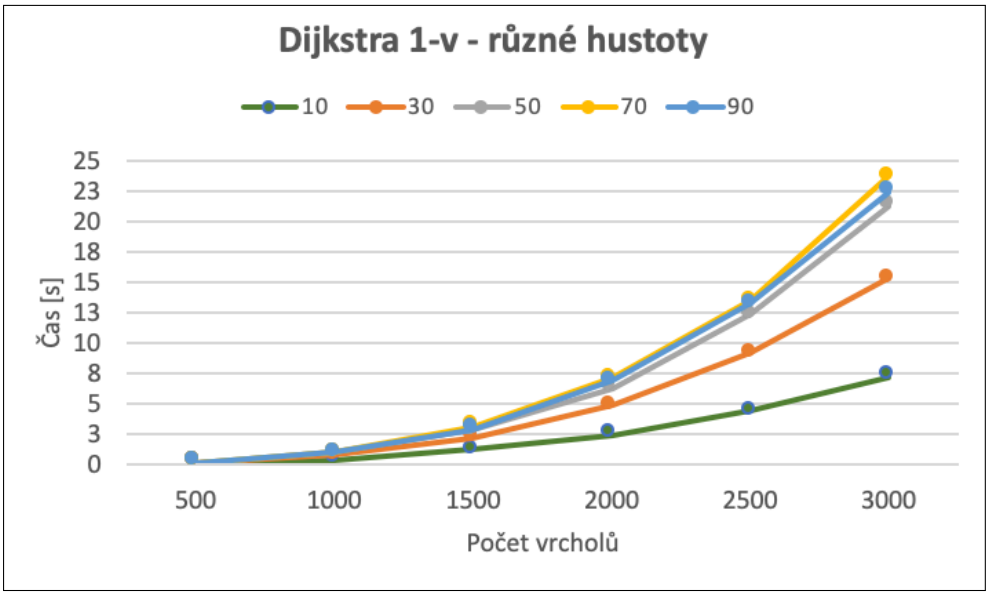
V této třídě Dijkstrův algoritmus probíhá tolikrát, kolik je vrcholů v množině D^- (tzn. vrcholy se zápornou δ)(viz kap. 2.2). Generátor generuje grafy, které mají průměrně polovinu všech vrcholů v množině D^-. Toto omezení by bylo možné ovlivnit v závěrečné fázi generátoru při náhodném přidávání hran tak, aby toto přidávání bylo specifikováno. Lze očekávat, že s rostoucí množinou D^- bude klesat efektivita výpočtu této třídy.

Na grafu 5.1 je znázorněno paralelní zrychlení pro grafy s různou hustotou hran. Hustota hran je podíl počtu hran v grafu k maximálnímu počtu hran.



Obrázek 5.1: Paralelní zrychlení třídy *CDijkstra* s různou hustotou hran

Na grafu 5.2 lze vidět jednovláknové testy s různými hustotami hran vstupního grafu.



Obrázek 5.2: Testování třídy *CDijkstra* s různou hustotou hran

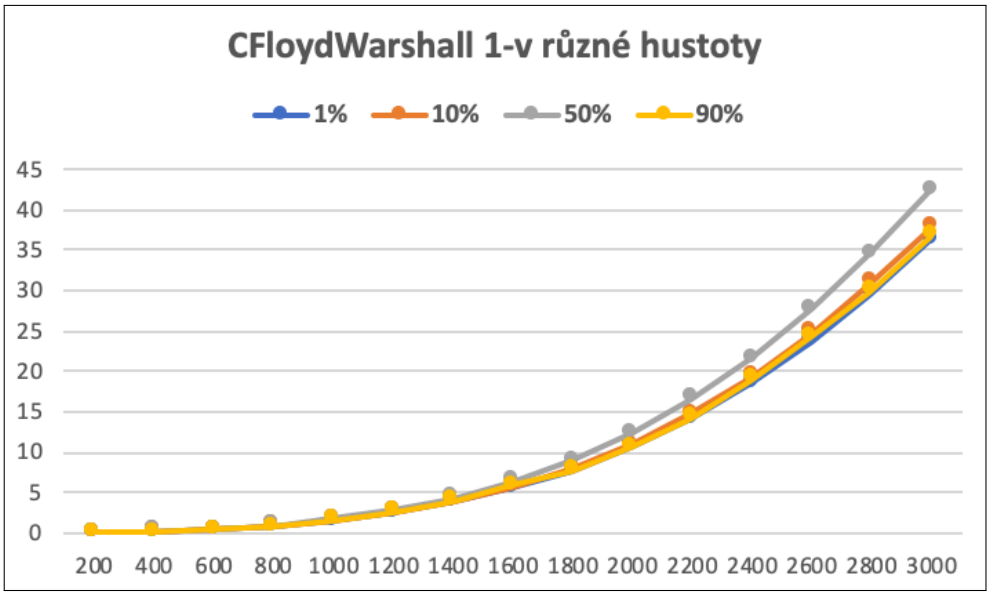
Testy s různým počtem vláken potvrzují efektivitu použití více vláken. Jak lze vidět stupeň efektivity závisí na hustotě hran vstupního grafu (viz 5.1). Důvodem vyššího paralelního zrychlení pro hustotu hran 90% je to, že paralelizovaná část výpočtu je pro tuto hustotu časově nejnáročnější.

Podle očekávání testy potvrzují, že s rostoucí počtem vrcholů ve vstupním grafu roste časová náročnost výpočtu. (viz 5.2) Důvodem prohození pořadí 70% a 90% je to, že při 70% se ve výpočtu vytváří větší výstupní struktura (viz tabulka 5.1).

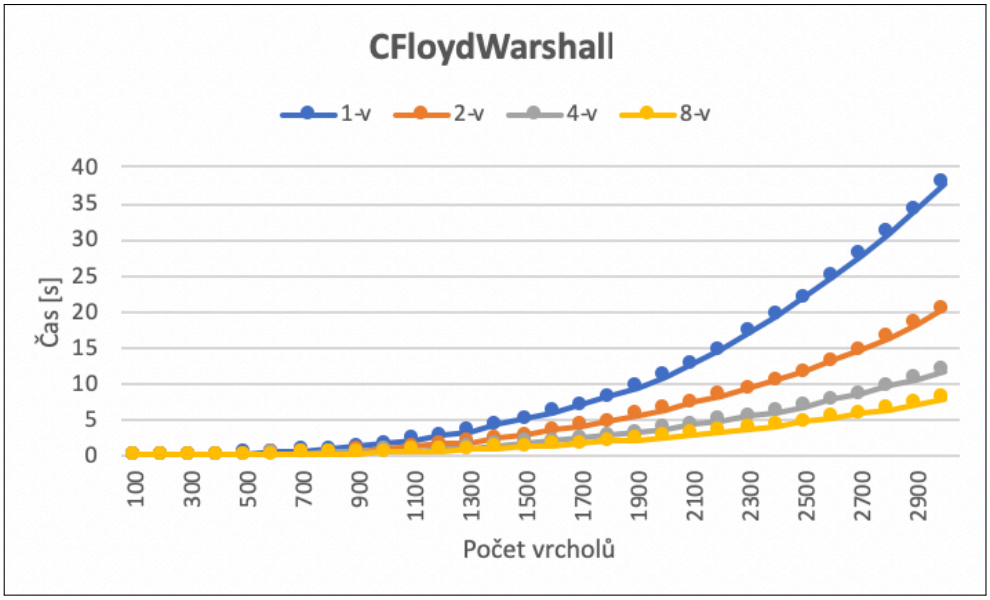
5.4 Testování třídy CFloydWarshall

Nemusíme se zabývat otázkou, z kolika vrcholů musíme počítat nejkratší vzdálenost, protože Floydův-Warshallův algoritmus počítá nejkratší vzdálenosti vždy ze všech vrcholů.

