FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Algoritmy pro řešení neorientovaného a orientovaného Problému čínského listonoše
Student:	Martin Vítek
Vedoucí:	doc. Ing. Ivan Šimeček, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

1) Nastudujte a popište problém čínského listonoše (CPP) a jeho varianty [1,2]
2) Popište algoritmy pro řešení variant UCPP a DCPP pomocí: síťových toků [3], Maďarské metody [4]
3) Implementujte algoritmy popsané v bodě 2 v jazyce C++
4) Optimalizujte implementované algoritmy paralelizací pomocí OpenMP
5) Porovnejte vaši implementaci s již existujícími (např. [5]).

[1] Eiselt H. A.; Gendreau M.; Laporte G., Arc Routing Problems, Path 1: The Chinese Postman Problem. Operations Research Vol. 43, No. 2, 1995: s. 231-242
[2] Eiselt H. A.; Gendreau M.; Laporte G., Arc Routing Problems, Path 2: The Rural Postman Problem. Operations Research Vol. 43, No. 3, 1995: s. 399-414
[3] Ahuja, R.; Magnanti, T.; Orlin, J.: Networks Flows: Theory, Algorithms, and Practice, ISBN 013617549X. 1993
[4] Harold W. Kuhn, The Hungarian Method for the assignment problem. Naval Research Logistics Quarterly, 2, 1955: s. 83–97
[5] Razák M., Algoritmy pro řešení problému čínského listonoše. Praha, 2019.

Elektronicky schválil/a doc. Ing. Jan Janoušek, Ph.D. dne 11. února 2021 v Praze.

Bakalářská práce

ALGORITMY PRO ŘEŠENÍ NEORIENTOVANÉHO A ORIENTOVANÉHO PROBLÉMU ČÍNSKÉHO LISTONOŠE

Martin Vítek

Fakulta informačních technologií ČVUT v Praze
Katedra počítačových systémů
Vedoucí: doc. Ing. Ivan Šimeček, Ph.D.
13. května 2021

Obsah

Seznam obrázků

Seznam tabulek

Chtěl bych poděkovat především vedoucímu této práce doc. Ing. Ivanu Šimečkovi, Ph.D. za veškeré rady a velmi pohotové reakce. Dále bych chtěl poděkovat celé své rodině a přátelům za jejich pomoc a podporu, zejména Jiřímu Frýdlovi.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů 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 předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, 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 způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). 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 způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 10. května 2021 .....

Abstrakt

Tato bakalářská práce se zabývá Problémem čínského listonoše, konkrétně jeho neorientovanou a orientovanou variantou. Problém čínského listonoše je optimalizační úloha z teorie grafů. V úvodu je tento problém rozebrán spolu se zavedením definic pojmů používaných v následujících částech práce. Obě varianty problému jsou rozděleny do jednotlivých kroků. Pro každý dílčí krok jsou popsány algoritmy, které ho řeší. V implementační části práce jsou algoritmy implementovány a spojeny do komplexního řešení Problému čínského listonoše. Algoritmy jsou následně paralelizovány pomocí knihovny OpenMP. Jednotlivé implementace algoritmů jsou testovány a porovnány mezi sebou.

Klíčová slova teorie grafů, Problém čínského listonoše, eulerovský tah, paralelizace

Abstract

This bachelor thesis deals with the Chinese postman problem, especially with undirected and directed versions. The Chinese postman problem is an optimization problem from graph theory. In the introduction, the problem is presented along with the definition of terms used in the following sections. Both versions of the problem are divided into simple steps. For each of these steps some algorithms are presented. In the implementation part of this thesis, those algorithms are implemented and merged into a single solution to the Chinese postman problem. Implemented algorithms are parallelized using the OpenMP library. Algorithms are then tested and compared to each other.

Keywords graph theory, Chinese postman problem, eulerian cycle, parallelization

Seznam zkratek

CPP	Chinese Postman Problem
HW	Hardware
VLSI	Very Large Scale Integration
UCPP	Undirected CPP
DCPP	Directed CPP
MCPP	Mixed CPP
WPP	Windy Postman Problem
RPP	Rural Postman Problem
BGL	Boost Graph Library

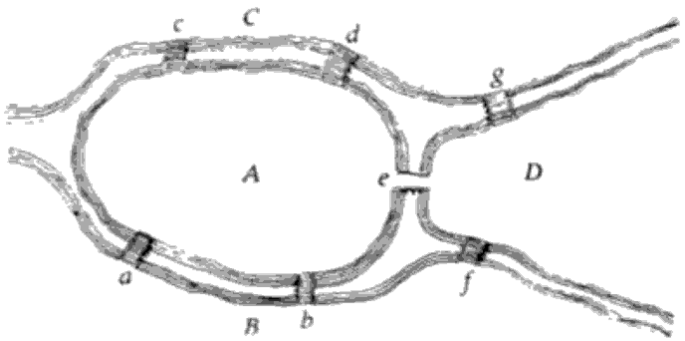
Úvod

Problém čínského listonoše (anglicky Chinese postman problem – CPP) je úloha z teorie grafů. V základní formě jde o hledání nejkratší trasy pro pošťáka, který musí roznést poštu do každé ulice a vrátit se zpět na poštu. Analogicky lze problém vyjádřit jako hledání nejkratší trasy grafem, která projde všechny hrany a skončí ve výchozím vrcholu. Tento graf může reprezentovat mapu města, kde hrany jsou ulicemi a vrcholy křižovatkami.

Podle typu hran lze rozlišovat jednotlivé varianty Problému čínského listonoše. Podrobněji jsou tyto varianty popsány v části 1.3. Tato bakalářská práce se zabývá hlavně neorientovanou a orientovanou variantou s celočíselným kladným ohodnocením hran.

Předchůdce problému lze nalézt v článku švýcarského matematika Leonharda Eulera *Solutio Problematis ad Geometriam Situs Pertinentis*, který publikoval v roce 1736. Anglicky vyšel v roce 1986 v knize *Graph Theory 1736 – 1936* [1]. Euler jako první nastolil problém, který vešel ve všeobecnou známost jako hádanka *sedmi mostů města Královce*. Tato hádanka je prvním vyjádřením problému, který se stal základním kamenem pro rozvoj nového odvětví matematiky – teorie grafů. Podstata hádanky je nalezení trasy, která projde každý z mostů v Královci právě jednou a skončí ve výchozím bodě. Euler ve svém článku ukázal, že tento problém nemá řešení, aniž by trasa nějaký most neprošla vícekrát.

Praktické využití Problému čínského listonoše netřeba hledat pouze v roznášení poštovních zásilek. Řešení lze uplatnit také například u čištění ulic, trasování linek autobusů [2], testování HW systémů [3], návrhů VLSI obvodů [4] nebo v částicové fyzice, konkrétně ve fyzice kondenzovaných látek [5]. Dále se Problém čínského listonoše objevuje jako podproblém jiných úloh. Kupříkladu u hledání nejkratšího cyklu procházejícího všemi vrcholy. Ten lze řešit převodem na Problém obchodní cestující nebo přímo pomocí řešení CPP jako jednoho z podproblémů [6].



Obrázek 1 Eulerův nákres Královce s mosty označenými malými písmeny. Zdroj: [1]

Kapitola 1
Představení problému

Pro formální zavedení Problému čínského listonoše je třeba si nejprve definovat některé pojmy z teorie grafů.

1.1 Základní definice

▶ **Definice 1.1.** Graf G *je uspořádaná dvojice neprázdné konečné množiny vrcholů* V *a množiny hran* E*. Značí se* $G = (V, E)$*.*

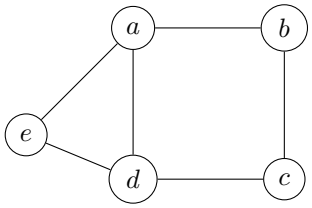
Tato základní definice je velmi volná v tom, co se může brát jako vrcholy a hrany. Definice vůbec neříká jaký vztah mají jednotlivé hrany a vrcholy. Do této definice se poměrně jednoduše schovají i multigrafy, které budou definovány později. Kvůli zmíněné volnosti jsou níže definovány neorientované a orientované grafy, které jsou více restriktivní z pohledu struktury.

▶ **Definice 1.2.** Neorientovaný graf *je graf, který jako množinu hran* E *má podmnožinu množiny všech neuspořádaných dvojic vrcholů* V*. Hranu mezi vrcholy* u *a* v *značí se* $\{u, v\}$ *a nazývá se* neorientovaná*.*

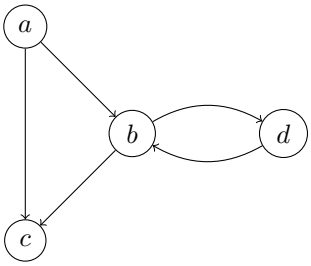
▶ **Definice 1.3.** Orientovaný graf *je graf, který jako množinu hran* E *má podmnožinu množiny všech uspořádaných dvojic vrcholů* V*. Hranu z vrcholu* u *do vrcholu* v *značí se* (u, v) *a nazývá se* orientovaná*.*

V těchto dvou definicích se množina hran dá chápat v jistém smyslu jako binární relace na množině vrcholů V. Vrcholy dává do vztahu, který se dá nazvat *sousedí s*. Například na obrázku 1.1 jsou vrcholy a, b, c, d, e. Vrchol a sousedí s vrcholem b, takže množina hran tohoto grafu jistě obsahuje dvojici a, b ($\{a, b\} \in E$).

Je třeba si dát ovšem pozor na ten fakt, že u neorientovaných grafů jsou hrany reprezentovány jako neuspořádané dvojice a binární relace je množina uspořádaných dvojic. Zde intuice může



Obrázek 1.1 Neorientovaný graf



Obrázek 1.2 Orientovaný graf

mást a někdo si může myslet, že pokud vrchol x je v relaci (sousedí s) vrcholem y, tak y musí být v relaci s (sousedit s) x. Tato vlastnost relace se nazývá symetrie. U neorientovaných grafů tomu tak opravdu je, ale problém nastane u orientovaných grafů, kde hrany jsou tzv. *orientované* – průchodné pouze v jednom směru – a tedy tato relace nemusí být symetrická.

Na obrázku 1.2 je orientovaný graf, který v množině hran obsahuje hranu (a, b), ale neobsahuje hranu (b, a). Je tedy možné se dostat z vrcholu a do vrcholu b, ale zpět už ne.

▶ **Definice 1.4.** *Multigraf je trojice (V, E, ϵ) taková, že*

- *V, E jsou disjunktní množiny, kde V je neprázná konečná množina*
- *$a\ \epsilon : E \to \binom{V}{2}$ je zobrazení, které přiřazuje každé hraně její dva konce, $\binom{V}{2}$ značí množinu neuspořádaných dvojic ve V.*

Definice multigrafu výše odpovídá zobecnění definice neorientovaného grafu, protože ϵ zobrazuje do množiny neuspořádaných dvojic. Tato definice neumožňuje vytvořit graf, který by obsahoval smyčky – hrany začínající a končící v jednom vrcholu. Smyčky však nejsou důležité pro řešení CPP. Jednoduchou úpravou zobrazení ϵ lze definovat orientovanou variantu multigrafu. Stačí aby ϵ zobrazovalo do množiny uspořádaných dvojic vrcholů a hrany multigrafu budou mít danou orientaci.

▶ **Definice 1.5.** *Graf se nazývá* úplný *pokud množina hran obsahuje všechny dvojice vrcholů. Úplný graf o n vrcholech se značí K_n.*

▶ **Definice 1.6.** *Graf $G = (V, E)$ se nazývá* bipartitní *pokud existuje rozdělení vrcholů do dvou množin $A, B \subseteq V \wedge A \cap B = \emptyset \wedge A \cup B = V$ takové, že pro každou hranu platí $\forall \{u, v\} \in E, u \in A \wedge v \in B$. Úplný bipartitní graf, kde n a m jsou velikosti partit se značí $K_{n,m}$.*

▶ **Definice 1.7.** Stupeň vrcholu *v neorientovaném grafu $G = (V, E)$ je počet hran spojující daný vrchol s jinými vrcholy. Značí se* $\deg_G(v) = |\{u \mid u \in V \wedge (v, u) \in E\}|$

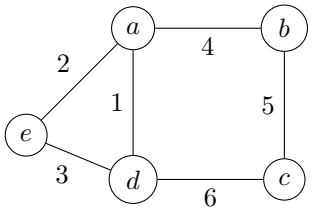
Stupeň vrcholu udává počet vrcholů, které sousedí s daným vrcholem. Na obrázku 1.1 je stupeň vrcholu a roven 3, tedy $\deg_G(v) = 3$.

