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 letního semestru 2019/20

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.

30. července 2020

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 30. července 2020 . . . . . .

Odkaz na tuto práci

Razák, Matěj. *Algoritmy pro řešení problému čínského listonoše*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2020.

Abstrakt

Tato bakalářská práce se věnuje vybraným algoritmům pro řešení problému čínského listonoše (varianta DCPP), tj. jednou z optimalizačních úloh z oboru teorie grafů. Úkolem je najít nejkratší cestu listonoše na jeho cestě 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. Je uvedena teoretickou částí s definicemi základních principů řešení a popisem jednotlivých algoritmů řešení. Cílem práce je použít různé dílčí algoritmy pro jednotlivé fáze řešení celého problému a jejich následné porovnání. Realizace používá paralelizaci a porovnává její použití s verzí sekvenční. Programy jsou řešeny v jazyce C++.

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 challenge is to find the shortest path of a

postman on his way through all the streets (those are the edges of the graph), with the fact that he must eventually return to the starting point. The work is devoted to the version with oriented weighted edges. The work starts with the theoretical part. It contains definitions of basic solution principles and the description of separate solution algorithms. The aim of the thesis is to use various partial algorithms for individual phases of the solution of the whole problem. Further more to compare some methods. The implementation uses the parallelization and compares it with the sequential version. Programs are solved in C ++.

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

Obsah

Úvod **1**

1 Cíle práce **3**

2 Analýza současného stavu řešení problému **5**
2.1 Základní pojmy teorie grafů 5
2.1.1 Hrany a vrcholy, stupeň vrcholu 6
2.1.2 Typy grafů 7
2.1.3 Eulerovský graf 8
2.2 Eulerova úloha sedmi mostů 9
2.3 Problém čínského listonoše 10
2.3.1 Historie 10
2.3.2 Typy úloh 11
2.3.3 Metodika řešení 11
2.4 Složitost 12

3 Řešení problému čínského listonoše (metody a algoritmy) **15**
3.1 Hledání nejkratších cest 15
3.1.1 Dijkstrův algoritmus 16
3.1.2 Floyd-Warshallův algoritmus 17
3.2 Přiřazovací problém 18
3.2.1 Maďarská metoda 18
3.2.2 Munkresova modifikace maďarské metody 19
3.3 Toky v sítích 21
3.3.1 Teorie 21
3.3.2 Ford-Fulkersonův algoritmus 21
3.3.3 Max-Flow Min-Cost bipartitní ohodnocený graf 22
3.4 Hledání eulerovského cyklu 23
3.4.1 Hierholzerův algoritmus 23

4 Implementace řešení DCPP **25**
4.1 Použité prostředky . . . 25
4.1.1 Programovací jazyk implementace C++ . . . 25
4.1.2 Knihovny Boost . . . 25
4.1.3 Paralelní provedení . . . 25
4.2 Sekvenční struktura řešení . . . 26
4.2.1 Class CDCPP . . . 27
4.2.2 Class CGraph . . . 28
4.2.3 Class CGenerator . . . 29
4.2.4 Class CPathFinding . . . 30
4.2.4.1 Subclass CDijkstra . . . 31
4.2.4.2 Subclass CFloydWarshall . . . 32
4.2.5 Class CAssignment . . . 32
4.2.5.1 Subclass CNaive . . . 33
4.2.5.2 Subclass CHungarian . . . 33
4.2.5.3 Subclass CBipartiteMatching . . . 34
4.2.6 Class CEuler . . . 36
4.3 Paralelní struktura řešení . . . 36
4.3.1 Třída CDijkstra . . . 36
4.3.2 Třída CFloydWarshall . . . 37
4.3.3 Třída CHungarian . . . 37
4.3.4 Třída CBipartiteMatching . . . 37
4.4 Použití programu . . . 38
4.4.1 Vstupní data . . . 38
4.4.2 Výstupní data . . . 38
4.4.3 Spuštění programu . . . 38

5 Testování **39**
5.1 Hardware a software . . . 39
5.2 Způsob testování . . . 39
5.3 Testování třídy CDijkstra . . . 40
5.4 Testování třídy CFloydWarshall . . . 41
5.5 Porovnání tříd CDijkstra a CFloydWarshall . . . 42
5.6 Testování třídy CHungarian . . . 43
5.7 Testování třídy CBipartiteMatching . . . 44
5.8 Testování třídy CNaive . . . 45
5.9 Porovnání tříd CHungarian a CBipartiteMatching . . . 46
5.10 Testování třídy CDCPP . . . 47
5.11 Shrnutí testování . . . 48
5.12 Test správnosti implementace . . . 49

Závěr **51**

Literatura **53**

A Seznam použitých zkratek **57**

B Obsah přiložené paměťové karty **59**

Seznam obrázků

2.1 Cesta v grafu z u do v 6
2.2 Orientovaný ohodnocený graf s pěti vrcholy 6
2.3 Orientovaný graf G 7
2.4 Cesta v grafu 8
2.5 Silně souvislý graf 8
2.6 Bipartitní graf úplný 9
2.7 Eulerovský neorientovaný graf 9
2.8 Úloha sedmi mostů 10
2.9 Eulerův diagram 14

3.1 Věta o tocích 21

4.1 Fork-join schéma 26
4.2 Schéma řešení DCPP 27

5.1 Testování třídy *CDijkstra* s různým počtem vláken 40
5.2 Testování třídy *CDijkstra* s různou hustotou hran vstupního grafu 41
5.3 Testování třídy *CFloydWarshall* s různým počtem vláken 42
5.4 Jedno-vláknové porovnání tříd *CDijkstra* (30%), *CFloydWarshall* a *CDijkstra* (50%) 43
5.5 Testování třídy *CHungarian* s různým počtem vláken 44
5.6 Testování třídy *CBipartiteMatching* s různým počtem vláken . . . 45
5.7 Jedno-vláknové testování třídy *CNaive* 46
5.8 Jedno-vláknové porovnání tříd *CBipartiteMatching* a *CHungarian* 47
5.9 Jedno-vláknové porovnání *CDCPP* s různými hustotami hran vstupního grafu 48
5.10 Tabulka průměrných velikostí matic optimálního párování na základě počtu vrcholů a hustoty hran vstupního grafu 48

Seznam algoritmů

1 Dijkstrův algoritmus 16
2 Floyd-Warshallův algoritmus 17
3 Maďarská metoda 19
4 Munkresova maďarská metoda 20
5 Ford-Fulkersonův algoritmus 22
6 Max-Flow Min-Cost algoritmus 23
7 Hierholzerův algoritmus 24

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. Simuluje práci poštovního doručovatele (nebo příp. jiných doručovacích či obslužných společností). Listonoš má za úkol navštívit všechny ulice jemu přiřazené oblasti (zadanou množinu ulic), přičemž vychází a vrací se do téhož 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 úlohu 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.

Práce je rozdělena do pěti hlavních kapitol. V kapitole 1 jsou popsány cíle této práce. Kapitola 2 (Analýza současného stavu řešení) se dále člení na čtyři podkapitoly. První z nich 2.1 se věnuje relevantním základním pojmům z teorie grafů a uvádí použitou terminologii, druhá 2.2 uvádí Eulerovu úlohu sedmi mostů, třetí 2.3 je věnována teorii (a historii) úlohy problému čínského listonoše (CPP). 2.4 se zabývá stručně složitostí.

Následující kapitoly jsou již věnovány vlastnímu řešení bakalářské práce. Kapitola 3 je věnována teorii řešení DCPP, uvádí použité metody a algoritmy. Obsahuje čtyři podkapitoly (Hledání nejkratší cesty mezi vrcholy 3.1: Dijkstrův algoritmus, Floyd-Warshallův algoritmus, Přiřazovací problém 3.2: maďarská metoda, Toky v sítích 3.3: Ford-Fulkersonův algoritmus, Hledání

eulerovského cyklu 3.4: Hierholzerův algoritmus). Tato kapitola je přípravou pro vlastní implementaci řešení CPP.

Samotná realizace je tak popsána v kapitole 4. Je také rozdělena do čtyř podkapitol, které jsou většinou dále ještě členěny do podsekcí. Kapitola tak 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 i vlastní testy a porovnání výsledných implementací algoritmu. Následuje Závěr a Seznam literatury. Programovací část je uložena na přiloženém paměťovém mediu (viz Příloha B).

Kapitola 1

Cíle práce

Úvodním dílčím cílem je nastudování potřebné teorie a stavu současného bádání v 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, rozdělením výpočtů do více vláken a porovnání s jednovláknovou variantou řešení. Následuje srovnání jednotlivých použitých algoritmů z hlediska jejich efektivnosti.

Kapitola 2

Analýza současného stavu řešení problému

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

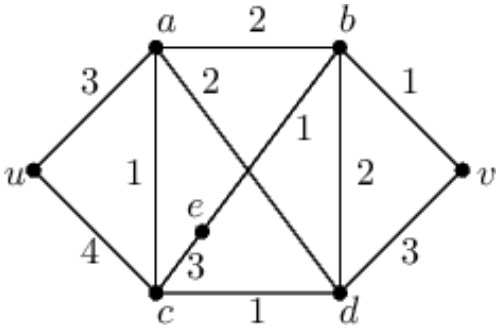
2.1 Základní pojmy teorie grafů

Aplikace teorie grafů [2, 3, 4] v praxi pokrývá celou řadu úloh. Tyto grafy pak zachycují jevy reálného světa. Jejich společným principem je zachycení struktury dané úlohy a jejích vlastností pomocí grafu, jehož základními prvky jsou hrany a vrcholy. Vhodnou reálnou situací je taková, kterou lze znázornit pomocí konečného počtu vrcholů a hran. Takto vypracovaný popis je prvním krokem k řešení daného problému.