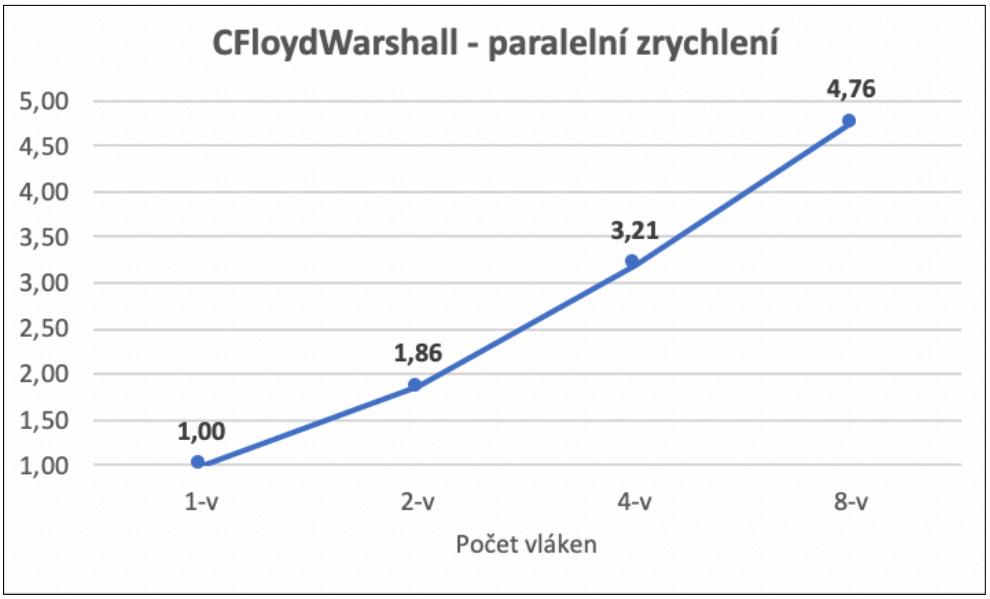
Na grafu 5.3 je znázorněna závislost výpočtu třídy *CFloydWarshall* na hustotě hran vstupního grafu. Na grafech 5.4 a 5.5 jsou znázorněny testy s různým počtem vláken.



Obrázek 5.3: Testování třídy *CFloydWarshall* s různou hustotou hran



Obrázek 5.4: Testování třídy *CFloydWarshall* s různým počtem vláken



Obrázek 5.5: Paralelní zrychlení třídy *CFloydWarshall*

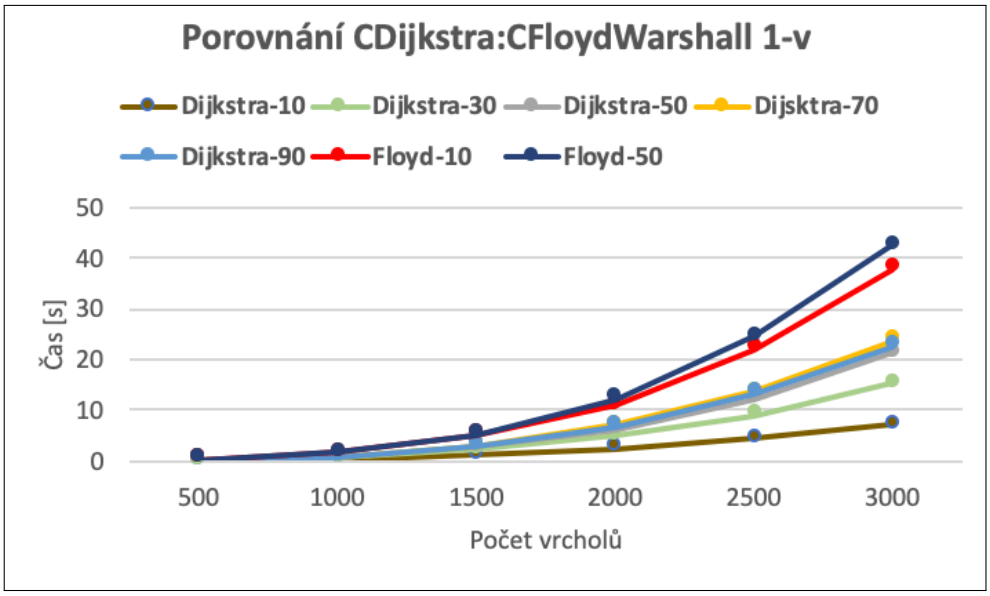
Na testu s různou hustotou hran vstupního grafu lze vidět, že výpočet varianty s 50% hustotou je pomalejší než ostatní varianty. Důvodem je vytváření větší výstupní struktury (viz tabulka 5.1).

Testy s různým počtem vláken potvrzují efektivitu použití více vláken.

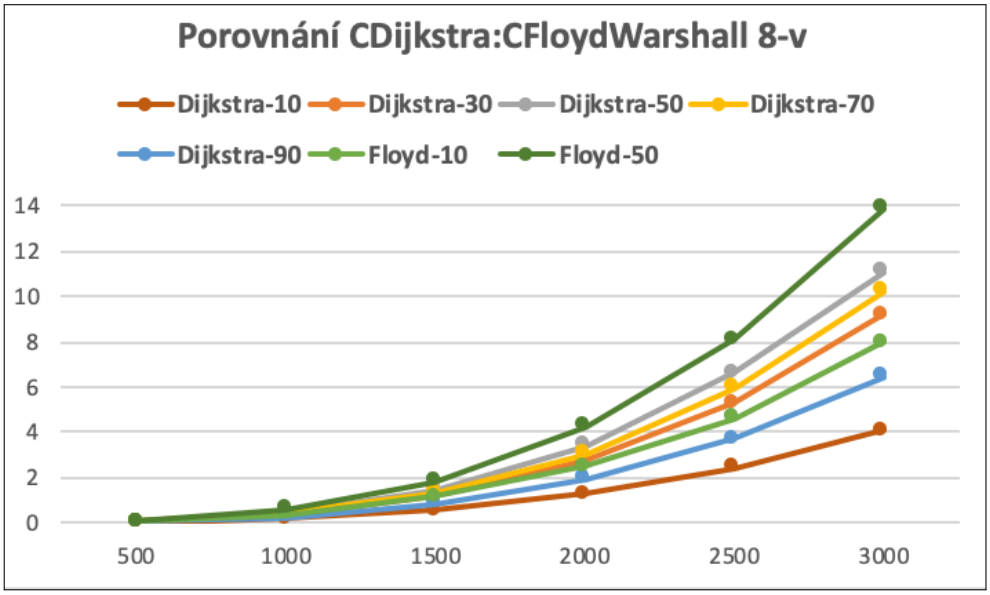
5.5 Porovnání tříd CDijkstra a CFloydWarshall