Definice stupně vrcholu výše je pouze pro neorientovaný graf. V orientovaném grafu je rozdíl v tom, že hrany mají směr a je tedy vhodné rozlišit takzvané vstupní a výstupní hrany vrcholu.

▶ **Definice 1.8.** Vstupní stupeň vrcholu *v orientovaném grafu $G = (V, E)$ je počet orientovaných hran končících ve vrcholu v. Značí se* $\deg^+_G(v) = |\{u \mid u \in V \wedge (u, v) \in E\}|$

▶ **Definice 1.9.** Výstupní stupeň vrcholu *v orientovaném grafu $G = (V, E)$ je počet orientovaných hran začínajících ve vrcholu v. Značí se* $\deg^-_G(v) = |\{u \mid u \in V \wedge (v, u) \in E\}|$.

▶ **Definice 1.10.** Stupeň vrcholu *v orientovaném grafu je součet vstupního a výstupního stupně vrcholu.* $\deg_G(v) = \deg^+_G(v) + \deg^-_G(v)$



Obrázek 1.3 Neorientovaný ohodnocený graf

V grafu na obrázku 1.2 má vrchol a vstupní stupeň vrcholu roven 0 a výstupní stupeň roven 2, protože z a vychází dvě hrany (do vrcholů b a c) a žádná v něm nekončí. To se zapíše takto: $\deg_G^+(v) = 0$ a $\deg_G^-(v) = 2$.

▶ **Věta 1.11.** *(o sudosti) Pro každý graf $G = (V, E)$ platí*

$$\sum_{v \in V} \deg_G(v) = 2\,|E|$$

.

Důkaz. Stupeň vrcholu je roven počtu hran obsahujících daný vrchol. Sečtením stupňů všech vrcholů se započte každá hrana dvakrát, což dává součet $2\,|E|$. ◀

▶ **Důsledek 1.12.** *V každém grafu je počet vrcholů lichého stupně sudý.*

▶ **Definice 1.13.** *Nechť $G = (V, E)$ je neorientovaný graf.* Hustotou *neorientovaného grafu G se nazývá poměr*

$$d = \frac{2|E|}{|V|(|V|-1)}.$$

▶ **Definice 1.14.** *Nechť $G = (V, E)$ je orientovaný graf.* Hustotou *orientovaného grafu G se nazývá poměr*

$$d = \frac{|E|}{|V|(|V|-1)}.$$

V literatuře se často mluví o hustých a řídkých grafech (anglicky dense a sparse). Intuice radí, že řídké grafy mají málo hran, kdežto husté mají mnoho hran a blíží se k úplným grafům. Definice výše definují hustotu grafu jako poměr počtu hran v grafu ku počtu možných hran v grafu. Každému grafu přiřadí hustotu z intervalu $\langle 0, 1\rangle$, kde 0 značí graf bez hran a 1 označuje úplný graf. Rozdíl v definici pro orientovaný a neorientovaný graf je kvůli různému maximálnímu počtu hran.

▶ **Definice 1.15.** *Nechť $G = (V, E)$ je graf a $w : E \to \mathbb{R}$ je zobrazení, které každé hraně $e \in E$ přiřadí číselnou hodnotu $w(e)$. Tato hodnota se nazývá* váha hrany *e, zobrazení se nazývá* váhová funkce *a graf se nazývá* ohodnocený.

Ohodnocený graf dává možnost jak každé hraně (dvojici vrcholů) přiřadit nějakou číselnou hodnotu, což pomáhá lépe a přesněji reprezentovat data v reálných situacích. Například u reprezentace sítě vodovodního potrubí grafem hrany budou představovat trubky a vrcholy budou propojení těchto trubek. Pomocí vah každé hraně (trubce) je možné přiřadit váhu, která by odpovídala například její délce, kapacitě nebo aktuálnímu průtoku.

Příklad ohodnoceného grafu je na obrázku 1.3. Graf obsahuje stejné vrcholy a hrany jako graf na obrázku 1.1, ale zde má každá hrana přiřazenou nějakou číselnou hodnotu. Například hrana $e = \{a, b\}$ má přiřazenou hodnotu $w(e) = 4$, tedy váha hrany e je 4.

▶ **Definice 1.16.** *Nechť $G = (V, E)$ je graf, posloupnost $(v_0, e_1, v_1, \ldots, e_n, v_n)$ se nazývá* sled *pokud:*

- $v_i \in V$ *pro všechna* $i \in \{1, \ldots, n\}$
- $e_i = (v_{i-1}, v_i) \in E$ *pro všechna* $i \in \{1, \ldots, n\}$

▶ **Definice 1.17.** *Sled v grafu se nazývá* tah *pokud se v něm neopakují hrany. Tah se nazývá* uzavřený *pokud začíná a končí ve stejném vrcholu. Tah se nazývá* eulerovský *pokud obsahuje všechny hrany grafu.*

▶ **Definice 1.18.** *Graf je* eulerovský *pokud v něm existuje uzavřený eulerovský tah.*

▶ **Věta 1.19.** *Neorientovaný graf je eulerovský právě tehdy, když je souvislý a každý stupeň vrcholu je sudý.*

Důkaz. Viz přednáška 10 v [7]. ◀

▶ **Věta 1.20.** *Orientovaný graf je eulerovský právě tehdy, když je silně souvislý a pro každý vrchol* $v \in V$ *platí* $\deg^+_G(v) = \deg^-_G(v)$.

Důkaz. Viz [8, s. 3] ◀

▶ **Definice 1.21.** *Sled v grafu se nazývá* cesta *pokud se v něm neopakují vrcholy.*

▶ **Definice 1.22.** *Neorientovaný graf* $G = (V, E)$ *se nazývá* souvislý *pokud v grafu pro každou dvojici různých vrcholů* $u, v \in V \wedge u \neq v$ *existuje nějaká cesta z* u *do* v .

▶ **Definice 1.23.** *Orientovaný graf* $G = (V, E)$ *se nazývá* slabě souvislý *pokud v grafu pro každou dvojici různých vrcholů* $u, v \in V \wedge u \neq v$ *existuje nějaká cesta z* u *do* v *nebo z* v *do* u.

Orientovaný graf $G = (V, E)$ *se nazývá* silně souvislý *pokud v grafu pro každou dvojici různých vrcholů* $u, v \in V \wedge u \neq v$ *existuje nějaká cesta z* u *do* v *a zároveň existuje nějaká cesta z* v *do* u.

▶ **Definice 1.24.** Váhou sledu v *ohodnoceném grafu se nazývá součet vah všech hran v sledu.*

Často se ohodnocení hran bere jako délka hran. Je vhodné definovat i vzdálenost dvou vrcholů grafu, které nejsou spojeny hranou.

▶ **Definice 1.25.** *Pro libovolné dva vrcholy* u, v *ohodnoceného grafu je vzdálenost* $d(u, v)$ *minimum z délek všech cest z* u *do* v*, případně* ∞*, pokud žádná* uv*-cesta neexistuje.*

▶ **Definice 1.26.** Nejkratší uv*-cesta je libovolná* uv*-cesta, jejíž délka je rovna vzdálenosti* $d(u, v)$.

Pokud $u = v$, poté se volí vzdálenost vrcholu od sebe samého jako 0. Pro neorientované grafy jistě platí, že $d(u, v) = d(v, u)$.

▶ **Definice 1.27.** *Párování v grafu* $G = (V, E)$ *je množina hran* $M \subseteq E$ *taková, že každý vrchol grafu* G *patří nejvýše do jedné hrany z* M*. Párování je* perfektní*, pokud každý vrchol grafu* G *patří právě do jedné hrany z* M.

▶ **Definice 1.28.** *Párování se nazývá* maximální*, pokud neexistuje párování, které má větší součet vah všech hran.*

Často se v teorii řeší maximální párování ve smyslu počtu hran, takové párování se nazývá maximální kardinální párování. Párování, u kterého se maximalizuje součet vah hran, se nazývá maximální vážené párování. Tento text zabývá hlavně váženým párováním, proto se maximálním párováním myslí maximální vážené párování, pokud nebude explicitně napsáno jinak.

▶ **Definice 1.29.** *Perfektní párování se nazývá* minimální*, pokud neexistuje perfektní párování, které by mělo menší součet vah všech hran.*

▶ **Věta 1.30.** *Minimální perfektní párování v grafu existuje právě tehdy, když existuje nějaké perfektní párování v grafu.*

Důkaz. ⇒: Minimální perfektní párování je perfektní párování.

⇐: V množině všech možných perfektních párování v grafu jistě existuje takový, jehož váha bude minimální, protože množina vrcholů grafu je konečná a tak je konečný i počet všech různých perfektních párování. ◀

Definice, věty, důkazy a důsledky uvedené výše byly převzaty a některé upraveny z materiálů předmětů BI-AG1 [9] a BI-AG2 [7].

1.2 Formulace problému

Problém čínského listonoše (anglicky Chinese Postman Problem – CPP) jako první formuloval čínský matematik Mei-Ko Kwan (romanizováno také jako Mei-Gu Guan) v článku z roku 1960, který byl v roce 1962 přeložen do angličtiny [10].

Původní formulace:

„Pošťák musí roznést dopisy v daném sousedství. Vyjde z pošty, musí projít každou ulici a skončit zase na poště. Jakou trasu má zvolit, aby ušel co nejkratší vzdálenost?“ [10]

Tuto formulaci poté převedl na optimalizační úlohu na grafu:

„Dáno souvislý graf obsahující $2n$ *vrcholů lichého stupně a zbytek vrcholů má sudý stupeň. Předpokládejme, že musíme přidat do grafu hrany s následujícími vlastnostmi: počet hran přidaných k vrcholu lichého stupně je lichý a počet hran přidaných k vrcholu sudého stupně je sudý. Musíme minimalizovat celkovou délku přidaných hran.“* [10]

V této úloze Guan tiše předpokládá, že se jedná o neorientovaný ohodnocený graf, kde váha hran představuje její délku. Pro orientovaný graf je třeba zaručit silnou souvislost grafu, což je ukázáno v kapitole 3. Přidáním hran podle popisu vznikne (multi)graf. Tento graf je eulerovský, protože souvislost grafu je dána a vrcholům sudého stupně se zvedne stupeň o sudé číslo a vrcholům lichého stupně se zvedne stupeň o liché číslo a tím pádem všechny vrcholy budou mít sudý stupeň.

1.3 Varianty CPP

Z definice problému vyplývá, že existují různé varianty a to nejen pro neorientované a orientované grafy. Přirovnáním k hledání nejkratší trasy pro pošťáka lze zjistit, že v určitých případech nelze na reálnou situaci aplikovat přímo jednu ze zmíněných variant a je potřeba úlohu upravit.

Základním typem CPP je již zmíněná neorientovaná varianta (UCPP – Undirected CPP). V grafu jsou pouze neorientované hrany, což znamená, že je možné projít hranou (ulicí) v jakémkoli směru. Vlastnosti UCPP:

- pouze neorientované hrany
- lze řešit v polynomiálním čase [11]
- definováno v [10]

Druhým typem je orientovaná varianta (DCPP – Directed CPP). Graf obsahuje pouze orientované hrany, které lze procházet pouze v jednom směru. Orientace hran dovoluje modelovat situace jako například jednosměrné ulice, avšak neexistuje možnost jak modelovat obousměrné ulice. Přidání hrany v opačném směru by nepomohlo, protože by tím byl požadován průchod ulicí oběma směry, což není stejné jako možnost průchodu nezávisle na směru. Vlastnosti DCPP:

- pouze orientované hrany
- lze řešit v polynomiálním čase [11]
- definováno v [2]

Přesnější modelování reálných situací a řešení výše zmíněného problému přináší smíšená úloha (MCPP – Mixed CPP). V této úloze je dovoleno používat jak orientované, tak neorientované hrany. Neorientované hrany se použijí na reprezentaci ulic, které jsou obousměrné a stačí je projít jen jednou v nějakém směru. Orientované hrany mohou reprezentovat jednosměrné ulice nebo například široké ulice, které je potřebné projít v obou směrech. Vlastnosti MCPP:

- neorientované i orientované hrany
- NP-těžký problém [12], [13]
- definováno v [12]

V reálném prostředí často nastává situace, kdy obtížnost průchodu ulicí závisí na směru průchodu. Příkladem je ulice v kopci nebo vítr kladoucí odpor při chůzi proti němu. Řešení přináší asymetrická varianta CPP (WPP – Windy Postman Problem). Tato varianta pracuje nad neorientovaným grafem a každé neorientované hraně přiřazuje váhu podle směru průchodu. Vlastnosti WPP:

- neorientované hrany
- váha hrany závisí na směru průchodu
- NP-težký problém [14]
- definováno v [12]

Je vhodné zmínit, že WPP se dá řešit v polynomiálním čase pokud pro každý cyklus v grafu platí, že součet vah hran v jednom směru je roven součtu vah hran ve směru opačném [14]. Tato podmínka není složitá splnit. Pokud váha hrany (ulice) určuje například stoupání (resp. klesání), pak tato podmínka bude splněna. Každý cyklus bude mít součet roven 0 v obou směrech, protože cyklus začíná a končí ve stejné výšce. Jednoduše řečeno, co cesta nastoupá musí i klesnout.