Graf G lze pak 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 úsečky spojující tyto body. Obrázek 2.1 znázorňuje jednu z typických úloh teorie grafů, totiž zadání hledání nejkratší cesty (v tomto případě z bodu u do bodu v). Vrcholy jsou v tomto případě místa (např. města, městské čtvrtě, či domy, v tomto případě a, b, ...), hrany jejich vzdálenosti (případně další vlastnosti těchto spojnic, v tomto případě jsou hranám přiřazeny hodnoty 1, 2, 3 a 4).



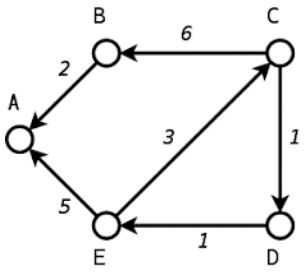
Obrázek 2.1: Cesta v grafu z u do v

Zdroj:https://kam.mff.cuni.cz/~sbirka/images/variants/20435206805bfe8614bba3a.png

2.1.1 Hrany a vrcholy, stupeň vrcholu

Hrana **orientovaná** 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).

Ohodnocená hrana – ohodnocení vyjadřuje kvantitu nebo kvalitu vztahů mezi vrcholy (např. vzdálenost nebo průchodnost)



Obrázek 2.2: Orientovaný ohodnocený graf s pěti vrcholy

Zdroj:https://www.vutbr.cz/www_base/zav_prace_soubor_verejne.php?file_id=115691

Z výše uvedených typů hran lze pak vytvořit různé kombinace (např. orientovaná ohodnocená hrana).

Rozlišujeme dva případy: **stupeň vrcholu** v neorientovaném a orientovaném grafu (viz kap. 2.1.2).

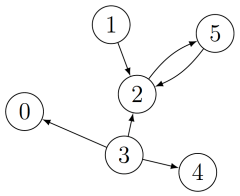
V neorientovaném grafu platí, že stupeň vrcholu je počet hran daného vrcholu.

$deg(u) = |\{e \in E | u \in e\}|$

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 graf G

Zdroj: [4]

Na obrázku 2.3 jsou následující stupně vrcholů (vrchol - deg^+ - deg^-):
0-1-0
1-0-1
2-3-1
3-3-0
4-1-0
5-1-1

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

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

Uzavřený tah je tah který začíná i končí ve stejném vrcholu. Uzavřený tah nazveme **cyklus**.

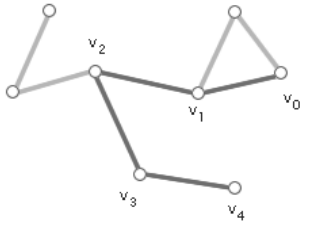
Délka cesty je součet ohodnocení hran v dané cestě.

Nejkratší uv**-cesta** je cesta, která má minimum z délek všech uv-cest.

Výše uvedené definice jsou převzaty z [4, přednáška 12][5, kap. 20].

2.1.2 Typy grafů

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



Obrázek 2.4: Cesta v grafu

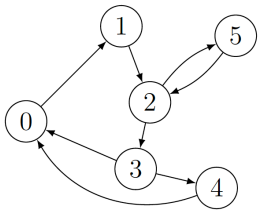
Zdroj:http://kdm.karlin.mff.cuni.cz/diplomky/lukas_jirovsky_dp/zakladni-pojmy/img/cesta_v_grafu_priklad.png

Graf G je **Souvislý**, jestliže v něm pro každé jeho dva vrcholy u, v existuje uv-cesta. Jinak je G **nesouvislý**.

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

Orientovaný graf $G = (V, E)$ nazveme **silně souvislý**, 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.

Pro **slabě souvislý** graf platí, že pro každé dva vrcholy u, v existuje alespoň jedna z cest z u do v, nebo z v do u.



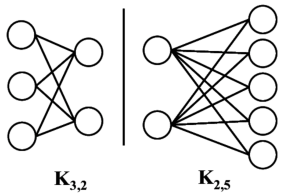
Obrázek 2.5: Silně souvislý graf

Zdroj: [4]

Bipartitní graf $G = (X \cup Y, E)$, kde X a Y jsou partity G, je graf, jehož množinu vrcholů je možné rozdělit na dvě disjunktní množiny. Každá z těchto množin obsahuje pouze hrany do druhé, tj. neexistuje hrana, která by spojovala dva vrcholy v téže množině (partitě). Úplný bipartitní graf značíme $K_{n,m}$.

2.1.3 Eulerovský graf

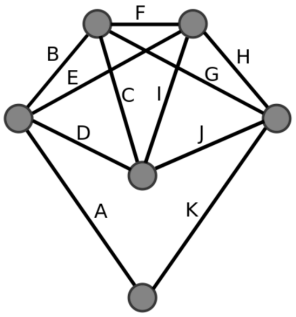
Jeho principem je „kresba jedním tahem“, tedy zadání: nakreslete daný graf jedním uzavřeným tahem bez zvednutí tužky z papíru (žádná hrana se neobtahuje dvakrát). Tento tah začíná i končí ve stejném vrcholu.



Obrázek 2.6: Bipartitní graf úplný

Zdroj:`https://www.researchgate.net/figure/Examples-of-complete-bipartite-graphs_fig4_265434438`

Eulerovský graf je takový souvislý neorientovaný graf, který má všechny vrcholy sudého stupně, tzn. existuje uzavřený tah, který obsahuje všechny jeho hrany.



Obrázek 2.7: Eulerovský neorientovaný graf

Zdroj:`https://cs.wikipedia.org/wiki/Soubor:Labelled_Eulergraph.svg`

Věta: Orientovaný graf G je eulerovský, právě tehdy když G je silně souvislý a pro všechny vrcholy v platí $deg^+(v) = deg^-(v)$. Důkaz lze najít [6].

Tah se nazývá **eulerovský**, když prochází všechny hrany grafu právě jednou. Navíc je tento tah uzavřený, pokud se vrací do výchozího bodu. Uzavřenému tahu se říká **eulerovský cyklus**. **Graf** je **eulerovský**, když v něm existuje eulerovský cyklus.

2.2 Eulerova úloha sedmi mostů

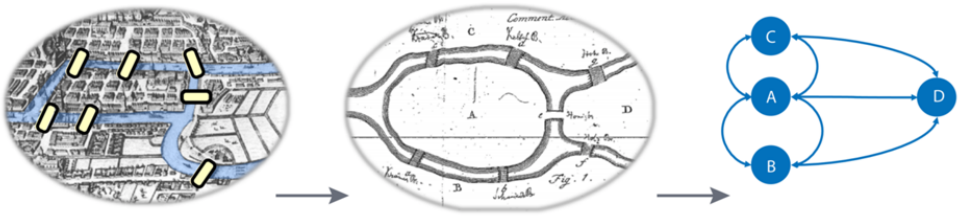
Příkladem aplikace eulerovského grafu je řešení úlohy sedmi mostů [7, 8].

Řešení problému sedmi mostů v Královci Leonhardem Eulerem bývá považováno za ustavující moment teorie grafů (1735).

Zadání úlohy:

Královec bylo pruské město se sedmi mosty a otázka byla, zda je možné přejít všech sedm bez toho, aby člověk šel přes některý most dvakrát. Zjednodušením úlohy na úlohu propojených čar se Eulerovi povedlo dokázat, že to nelze.

Máme sedm mostů přes řeku. Mosty jsou hrany (je jich sedm), dvě nábřeží a dva ostrovy tvoří celkem čtyři vrcholy (čtyři městské čtvrti).



Obrázek 2.8: Úloha sedmi mostů

Zdroj:https://www.oreilly.com/library/view/graph-algorithms/9781492047674/assets/gral_0101.png

Euler dokázal že úloha sedmi mostů nemá řešení. Ačkoliv je graf souvislý, všechny jeho vrcholy jsou lichého stupně.

2.3 Problém čínského listonoše

2.3.1 Historie

Chinese Postman Problem [9, 10], problém čínského listonoše (správně ovšem čínský problém listonoše) formuloval ve svém článku v roce 1962 čínský vědec v oblasti matematického programování Mei-Ko Kwan (nar. 1934 v Šanghaji).

Jedná se o optimalizační úlohu následujícího zadání: Listonoš musí doručit poštu v dané oblasti. Musí projít všechny ulice oblasti a vrátit se zpátky na poštovní úřad. Jak si má naplánovat cestu tak, aby ušel co nejkratší vzdálenost.

Kwan se specializoval na problém zkoumání 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, z oboru tzv. okružních úloh. 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.

2.3.2 Typy úloh

Původní nejjednodušší úloha (s neorientovanými hranami) byla časem rozvedena na další varianty CPP [11, 12].

Typy úloh CPP rozlišujeme podle typu grafu, kterým je úloha popsaná. Těchto typů je celá řada. Následující jsou některé z nich.

Původní 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) [13] – 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 (URCPP, DRCPP, UWRCPP apod.)

UCPP, DCPP patří do třídy složitosti P a MCPP, RPP a WPP jsou NP-složité.

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

2.3.3 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 [14] zavádíme následující:
$\delta(v) = deg^-(v) - deg^+(v)$
Jestliže $\delta(v) = 0$, potom v je vyvážený vrchol. Jinak je nevyvážený.
Graf je eulerovský, když každý jeho vrchol je vyvážený.

Eulerovský cyklus grafu je optimální trasa listonoše, protože každou hranu prochází právě jednou.

D^+ je množina vrcholů s kladnou δ
$D^+ = \{v|\delta(v) > 0\}$

D^- je množina vrcholů se zápornou δ
$D^- = \{v|\delta(v) < 0\}$

Jestliže graf není eulerovský, musíme:

- 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

Vždy se všechny prvky v přiřazovacím problému musí spárovat, jinak by některé vrcholy zůstaly nevyvážené. V implementaci to znamená, že matice s hodnotami párování 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 3.4).