Tyto testy se vztahují na vygenerované grafy s průměrně polovinou všech vrcholů v množině D^-. Čím by byla množina D^- větší, tím by se výsledné časy výpočtů třídy *CDijkstra* zhoršily. Tím by se také změnil moment, kdy by mohlo být výhodnější použít třídu *CFloydWarshall*.

Následující testy (provedení s jedním a osmi vlákny) porovnává efektivitu použití tříd *CDijkstra* a *CFloydWarshall*. Snažíme se zjistit pro jaké hustoty hran v grafu je časově výhodnější využívat třídu *CFloydWarshall*. (viz obr. 5.6 a 5.7)



Obrázek 5.6: Jednovláknové porovnání tříd *CDijkstra* a *CFloydWarshall* pro různé hustoty

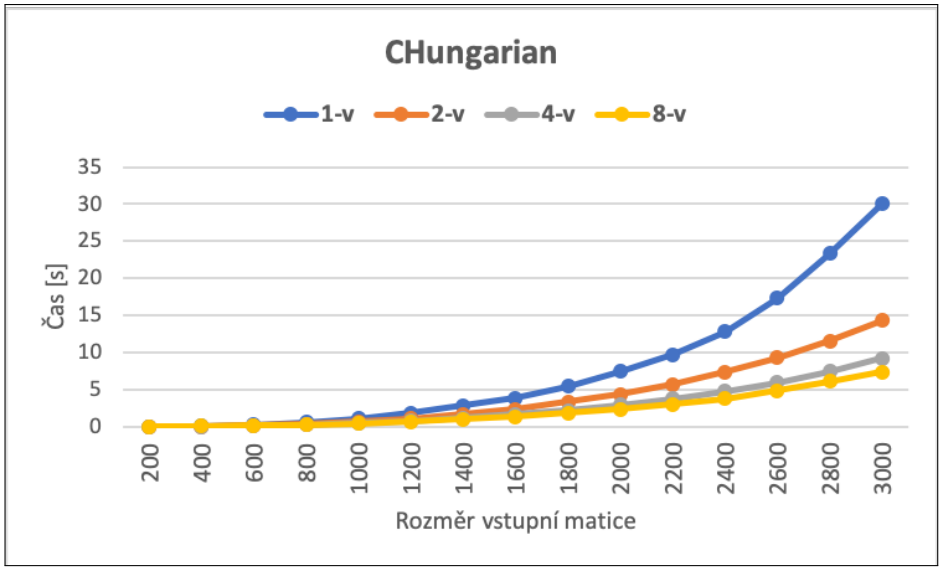


Obrázek 5.7: Osmivláknové porovnání tříd *CDijkstra* a *CFloydWarshall* pro různé hustoty

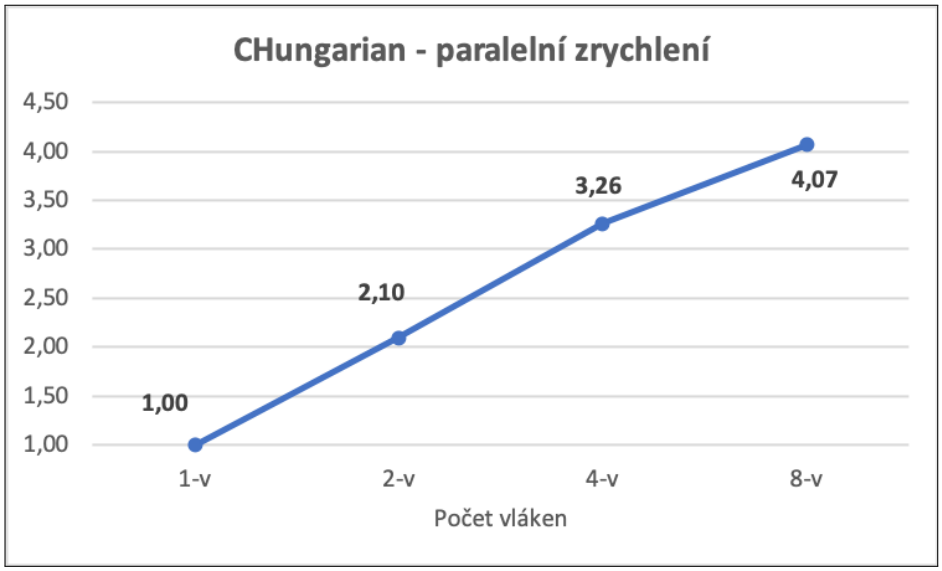
Z testů vyplynulo, že z hlediska časové složitosti je v případě provedení s jedním vláknem i s osmi vlákny výhodnější pro testované hustoty hran používat třídu *CDijkstra*.

5.6 Testování třídy CHungarian

V této třídě je na vstupu matice s náhodně generovanými kladnými hodnotami. Na grafech 5.8 a 5.9 jsou znázorněny testy s různým počtem vláken.



Obrázek 5.8: Testování třídy *CHungarian* s různým počtem vláken



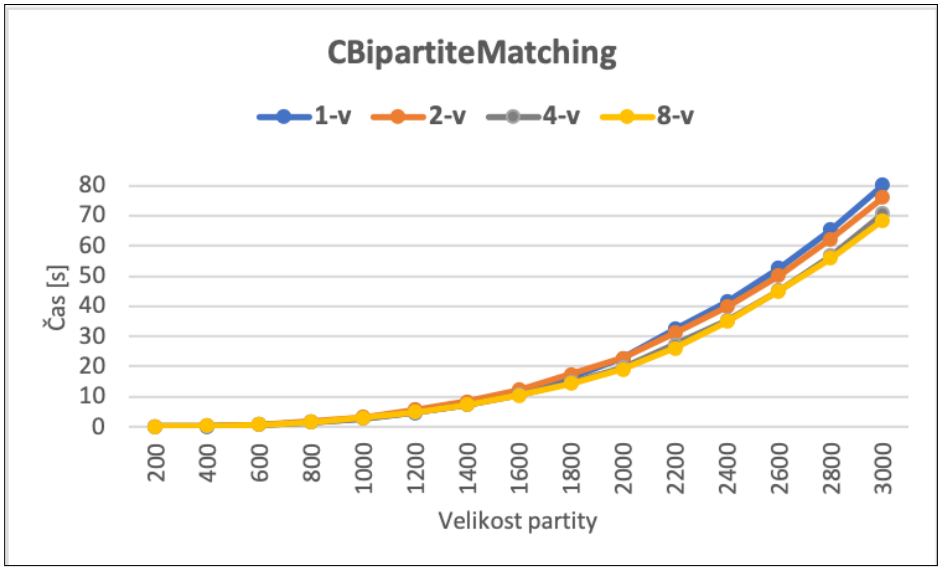
Obrázek 5.9: Paralelní zrychlení třídy *CHungarian*

Testy s různým počtem vláken potvrzují efektivitu použití více vláken.