Další situací, která se často objevuje, je přidání ulic, které není potřeba projít, ale lze je využít ke zkrácení celkové délky trasy. Tato varianta se nazývá problém venkovského pošťáka (RPP – Rural Postman Problem). V tomto případě se nepožaduje, aby řešení obsahovalo všechny hrany, ale pouze podmnožinu hran – povinné hrany. Vlastnosti RPP:

- neorientované i orientované varianty
- množina povinných a nepovinných hran
- NP-težký problém [15]
- definováno v [15]

V případě velmi velkého vstupu se lze uchýlit k aproximačním algoritmům a heuristikám. Například u WPP, pokud podmínka na stejnou délku cyklů v obou směrech není splněna, ale rozdíl těchto délek je „malý“, pak lze využít k řešení aproximační algoritmus popsaný v [14]. Další aproximační algoritmy pro WPP jsou v [16]. Aproximační algoritmy pro MCPP lze nalézt v [2] a podrobněji rozebrané v [17]. Heuristiky k řešení neorientovaných i orientovaných variant problému venkovského pošťáka jsou popsány v [15].

Kapitola 2

Neorientovaný problém čínského listonoše

Pro jednoduchost v této kapitole je pod pojmem graf míněn souvislý neorientovaný ohodnocený graf, pokud nebude explicitně napsáno jinak. Následující podkapitoly předpokládají, že vstupní grafy jsou pouze kladně celočíselně ohodnocené.

Původní formulace problému od matematika Mei-Gu Guana zní: „*Dáno souvislý graf obsahující* $2n$ *vrcholů lichého stupně a zbytek vrcholů mají sudý stupeň. Předpokládejme, že musíme přidat do grafu hrany s následujícími vlastnostmi: počet hran přidaných k vrcholu lichého stupně je lichý a počet hran přidaných k vrcholu sudého stupně je sudý. Musíme minimalizovat celkovou délku přidaných hran.*“ [10]

Sám Guan ve stejném článku ukázal, že pro množinu přidaných hran musí platit následující podmínky, aby řešení problému bylo optimální:

- Maximálně jedna hrana je přidána mezi každou dvojici vrcholů – maximálně jedna kopie.
- Pro každý cyklus v rozšířeném grafu platí, že celková váha přidaných hran není větší než polovina váhy celého cyklu.

Jeho důkaz byl konstruktivní, čímž ukázal nejen, za jakých podmínek je řešení optimální, ale také jak ho nalézt [10].

V roce 1973, tedy 11 let po zveřejnění anglického překladu původního článku, publikovali američtí matematici Jack Edmonds a Ellis L. Jonhson článek [11], ve kterém ukázali, že Problém čínského listonoše lze redukovat na hledání maximálního párování, které je řešitelné v polynomiálním čase, což Edmonds ukázal již v roce 1965 [18].

Ve formě celočíselného programování se problém dá formulovat jako minimalizace přidání vah takto:

Nechť $G = (V, E)$ je graf a transformuje se ho do grafu G', ve kterém budou všechny vrcholy sudého stupně. Nechť x_{ij} je počet kopií hran (cest) $\{v_i, v_j\}$ $(i < j)$ potřebných k transformaci G. Nechť $N(v_i)$ je množina vrcholů sousedící s vrcholem v_i. Množina vrcholů lichého stupně se označuje jako T.

Minimalizuj

$$\sum_{\{v_i,v_j\}\in E} c_{ij}\, x_{ij}$$

přičemž:

$$\sum_{v_j \in N(v_i)} x_{ij} \equiv \begin{cases} 1 \pmod 2 & \text{if } v_i \in T \\ 0 \pmod 2 & \text{if } v_i \in V \setminus T \end{cases}$$
$$x_{ij} \in \{0, 1\} \quad (\{v_i, v_j\} \in E),$$

kde c_{ij} je délka nejkratší cesty mezi vrcholy v_i a v_j [2].

Řešení UCPP se rozdělí na postupné řešení dílčích úkolů, které se dají řešit různými metodami/algoritmy v následujících krocích.

- **Krok 1.** Nalézt všech $2k$ vrcholů lichého stupně.

  Z Věty o sudosti 1.11 a jejího důsledku 1.12 je zřejmé, že v grafu je sudý počet vrcholů lichého stupně. Tento krok nevyžaduje použití nějakého sofistikovaného algoritmu, provede se jednoduchou iterací přes vrcholy grafu a získá se stupeň vrcholu $\deg_G(v)$.

- **Krok 2.** Vyhledat nejkratší cesty mezi dvojicemi vrcholů lichého stupně.

- **Krok 3.** Sestrojit úplný graf K_{2k} s vahami spočítanými v kroku 2.

- **Krok 4.** Nalézt minimální perfektní párování v K_{2k}.

  Perfektní párování v K_{2k} jistě existuje, stačí párovat postupně dvojice vrcholů v nějakém pořadí vrcholů a tím pádem z Věty 1.30 plyne, že minimální perfektní párování v kroku 4 existuje.

- **Krok 5.** Vytvořit upravený graf přidáním do původního grafu hrany podél cest z kroku 2 mezi vrcholy párování z kroku 4.

- **Krok 6.** Nalézt eulerovský tah v upraveném grafu.

2.1 Hledání nejkratších cest

Pro nalezení optimálního řešení je třeba nalézt minimální perfektní párování. Do toho je však třeba spočítat váhy mezi všemi dvojicemi vrcholů lichého stupně. Pokud se nad vahami hran uvažuje jako na jejich délkami, tak se budou hledat nejkratší cesty.

Algoritmů k hledání nejkratších cest v grafu je známo několik. Každý má své výhody a nevýhody. V této kapitole jsou ukázány dva základní. Tím prvním algoritmem je asi ten nejznámější, Dijkstrův, který hledá nejkratší cesty z jednoho vrcholu do ostatních. Druhým algoritmem je Floydův–Warshallův algoritmus. Ten hledá nejkratší cesty mezi všemi dvojicemi vrcholů grafu najednou.

Níže popsané algoritmy využívají faktu, že pokud existuje nejkratší cesta z vrcholu a do vrcholu b a někde na té cestě je vrchol c, tak část cesty mezi vrcholy a a c je nejkratší cesta mezi těmito vrcholy. Díky tomu je možné konstruovat nejkratší cesty postupně [19].

2.1.1 Dijkstrův algoritmus

Dijkstrův algoritmus je algoritmus na hledání nejkratších cest v grafu. Představil ho nizozemský matematik Dijkstra ve svém článku v roce 1959 [19]. Algoritmus pracuje správně na nezáporně ohodnocených grafech [20].

Algoritmus popsaný v pseudokódu 1, pracuje s třemi stavy vrcholů. Stav *nenalezený* reprezentuje vrchol, který ještě nebyl nenavštíven. *Otevřený* je vrchol, který algoritmus již navštívil, ale ještě k němu nebyla nalezena nejkratší cesta. *Uzavřený* vrchol již byl navštíven a také k němu

byla nalezena nejkratší cesta. Algoritmus je konečný, protože v každé iteraci uzavře právě jeden vrchol, ke kterému se již nebude vracet. Důkaz korektnosti lze nalézt v [20, s. 146–148].

Algoritmus 1: Dijkstrův algoritmus

Vstup : Graf G, počáteční vrchol v_0
Výstup: Pole vzdáleností dist, pole předchůdců prev

```
1  Pro všechny vrcholy v:
2      stav(v) ← nenalezený
3      dist(v) ← +∞
4      prev(v) ← nedefinováno
5  dist(v_0) ← 0
6  stav(v_0) ← otevřený
7  Dokud existují otevřené vrcholy:
8      Vyber otevřený vrchol v, jehož dist(v) je nejmenší
9      Pro všechny sousedy w vrcholu v:
10         Pokud dist(w) > dist(v) + d(v, w):
11             dist(w) ← dist(v) + d(v, w)
12             stav(w) ← otevřený
13             prev(w) ← v
14     stav(v) ← uzavřený
```

Při uložení všech dat do pole vznikne algoritmus, který poběží v čase $\mathcal{O}(|V|^2)$. Inicializace trvá $\mathcal{O}(|V|)$ a vnějším cyklem algoritmus projde právě $|V|$-krát. V cyklu se pokaždé hledá minimum ze vzdáleností vrcholů, což trvá $\mathcal{O}(|V|)$ a poté se prochází jednotliví sousedi daného vrcholu [20].

Pro zlepšení asymptotické složitosti tohoto algoritmu je možné využít nějakou datovou strukturu, která umožňuje rychle vyhledávat a extrahovat minimum a snižovat hodnoty. Struktura, která zvládá tyto operace, se nazývá *prioritní fronta*. Existují implementace prioritní fronty, které dokáží nalézt a extrahovat minimum rychleji než prostým průchodem v poli, ale zaplatí se za to zpomalením operace snížení existující hodnoty, která je v poli trvá konstantní dobu.

Jednou implementací je *binární halda*, která vytvoří z prvků binární strom, jehož kořen je nejmenší prvek. Výhoda binární haldy je její jednoduchost. Druhou implementací je *Fibonacciho halda*, která si také tvoří strom a dává ještě lepší asymptotickou složitost u některých operací. Na rozdíl od binární haldy je Fibonacciho halda poměrně složitá na implementaci a lepší asymptotická složitost se projeví až u velkých vstupů. Více o binární haldě se lze dozvědět v přednášce 4 v [9] a v [20, s. 84–89], více o Fibonacciho haldě v přednášce 8 v [7] a v [20, s. 422–428].

Při použití binární haldy pro ukládání vrcholů vyjde asymptotická složitost Dijkstrova algoritmu jako $\mathcal{O}((|V| + |E|) \cdot \log|V|)$. Při použití Fibonacciho haldy se asymptotická složitost sníží na $\mathcal{O}(|E| + |V| \cdot \log|V|)$ [20, s. 148–149].

Zrekonstruovaní cesty z vrcholu u do vrcholu v z pole předchůdců je třeba provádět postupně odzadu. Na začátku bude cestu tvořit pouze vrchol v. Poté se bude na začátek této cesty přidávat předchůdce prvního vrcholu cesty, dokud cesta nebude začínat ve vrcholu u. Cesta z u do v neexistuje, právě když předchůdce v není definován.

K získání nejkratší cesty mezi všemi dvojicemi vrcholů lichého stupně lze Dijkstrův algoritmus opakovaně spouštět z jednotlivých vrcholů.

2.1.2 Floydův–Warshallův algoritmus

Floydův–Warshallův algoritmus hledá nejkratší cesty mezi všemi vrcholy grafu najednou. Graf může být orientovaný i neorientovaný. Algoritmus pracuje správně na grafu, který obsahuje hrany záporné délky, ale nesmí obsahovat záporné cykly. Své jméno má po dvou vědcích, Robertu W. Floydovi a Stephenovi Warshallovi, kteří ho publikovali nezávisle na sobě v roce 1962 [21], [22]. Nebyli ovšem první, kteří tento algoritmus představili. Již v roce 1959 francouzský matematik

Bernard Roy publikoval algoritmus na hledání tranzitivního uzávěru, který je ekvivalentní hledání nejkratších cest [23].

Algoritmus bere na vstupu matici délek hran, značí se ji D^0, ijtá pozice matice obsahuje délku hrany mezi $i$tým a $j$tým vrcholem. Pokud hrana neexistuje, tak ijtá pozice je dána jako $D^0_{i,j} = \infty$. Poté se postupně počítá matice vzdáleností D^k, která na ijté pozici obsahuje délku nejkratší cesty z vrcholu i do vrcholu j pouze za použití prvních k vrcholů jako vnitřních vrcholů. Matice $D^{k+1}_{i,j}$ se spočítá jako $\min(D^k_{i,j}, D^k_{i,k} + D^k_{k,j})$. Na konci matice D^n (n značí počet vrcholů) obsahuje délky nejkratších cest mezi dvojicemi vrcholů nebo ∞, pokud cesta neexistuje. Důkaz korektnosti lze nalézt v [20, s. 154–155].