Matematické zapsání problému:

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

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

přičemž:

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

2.4 Složitost

Tato kapitola uvádí základní termíny teorie složitosti algoritmů pro potřeby této práce. [15, 16]

Při řešení úloh na počítačích potřebujeme nástroj, kterým je možné „změřit“ a porovnat efektivitu a rychlost jednotlivých algoritmů. K tomu slouží tzv. **asymptotická složitost algoritmu**, která umožňuje klasifikaci algoritmů. Tato složitost určuje operační náročnost programu tak, že zjišťuje časovou závislost chování algoritmu na velikosti vstupních dat.

Definice: Nechť $f, g : N \to R$ jsou dvě funkce. Řekneme, že funkce $f(n)$ *je třídy* $O(g(n))$, jestliže existuje taková kladná reálná konstanta c, že pro *skoro všechna* n platí $f(n) \leq cg(n)$. Skoro všemi n se myslí, že nerovnost může selhat pro konečně mnoho výjimek, tedy že existuje nějaké přirozené n_0 takové,

že nerovnost platí pro všechna $n \geq n_0$. Funkci $g(n)$ se pak říká asymptotický horní odhad funkce $f(n)$. [16, kap. 2.4]

Složitost algoritmu je pak vyjádřena zařazením do třídy složitosti. Zařazení odpovídá funkci (stejné třídy), která ohraničuje jeho složitost, funkce $f(n)$. Podle typu této funkce dělíme algoritmy do kategorií funkce logaritmické $O(logn)$, lineární $O(n)$ či polynomiální $O(n^k)$ až po exponenciální $O(k^n)$ nebo faktoriálovou $O(n!)$. Poslední dvě jsou ovšem už považované za neefektivní. Pro řešení rozsáhlých a složitých úloh se pak používají algoritmy aproximační.

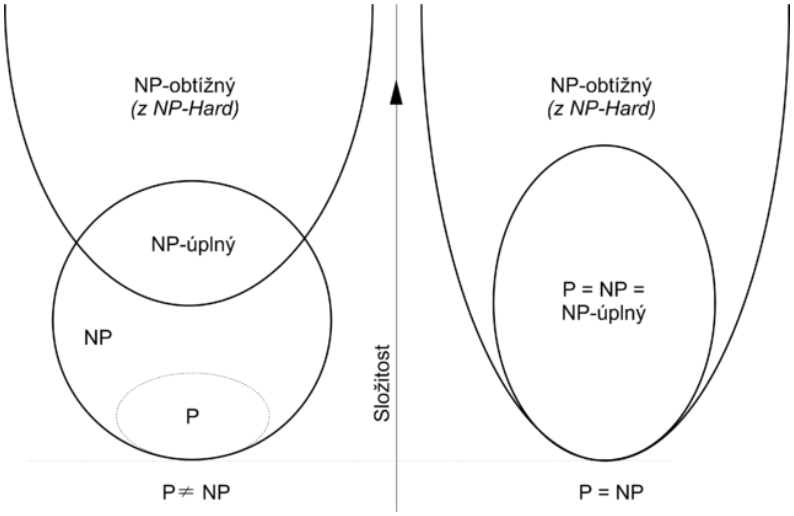
Následně řadíme i úlohy do tříd složitosti (viz obr. 2.9). Mezi neznámější třídy složitosti patří třídy P a NP.
Stručné definice čtyř příslušných tříd složitosti řešeným problémů:

Třída P obsahuje problémy, které lze vyřešit v polynomiálním čase. Obsahuje tedy problémy, které lze účinně vyřešit.

Třída NP obsahuje problémy, které lze ověřit v polynomiálním čase. Například problém obchodního cestujícího. Správné řešení lze zkontrolovat (ověřit) poměrně snadno, ale skutečné řešení může trvat nekonečně dlouho.

Třída NP-složité obsahuje problémy, které nelze v polynomu ani ověřit (zkontrolovat). Úlohy lze redukovat na NP v polynomiálním čase.

Třída NP-úplný obsahuje problémy skutečně velmi obtížné. Pokud existuje rychlejší způsob řešení NP-úplný, pak se NP-úplný stává P.



Obrázek 2.9: Eulerův diagram

Zdroj: Commons Wikipedia `https://shorturl.at/fkrGN`

Kapitola 3

Řešení problému čínského listonoše (metody a algoritmy)

CPP je optimalizační úloha. Při řešení problému CPP 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 k těm které byly použity v implementaci (viz kap. 4).

Jedná se o následující algoritmy:

- hledání nejkratší cesty mezi vrcholy (Dijkstrův algoritmus, Floyd-Warshallův algoritmus) (viz kap. 3.1)
- algoritmy řešící přiřazovací problém (Maďarská metoda) (viz kap. 3.2)
- toky v sítích (Ford-Fulkersonův algoritmus, Max-Flow Min-Cost) (viz kap. 3.3)
- hledání eulerovského cyklu (Hierholzerův algoritmus) (viz kap. 3.4)

Pro jednotlivé algoritmy (resp. metody) je uvedeno:

- princip algoritmu, základní vlastnosti
- základní věty a důkaz korektnosti algoritmu (resp. odkaz na důkaz)
- 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 nevyváže-

nými 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 vybrané algoritmy, konkrétně Floyd-Warshallův algoritmus (hledání nejkratší cesty v grafu mezi všemi dvojicemi vrcholů grafu) a Dijkstrův algoritmus (z daného vrcholu grafu do všech ostatních vrcholů grafu).

3.1.1 Dijkstrův algoritmus

Jde o jeden z nejznámějších algoritmů pro hledání nejkratších cest. Uvedl ho Edsger Dijkstra v roce 1959. [17]

Dijsktrův algoritmus [16, s. 146-148][4, přednáška 12] slouží k nalezení nejkratší cesty v grafu. Je konečný, protože v každém průchodu cyklu 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 kladně ohodnocenými hranami.

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. Vlna se však šíří na základě vzdálenosti od výchozího vrcholu. Vlna tedy zpracovává jen vrcholy, ke kterým již byla nalezena nejkratší cesta.

Algoritmus 1: Dijkstrův algoritmus

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

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

Důkaz korektnosti lze najít v [2, s. 32-33]

Složitost [16, 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 Floyd-Warshallův algoritmus

Floyd-Warshallův algoritmus [16, s. 154-155][5, kap. 16] zpracovává orientovaný i neorientovaný graf. Graf může obsahovat i hrany záporné délky (nikoliv záporné cykly). Algoritmus porovnává všechny možné cesty v grafu mezi všemi dvojicemi vrcholů. U cest stejné délky vybírá tu s nejmenším počtem hran. Pracuje tak, že postupně vylepšuje odhad na nejkratší cestu. Končí tak odhadem optimálním.

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

Popis algoritmu: Na vstupu je matice délek. 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.

Algoritmus 2: Floyd-Warshallův algoritmus

Input: Graf G
Output: 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] = v;
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[v] ← next[i][k];
11 |  |  |  end
12 |  |  end
13 |  end
14 end
15 return dist, next
```

Důkaz korektnosti lze najít v [16, s. 154-155]

Složitost[16, 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). [18]

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 hodnocení 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 hran. V opačném případě jde o nevyvážené přiřazení.

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 n. Jedním z nich je i **maďarská metoda** (pro řešení vyváženého přiřazovacího problému).

3.2.1 Maďarská metoda

Maďarská metoda [19, 20] kombinuje prostředky lineárního programování a teorie grafů. Je nazvána podle prací maďarských vědců J. Egerváryho a Denese Koeniga, jejichž práci využil a zpracoval H.W. Kuhn. Představil ji ve své stati [18] a nazval maďarskou metodou.

Metoda nabízí algoritmus, který redukuje obecný přiřazovací problém na problém dvou stavů, 0 a 1. Problémy, které jsou jako úloha lineárního programování příliš složité při velkém objemu dat, se zjednoduší nasazením prostředků teorie grafů do stavu, který umožňuje nalézat řešení.

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, horizontální řady představují pracovníky a vertikální 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.

Popis jádra metody (pro čtvercovou matici):

Prvek v i-tém řádku a j-tém sloupci značí např. náklady (cenu práce) j-tého úkolu i-tým pracovníkem.

Algoritmus 3: Maďarská metoda

Input: Matice párování
Output: Optimální párování této matici

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 minimálního počtu horizontálních a vertikálních čar;
4 **Test optimality**: jestliže minimální počet označených čar je n, pak optimální přiřazení je možné a je konec. 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. [21] Přidal algoritmické řešení hledání minimálního počtu čar, které pokrývají všechny

nuly a maximální množiny nezávislých nul. Viz algoritmus 4.

Algoritmus 4: Munkresova maďarská metoda

```
   Input: Matice párování
   Output: 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;
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;
24 |  |  |  Star Z;
25 |  |  |  while there exists another already starred zero Z* in the column of Z do
26 |  |  |  |  Unstar Z*;
27 |  |  |  |  Rename the primed zero in the row of Z*, 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 [21].

Složitost[22]: původní Munkresův algoritmus má složitost $O(n^4)$. V článku [22] je prezentováno vylepšení algoritmu na $O(n^3)$. Viz také dále v kapitole implementace 4.2.5.2.

3.3 Toky v sítích

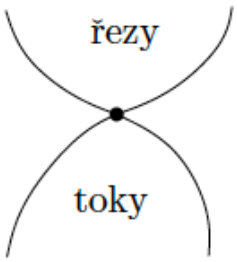
Přiřazovací problém lze převést na problém hledání maximálního toku v sítích (angl. Network Flow) [23] [16, kap. 14], konkrétně na nalezení maximálního toku s minimální cenou v bipartitním ohodnoceném grafu (angl. Max-Flow Min-Cost Bipartite Weighted Graph).

3.3.1 Teorie

Nenasycené cestě ze zdroje do stoku říkáme také zlepšující cesta (hlavně v popisu algoritmů). Tok nazveme nasycený, když každá cesta ze zdroje do stoku je nasycená. Nasycená cesta je tedy taková, podél níž se tok nedá zvětšit. Cestě, která není nasycená, říkáme nenasycená. Tok je maximální, právě když je nasycený.