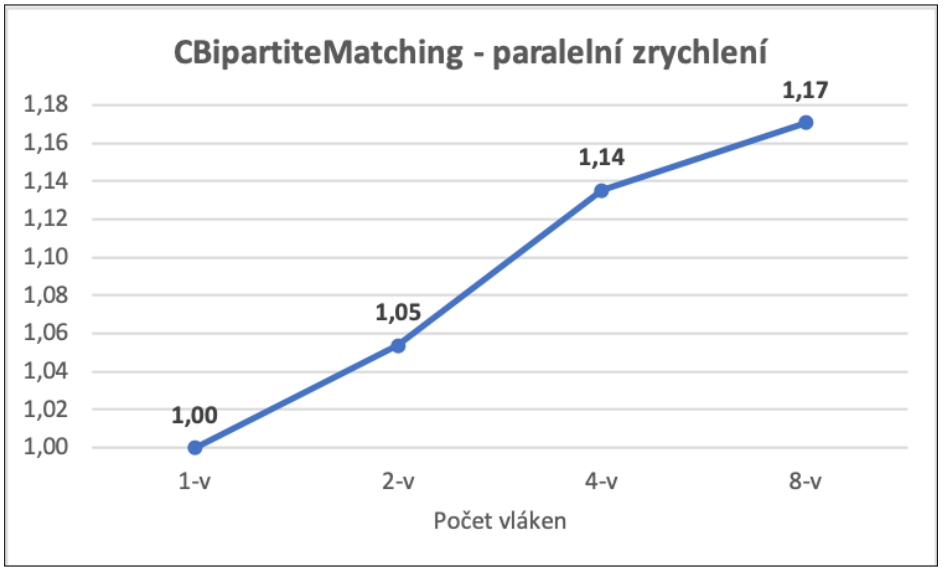
5.7 Testování třídy CBipartiteMatching

V této třídě je na vstupu bipartitní graf s náhodně generovanými kladnými hodnotami hran.

Na grafech 5.10 a 5.11 jsou znázorněny testy s různým počtem vláken.



Obrázek 5.10: Testování třídy *CBipartiteMatching* s různým počtem vláken

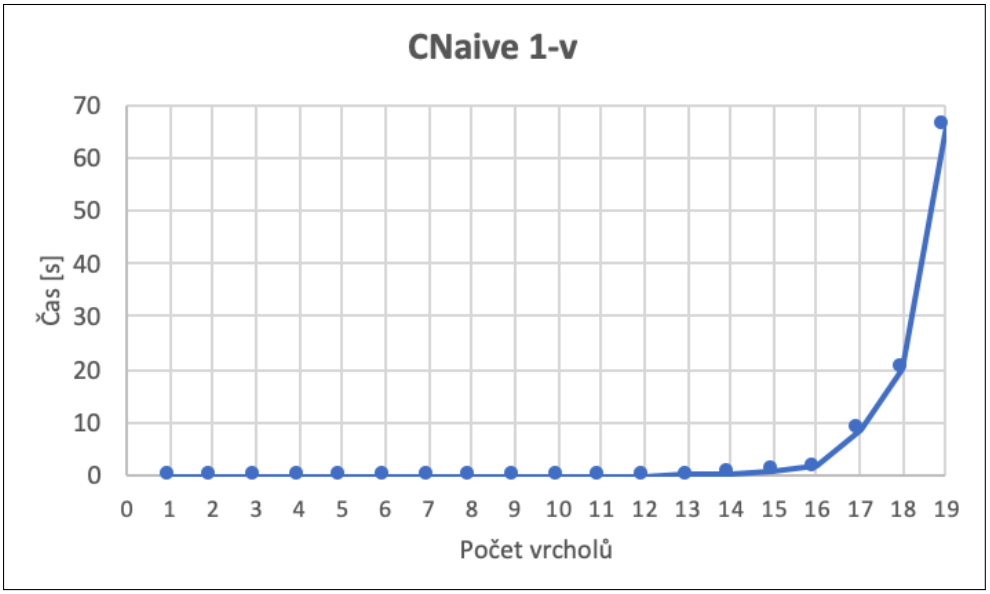


Obrázek 5.11: Paralelní zrychlení třídy *CBipartiteMatching*

Jak lze z testů vidět, paralelizace má jen omezenou účinnost.

5.8 Testování třídy CNaive

Tento test má za cíl demonstrovat, jak neefektivní je toto naivní řešení problému (s asymptotickou složitostí $O(n * n!)$) ve srovnání s ostatními řešeními. Proto ani nebylo prováděno pro více vláken.

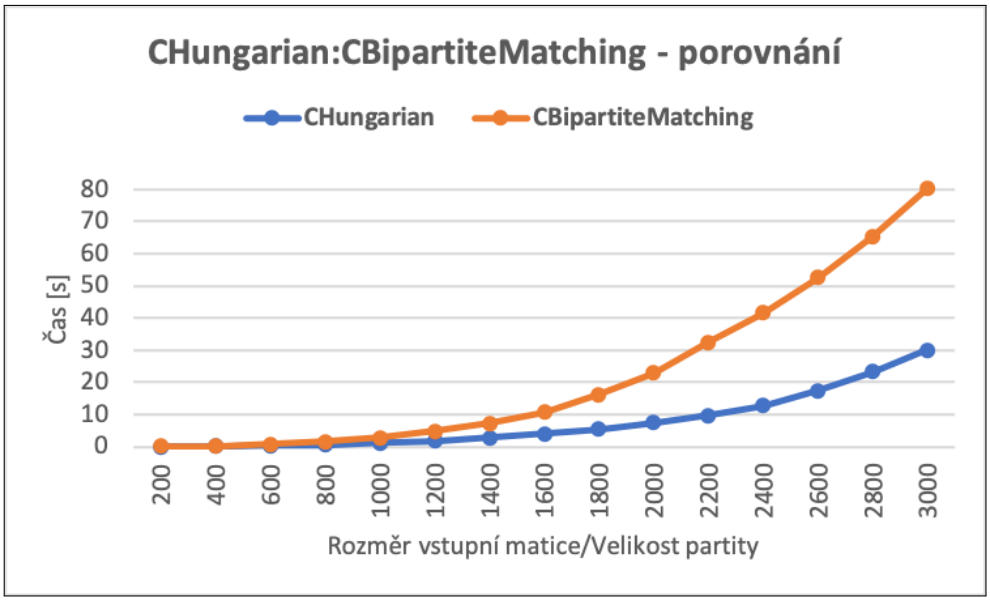


Obrázek 5.12: Jednovláknové testování třídy *CNaive*

5.9 Porovnání tříd CHungarian a CBipartiteMatching

Na vstupu máme matice (resp. bipartitní graf) s náhodně generovanými kladnými hodnotami.