Algoritmus 2: Floydův–Warshallův algoritmus

Vstup : Matice délek D^0
Výstup: Matice vzdáleností dist, matice následníků next

```
1 dist ← D^0
2 Pro každou dvojici i, j:
3     next(i)(j) ← j
4 Pro k = 0, ..., n − 1:
5     Pro i = 1, ..., n:
6         Pro j = 1, ..., n:
7             Pokud dist(i)(j) > dist(i)(k) + dist(k)(j):
8                 dist(i)(j) ← dist(i)(k) + dist(k)(j)
9                 next(i)(j) ← next(k)(j)
```

Při bližší prozkoumání je možné si všimnout, že v algoritmu 2 není použito pro každé k vlastní matice, ale vše se počítá v matici, která bude obsahovat výsledek. V případě přepisovaní hodnot přímo na místě, není možné rozlišit, zda se čte $D^k_{u,v}$ nebo $D^{k+1}_{u,v}$. Jak ukáže lemma 2.1, obě hodnoty jsou si rovny a tedy nebude vadit přepisování matice. Tímto se radikálně zredukuje používaný prostor v paměti.

▶ **Pomocné tvrzení 2.1.** *Pro všechna i, j, k platí $D^{k+1}_{k+1,j} = D^k_{k+1,j}$ a $D^{k+1}_{i,k+1} = D^k_{i,k+1}$.*

Důkaz. Levá a pravá strana rovností se liší pouze tím, zda lze použít $(k+1)$ní vrchol jako vnitřní vrchol. Ten se ale jako vnitřní vrchol používat nebude, protože je již použit jako počáteční, resp. koncový, vrchol. ◀

Jak je vidět z pseudokódu 2, Floydův–Warshallův algoritmus je velmi jednoduchý na implementaci a jeho časová složitost je $\Theta(|V|^3)$ [20].

2.2 Hledání párování

Dalším krokem po nalezení nejkratších cest v grafu mezi vrcholy lichého stupně je konstrukce úplného grafu K_{2k} s právě napočítanými délkami jako váhami mezi vrcholy. Váhy hran budou jistě kladné, protože vstupní graf je kladně ohodnocený a tedy nejkratší cesty mají kladnou délku. V tomto grafu se poté bude hledat minimální perfektní párování. Toto párování určí, které hrany se budou muset projít vícekrát a jeho minimalita zaručí optimalitu řešení.

2.2.1 Minimální perfektní párování

Problém hledání minimálního perfektního párování v úplném grafu lze převést na problém hledání maximálního párování v upraveném úplném grafu.

Upravený graf bude obsahovat stejné vrcholy a hrany, ale změní se pouze váhová funkce. Nechť $m = \max_{f \in E}(w(f)) + 1$. Nová váhová funkce bude mít předpis $w'(e) = m - w(e)$. Nová váha hrany bude rovna rozdílu maxima z vah původních hran a váhy dané hrany. K této nové váze se přičte ještě jednička, která zaručí, že váhy všech hran budou ostře větší než nula.

▶ **Pomocné tvrzení 2.2.** *Maximální párování v upraveném grafu je perfektní.*

Důkaz. Pro spor lze předpokládat, že maximální párování by nebylo perfektní. Protože graf má sudý počet vrcholů, jistě existují alespoň dva vrcholy u, v, které nejsou v párování. Protože graf je úplný, v grafu existuje hrana mezi u a v, ta má jistě kladnou váhu. Protože tato hrana není v párování a v párování není ani jeden z jejích konců, párování lze o tuto hranu zvětšit, čímž vznikne spor s tím, že párování bylo maximální. ◀

▶ **Pomocné tvrzení 2.3.** *Maximální párování v upraveném grafu odpovídá minimálnímu perfektnímu párování v původním grafu.*

Důkaz. Z lemma 2.2 vyplývá, že maximální párování v upraveném grafu je perfektní. V upraveném grafu se hledá maximální párování, tedy takové, které má maximální součet vah hran. Hrany v upraveném grafu mají váhu rovnou rozdílu maxima (+1) a váhy v původním grafu. Hledá se tedy párování, ve kterém mají hrany co největší rozdíl od maxima z vah, což odpovídá hledání minimálního perfektního párování v původním grafu. ◀

Z těchto lemmat vyplývá, že problém hledání minimálního perfektního párování lze převést na hledání maximálního párování, což je poměrně rozšířený problém v teorii grafů.

2.2.2 Maximální párování

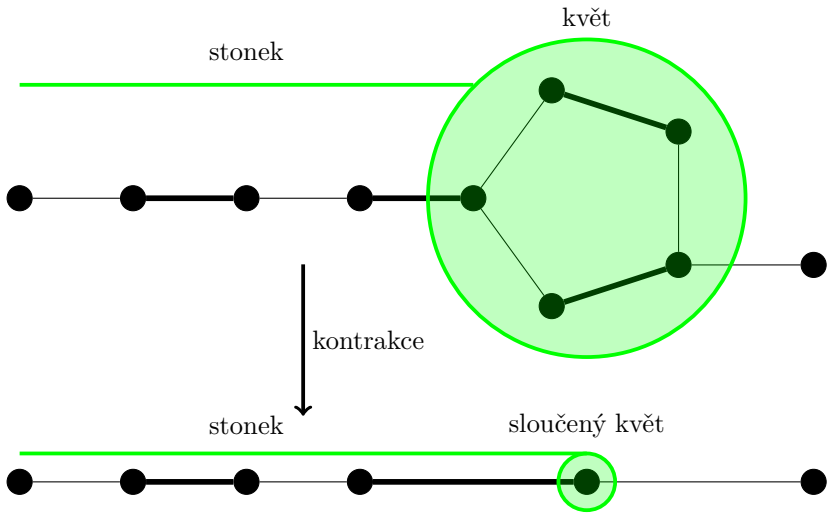
Hledáním maximálního párování v obecném grafu se zabýval Jack Edmonds v sérii svých článků, zejména v tom z roku 1965 [18] a později přímo ve spojitosti s Problémem čínského listonoše v roce 1973 [11]. Do té doby se matematici zabývali hledáním maximálního párování pouze v bipartitních grafech.

Edmonds představil algoritmus, který umí nalézt maximální párování v obecném grafu. Tento algoritmus se nazývá Edmondsův nebo také květinový (anglicky blossom algorithm) podle pojmenování cyklů liché délky. Tato práce se nezabývá zkoumáním Edmonsova algoritmu do hloubky, protože se jedná o poměrně složité téma, které by možná samo vydalo na bakalářskou práci. Z těchto důvodů je zde pouze nástin, jak algoritmus funguje.

Základní pseudokód algoritmu se dá zapsat velmi jednoduše, jak je vidět v algoritmu 3. Využívá Bergeho věty, která říká, že párování je maximální právě když neexistuje zlepšující cesta [24]. Zlepšující cestou se nazývá cesta v grafu, která zvýší počet hran nebo váhu párování v závislosti na tom jaký problém řešíme. Obtížná část algoritmu je právě v hledání zlepšující cesty.

Algoritmus 3: Edmondsův algoritmus

Vstup : Graf G, základní párování M
Výstup: Maximální párování
1 $P \leftarrow$ zlepšující cesta na G s párováním M
2 Pokud P je neprázdná:
3 vylepši párování M podél cesty P
4 rekurzivně spočítej maximální párování na G s vylepšeným párováním
5 Jinak:
6 vrať párování M



Obrázek 2.1 Kontrakce květu do jednoho vrcholu

Květinový algoritmus zakládá na algoritmu hledání maximálního párování v bipartitní grafu, který postupně hledá zlepšující cesty, o které lze párování zvětšit. Problém u grafů, které nejsou bipartitní, nastane, když algoritmus narazí na kružnici obsahující lichý počet vrcholů. Taková kružnice jistě existuje v každém grafu, který není bipartitní. Algoritmus pak může vracet nesprávné výsledky. Situaci, kdy narazí na kružnici liché délky, květinový algoritmus detekuje a kružnici sloučí do nového pseudovrcholu, kterým poté kružnici nahradí a rekurzivně začne hledat zlepšující cestu v grafu bez dané kružnice lichého stupně.

Na obrázku 2.1 je vidět jak funguje kontrakce kružnice liché délky, v tomto případě délky 5. V této kružnici mohou být v párování maximálně 2 hrany. Tlusté čáry označují hrany párování. Vrcholy, které nejsou součástí párování, se nazývají volné. Pokud v grafu existuje cesta, která začíná a končí volným vrcholem a hrany cesty postupně střídají zda jsou v párování, lze ji využít k zvětšení párování co do počtu hran.

Podrobný popis fungování algoritmu a důkaz korektnosti lze nalézt v [11]. Ukázka algoritmu s obrázky, které ukazují, jak jednotlivé kroky algoritmu funguje jsou v [25].

Složitost Edmondsova algoritmu v základní jednoduché formě je $\mathcal{O}(|E||V|^2)$ [25]. S hojným použitím prioritních front lze asymptotickou složitost snížit na $\mathcal{O}(|E||V|\log|V|)$ [26].

2.3 Hledání eulerovského tahu

Po nalezení minimálního párování v K_{2k} stačí doplnit hrany mezi vrcholy z párování. Hrany, které budou doplněny/zduplikovány jsou určeny nejkratšími cestami mezi vrcholy párování. Doplněním hran vzroste stupeň vrcholům lichého stupně – koncovým vrcholům cest – stupeň o 1. Vrcholům sudého stupně vzroste stupeň o 2, pokud jsou vnitřními vrcholy jedné z cest, podél které se přidávali hrany. Ostatním vrcholům sudého stupně se stupeň nezmění. V (multi)grafu s doplněnými hranami mají všechny vrcholy sudý stupeň a je souvislý, je tedy eulerovský. Posledním krokem je v něm nalézt eulerovský tah.

K nalezení eulerovského tahu se dá použít Hierholzerův algoritmus, který byl představen německým matematikem Carlem Hierholzerem v roce 1873. Algoritmus používá proceduru Tah, kterou volá postupně na vrcholech, které jsou incidentní[†] s neprošlými hranami. Procedura Tah vrací nějaký uzavřený tah z vrcholu v. Výsledný tah je vrácen jako seznam vrcholů tahu.

[†]Vrchol je incidentní s hranami, které začínají nebo končí v daném vrcholu.

Hlavní část Hierholzerova algoritmu je popsána v pseudokódu 4. Začíná se s prázdným seznamem *done*, do kterého se postupně ukládají vrcholy výsledného eulerovského tahu. Druhý seznam $left$ je seznam navštívených vrcholů, ale tyto vrcholy ještě mohou být incidentní s hranou, kterou algoritmus nezahrnul do výsledného tahu.

Procedura Tah(v)

Vstup : Eulerovský graf G, vrchol v
Výstup: Uzavřený tah
$C \leftarrow$ prázdný seznam
Dokud v je incidentní s alespoň jednou hranou:
    Nechť $e = \{v, u\}$ je taková hrana
    Přidej u na konec C
    Smaž hranu e z grafu G
    $v \leftarrow u$
Vrať C

Algoritmus 4: Hierholzerův algoritmus

Vstup : Eulerovský graf G
Výstup: Uzavřený eulerovský tah
Nechť v je libovolný vrchol G
done $\leftarrow$ prázdný seznam
$left \leftarrow Tah(v)$
Připoj v na začátek $left$
Dokud $left$ je neprázdný:
    Odeber první vrchol ze $left$ a označ ho u
    Pokud u je incidentní s alespoň jednou hranou:
        Připoj $Tah(u)$ na začátek $left$
    Vlož u na konec *done*
Vrať *done*

Správnost algoritmus lze nalézt v přednášce 10 v [7]. Složitost algoritmu je $\mathcal{O}(|E|)$ [7]. Pro dosažení této složitosti je třeba umět mazat hrany v konstantním čase. Algoritmus sice požaduje mazání hran, ale jednoduchou úpravou lze zařídit, aby místo smazání hrany stačilo označení hrany jako smazané a už se k dané hraně nevracet.

Další algoritmus, který hledá eulerovský tah v grafu se jmenuje Fleuryho. Ten prochází hrany tak, že v každém kroku vybere hranu, která by nerozpojila graf na dvě části. Implementace však potřebuje umět detekovat takzvané mosty, což navyšuje asymptotickou složitost nad optimálních $\mathcal{O}(|E|)$.

Kapitola 3

Orientovaný problém čínského listonoše

Pro jednoduchost v této kapitole je pod pojmem graf míněn silně souvislý orientovaný ohodnocený graf, pokud nebude explicitně napsáno jinak. Následující podkapitoly předpokládají, že vstupní grafy jsou pouze kladně celočíselně ohodnocené.

Pokud by vstupní graf nebyl silně souvislý, ale jen slabě souvislý, tak by nad ním nebylo možné řešit CPP. V takovém grafu jistě nebude existovat uzavřený tah, který by prošel všechny hrany ani když se některé hrany znásobí. Pokud by v slabě souvislém grafu existoval eulerovský tah, tak by graf byl i silně souvislý. To platí, protože tah lze zkrátit na cestu a eulerovský tah navštíví všechny vrcholy.