Hlavní věta o tocích: Pro každou síť se velikost maximálního toku rovná kapacitě minimálního řezu:

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



Obrázek 3.1: Věta o tocích

Zdroj: [5]

Důkaz této věty lze najít v [5, kap. 8, s. 13-15]

3.3.2 Ford-Fulkersonův algoritmus

Ford-Fulkersonův algoritmus [16] 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: Ford-Fulkersonův algoritmus

Input: Graph G with flow capacity c, a source node s, a sink node t
Output: Maximum flow f from s to 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 residual 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 (angl. Breadth-First Search), nazývá se algoritmus **Edmonds-Karpův**.

Složitost [5, kap. 8, s. 16]: Edmonds-Karpův algoritmus má složitost $O(|V||E|^2)$.

3.3.3 Max-Flow Min-Cost bipartitní ohodnocený graf

Nejprve se vytvoří potřebný graf, ve kterém se hledá maximální tok (Max-Flow) s minimální cenou (Min-Cost). [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. V takto sestrojen síti již lze

najít optimální párování.

Algoritmus 6: Max-Flow Min-Cost algoritmus

Input: Bipartite Graph G with flow capacity c, a source node s, a sink node t
Output: Optimal matching 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) is calculated length of a shortest path from
   |  |     s to 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
12 |  Update matching M via path p;
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 cyklu

K nalezení eulerovského cyklu (viz kap. 2.1.3) využívám Hierholzerův algoritmus.

3.4.1 Hierholzerův algoritmus

Hierholzerův algoritmus [5] 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