Na grafu 5.13 je jednovláknové porovnání tříd *CBipartiteMatching* a *CHungarian*.



Obrázek 5.13: Jednovláknové porovnání tříd *CBipartiteMatching* a *CHungarian*

Z testu vyplývá, že výpočet třídou *CHungarian* je značně efektivnější.

Důvodem může být, že úvodní krok *CHungarian* najde lepší výchozí párování, a tím získá náskok oproti *CBipartiteMatching*.

Vzhledem k tomu, že *CHungarian* má výrazně lepší paralelní zrychlení, situace se nezmění ani pro více vláken.

5.10 Testování třídy CDCPP

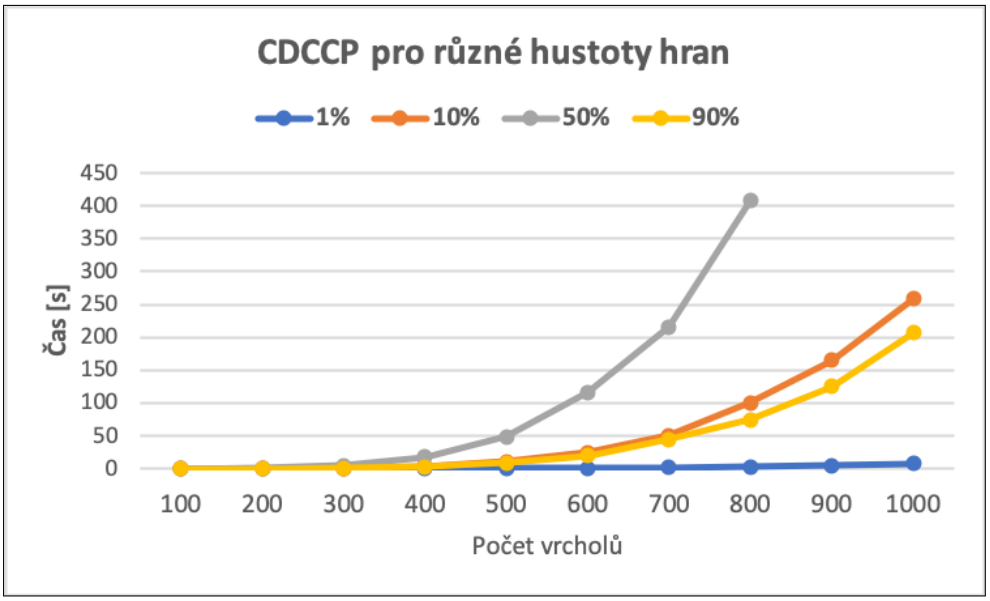
V této podkapitole se testuje kompletní implementace DCPP.

Na základě hustoty hran vstupních grafů se testují čtyři varianty. Volba zvolených algoritmů se určuje podle výsledků předchozích dílčích testů.

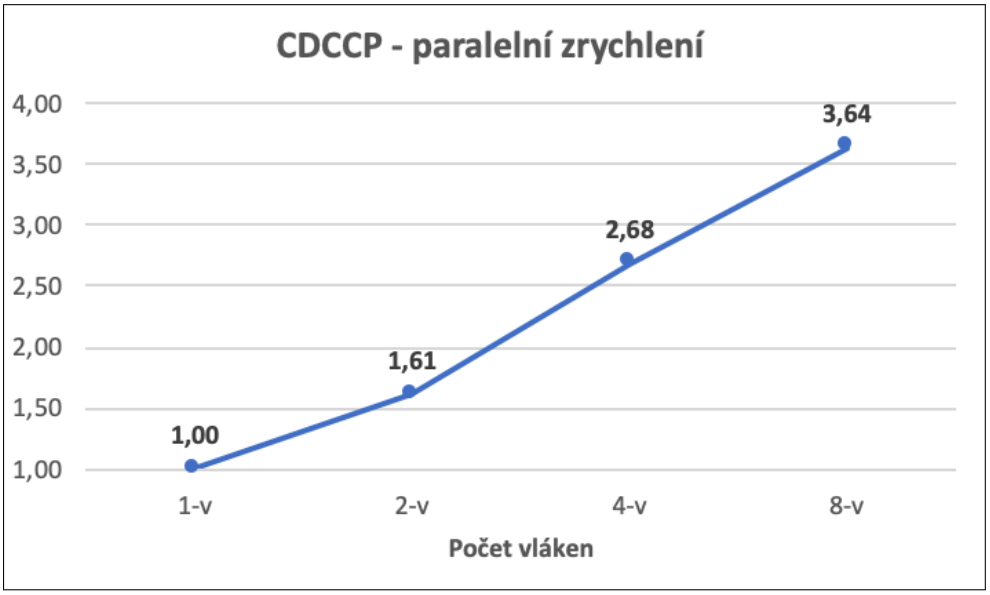
Na přiřazovací problém se nasadí třída *CHungarian* pro všechny varianty, protože se ukázala efektivnější než třída *CBipartiteMatching* (viz obr. 5.13).

Na hledání nejkratších cest se nasadí třída *CDijkstra*, která se ukázala v testovaných variantách efektivnější než třída *CFloydWarshall* (viz obr. 5.6).

Na hledání eulerovského cyklu se aplikuje třída *CEuler*.



Obrázek 5.14: Jednovláknové porovnání třídy *CDCPP* s různými hustotami hran vstupního grafu



Obrázek 5.15: Paralelní zrychlení třídy *CDCPP* s hustotou hran 1% vstupního grafu

Na obrázku 5.14 je vidět, že časová složitost výpočtu pro graf s hustotou hran 50% je značně horší než pro ostatní varianty.

Důvod tohoto chování lze vidět na tabulce 5.1. Průměrná velikost hledaného optimálního párování roste výrazně rychleji pro grafy s hustotou hran 50% než pro jiné varianty.