Věta 1.20 říká, že graf je eulerovský právě tehdy, když je silně souvislý a všechny vrcholy mají stejný vstupní a výstupní stupeň. Silná souvislost je zaručena již na vstupu. Graf však může obsahovat takzvané nevyvážené vrcholy, vrcholy jejichž vstupní stupeň se liší od výstupního stupně. Pokud se vstupní a výstupní stupně vrcholu rovnají, nazývá vyvážený. V grafu je potřeba znásobit některé hrany, aby se tyto nevyvážené vrcholy staly vyváženými a šlo v grafu nalézt eulerovský tah.

Nechť $\delta(v) = \deg_G^-(v) - \deg_G^+(v)$ určuje rozdíl výstupního a vstupního stupně. Vrchol je vyvážený, když $\delta(v) = 0$. D^+ je množina vrcholů s kladnou δ $D^+ = \{v \in V | \delta(v) > 0\}$. D^- je množina vrcholů se zápornou δ $D^- = \{v \in V | \delta(v) < 0\}$.

Ve formě celočíselného programování lze orientovanou variantu CPP formulovat jako minimalizace přidání vah takto:

Minimalizuj

$$\sum_{v_i \in D^-} \sum_{v_j \in D^-} c_{ij}\, x_{ij}$$

přičemž:

$$\begin{aligned} \sum_{v_j \in D^+} x_{ij} &= -\delta(v_i) \quad (v_i \in D^-) \\ \sum_{v_i \in D^-} x_{ij} &= \delta(v_j) \quad (v_j \in D^+) \\ x_{ij} &\geq 0 \quad (v_i \in D^-, v_j \in D^+), \end{aligned}$$

kde c_{ij} je délka nejkratší cesty z vrcholu v_i do v_j. x_{ij} odpovídá počtu průchodů hranami mezi v_i a v_j navíc [2].

Řešení DCPP se rozdělí na postupné řešení dílčích úkolů, které se dají řešit různými metodami/algoritmy v následujících krocích.

- **Krok 1.** Nalézt nevyvážené vrcholy.
- **Krok 2.** Nalézt nejkratší cesty z vrcholů z D^- do vrcholů z D^+.
- **Krok 3.** Sestrojit úplný bipartitní graf $K_{n,n}$, kde v první partitě budou vrcholy z D^- a v druhé vrcholy z D^+. Váhy odpovídají délkám nejkratších cest z kroku 2.

  Každý vrchol do $K_{n,n}$ se musí přidat v závislosti na hodnotě $\delta(v)$. Každý vrchol bude přidán právě $|\delta(v)|$krát.
- **Krok 4.** Nalézt minimální perfektní párování v grafu $K_{n,n}$.
- **Krok 5.** Přidat do grafu hrany podél nejkratších cest z kroku 2 mezi vrcholy odpovídající párování z předchozího kroku.

  Každému vrcholu z D^+ se přiřadí právě $\delta(v)$ vrcholů a získá navíc $\delta(v)$ vstupních hran a tím se stane vyváženým. Obdobně vrcholy z D^- se stanou vyváženými. Ostatním vrcholům se může změnit stupeň, ale zůstanou stále vyvážené.
- **Krok 6.** Nalézt eulerovský tah v upraveném grafu.

3.1 Hledání nejkratších cest

Dalším krokem po identifikaci nevyvážených vrcholů je nalezení nejkratších cest v grafu mezi vrcholy z D^- a vrcholy z D^+. Jedná se vždy o cesty začínající v nějakém vrcholu z D^- a končící ve vrcholu z D^+. Tyto cesty jsou poté přidány do grafu.

Kapitola 2.1 se zabývá hledáním nejkratších cest v neorientovaném grafu. Oba představené algoritmy – Dijkstrův i Floydův–Warshallův – umí hledat nejkratší cesty i v orientovaném grafu. Ani jeden z algoritmů nijak nevyužívá (ne)orientovanosti hran. Podrobnosti o Dijkstrově algoritmu jsou v kapitole 2.1.1. Floydův–Warshallův algoritmus je popsán v kapitole 2.1.2.

3.2 Maďarská metoda

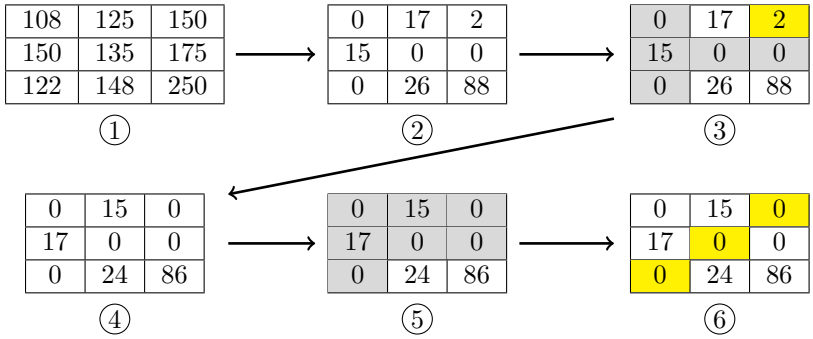
Následujícím krokem po nalezení nejkratších cest je hledání optimálního přiřazení. Přidáním cest tohoto přiřazení do grafu vznikne eulerovský graf. Každému vrcholu z D^+ (resp. D^-) se musí přiřadit $\delta(v)$ (resp. $-\delta(v)$) vrcholů z D^- (resp. D^+). Nalezení optimálního přiřazení je takzvaný Přiřazovací problém (anglicky Assignment problem).

Přiřazovací problém je základní optimalizační úloha. Ukázkovým příkladem je úloha rozdělení určitého počtu pracovníků k množině úkolů. Každý pracovník může být přiřazen k libovolnému úkolu. Navíc každé přiřazení pracovníka k úkolu má určitou cenu. Cílem je získat přiřazení co nejvíce úkolů při minimalizaci (maximalizaci) ceny. V případě, kdy je počet pracovníků rovný počtu úkolů, se jedná o vyvážené přiřazení.

V teorii grafů Přiřazovací problém odpovídá hledání párování určité velikosti v ohodnoceném bipartitním grafu, ve kterém je součet vah hran párování minimální. V případě vyváženého přiřazení jsou obě partity grafu stejně velké.

Algoritmů na hledání optimálního přiřazení je několik. Jedním z nich je takzvaná Maďarská metoda.

Maďarská metoda je algoritmus na řešení Přiřazovacího problému. Algoritmus popsal americký matematik Harold W. Kuhn ve svém článku [27]. Kuhn navazoval na práci dvou maďarských matematiků J. Egerváryho a D. Königa, podle kterých algoritmus pojmenoval právě Maďarská metoda.



Obrázek 3.1 Ukázka průběh Maďarské metody

Prvním problémem, který Kuhn představil ve svém článku, je přiřazení n pracovníků k n úkolům na základě jejich kvalifikace. Zadání problému bylo uspořádáno do matice $n \times n$, kde řádky odpovídaly pracovníkům a sloupce úkolům. Matice obsahovala pouze 1 a 0. Matice na ijté pozici obsahovala 1 právě když $i$tý pracovník byl kvalifikovaný pro $j$tý úkol. Cílem bylo nalézt takové přiřazení, ve kterém je k co nejvíce úkolům přiřazen kvalifikovaný pracovník.

Druhý představený problém je zobecněnou variantou prvního. Matice neobsahuje pouze 1 a 0, ale každé přiřazení má nějakou váhu, v tomto případě celočíselnou kladnou. Cílem je nalézt takové přiřazení pracovníků k úkolům, které maximalizuje součet jednotlivých vah přiřazení.

Algoritmus 5: Maďarská metoda

Vstup : Matice vah $n \times n$
Výstup: Optimální přiřazení

1 Od každého řádku matice odečti minimum daného řádku
2 Od každého sloupce matice odečti minimum daného sloupce
3 Označ všechny nuly v matici minimálním pokrytím horizontálních a vertikálních čar
4 Dokud počet čar pokrytí je menší než n:
5 $min \leftarrow$ nejmenší hodnota, která není pokrytá žádnou čárou
6 Ke každému pokrytému řádku přičti min
7 Od každého nepokrytého sloupce odečti min
8 Přepočítej pokrytí nul čarami
9 Vrať pozice n nezávislých nul jako optimální přiřazení

Pseudokód 5 popisuje princip fungování algoritmu, který se snaží minimalizovat celkovou cenu. Nezávislé nuly, které se hledají v posledním kroku, jsou takové, pro které platí, že žádné dvě nuly spolu nesdílí řádek ani sloupec.

▶ **Příklad 3.1.** Pořádá se večírek, na kterém má být pohoštění a živá hudba. Po večírku je potřeba uklidit. Tři různé firmy, označené A, B a C, nabízejí služby podle ceníku v tabulce 3.1. Každá firma může zařizovat pouze jednu z požadovaných služeb. Jak každé firmě zadat službu, aby byla celková cena minimální?

	Cena za hudbu	Cena za jídlo	Cena za úklid
Firma A	\$108	\$125	\$150
Firma B	\$150	\$135	\$175
Firma C	\$122	\$148	\$250

Tabulka 3.1 Ceník služeb

Na obrázku 3.1 je vyobrazen průběh algoritmu maďarské metody postupně v krocích. V prvním kroku se začíná s tabulkou cen, která je přebrána z tabulky 3.1. V kroku 2 se od každého řádku odečetla nejmenší hodnota řádku a od každého sloupce se odečetla nejmenší hodnota sloupce. Dalším krokem algoritmu je nalezení minimální pokrytí čar, což je vyobrazeno zešedivěním 1. sloupce a 2. řádku. Počet čar pokrytí není roven 3, takže se musí nalézt minimální nepokrytý prvek. Tím je zvýrazněné číslo 2. V kroku 4 se k řádku 2 přičte číslo 2 a od sloupců 2 a 3 se odečte číslo 2. Tímto vznikne nová nepokrytá nula na místě minimálního prvku. Nové minimální pokrytí čarami používá tři čáry, což je zobrazeno v kroku 5. Existují tedy 3 nezávislé nuly, které jsou zvýrazněny v poslední tabulce.

Z nalezení těchto nezávislých nul vyplývá, že nejvýhodnější přiřazení je, že firma A bude dělat úklid, firma B bude mít na starosti jídlo a firma C se obstará živou hudbu. Celková cena je $407. Lze jednoduše ověřit, že se jedná o optimální cenu, pomocí vyzkoušení všech 6 různých přiřazení. Tento příklad byl přebrán z [28].

Americký matematik James R. Munkres ve svém článku [29] z roku 1957 podrobně rozvedl Kuhnovu Maďarskou metodu. Popsal algoritmus hledání minimálního počtu čar pokrývající všechny nuly a hledání největší množiny nezávislých nul.

Pseudokód Munkresovy Maďarské metody je rozepsán v algoritmu 6. Při porovnání algoritmů 5 a 6 si lze všimnout určitých podobností. Algoritmus označuje nulové prvky matice. Nula v matici může být označena hvězdičkou, čárkou nebo nemá žádné označení. Na začátku jsou všechny prvky matice bez označení.

První 3 kroky v kratším zápisu algoritmu odpovídají krokům 1–11 v Munkresově variantě. Nulové prvky matice označené hvězdičkou jsou aktuálním nejlepším přiřazením, které se zatím podařilo nalézt. Pokud je hvězdičkou označených nul stejně jako sloupců v matici, algoritmus našel optimální přiřazení v matici.

Kroky 12–35 jsou podrobným popisem jak hledat nezávislé nuly a jak zvětšovat velikost přiřazení, aby se pokryly všechny sloupce. V této části se objevuje druhý typ označení nul čárkami. Pomocí nul označených čárkami se zvětšuje přiřazení, což je vidět v krocích 18–29. Kroky od 30 dál se vykonávají v případě, kdy neexistuje nepokrytá nula a počet čar pokrytí je menší než n. Je třeba nalézt minimální nepokrytý prvek matice a ten přičíst ke všem pokrytým řádkům a odečíst od všech nepokrytých sloupců. Tím se jistě získá nová nepokrytá nula a lze hledat nějaké lepší přiřazení.

Asymptotická složitost popsané Munkresovy varianty je $\mathcal{O}(n^4)$, pro matici $n \times n$ jako vstup. V článku [30] je však ukázáno jak zlepšit asymptotickou složitost na $\mathcal{O}(n^3)$. Zlepšení spočívá v zrychlení zdlouhavého odečítání a přičítání minimálních prvků matice v krocích 37–45. Využívá k tomu pomocná pole k hledání minima a línému přičítání a odčítání.