Input: Eulerovský graf G
Output: 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;
```

Důkaz korektnosti lze najít v [5, kap. 20, s. 48-49].

Složitost [5, 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.

Kapitola 4

Implementace řešení DCPP

4.1 Použité prostředky

Popisuji využívaný programovací jazyk implementace a prostředky používané při paralelizaci řešení. Odlišuji sekvenční a paralelní provedení.

4.1.1 Programovací jazyk implementace C++

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 eistuje také mnoho vhodných knihoven, které využívám pro ulehčení implementace.

4.1.2 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.

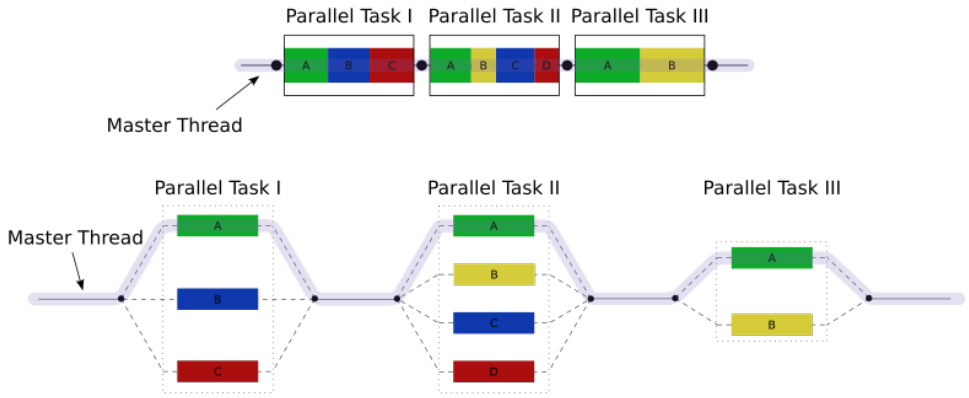
4.1.3 Paralelní provedení

Jednou z hlavních náplní mé implementace je využití paralelizace [26]. To znamená možnost využití více vláken najednou na vhodných místech algoritmů. Důsledkem je (ideálně) výrazné zrychlení výsledného algoritmu.

Na toto paralelní provedení využívám knihovnu OpenMP [27, 28]. 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, prováděna a ukončována vlákna. V ostatních částech je pouze hlavní vlákno.



Obrázek 4.1: Fork-join schéma

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

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

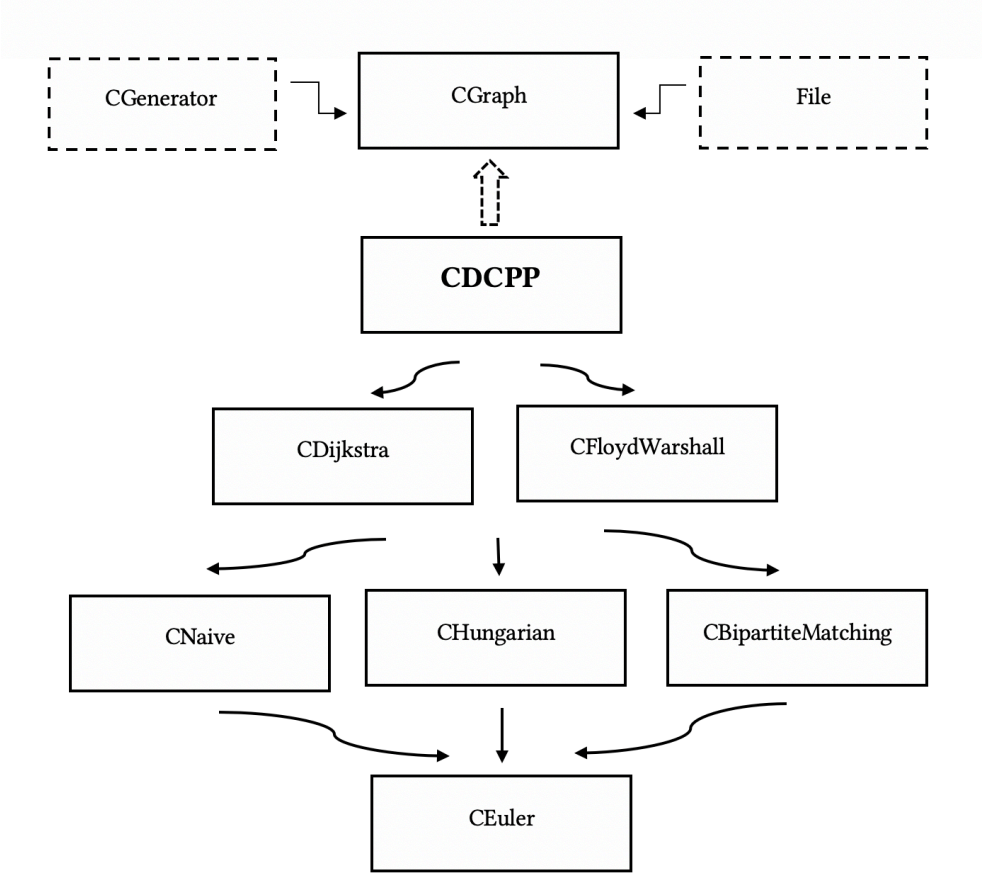
Paralelizace lze rozdělit na dva základní druhy přístupy:

- **datový paralelismus** - prvky větší datové struktury jsou rovnoměrně rozděleny mezi jednotlivá vlákna
- **funkční paralelismus** - program je rozdělen na jednotlivé kusy kódu, které se dají paralelizovat

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 následujícím obrázku je znázorněno hrubé schéma mého řešení DCPP.



Obrázek 4.2: Schéma řešení DCPP

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 algoritmy 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*, která drží instanci předem zvolené podtřídy. Tato třída vypočítá nejkratší vzdálenosti potřebných vrcholů a uloží je do matice vzdáleností.

Posléze se tato matice pošle instanci třídy *CAssignment* (stejně jako třída *CPathFinding* drží instanci zvolené podtřídy), která v této matici 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 $O(n^3)$.

4.2.2 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 všech vrcholů,

m_Graph pole listů držící všechny hrany v grafu.

Seznam metod:

NewGraph Vytvoří nový graf ze souboru,

AddEdge přidá do grafu určitou hranu,

PrintGraph vypíše graf ve formátu, ze kterého může vytvořit nový graf.

Třída *CGraph* drží informace o grafu. Graf je uložen jako pole spojových seznamů pomocí struktury list. Hrana je reprezentována pomocí struktury *struct* s atributy *name* a *weight*.

Vzhledem k tomu že nevíme, jak moc bude zadaný graf řídký, uložení grafu jako matice sousednosti by nemuselo být časově ani datově výhodné. List nám přináší konstantní složitost při vkládání hran. V programu se nevyužívá hledání určitého vrcholu, takže pomalost listu v tomto ohledu nevadí.

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.

4.2.3 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, aby graf tvořil silně souvislý graf,

CreatingEdge přidají se do silně souvislého grafu 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.

Na vytvoření silně souvislého grafu se využívá modifikovaný Tarjanův algoritmus. [29]

Nejprve vytvoří kostru grafu postupným přidáváním vrcholu hranou k již existující množině vrcholů.

Dále se spustí modifikovaný Tarjanův algoritmus. Při detekci nové silně souvislé komponenty se místo ní přidá hrana, která zabrání vzniku nové komponenty.

V poslední fázi se přidávají náhodné hrany do počtu zadaného uživatelem. 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.

Vytvářející 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.

4.2.4 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_oddIndexRow vektor vrcholů tvořící indexy řádků výsledné matice délek nejkratších cest,

m_oddIndexCol vektor vrcholů tvořící indexy sloupců výsledné matice délek nejkratších cest.

Seznam metod:

Calculate Abstraktní metoda,

FindPathFromIndex přiřadí k dvěma indexům z výsledné matice odpovídající vrcholy z grafu a nejkratší cestu mezi nimi,

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

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

V metodě *FillOddIndex* se do atributů *m_oddIndexRow* a *m_oddIndexCol* vloží názvy vrcholů tvořící indexy výsledné matice vzdáleností. Ve vektoru *m_oddIndexRow* se každý vrchol v se zápornou δ opakuje $|\delta(v)|$-krát. To samé v *m_oddIndexCol* s vrcholy s kladnou δ.

Metoda *FindPathFromIndex* vypočítá jaké vrcholy patří zadaným indexům z výsledné matice vzdáleností a vrátí hrany patřící do odpovídající nejkratší cesty.

4.2.4.1 Subclass CDijkstra

Seznam atributů:

m_startPoints Pole vrcholů z množiny D^-.

Seznam metod:

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

Algorithm Výpočet Dijkstrova algoritmu pro konkrétní vrchol s pamatováním předchůdců v cestě,

CreateOddMatrix vytváří potřebnou matici vzdáleností z naměřených hodnot.

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 na každý vrchol z *m_startPoints* volá metoda *Algorithm*, která 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 *CreateOddMatrix* vytvoří matice vzdáleností nevyvážených vrcholů pomocí atributů *m_oddIndexRow* a *m_oddIndexCol*.

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 ne vícekrát než $O(|V|)$. Vytvoření výsledné matice vzdáleností má složitost $O(|V|^2)$.

Celkově má tedy tento výpočet složitost $O(|V|+|V||E|+|V|^2log|V|+|V|^2)$, což pro nejhorší možnost (tj. úplný graf) je $O(|V|^3)$. Avšak pro řídké grafy může být blíže k $O(|V|^2log|V|)$. Navíc při menší početnosti vrcholů v D^- je složitost výrazně menší.

4.2.4.2 Subclass CFloydWarshall

Seznam metod:

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

Algorithm samotný výpočet Floyd-Warshallova algoritmu s pamatováním následujícího vrcholu v každé cestě.

Tato třída řeší problém nalezení nejkratších cest za pomoci Floyd-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. Ve výpočtu se přidá příkaz $m\_pathMatrix(i)(j) = m\_pathMatrix(i)(k)$ za příkaz porovnání. 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 Floyd-Warshallův algoritmus.