		Počet vrcholů						
		100	**200**	**300**	**400**	**500**	**600**	**700**
Hustota	**10(90)%**	150	470	850	1350	1900	2500	3100
	50%	280	800	1400	2300	3150	4100	5250

Tabulka 5.1: Tabulka průměrných velikostí hledaných optimálních párování na základě počtu vrcholů a hustoty hran vstupního grafu

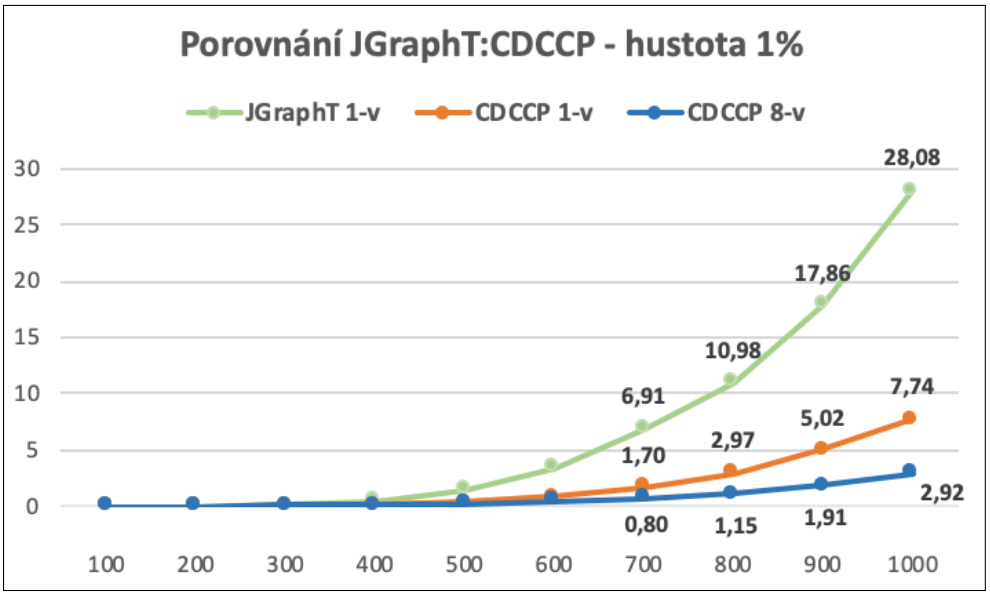
Při použití mého generátoru náhodných grafů se ukazuje, že přiřazovací problém má v rámci celkového řešení s rostoucí velikostí vstupního grafu vyšší časovou váhu než problém hledání nejkratších cest.

Testy s různým počtem vláken pro vstupní graf s hustotou hran 1% potvrzují efektivitu použití více vláken.

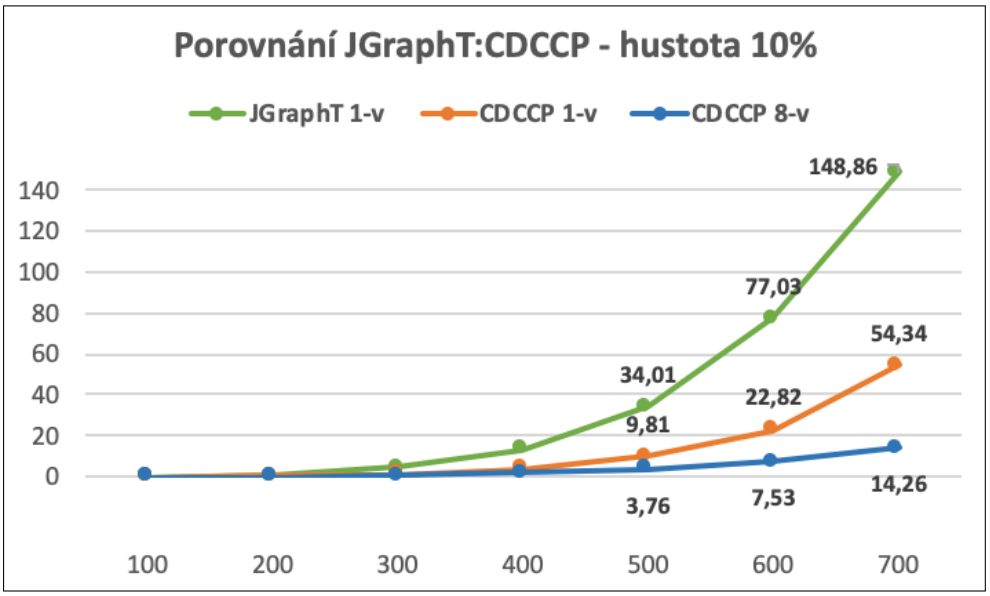
5.11 Porovnání s existující implementací

Jako existující implementace byl použit Chinese postman problem z webu JGraphT (A Java library of graph theory data structures and algorithms) [31]. Implementace používá Dijkstrův algroritmus (s použitím párovací haldy), Munkresovu maďarskou metodu (avšak bez modifikace $O(n^4) \rightarrow O(n^3)$) a Hierholzerův algoritmus. Nepodporuje paralelizaci. Je psaná v Javě.

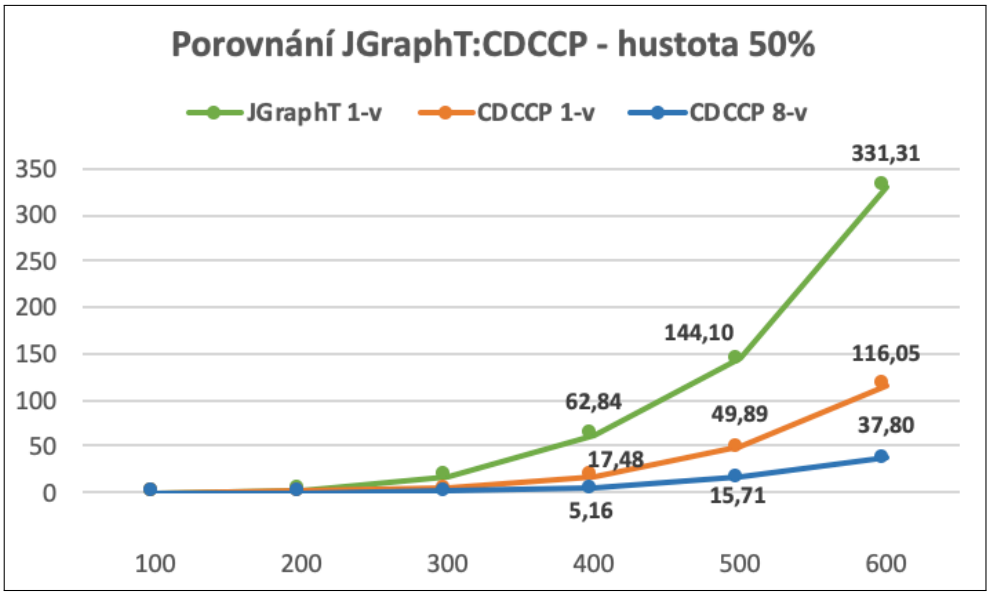
Na grafech 5.16 , 5.17 , 5.18 je výsledek porovnání.