Algoritmus 6: Munkresova Maďarská metoda

Vstup : Matice vah M $n \times n$
Výstup: Optimální přiřazení

Pro každý řádek r matice M:
    $r \leftarrow r -$ minimum řádku r
Pro každý sloupec c matice M:
    $c \leftarrow c -$ minimum sloupce c
Pro každou nulu v matici M:
    Pokud ve stejném řádku a sloupci není již označená nula hvězdičkou:
        Označ takovou nulu hvězdičkou
$lines \leftarrow 0$
Pro každý sloupec c v M, který obsahuje nulu označenou hvězdičkou:
    Pokryj sloupec c
    $lines \leftarrow lines + 1$
Opakuj dokud $lines = n$:
    Dokud existuje v matici M nula Z, jejíž řádek a sloupec nejsou pokryty:
        Označ Z čárkou
        Pokud v řádku nuly Z existuje nula Z^* označená hvězdičkou:
            Pokryj řádek nuly Z^*
            Odkryj sloupec nuly Z^*
        Jinak:
            Zruš označení čárkou nuly Z
            Označ hvězdičkou nulu Z
            Dokud existuje další nula Z^+ označená hvězdičkou ve sloupci nuly Z:
                Zruš označení hvězdičkou u Z^+
                $Z \leftarrow$ nula označená čárkou z řádku nuly Z^+
                Zruš označení čárkou u Z
                Označ hvězdičkou Z
            $lines \leftarrow lines + 1$
            Zruš aktuální pokrytí řádků a sloupců matice
            Pro každý sloupec c matice M obsahující nulu označenou hvězdičkou:
                Pokryj sloupec c
    Pokud $lines < n$:
        $h \leftarrow$ minimální nepokrytý prvek matice M
        Pro každý pokrytý řádek r matice M:
            $r \leftarrow r + h$
        Pro každý nepokrytý sloupec c matice M:
            $c \leftarrow c - h$
Vrať pozice nul označených hvězdičkou jako optimální přiřazení

3.3 Toky v sítích

Přiřazovací problém lze převést na problém hledání maximálního toku v síti, konkrétně na hledání maximálního toku s minimální cenou v síti, která vznikne z ohodnoceného bipartitního grafu. [31] K představení takového algoritmu je třeba definovat několik pojmů okolo toků v síti.

3.3.1 Teorie

Definice, věty a důkazy v této podkapitole jsou převzaty z [7] a z [32].

▶ **Definice 3.2.** *Uspořádaná čtveřice (G, z, s, c) se nazývá síť, kde $G = (V, E)$ je orientovaný graf, z a s jsou dva různé vrcholy G (nazývají se zdroj a stok) a kapacita $c : E \to \mathbb{R}_0^+$ je funkce ohodnocující hrany nezápornými reálnými čísly.*

▶ **Definice 3.3.** Tok v síti *je každá funkce $f : E \to \mathbb{R}_0^+$ splňující:*

- *Pro každou hranu $e \in E$ platí $0 \leq f(e) \leq c(e)$.*
- *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 3.4.** Velikost toku *je*

$$w(f) = \sum_{(z,x)\in E} f(z,x) - \sum_{(x,z)\in E} f(x,z)$$

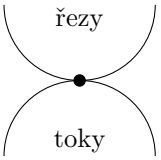
.

▶ **Definice 3.5.** Řezem *mezi zdrojem z a stokem s v síti (G, s, z, c), kde $G = (V, E)$, je množina hran $R \subseteq E$ taková, že v síti (G', z, s, c) neexistuje žádná orientovaná cesta ze zdroje do stoku, kde $G' = (V, E \setminus R)$.*

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

▶ **Věta 3.6** (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 3.2 Ilustrace Hlavní věty o tocích [7]

Důkaz. Důkaz Hlavní věty o tocích lze nalézt v přednášce 3 v [7]. ◀

▶ **Definice 3.7.** Cesta v síti (G, z, s, c) *(ve zbytku podkapitoly pouze cesta) je posloupnost $(v_0, e_1, v_1, \ldots, v_{m-1}, e_m, v_m)$, kde vrcholy $v_0, \ldots, v_m$ jsou navzájem různé vrcholy sítě a pro každé i je $e_i = (v_{i-1}, v_i)$ nebo $e_i = (v_i, v_{i-1})$.*

Tato definice cesty v síti se liší od definice cesty v grafu tím, že povoluje procházet orientované hrany i proti směru orientace.

▶ **Definice 3.8.** *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 směru je $f(e_i) = 0$.*
Cestě, která není nasycená, se říká nenasycená.
Nenasycená cesta ze zdroje do stoku se také často označuje jako zlepšující cesta.
Tok se nazývá nasycený, *když každá cesta ze zdroje do stoku je nasycená.*

▶ **Věta 3.9.** *Tok f je maximální, právě když je nasycený. Pro každý maximální tok f existuje řez R takový, že $w(f) = c(R)$.*

Důkaz. Důkaz lze nalézt v přednášce 3 v [7]. ◀

▶ **Definice 3.10.** *Uspořádaná pětice (G, z, s, c, a), kde (G, z, s, c) je síť a $a : E \to \mathbb{R}_0^+$ je funkce cenová funkce, se nazývá* síť s cenovou funkcí *a.*

▶ **Definice 3.11.** Cena toku *f v síti s cenovou funkcí a je rovna:*

$$\sum_{e \in E} a(e) f(e)$$

.

Poslední dvě definice přidávají možnost každé hraně v síti přiřadit hodnotu představující cenu využití této hrany v toku. Vzniká tak nový problém, hledání maximálního toku v síti při minimalizaci ceny.

▶ **Definice 3.12.** *Nechť $N = (G, z, s, c, a)$ je síť s cenovou funkcí a.* Reziduální síť *vzhledem k toku f se definuje jako pětice $N_f = (G_f, z, s, c_f, a)$, kde $G_f = (V, E_f)$,*

$$E_f = \{e \in E \mid f(e) < c(e)\} \cup \{\overleftarrow{e} \mid e \in E \wedge f(e) > 0\},$$

$\overleftarrow{e}$ značí opačnou orientaci hrany e a $c_f : E_f \to \mathbb{R}_0^+$ je definována jako: $c_f(e) = c(e) - f(e)$ a $c_f(\overleftarrow{e}) = f(e)$ pro každou hranu $e \in E$. Cenová funkce a je rozšířena na E_f jako $a(\overleftarrow{e}) = -a(e)$ pro každou hranu $e \in E$ a původním hranám přiřazuje původní hodnotu.

3.3.2 Fordův–Fulkersonův algoritmus

Důkaz Hlavní věty o tocích v [7] poskytuje návod jak v případě existence nenasycené cesty ji nasytit a zvětšit aktuální tok. Takovýto postup představili američtí matematici Lester R. Ford a Delbert R. Fulkerson ve svém článku [33] z roku 1956. Algoritmus se po nich nazývá Fordův–Fulkersonův.

V pseudokódu 7 je popsán jak funguje Fordův–Fulkersonův algoritmus. Algoritmus postupně hledá zlepšující – nenasycené – cesty v síti mezi zdrojem a stokem. Když takovou cestu P nalezne, spočítá ε_P podle následujícího předpisu:

$$\begin{aligned}
\varepsilon_1 &= \min\{c(e) - f(e) \mid e \in P \text{ je orientovaná po směru}\} \\
\varepsilon_2 &= \min\{f(e) \mid e \in P \text{ je orientovaná proti směru}\} \\
\varepsilon_P &= \min\{\varepsilon_1, \varepsilon_2\}
\end{aligned}$$

Poté vylepší tok f tak, že vytvoří nový tok f', který zvětší o ε_P následovně:

$$f'(e) = \begin{cases} f(e) + \varepsilon_P & e \in P \text{ je orientovaná ze zdroje do stoku} \\ f(e) - \varepsilon_P & e \in P \text{ je orientovaná proti směru} \\ f(e) & e \notin P \end{cases}$$

Algoritmus 7: Fordův–Fulkersonův algoritmus

Vstup : Síť (G, z, s, c)
Výstup: Maximální tok f

```
1 Pro každou hranu e:
2     f(e) ← 0
3 Dokud existuje zlepšující cesta:
4     Nalezni zlepšující cestu P
5     Vypočítej ε_P
6     Vylepši tok podél cesty P o ε_P
7 Vrať maximální tok f
```

Tím se nasytí cesta P a cyklus opět začíná hledat nějakou další zlepšující cestu v síti vzhledem k novému toku f'. Pokud žádná nová zlepšující cesta neexistuje, pak je podle Věty 3.9 tok maximální a algoritmus ho vrátí. Toto tvrzení platí pro sítě s celočíselnými a racionálními kapacitami. Pro reálné kapacity nemusí algoritmus doběhnout a ani nemusí konvergovat k maximálnímu toku. Důkaz korektnosti lze nalézt v [7].

Algoritmus neurčuje jak přesně se mají hledat zlepšující cesty. Pokud se zlepšující cesty hledají pomocí prohledávání do šířky, algoritmus se nazývá Edmondsův–Karpův [34]. Asymptotická složitost Edmondsova–Karpova algoritmu je $\mathcal{O}(|V||E|^2)$ [35].

3.3.3 Párování v ohodnoceném bipartitním grafu

K nalezení maximálního párování s minimální cenou lze využít modifikace Hopcroftova–Karpova algoritmu. Ten hledá maximální párování, ve smyslu počtu hran, v neohodnoceném bipartitním grafu. Algoritmus byl představen v roce 1973 americkými matematiky Johnem Hopcroftem a Richardem Karpem v článku [36].

Nechť $(A \cup B, E)$ je ohodnocený bipartitní graf, kde A, B jsou partity grafu a w je váhová funkce. Algoritmus 8 bere na vstupu síť, která se vytvoří následujícím postupem. Každý vrchol grafu se přidá do sítě. Za každou hranu $\{u, v\} \in E \wedge u \in A \wedge v \in B$ se do sítě přidá orientovaná hrana $e = (u, v)$ s cenou $a(e) = w(\{u, v\})$ rovné váze v původním grafu. Všechny zatím přidané hrany v síti jsou orientované z partity A do partity B. Do sítě se přidají dva nové vrcholy z a s, které budou zdroj a stok. Pro každý vrchol $u \in A$ se do sítě přidá hrana (z, u) s nulovou cenou. Pro každý vrchol $v \in B$ je přidá hrana (v, s) také s nulovou cenou. Všechny hrany v síti mají kapacitu $c(e) = 1$. Takto vybudovaná síť je vstupem algoritmu, který hledá optimální párování pomocí hledání maximálního toku při minimální ceně.

Algoritmus 8 postupně hledá cesty v reziduální síti, kterými by mohl vylepšit tok při minimalizaci ceny. Jakmile nalezne cestu s minimální vahou/cenou, odstraní první a poslední hrany cesty z G_f. To jsou hrany, které spojují vrcholy partity A, respektive B, se zdrojem, respektive se stokem. Poté upraví cenu hran, které nejsou podél cesty, pomocí funkce d, která představuje vzdálenost vrcholu od stoku. Toto upravení cen zaručí, že budou nezáporné a bude možné k hledání cesty použít Dijkstrův algoritmus. Poté následuje otočení hran hran cesty a nastavení ceny na 0. Upravení párování proběhne tak, že se jako párování berou hrany cesty P orientované z A do B.

Důkaz korektnosti algoritmu lze nalézt v [32]. Asymptotická složitost s využitím Dijkstrova algoritmu s Fibonacciho haldou je $\mathcal{O}(|V||E| + |V|^2 \log|V|)$ [32].

Algoritmus 8: Párování v ohodnoceném bipartitním grafu

Vstup : Síť $N = (G, z, s, c)$ vytvořená z bipartitního grafu
Výstup: Párování M

$M \leftarrow \emptyset$
Dokud existuje cesta z z do s v reziduální síti N_f:
    Nalezni takovou cestu P s minimální vahou
    Smaž první a poslední hranu cesty P z G_f
    Pro každou hranu $(u, v) \notin P$:
        $a(u, v) \leftarrow a(u, v) + d(u) - d(v)$
    Pro každou hranu $(u, v) \in P$:
        $(u, v) \leftarrow (v, u)$
        $a(v, u) \leftarrow 0$
    Uprav párování M podél P

3.4 Hledání eulerovského tahu

Posledním krokem řešení DCPP je nalezení eulerovského tahu v upraveném grafu. Doplněním hran podél nejkratších cest mezi vrcholy párování se každý nevyvážený vrchol stane vyváženým a již vyvážené vrcholy zůstanou stále vyvážené. Výsledný graf je tedy podle Věty 1.20 eulerovský a tedy v něm existuje eulerovský tah.