V konečném kroku se metodou *CreateOddMatrix* vytvoří matice vzdáleností nevyvážených vrcholů pomocí atributů *m_oddIndexRow* a *m_oddIndexCol*.

První fáze výpočtu má asymptotickou složitost $O(n^2)$, protože vyplňujeme matice. Samotný algoritmus má složitost $O(n^3)$. Poslední krok vytvoření výsledné matice má opět $O(n^2)$. Dohromady má tento výpočet složitost $O(n^2 + n^3 + n^2)$, což je $O(n^3)$.

4.2.5 Class CAssignment

Seznam atributů:

m_matrix Matice párování.

Seznam metod:

Calculate Abstraktní metoda.

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

4.2.5.1 Subclass CNaive

Seznam atributů:

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 strom.

Algoritmus postupně zkouší všechny možné permutace. Což dává asymptotickou složitost $O(n!)$.

V každém volání se vytvoří nové pole, do kterého se nakopíruje staré pole a nová informace na určité místo v poli. Na každé úrovni vnoření se ukládá do pole určité číslo na různé pozice v poli.

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 přepíše. Druhá část podmínky je, když je lokální řešení již v průběhu výpočtu horší. Tak se současné řešení ukončí.

4.2.5.2 Subclass CHungarian

Seznam atributů:

m_step Udává následující krok algoritmu,

m_mask matice držící informaci o typu prvku na dané pozici,

m_col pole určující informaci, zdali je daný sloupec kryt,

m_row pole určující informaci, zdali je daný sloupec 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 *Step6*,

m_colAdjust pole úprav slouců matice z metody *Step6*.

Seznam metod:

Calculate Řídí posloupnost kroků algoritmu pomocí atributu *m_step* a příkazu *switch*,

Step1 odčítá od řádků a sloupců maximum odpovídajícího řádku či sloupce,

Step2 označuje dosud nepokryté nuly,

Step3 součet počtu označených nul,

Step4 obsahuje *while* cyklus, který probíhá, dokud je v matici nepokrytá nula,

FoundNoncovZero výpočet v případě nalezení nepokryté nuly,

Step5 provádí se, když na řádku nepokryté nuly není označená nula,

Step6 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 $O(n^3)$.

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

Algoritmus je rozdělen do šesti hlavních metod Step1 až Step6 a metody *Calculate*.

Původní Mukresova implementace má asymtotickou složitost $O(n^4)$. Ta je způsobena výpočtem modifikace matice probíhajícího v metodě *Step6*. Tento výpočet má složitost $O(n^2)$ a může proběhnout nejvíce n^2-krát. Jak je vidět z článku [22] 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*.

4.2.5.3 Subclass CBipartiteMatching

Seznam atributů:

m_source *Set* držící nevyužité hrany vedoucí ze zdroje,

m_sink *vector* držící nevyužité hrany vedoucí do stoku.

Seznam metod:

Calculate Spouští samotný výpočet,

Dijkstra spouští Dijkstrův algoritmus hledající zlepšující cestu,

Recalibrating přepočítání matice, aby nevznikly žádné záporné hrany,

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 matici nejkratších vzdáleností M a vrací seznam optimálního párování.

Nejprve se vytvoří potřebný bipartitní graf $G = (X \cup Y, E)$ se zdrojem a stokem.

Hrany vedené ze zdroje do parity X obsahující vrcholy se záporným stupněm jsou uloženy do atributu *m_source*. Na atribut *m_source* je využita struktura *set*. Využívá se rychlé mazání hrany, kterou využila zlepšující cesta a zároveň není potřeba konstantní nalezení této hrany, které by jiné struktury nabídly.

Naopak hrany vedené z parity Y do stoku je třeba najít rychle, a tak se využije struktura *vector*.

Ceny hran mezi paritami jsou reprezentovány v matici M. I-tý řádek odpovídá i-tému vrcholu z parity X a j-tý sloupec odpovídá j-tému vrcholu z parity Y. Vzniklé hrany z parity Y do X mají vždy cenu 0. Jsou označené v matici jako -1, pro rozpoznání od hran z parity X.

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

Dále se přepočítává matice pomocí vzorečku $M(i,j) = M(i,j)+d(u)-d(v)$, 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 $O(n^2)$.

V konečném kroku musíme invertovat jednotlivé hrany ze zlepšující cesty, vymazat využitou hranu ze zdroje, aktualizovat momentální optimální párování. To vše lze zvládnout lineárně.

Tyto kroky výpočtu se provádějí n-krát, kde n je počet vrcholů v jedné paritě.

Celková složitost algoritmu je pak $O(|V|^2 log|V| + |V||E| + |V|^3 + |V|^2)$. Což nám dává $O(n^3)$.

4.2.6 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 vždy přijatý graf eulerovský (viz kap. 2.1.3).

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

Celkový výpočet má tedy asymptotickou složitost $O(m)$ vzhledem k počtu hran.

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 $O(n)$ vzhledem k počtu hran 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 největší časovou složitost smyčka, která provádí jednotlivé výpočty Dijkstrova algoritmu. My však nemusíme tyto výpočty provádět sekvenčně, jelikož na sobě nezávisí. 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. To by se však mohlo uplatnilo pouze v případě, kdy máme větší počet vláken než kolikrát počítáme Dijkstrův algoritmus. To však průměrně nastane jenom v menších grafech, kde paralelizace není tak užitečná.

4.3.2 Třída CFloydWarshall

Ve třídě má největší časovou složitost samotný Floyd-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.

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 *Step4*, kde v cyklu hledáme nepokryté nuly).

4.3.4 Třída CBipartiteMatching

Ve třídě *CDijkstra* má největší časovou složitost smyčka, která v každé iteraci hledá zlepšující cestu, přepočítává hodnoty matice a aktualizuje momentální párování.

Aktualizace párování má pouze lineární časovou složitost. Hledání zlepšující cesty a přepočítávání matice mají však asymptotickou složitost $O(n^2)$.

Na přepočítávání matice lze využít datovou paralelizaci za použití direktiv *parallel* a for. Výpočet uvnitř cyklů je konstantní, proto můžeme využít statické přidělení iterací.

Hledání zlepšující cesty je řešeno Dijkstrovým algoritmem. Jelikož si graf ukládáme do matice, kde hrany vrcholu jsou v řádku či sloupci, můžeme procházené hrany rozdělit mezi vlákna. Na to použijeme datovou paralelizaci pomocí direktiv *parallel* a for. Výpočet pro každou hranu je konstantní. Proto opět využijeme statické přidělení iterací.