Obrázek 5.16: Porovnání třídy *CDCPP* a JGraphT s hustotou hran 1% vstupního grafu



Obrázek 5.17: Porovnání třídy *CDCPP* a JGraphT s hustotou hran 10% vstupního grafu



Obrázek 5.18: Porovnání třídy *CDCPP* a JGraphT s hustotou hran 50% vstupního grafu

Implementace třídy *CDCPP* je efektivnější. Jak je vidět v některých případech může dosahovat i 10-ti násobné zrychlení. Důvody mohou být následující: rozdílná implementace algoritmů, použití různých programovacích prostředků (C++ vs. Java), g++ optimalizace (-O3).

Je třeba poznamenat, že naprostá většina dalších existujících implementací je určena pro jinou variantu CPP než orientovanou. Na githubu nalezená implementace pro orientovanou variantu CPP ([32]) nefunguje.

5.12 Shrnutí testování

Testování ukázalo, že pro hledání nejkratších cest při použití jednoho vlákna i osmi vláken je časově výhodnější použít třídu *CDijkstra* (viz obr. 5.6).

V případě přiřazovacího problému ukázalo, že je časově výhodnější použít třídu *CHungarian* (viz obr. 5.13).

V případě CDCPP ukázalo, že výpočet pro vstupní graf s 50% hustotou hran je průměrně časově náročnější než výpočet pro řídší či hustší graf (viz obr. 5.14).

Co se týká paralelizace, testování ukázalo, že pro třídy *CDijkstra* (viz obr. 5.1), *CFloydWarshall* (viz obr. 5.4) a *CHungarian* (viz obr. 5.8) je efektivní. Pro třídu *CBipartiteMatching* (viz obr. 5.10) má paralelizace omezenou efektivitu.

Testování s existujícím řešením prokázalo vyšší efektivitu mého řešení.

5.13 Testy správnosti

Testy správnosti lze rozdělit do několika skupin.

Nejprve jsem porovnával výsledky dílčích algoritmů (*CDijkstra* vs. *CFloydWarshall* a *CHungarian* vs. *CBipartiteMatching* vs. *CNaive*).

Na vstupu byl graf až s 3000 vrcholy (*CDijkstra*,*CFloydWarshall*), resp. matice až s 3000 řádky (*CHungarian*), resp. bipartitní graf až s 3000 vrcholy v jedné partitě (*CBipartiteMatching*), resp. matice až s 19 řádky (*CNaive*). Test správnosti není ovlivněn hustotou hran vstupního grafu (*CDijkstra*, *CFloydWarshall*).

Dalším krokem bylo testování kompletní implementace třídy *CDCPP* s použitím různých dílčích algoritmů a porovnání výsledků.

Na vstupu byl graf až s 800 vrcholy a s různou hustotou hran (1%, 10%, 50%, 90%). Navíc jsem testoval variantu, kdy vstupní graf je již eulerovský (využil jsem graf s hustotou hran 100%).

Pozitivní výsledek těchto testů mi umožnil nadále použít vždy už jenom jednu vybranou kombinaci (*CDijkstra*, *CHungarian*).

Dalším krokem bylo porovnání kompletní implementace třídy *CDCPP* s existujícím řešením (JGraphT). Metoda testování bylo shodná s metodou uvedenou v předchozím kroku.

Testování v každém předchozím kroku bylo prováděno na řádově stovkách instancí. Testování prokázalo správnost mého řešení (porovnávané výsledky s JGraphT byly shodné).

Závěr

Hlavním cílem práce bylo navržení a implementace programu pro řešení DCPP varianty problému čínského listonoše za použití paralelizace. Práce dokumentuje přípravu i vlastní postup implementace algoritmů problému od stanovení cílů, přes uvedení relevantních termínů teorie grafů (viz kap. 2) a algoritmů řešení DCPP (viz kap. 3), až k realizaci vlastního řešení práce.

Implementace je popsaná v kapitole 4. Popsané algoritmy byly implementovány v jazyce C++. Následovala paralelizace vybraných částí řešení. Testování je popsáno v kapitole 5. Probíhalo na fakultním svazku STAR. Zaměřeno bylo na srovnání efektivity (časové náročnosti) jednotlivých algoritmů a jejich paralelizace s použitím různého počtu vláken. Potvrdilo se, že paralelní algoritmy jsou efektivnější než sekvenční. Testování s existujícím řešením prokázalo výraznou efektivitu mého řešení.

Další možnosti rozšíření práce by mohlo být např. modifikace třídy *CBipartiteMatching* na řešení problému B-matching, rozšíření generátoru náhodných grafů i na tvorbu jiných variant grafů (např. přidání parametru velikosti množiny D^-) nebo porovnání DCPP s jinými typy úloh CPP.

Literatura

[1] Kwan, M.-K.: Graphic programming using odd and even points. *Chinese Math.*, ročník 1, 1962: s. 237–277, [cit. 2020-10-1]. Dostupné z: https://www.math.uni-bielefeld.de/documenta/vol-ismp/16_groetschel-martin-yuan-ya-xiang.pdf

[2] Malík, M.; Suchý, O.; Tvrdík, P.; aj.: Algoritmy a grafy 1. 2018/2019, [Soubor přístupný po přihlášení do sítě ČVUT]. Dostupné z: https://courses.fit.cvut.cz/BI-AG1/media/lectures/bi-ag1-p2-handout.pdf

[3] Suchý, O.; Valla, T.: Algoritmy a grafy 2. 2017/2018, [Soubor přístupný po přihlášení do sítě ČVUT], [cit. 2019-02-04]. Dostupné z: https://courses.fit.cvut.cz/BI-AG2/media/lectures/bi-ag2-p-vse.pdf

[4] Lu, L.: Math 777 Graph Theory I Lecture Note 3. 2005, [cit. 2020-03-03]. Dostupné z: http://people.math.sc.edu/lu/teaching/2005fall_776/lecture3.pdf

[5] Hliněný, P.: Základy teorie grafů. *Brno: Masarykova Univerzita*, 2010, [cit. 2020-10-1]. Dostupné z: https://is.muni.cz/do/1499/el/estud/fi/js10/grafy/Grafy-text10.pdf

[6] Black, P. E.: *Dictionary of algorithms and data structures*. National Institute of Standards and Technology Gaithersburg, 2004, [cit. 2020-10-1]. Dostupné z: https://xlinux.nist.gov/dads/?