Podobně jako algoritmy na hledání nejkratších cest není hledání eulerovského tahu závislé na tom, zda vstupní graf je orientovaný či ne. Hierholzerův algoritmus je podrobně popsán v kapitole 2.3. Pro správné fungování algoritmu je třeba správně určit, se kterými hranami je vrchol incidentní. V neorientovaném grafu to jsou jak hrany začínající ve vrcholu, tak i ty, které v něm končí. To by ovšem způsobilo, že algoritmus by mohl procházet hrany i proti jejich orientaci. Proto je třeba označit jako incidentní hrany pouze takové, které v daném vrcholu začínají.

3.5 Modifikovaný Tarjanův algoritmus

Pro testování implementace je třeba generovat vstupní data, v orientovaném případě silně souvislé grafy. K získání pseudonáhodného silně souvislého grafu se využije modifikace Tarjanova algoritmu představená v [37]. Pro popis algoritmu je třeba zadefinovat komponenty silné souvislosti.

▶ **Definice 3.13.** *Nechť $\leftrightarrow$ je binární relace na vrcholech grafu definovaná tak, že $x \leftrightarrow y$ právě tehdy, když existuje orientovaná cesta jak z x do y, tak z y do x.*

Ekvivalenční třídy takto definované binární relace indukují podgrafy, které se nazývají komponenty silné souvislosti.

Uvedená definice komponent silné souvislosti je převzata z [20].

Tarjanův algoritmus je algoritmus na hledání silně souvislých komponent v orientovaném grafu. Tarjanův algoritmus prochází graf do hloubky a u každého vrcholu si pamatuje různé proměnné, podle kterých určuje komponenty silné souvislosti. Jak algoritmus přesně funguje je popsáno v [20, s. 134–137].

Modifikace spočívá v nahrazení části programu, který detekuje novou komponentu silné souvislosti, za přidání hrany mezi komponenty tak, že je propojí dohromady.

Algoritmus 9 zobrazuje začátek běhu, ve kterém se inicializuje *in* pro každý vrchol na nedefinovanou hodnotu. Tímto se všechny vrcholy označí jako nenavštívené. Dále se globální počítadlo

T nastaví na 0. Algoritmus očekává slabě souvislý graf, v tomto případě strom, který má orientované hrany směrem od nějakého vrcholu v_0. V implementaci je to vždy vrchol s číslem 0.

Procedura ModifikovanýTarjan začíná nastavením proměnných in, low a $inverseIn$ pro daný vrchol na hodnotu globálního počítadla T. Proměnná $inverseIn$ není nezbytná pro korektní běh algoritmu, ale pouze umožňuje rychlé vyhledání vrcholu s určitou hodnotou in. Kroky 5–9 jsou identické s původním Tarjanovým algoritmem. V nich se procedura volá rekurzivně a počítá proměnné in a low.

Krok 10 odpovídá detekci nové komponenty silné souvislosti. Pokud se vrchol v nachází v jiné komponentě než v_0, pak se vytvoří nová hrana, která tyto komponenty spojí. Vybere se náhodný vrchol z komponenty vrcholu v a komponenty vrcholu v_0 a spojí se tyto vrcholy hranou. Poté se musí aktualizovat $low(v)$. Tato změna může vést k nesprávným hodnotám low u jiných vrcholů, avšak k těm se algoritmus již vracet nebude [37].

Důkaz korektnosti algoritmu lze nalézt v [37]. Asymptotická složitost je $\mathcal{O}(|V|)$ [37].

Algoritmus 9: Modifikovaný Tarjanův algoritmus

Vstup : Orientovaný slabě souvislý graf G, vrchol v_0
Výstup: Orientovaný silně souvislý graf

1 Pro všechny vrcholy u v G:
2 $in(u) \leftarrow$ nedefinováno
3 $T \leftarrow 0$
4 ModifikovanýTarjan(v_0)

Procedura ModifikovanýTarjan(v)

Vstup : Orientovaný graf G, vrchol v

1 $in(v) \leftarrow T$
2 $low(v) \leftarrow T$
3 $inverseIn(v) \leftarrow T$
4 $T \leftarrow T + 1$
5 Pro všechny sousedy w vrcholu v:
6 Pokud $in(v)$ není definován:
7 ModifikovanýTarjan(w)
8 $low(v) \leftarrow \min(low(w), low(v))$
9 $low(v) \leftarrow \min(low(v), in(w))$
10 Pokud $in(v) == low(v) \wedge v \neq v_0$:
11 $x \leftarrow$ náhodné celé číslo z intervalu $\langle in(v), T-1\rangle$
12 $y \leftarrow$ náhodné celé číslo z intervalu $\langle 0, T\rangle$
13 $a \leftarrow inverseIn(x)$
14 $b \leftarrow inverseIn(y)$
15 Přidej hranu z vrcholu a do vrcholu b
16 $low(v) \leftarrow y$

Kapitola 4

Implementace

Tato kapitola se věnuje implementaci řešení Problému čínského listonoše. Využity byly algoritmy popsané v předchozích kapitolách.

Jak již bylo v předchozích kapitolách zmíněno obě varianty se dají rozdělit na několik kroků/podproblémů za sebou. Tyto kroky jsou si v obou variantách velmi podobné a některé z nich se dají řešit stejnými algoritmy. Schéma na obrázku 4.1 zobrazuje jednotlivé stupně řešení jako sekvenci oddělených kroků (anglicky pipeline). Zeleně označené kroky odpovídají algoritmům, které byly představeny v kapitolách 2 a 3. Samotné implementace algoritmů se dají poměrně jednoduše vyměňovat. To uživateli umožňuje zvolit, jaké implementace se pro jaký krok mají použít nebo použít vlastní implementaci nějakého kroku.

Samotné řešení je rozděleno do několika složek, podle typu třídy. Složka *Graph* obsahuje základní datové struktury pro práci s grafy. Dále obsahuje třídy, které generují nebo načítají grafy ze souboru. Složka *Algorithm* obsahuje rozhraní a implementace algoritmů, které se použijí k řešení jednotlivých variant CPP. Složka *Pipeline* obsahuje implementace sekvencí kroků pro řešení neorientované a orientované varianty. Složka *Utils* obsahuje třídy, které přímo nesouvisí s CPP, ale pomáhají vytvořit výsledný program.

4.1 Použité technologie

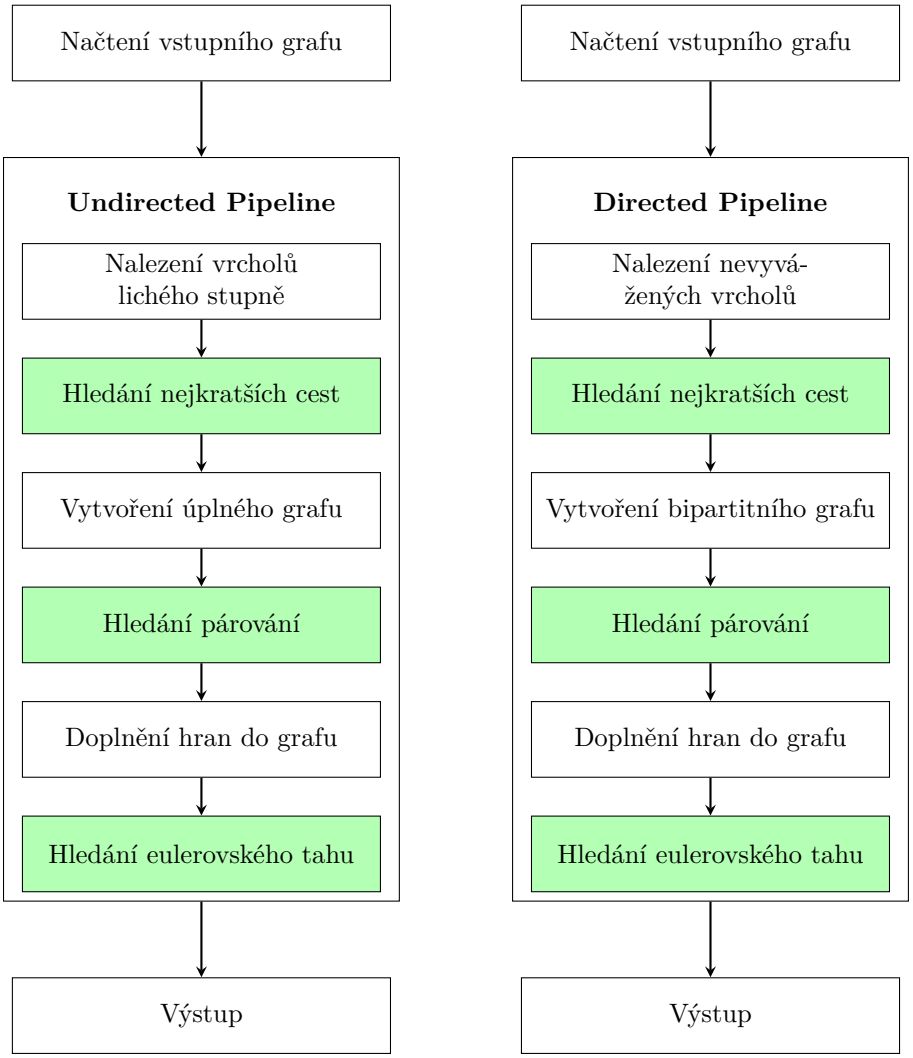
4.1.1 Programovací jazyk

V zadání této práce byl pro implementaci určen jazyk C++. Je to velmi rozšířený programovací jazyk a je takzvaně multiparadigmatický. V této práci je použit hlavně jako objektově orientovaný jazyk. Kód je rozdělen do tříd, které zabalují určitou funkcionalitu. Jedna z výhod C++ je kompilace přímo do strojového kódu, což umožňuje kompilátoru provádět nízkoúrovňové optimalizace.

Pro C++ existuje mnoho knihoven. Některé z nich jsou použity v této práci k ulehčení implementace a zkrácení vlastního kódu. Knihovny jsou často odladěné k co nejrychlejšímu běhu. Vlastní implementace některých algoritmů se tak těžko vyrovná, natož překoná, implementaci v knihovně.

4.1.2 Knihovny

Kromě standardní knihovny C++ byly využity i jiné knihovny třetích stran.



Obrázek 4.1 Schéma řešení neorientované (vlevo) a orientované (vpravo) varianty CPP

4.1.2.1 Boost

Boost je kolekce volně přístupných knihoven pro C++. Boost nabízí knihovny pro téměř jakékoliv užití. V této práci se využívá hlavně knihovna *Boost.Heap*, která obsahuje implementaci prioritních front. Prioritní fronta využita v této práci je Fibonacciho halda a to v implementaci Dijkstrova algoritmu. Další využitou knihovnou je *The Boost Graph Library* (zkráceně BGL) pro její implementaci algoritmu na hledání maximálního váženého párování. Bohužel se ukázalo, že implementace obsahuje chybu. Tato chyba je již nahlášena autorům.

Verze knihovny Boost použité v této práci je 1.75.0 [38].

4.1.2.2 LEMON

LEMON je knihovna napsaná v C++, která poskytuje efektivní implementace běžně užívaných datových struktur a algoritmů k práci s grafy a sítěmi. Knihovna LEMON byla použita jako náhrada za knihovnu Boost Graph Library. LEMON obsahuje fungující algoritmus hledání maximálního váženého párování. Na rozdíl od BGL, LEMON nepoužívá šablony k definici grafů, což přináší určité výhody [39].

Verze knihovny LEMON použité v této práci je 1.3.1 [26].

4.1.2.3 Blossom V

Blossom V je název implementace Edmondsova květinového algoritmu na hledání perfektního minimálního párování. Byla představena v článku [40] z roku 2009. Blossom V předčila svou efektivitou předchozí implementace. Je považována za nejefektivnější implementaci Edmondsova algoritmu [41].

Verze implementace Blossom V použité v této práci je 2.05.

4.1.2.4 OpenMP

Součástí zadání je paralelizace implementovaných algoritmů pomocí OpenMP. OpenMP je soubor direktiv překladače, proměnných a funkcí pro paralelní programování. Nabízí jednoduché vysokoúrovňové rozhraní pro vytvoření vícevláknového programu.

Paralelizace algoritmů se provede přidáním vhodných direktiv do těla algoritmů. Podrobný popis fungování lze nalézt v oficiální specifikaci OpenMP [42].

Verze OpenMP použité v této práci je 4.5.

4.2 Společné třídy

Vzhledem k podobnosti obou variant problému jsou některé některé třídy sdíleny mezi oběma variantami. Aby se implementace algoritmů mohly použít jak v neorientované, tak v orientované variantě, je třeba vytvořit rozhraní, které bude nezávislé na typu vstupního grafu.