Přidávání hran do prioritní fronty musíme dát do kritické sekce pomocí direktivy *critical*.

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 spojitý 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 spojitost 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 pak celková délka tohoto 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 algoritmu řešícímu část programu hledání nejkratších cest. Na třetím řádku číslo odpovídající algoritmu řešícímu přiřazovací problém.

Kapitola 5

Testování

Kapitola navazuje na předchozí kapitolu. Věnuje se 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 algoritmu 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 -fopenmp*.

5.2 Způsob testování

Pro generování vstupních grafů se používá generátor náhodných silně souvislých grafů, třída *CGenerator* (viz kap. 4.2.3). Pro generování náhodných čísel v maticích (pro testování tříd *CHungarian*, *CBipartiteMatching* a *CNaive*) se používá standardní knihovna *random*.

Testování lze rozdělit do tří částí:

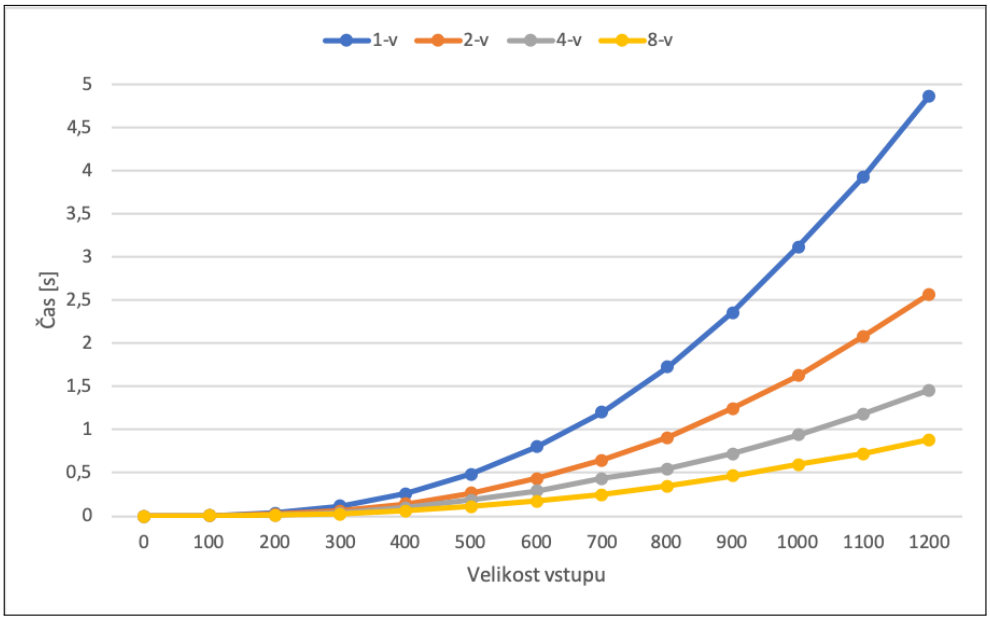
- testování jednotlivých tříd s porovnáním sekvenčního provedení s paralelním
- testování se záměrem porovnat třídy řešící stejný problém

- testování kompletní třídy *CDCPP*

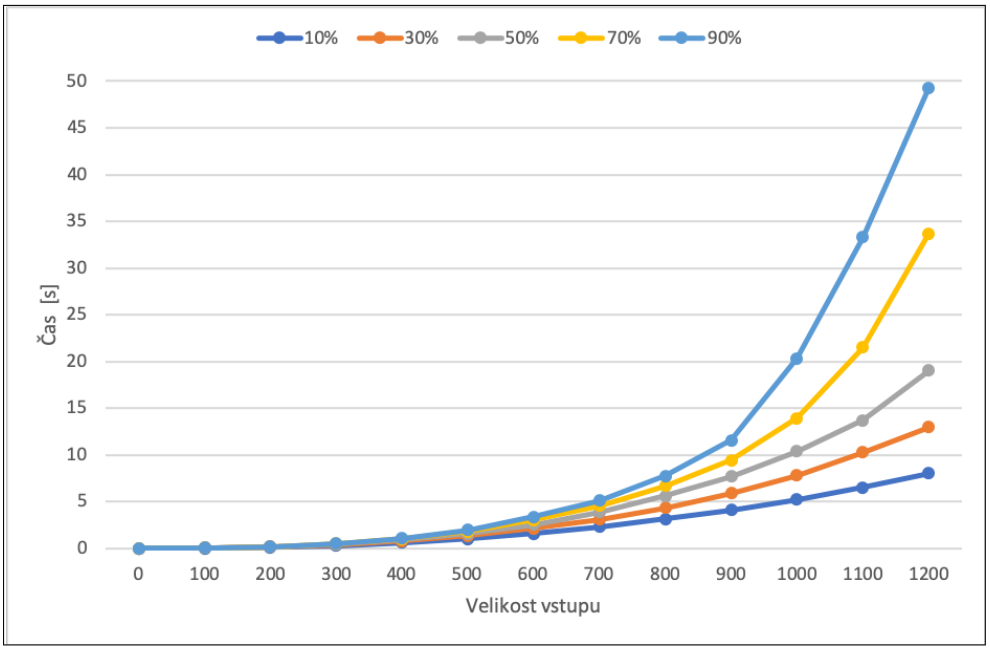
5.3 Testování třídy CDijkstra

V této třídě Dijkstrův algoritmus probíhá tolikrát, kolik je vrcholů v množině D^- (tzn. vrcholy se zápornou δ)(viz kap. 2.3.3). Generátor generuje grafy, které mají průměrně něco méně než polovinu všech vrcholů v množině D^-. Lze očekávat, že s rostoucí množinou D^- bude klesat efektivita výpočtu této třídy.

Na grafu 5.1 jsou znázorněny testy s různým počtem vláken. Na vstupu byl velmi řídký graf (konkrétně 1% z maximálního počtu hran). Na grafu 5.2 lze vidět jedno-vláknové testy s různými hustotami vstupního grafu.



Obrázek 5.1: Testování třídy *CDijkstra* s různým počtem vláken



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

Oba testy potvrzují, že s rostoucí počtem vrcholů ve vstupním grafu roste časová náročnost výpočtu.

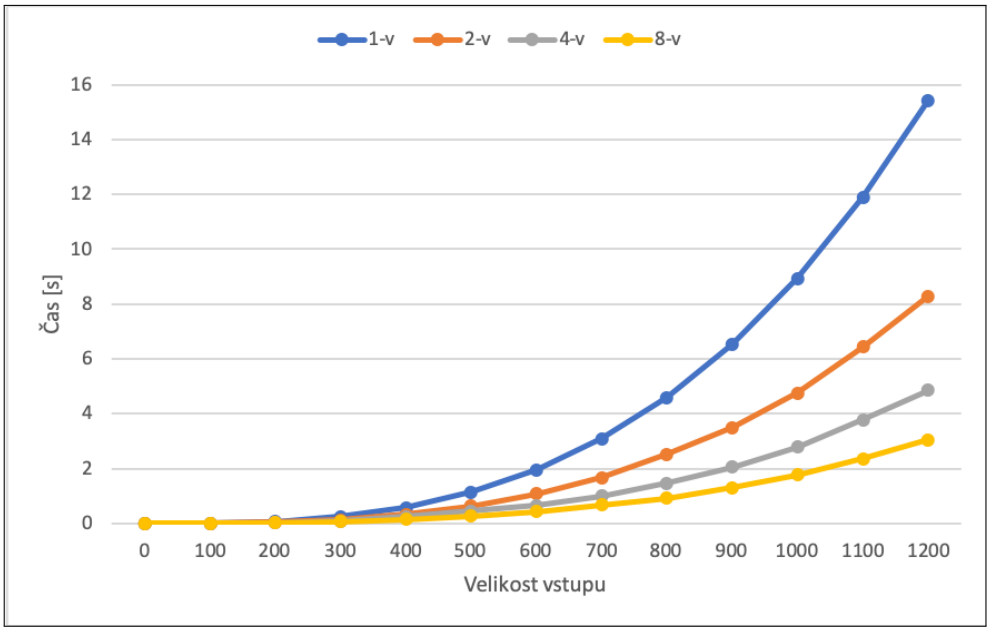
Testy s různým počtem vláken potvrzují efektivitu použití více vláken.

Testy s různou hustotou hran potvrzují závislost asymptotické složitosti Dijkstrova algoritmu na počtu hran.

5.4 Testování třídy CFloydWarshall

Nemusíme se zabývat otázkou, z kolika vrcholů musíme počítat nejkratší vzdálenost, protože Floyd-Warshallův algoritmus počítá nejkratší vzdálenosti vždy ze všech vrcholů. Stejně tak nezáleží na počtu hran.

Na grafu 5.3 jsou znázorněny testy s různým počtem vláken.



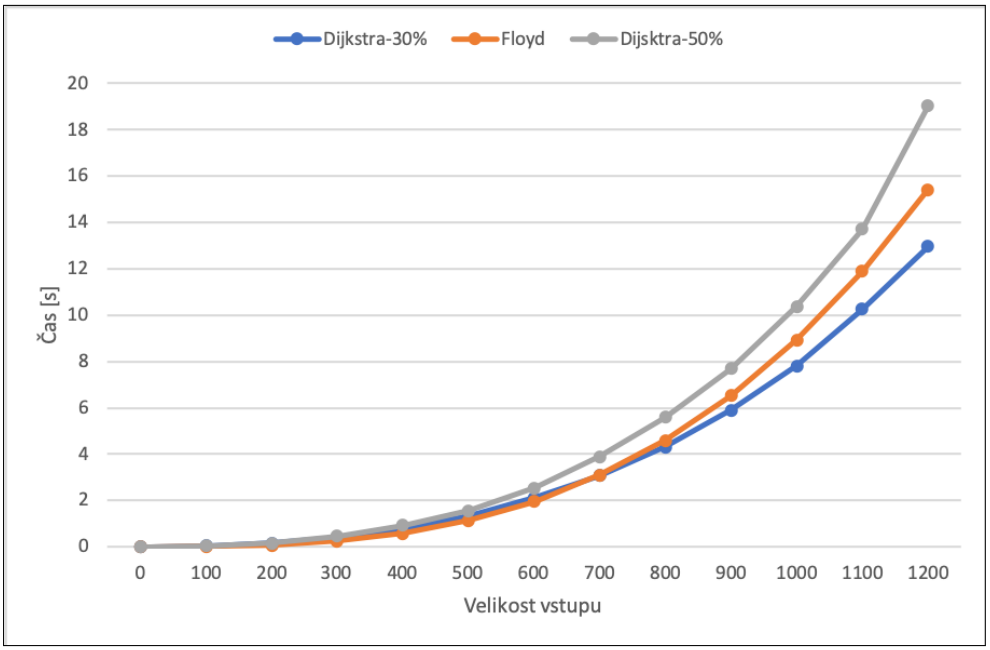
Obrázek 5.3: Testování třídy *CFloydWarshall* s různým počtem vláken

Testy s různým počtem vláken potvrzují efektivitu použití více vláken.

5.5 Porovnání tříd CDijkstra a CFloydWarshall

Následující test (jedno-vláknové provedení) porovnává efektivitu použití tříd *CDijkstra* a *CFloydWarshall*. Snažíme se zjistit od jaké hustoty hran v grafu je časově výhodnější využívat třídu *CFloydWarshall*.

Tento test se vztahuje na vygenerované grafy s průměrně něco méně než 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 je výhodnější použít třídu *CFloydWarshall*.



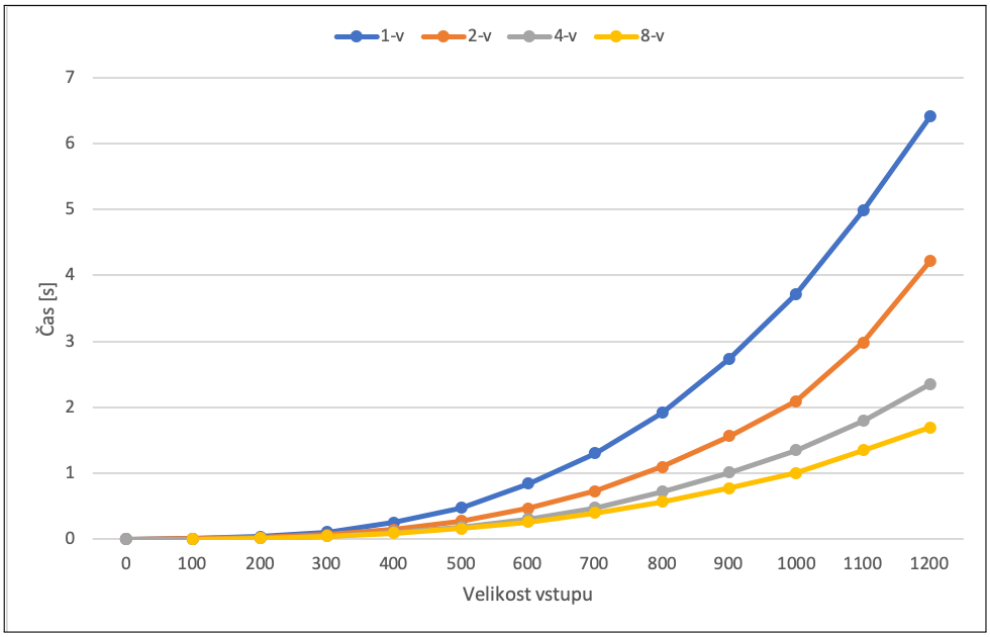
Obrázek 5.4: Jedno-vláknové porovnání tříd *CDijkstra* (30%), *CFloydWarshall* a *CDijkstra* (50%)

Z testu vyplynulo, že z hlediska časové složitosti je výhodnější používat třídu *CFloydWarshall* od hustoty hran ve vstupním grafu mezi 30% a 50%.

5.6 Testování třídy CHungarian

V této třídě je na vstupu matice s náhodně generovanými kladnými hodnotami.

Na grafu 5.5 jsou znázorněny testy s různým počtem vláken.



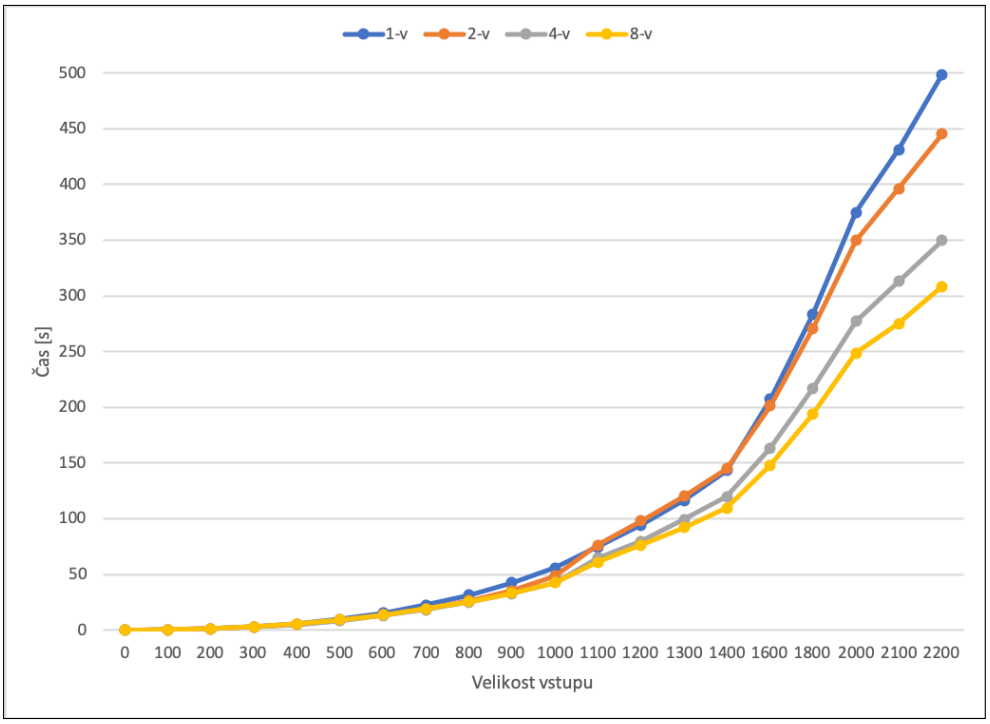
Obrázek 5.5: Testování třídy *CHungarian* s různým počtem vláken

Testy s různým počtem vláken potvrzují efektivitu použití více vláken.

5.7 Testování třídy CBipartiteMatching

V této třídě je na vstupu matice s náhodně generovanými kladnými hodnotami.

Na grafu 5.6 jsou znázorněny testy s různým počtem vláken.



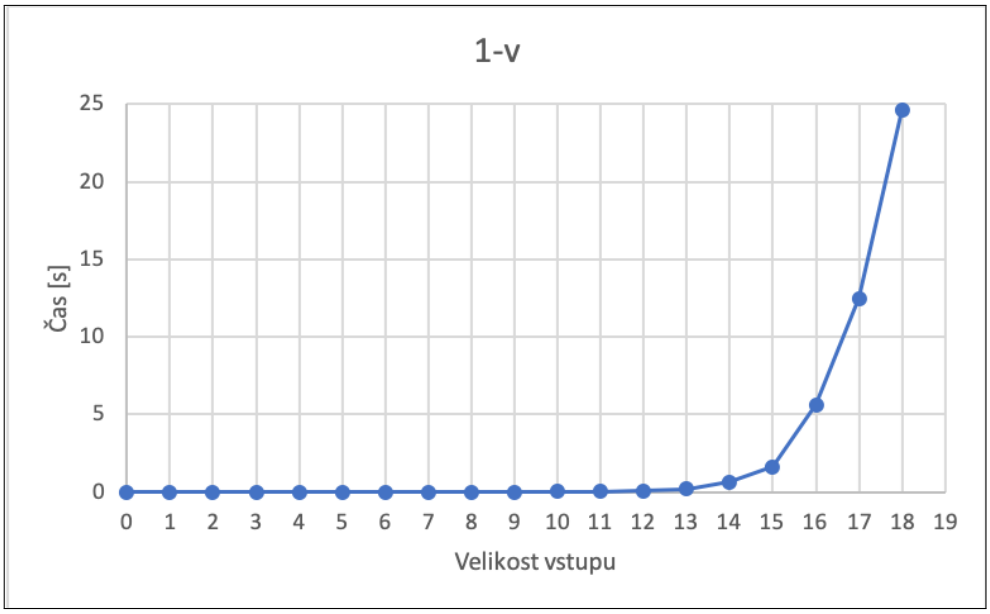
Obrázek 5.6: Testování třídy *CBipartiteMatching* s různým počtem vláken

Z testu vyplývá, že na rozdíl od testů předchozích tříd je paralelizace výhodná až od větší velikosti vstupu.

Jistou hrubost grafu lze vysvětlit menším počtem testovacích případů vzhledem k vysoké časové náročnosti algoritmu.

5.8 Testování třídy CNaive

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

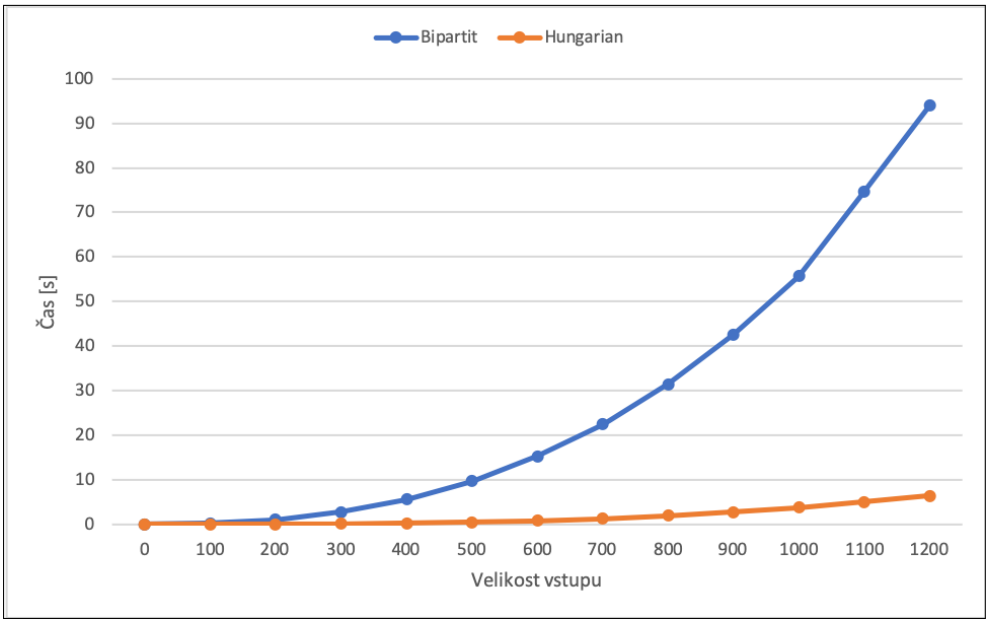


Obrázek 5.7: Jedno-vláknové testování třídy *CNaive*

5.9 Porovnání tříd CHungarian a CBipartiteMatching

Na vstupu máme matice s náhodně generovanými kladnými hodnotami.

Na grafu 5.8 je jedno-vláknové porovnání tříd *CBipartiteMatching* a *CHungarian*.



Obrázek 5.8: Jedno-vláknové porovnání tříd *CBipartiteMatching* a *CHungarian*

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

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

5.10 Testování třídy CDCPP

V této podkapitole se testuje kompletní implementace DCPP.

Na základě hustoty hran vstupních grafů se testují tři varianty. Volba třídy se určuje podle výsledků předchozích dílčích testů.

Na přiřazovací problém se nasadí třída *CHungarian* pro všechny varianty.

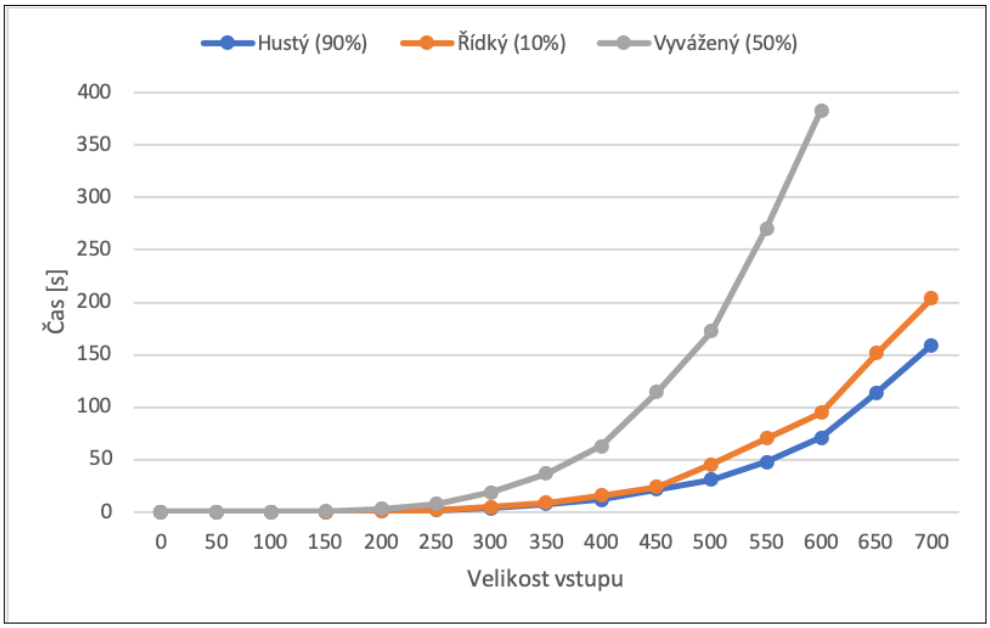
Na hledání nejkratších cest se nasadí třída *CFloydWarshall* pro hustoty hran 90% (vstupního grafu) a 50% a třída *CDijkstra* pro hustotu hran 10%.

Na hledání eulerovského cyklu se aplikuje třída *CEuler*.

Varianta 1 Hustota hran ve vstupním grafu je 90%. Vybrané třídy: *CFloydWarshall*, *CHungarian* a *CEuler*.

Varianta 3 Hustota hran ve vstupním grafu je 10%. Vybrané třídy: *CDijkstra*, *CHungarian* a *CEuler*.

Varianta 2 Hustota hran ve vstupním grafu je 50%. Vybrané třídy: *CFloydWarshall*, *CHungarian* a *CEuler*.



Obrázek 5.9: Jedno-vláknové porovnání *CDCPP* s různými hustotami hran vstupního grafu

Na obrázku 5.9 je vidět, že časová složitost výpočtu pro vyvážený (50%) graf je značně horší než pro ostatní varianty.

Důvod tohoto chování lze vidět v tabulce na obrázku 5.10. Jak je vidět, průměrná velikost matice hledaného optimálního párování roste výrazně rychleji pro vyvážené grafy než pro jiné varianty.