[7] Edmonds, J.; Johnson, E. L.: Matching, Euler tours and the Chinese postman. *Mathematical Programming*, ročník 5, č. 1, 1973: s. 88–124, [cit. 2020-10-1]. Dostupné z: https://web.eecs.umich.edu/~pettie/matching/Edmonds-Johnson-chinese-postman.pdf

[8] Grötschel, M.; Yuan, Y.-x.: Euler, Mei-Ko Kwan, Königsberg, and a Chinese Postman. *Documenta Mathematica · Extra Volume*, 2012: s. 43–50, [cit. 2020-10-20]. Dostupné z: https://www.math.uni-bielefeld.de/documenta/vol-ismp/16_groetschel-martin-yuan-ya-xiang.pdf

[9] Gordenko, M. K.; Avdoshin, S. M.: The Mixed Chinese Postman Problem. *Trudy ISP RAN/Proc. ISP RAS*, ročník 29, č. 4, 2017, doi:10.15514/ISPRAS-2017-29(4)-7, [cit. 2020-10-1]. Dostupné z: http://www.mathnet.ru/links/aa8e6abc0075c8bf8fc80146f257d3a0/tisp238.pdf

[10] Eiselt, H. A.; Gendreau, M.; Laporte, G.: Arc Routing Problems, Part I: The Chinese Postman Problem. *Operations Research*, ročník 43, č. 2, 1995: s. 231–242, [cit. 2020-10-1]. Dostupné z: https://pubsonline.informs.org/doi/pdf/10.1287/opre.43.2.231

[11] Guan, M.: On the Windy Postman Problem. *Discrete Applied Mathematics*, ročník 9, č. 1, 1984: s. 41–46, [cit. 2020-10-1]. Dostupné z: https://core.ac.uk/reader/82387012

[12] Thimbleby, H.: The directed Chinese Postman Problem. *Software: Practice and Experience*, ročník 33, č. 11, 2003: s. 1081–1096, [cit. 2020-10-1]. Dostupné z: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.106.4691&rep=rep1&type=pdf

[13] Dijkstra, E. W.: A Note on Two Problems in Connexion with Graphs. *Numerische mathematik*, ročník 1, č. 1, 1959: s. 269–271, [cit. 2020-11-11]. Dostupné z: http://www-m3.ma.tum.de/foswiki/pub/MN0506/WebHome/dijkstra.pdf

[14] Mareš, M.; Valla, T.: *Průvodce labyrintem algoritmů*, ročník 15. publikace. Praha: CZ.NIC, z.s.p.o, první vydání, 2017, ISBN 8088168198.

[15] Kolář, J.: *Teoretická informatika*. Česká informatická společnost, 2000.

[16] Ahuja, R.; Magnanti, T.; Orlin, J.: Networks Flows: Theory, Algorithms, and Practice, ISBN 013617549X. 1993.

[17] Šmerek, M.: Přiřazovací problém. [cit. 2019-04-20]. Dostupné z: https://moodle.unob.cz/mod/resource/view.php?id=24001

[18] KUHN, H. W.: The Hungarian Method for the assignment problem. *Naval research logistics quarterly*, 1955: s. 83–97, [cit. 2020-11-11]. Dostupné z: https://web.eecs.umich.edu/~pettie/matching/Kuhn-hungarian-assignment.pdf

[19] Munkres, J.: Algorithms for the assignment and transportation problems. *Journal of the society for industrial and applied mathematics*, ročník 5, č. 1, 1957: s. 32–38, [cit. 2020-11-11]. Dostupné z: https://web.eecs.umich.edu/~pettie/matching/Munkres-variant-of-Hungarian-alg.pdf

[20] Wong, J. K.: A new implementation of an algorithm for the optimal assignment problem: An improved version of Munkres' algorithm. *BIT Numerical Mathematics*, ročník 19, č. 3, 1979: s. 418–424, [cit. 2020-09-11]. Dostupné z: https://link.springer.com/article/10.1007/BF01930994

[21] Edmonds, j.; Karp, R.: Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *Journal of the Association for Computing Machinery*, ročník 19, č. 2, 1972: s. 248–264, [cit. 2020-10-1]. Dostupné z: https://web.eecs.umich.edu/~pettie/matching/Edmonds-Karp-network-flow.pdf

[22] Kleinberg, R.: Introduction to Algorithms. Spring 2010, [cit. 2020-12-1]. Dostupné z: https://www.cs.cornell.edu/courses/cs4820/2012sp/handouts/edmondskarp.pdf

[23] Hopcroft, J. E.; Karp, R. M.: An n^5/2 algorithm for maximum matchings in bipartite graphs. *SIAM Journal on computing*, ročník 2, č. 4, 1973: s. 225–231.

[24] Zwick, U.: Analysis of Algorithms. 2005/2006, [cit. 2020-02-02]. Dostupné z: http://www.cs.tau.ac.il/~zwick/grad-algo-06/min-cost-flow.pdf

[25] Mayr, E.: Praktikum Algorithmen-Entwurf (Teil 6), Nov. 2002, 6–11. *Technische Universität München*, [cit. 2020-05-10]. Dostupné z: http://wwwmayr.in.tum.de/lehre/2002WS/algoprak/part6.ps.gz

[26] Maurer, P. M.: Generating strongly connected random graphs. In *Proceedings of the International Conference on Modeling, Simulation and Visualization Methods (MSV)*, 2017, s. 3–6, [cit. 2020-10-1]. Dostupné z: https://csce.ucmss.com/cr/books/2017/LFS/CSREA2017/MSV3359.pdf

[27] Parhami, B.: *Introduction to parallel processing: algorithms and architectures*. New York: Plenum Press, 1999, [cit. 2020-10-1]. Dostupné z: https://doc.lagout.org/science/0_ComputerScience/2_Algorithms/IntroductiontoParallelProcessing-AlgorithmsandArchitectures.pdf

[28] OpenMP: OpenMP. 2019, [cit. 2020-06-06]. Dostupné z: https://www.openmp.org/specifications/

[29] Šimeček, I.; Langr, D.: *Moderní počítačové architektury a optimalizace implementace algoritmů*. ČVUT v Praze, druhé vydání, 2016, ISBN 9788001060353.

[30] Super Micro Computer, I.: SuperServer F618R3-FT. 2020, [cit. 2020-05-30]. Dostupné z: https://www.supermicro.com/products/system/4u/F618/SYS-F618R3-FT.cfm

[31] Kinable, J.: Chinese Postman. 2020, [cit. 2020-10-30]. Dostupné z: https://jgrapht.org/javadoc-1.4.0/org/jgrapht/alg/cycle/ChinesePostman.html

[32] Soragna, A.: Chinese-Postman-Problem. 2018, [cit. 2020-11-11]. Dostupné z: https://github.com/alsora/chinese-postman-problem

Příloha A

Seznam použitých zkratek

CPP Chinese Postman Problem

BFS Breadth-First Search

DCPP Directed CPP

DFS Depth-First Search

MCPP Mixed CPP

RPP Rural Postman Problem

UCPP Undirected CPP

WPP Windy Postman Problem

WRCPP Windy Rural CPP

Příloha **B**

Obsah přiložené paměťové karty

- `readme.txt` ..... stručný popis obsahu paměťové karty
- `exe` ..... adresář se spustitelnou formou implementace
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `Zadání_BP.pdf` ..... zadání práce ve fromátu PDF
  - `BP_Razak_Matej_2021.pdf` ..... text práce ve formátu PDF