Velikost vstupu	50	100	150	200	250	300	350	400	450	500	550	600	650	700
10(90)%	55	150	300	470	660	850	1100	1350	1600	1900	2150	2500	2800	3100
50%	100	280	500	800	1100	1400	1800	2300	2700	3150	3600	4100	4700	5250

Obrázek 5.10: Tabulka průměrných velikostí matic optimálního 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.

5.11 Shrnutí testování

Testování ukázalo, že volba efektivnější metody v případě tříd pro hledání nejkratších cest závisí na hustotě hran vstupního grafu. (viz obr. 5.4)

V případě přiřazovacího problému ukázalo, že je časově výhodnější použít třídu *CHungarian* (viz obr. 5.8).

V případě celého DCPP ukázalo, že výpočet pro vstupní graf s 50% hustotou hran je průměrně časově náročnější než výpočet pro řídší či hustší graf (viz obr. 5.9).

Co se týká paralelizace, testování ukázalo, že pro třídy *CDijkstra* (viz obr. 5.1), *CFloydWarshall* (viz obr. 5.3) a *CHungarian* (viz obr. 5.5) je efektivní již od menších velikostí vstupu (řádově stovky) a pro třídu *CBipartiteMatching* (viz obr. 5.6) až vstup v řádu tisíců.

5.12 Test správnosti implementace

Nepodařilo se porovnat implementovanou realizaci s jinou, která by plně vyhovovala. Omezením bylo zejména to, že se má jednat o variantu DCPP implementovanou v c++. Nalezené implementace byly nevyhovující (UCPP, java, python).

Na webu *Postman Problem* (`https://www-m9.ma.tum.de/graph-algorithms/directed-chinese-postman/index_en.html`) je k dispozici online aplikace pro řešení zadané úlohy DCPP.

Tato aplikaci umožňuje kontrolu správnosti výsledků implementace, ale neumožňuje měření efektivnosti algoritmu. Nedává k dispozici žádné časové údaje.

Porovnávané výsledky souhlasily.

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 studium stávajících teorií (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.

Další možnosti rozšíření práce by mohlo být např. porovnání s jiným už existujícím řešením DCPP, 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. Dostupné z: https://www.math.uni-bielefeld.de/documenta/vol-ismp/16_groetschel-martin-yuan-ya-xiang.pdf

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

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

[4] Suchý, O.; Valla, T.: 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

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

[6] Lu, L.: Graph Theory I Lecture note 3. 2005. Dostupné z: http://people.math.sc.edu/lu/teaching/2005fall_776/lecture3.pdf

[7] Grötschel, M.; Yuan, Y.-x.: Euler, mei-ko kwan, königsberg, and a chinese postman. *Optimization Stories*, 2012: str. 43. Dostupné z: https://www.math.uni-bielefeld.de/documenta/vol-ismp/16_groetschel-martin-yuan-ya-xiang.pdf

[8] Šišma, P.: Leonhard Euler. *Rozhledy matematicko-fyzikální*, ročník 82, č. 4, 2007: s. 21–32. Dostupné z: https://dml.cz/bitstream/handle/10338.dmlcz/146218/Rozhledy_082-2007-4_5.pdf

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

[10] Hopkins, B.; Wilson, R. J.: The truth about Königsberg. *The College Mathematics Journal*, ročník 35, č. 3, 2004: s. 198–207. Dostupné z: https://www.maa.org/sites/default/files/pdf/upload_library/22/Polya/hopkins.pdf

[11] Gordenko, M. K.; Avdoshin, S. M.: The mixed chinese postman problem. *Works of the Institute of System Programming RAN*, ročník 29, č. 4, 2017. Dostupné z: http://www.mathnet.ru/links/a6bcd8723ba1c1b7f10c96e28dda771c/tisp238.pdf

[12] 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. Dostupné z: https://pubsonline.informs.org/doi/pdf/10.1287/opre.43.2.231

[13] Guan, M.: On the windy postman problem. *Discrete Applied Mathematics*, ročník 9, č. 1, 1984: s. 41–46. Dostupné z: https://core.ac.uk/reader/82387012

[14] Thimbleby, H.: The directed chinese postman problem. *Software: Practice and Experience*, ročník 33, č. 11, 2003: s. 1081–1096. Dostupné z: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.106.4691&rep=rep1&type=pdf

[15] Černý, J.: *Základní grafové algoritmy*. České vysoké učení technické, 2013. Dostupné z: https://docplayer.cz/4802558-Zakladni-grafove-algoritmy.html

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

[17] Dijkstra, E. W.; aj.: A note on two problems in connexion with graphs. *Numerische mathematik*, ročník 1, č. 1, 1959: s. 269–271.

[18] KUHN, H. W.: The Hungarian method for the assignment problem. *Naval research logistics quarterly*, 1955. Dostupné z: https://tom.host.cs.st-andrews.ac.uk/CS3052-CC/Practicals/Kuhn.pdf

[19] Přiřazovací problém. Dostupné z: https://www.turnovfree.net/~stybla/skola/czu/tretak/sam/c/mat/ostatni/prirazovaci_problem.pdf

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

[21] 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. Dostupné z: https://web.eecs.umich.edu/~pettie/matching/Munkres-variant-of-Hungarian-alg.pdf

[22] 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. Dostupné z: https://link.springer.com/article/10.1007/BF01930994

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

[24] Zwick, U.: Analysis of Algorithms. 2005/2006. 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*. Dostupné z: http://wwwmayr.in.tum.de/lehre/2002WS/algoprak/part6.ps.gz

[26] Parhami, B.: *Introduction to parallel processing: algorithms and architectures*. New York: Plenum Press, 1999. Dostupné z: https://bit.ly/2UyYftV

[27] OpenMP: OpenMP. 2019. Dostupné z: https://www.openmp.org/specifications/

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

[29] Maurer, P. M.: Generating strongly connected random graphs. In *Proceedings of the International Conference on Modeling, Simulation and Visualization Methods (MSV)*, The Steering Committee of The World Congress in Computer Science, Computer ..., 2017, s. 3–6. Dostupné z: https://csce.ucmss.com/cr/books/2017/LFS/CSREA2017/MSV3359.pdf

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

Příloha A

Seznam použitých zkratek

CPP Chinese Postman Problem

BFS Breadth-First Search

DCPP Directed CPP

DRCPP Directed Rural CPP

MCPP Mixed CPP

RPP Rural Postman Problem

UCPP Undirected CPP

URCPP Undirected Rural CPP

UWRCPP Undirected Windy Rural 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_2020.pdf` ..... text práce ve formátu PDF