4.2.1 Abstraktní třída Graph

Třída Graph je abstraktní třída, která určuje jaké metody by podtřída reprezentující graf měla implementovat. Tato třída je základním kamenem pro využití například Dijkstrova algoritmu nezávisle na tom, zda je graf orientovaný. Vrcholy jsou ukládány jako celá čísla v pořadí od 0. Jednotlivé hrany jsou reprezentovány pomocí struktury Edge.
Seznam metod:

- **addEdge** přidá do grafu hranu mezi vrchcoly s danou váhou,
- **setVertexCount** nastaví počet vrcholů grafu,

- **vertexCount** vrátí počet vrcholů grafu,
- **getWeight** vrátí váhu hrany mezi dvěma vrcholy,
- **getEdges** vrátí všechny hrany začínající v daném vrcholu,
- **operator<<** vypíše graf do streamu v jednoduchém formátu.

4.2.2 Struktura Edge

Struktura Edge reprezentuje hranu v grafu. Jedná se o trojici celých čísel. Dvě čísla určují vrcholy v grafu, které hrana spojuje. Proměnná *from* je začátek hrany a *to* označuje konec hrany. Třetím číslem, *weight*, je váha hrany.

Implementace obsahuje operátor porovnání a specializaci šablony struktury `std::hash` ze standardní knihovny. Tato implementace umožňuje použití jako klíče v `std::map`, avšak ve finální verzi programu se toho nevyužívá.

4.2.3 Třída Matrix

Třída Matrix je pomocná třída, která reprezentuje celočíselnou matici. Implementována je pomocí dvojitého použití šablony `std::vector`. Třída umožňuje pouze číst a zapisovat na danou pozici. Seznam metod:

- **get** vrátí prvek matice na dané pozici,
- **set** nastaví prvek matice na dané pozici na danou hodnotu,
- **rows** vrátí počet řádků matice,
- **cols** vrátí počet sloupců matice,
- **operator[]** vrátí referenci na specifikovaný řádek.

4.2.4 Třída Tour

Třída Tour reprezentuje tah v grafu jako sekvenci po sobě jdoucích vrcholů. Instance třídy si pamatuje i celkovou váhu tahu, takže není třeba počítat váhu po nalezení tahu, lze to počítat už při hledání tahu. Sekvence vrcholů je uložena jako `std::vector`.
Seznam metod:

- **empty** statická metoda, která vrací prázdný tah – tah nulové délky,
- **operator<<** vypíše celkovou váhu hran v tahu a poté jednotlivé vrcholy v pořadí v tahu.

4.2.5 Abstraktní třída PathFinder

Abstraktní třída PathFinder určuje jaké rozhraní musí implementovat algoritmy, aby mohly být použity pro hledání nejkratších cest v grafu při řešení CPP. Třída, která od této dědí, musí implementovat hledání nejkratší cesty mezi dvěma vrcholy a spočítání matice vzdáleností mezi dvojicemi vrcholů.
Seznam metod:

- **findPath** nalezne nejkratší cestu mezi dvěma vrcholy, vrátí její délku a do výstupního parametru uloží vrcholy, kterými cesta prochází,

- **getDistanceMatrix** spočítá a vrátí matici vzdáleností mezi dvojicemi vrcholů specifikovanými argumenty,
- **setGraph** metoda uloží graf, na kterém se budou počítat nejkratší vzdálenosti.

4.2.6 Třída Dijkstra

Třída Dijkstra dědí od třídy PathFinder. Implementuje Dijkstrův algoritmus, který hledá nejkratší cesty z jednoho vrcholu do ostatních. Pro získání matice vzdálenosti je Dijkstrův algoritmus postupně spouštěn z jednotlivých vrcholů. Pro co nejlepší asymptotickou časovou složitost je využita Fibonacciho halda z knihovny Boost.

Kromě implementace zděděných metod z třídy PathFinder tato třída navíc obsahuje metodu calculateFromTo na nalezení nejkratších cest z jednoho vrcholu. Tato metoda je samotnou implementací Dijkstrova algoritmu. Ta se používá například v implementaci algoritmu 8 pro hledání cesty s nejmenší váhou ze zdroje sítě. Metody findPath a getDistanceMatrix očekávají, že se graf nemění mezi voláním jednotlivých metod. Pro správné fungování je třeba zavolat znovu metodu setGraph, která resetuje dosud napočítané výsledky.
Seznam metod:

- **findPath** metoda zděděna z třídy PathFinder,
- **getDistanceMatrix** metoda zděděna z třídy PathFinder,
- **setGraph** metoda zděděna z třídy PathFinder,
- **calculateFromTo** nalezne nejkratší cestu z grafu z předaného vrcholu do vrcholů specifikovaných argumentem volání, očekává předání polí pro udržování vzdáleností a předchůdců, které mají velikost rovnu počtu vrcholů v grafu,
- **getUsedEdges** vrátí iterátory obsahující hrany použité v nejkratších cestách při posledním volání metody calculateFromTo.

Třída si při prvním volání metod findPath nebo getDistanceMatrix předpočítá matici vzdáleností a matici předchůdců, ze kterých poté vytváří výsledky. Toto umožňuje například opakovaně volat metodu findPath se stejnými parametry bez nutnosti opakovaného přepočítávání nejkratších cest. Předpočítání matice vzdáleností trvá $\mathcal{O}(k \cdot (|E| + |V| \log |V|))$, kde k značí počet vrcholů, ze kterých jsou nejkratší cesty hledány. V neorientované variantě to odpovídá počtu vrcholů lichého stupně a v orientované počtu vrcholů v D^- – takových jejichž rozdíl výstupního a vstupního stupně vrcholu je záporný. V obou variantách je tento počet až $|V|$, tudíž celková asymptotická složitost odpovídá $\mathcal{O}(|V||E| + |V|^2 \log |V|)$.

Po předpočítání matice vzdáleností a předchůdců jednotlivé volání metod findPath a getDistnaceMatrix trvá kratší dobu než když by se cesty musely vždy počítat znovu. Metoda findPath spočítá délku nejkratší cesty mezi dvěma vrcholy, což je díky matici vzdáleností konstantní operace a z matice předchůdců získá jednotlivé vrcholy podél nejkratší cesty. Extrakce vrcholů podél cesty z pole předchůdců se provádí od koncového vrcholu směrem k začátku. To trvá v nejhorším případě $\mathcal{O}(|V|)$. Metoda getDistanceMatrix pouze musí zkopírovat z matice vzdáleností vhodné hodnoty, to trvá maximálně $\mathcal{O}(|V|^2)$.

4.2.7 Třída FloydWarshall

Třída FloydWarshall dědí od třídy PathFinder. Implementuje Floydův–Warshallův algoritmus, který hledá nejkratší cesty mezi dvojicemi vrcholů v grafu. Floydův–Warshallův algoritmus začne s maticí vah hran a postupně počítá výslednou matici vzdáleností mezi všemi dvojicemi vrcholů v grafu.
Seznam metod:

- **findPath** metoda zděděna z třídy PathFinder,
- **getDistanceMatrix** metoda zděděna z třídy PathFinder,
- **setGraph** metoda zděděna z třídy PathFinder.

Podobně jako u třídy Dijkstra se při prvním volání metod findPath nebo getDistanceMatrix předpočítá matice vzdáleností a následníků. Spočítání těchto matic trvá $\Theta(|V|^3)$, nezávisle na tom, z kterých vrcholů se cesty mají počítat. Po spočítání matic asymptotická složitost findPath odpovídá vytvoření cesty z matice následníků, což je $\mathcal{O}(|V|)$. Metoda getDistanceMatrix pouze překopíruje vhodné hodnoty do výsledné matice a to se provede v čase $\mathcal{O}(|V|^2)$.

4.2.8 Abstraktní třída EulerTourFinder

Abstraktní třída EulerTourFinder určuje rozhraní třídám, které implementují algoritmy na hledání eulerovského tahu v grafu.
Seznam metod:

- **findEulerTour** nalezne uzavřený eulerovský tah v grafu pokud existuje.

4.2.9 Třída Hierholzer

Třída Hierholzer je podtřída abstraktní třídy EulerTourFinder. Třída hledá eulerovský tah v grafu pomocí Hierholzerova algoritmu. Tento algoritmus funguje korektně nezávisle na orientaci grafu avšak implementace je rozdílná pro neorientovaný a orientovaný graf.
Seznam metod:

- **findEulerTour** přetížená metoda pro abstraktní třídu Graph a třídy UndirectedGraph a DirectedGraph.

Metoda findEulerTour má tři varianty. První je zděděná z třídy EulerTourFinder, jako argument bere konstantní referenci na instanci třídy Graf a volá jednu z dalších metod podle typu daného grafu. Zbývající dvě varianty jsou speciální implementace pro neorientovaný graf a orientovaný graf.

Obě implementace používají iterátory, takže každá hrana je zpracována pouze jednou. U neorientované varianty se každá hrana vezme v potaz dvakrát, ale to je stále konstanta. Implementace pro neorientovaný graf se od té pro orientovaný liší kvůli reprezentaci neorientovaných hran. Každá hrana je v neorientovaném grafu uložena dvakrát, jednou u každého z koncových vrcholů. Aby bylo mazaní hran (nebo označování hran jako smazané) možné provést v konstantním čase, je třeba využít navíc matici, která obsahuje počet volných hran mezi dvěma vrcholy.

Asymptotická složitost metody findEulerTour pro orientovaný graf je lineární vzhledem k počtu hran vstupního grafu. Inicializace trvá $\mathcal{O}(|V|)$ a samotný algoritmus trvá $\mathcal{O}(|E|)$. Vstupní graf je vždy souvislý, takže platí $|V| \leq |E| + 1$ a celková složitost je tedy $\mathcal{O}(|E|)$.

Asymptotická složitost metody findEulerTour pro neorientovaný graf je oproti orientované variantě zatížena kopírováním matice počtu volných hran. Inicializace tedy trvá $\mathcal{O}(|V|^2)$ a vlastní algoritmus má složitost $\mathcal{O}(|E|)$. Celková složitost vychází jako $\mathcal{O}(|E|+|V|^2)$. Algoritmus by se dal implementovat lépe, avšak to by znamenalo nějakou změnu struktury Edge nebo komplexnější odebírání hran z grafu. Tato úprava není potřeba, protože kopírování matice má velmi malý vliv na celkovou dobu běhu programu.

4.2.10 Třída GraphLoader

Třída GraphLoader je pomocná třída pro načítání grafů ze souboru. GraphLoader umí načítat obě varianty grafu. Formát vstupu je následující. První číslo n určuje počet vrcholů. Poté

následuje libovolný počet trojic celých čísel a, b, w. Každé hraně grafu odpovídá jedna trojice. Číslo a určuje, ve kterém vrcholu hrana začíná. Číslo b určuje, ve kterém vrcholu hrana končí. Číslo w určuje váhu hrany. Přípustné hodnoty pro čísla a, b jsou z množiny $\{0, \ldots, n-1\}$. Váha hrany musí být nezáporná. Pro neorientované grafy stačí uvést hranu pouze v jednom směru. GraphLoader využívá rozhraní třídy Graph, které zaručuje přítomnost metody addEdge, která řeší přidání hrany mezi dva vrcholy.
Seznam metod:

- **loadUndirected** načte neorientovaný graf ze souboru nebo z vstupního streamu
- **loadDirected** načte orientovaný graf ze souboru nebo z vstupního streamu

4.2.11 Třída GraphGenerator

Třída GraphGenerator je pomocná třída pro generování pseudonáhodných neorientovaných a orientovaných grafů. Při volání konstruktoru je možné předat číslo, které bude použito jako *seed* pro generátor náhodných čísel. Jako generátor je použita implementace generátoru Mersenne Twister ve standardní knihovně `std::mt19937`. Váhy hran jsou generovány pomocí rovnoměrného rozdělení na intervalu $\langle 1, 100 \rangle$.
Seznam metod:

- **generateUndirected** vygeneruje pseudonáhodný souvislý neorientovaný graf se zadaným počtem vrcholů a hran
- **generateUndirectedWithDensity** spočítá z počtu vrcholů a hustoty počet hran a zavolá metodu generateUndirected
- **generateDirected** vygeneruje pseudonáhodný silně souvislý orientovaný graf se zadaným počtem vrcholů a hran
- **generateDirectedWithDensity** spočítá z počtu vrcholů a hustoty počet hran a zavolá metodu generateDirected

Metoda generateUndirected generuje souvislé neorientované grafy v několika krocích. První se otestuje zda požadovaný počet hran je ve správném rozmezí. Při n vrcholech musí být počet hran v intervalu $\langle n-1, n(n-1)/2 \rangle$. Poté se vygeneruje náhodný strom, který zaručí souvislost grafu. Poté se naleznou všechny hrany, které mohou být přidány do grafu, zamíchají se a do grafu se přidá prvních $m-n+1$ hran, kde m značí počet požadovaných hran. Zamíchání a výběr prvních k prvků je ekvivalentní náhodnému výběru k prvků, avšak takto je to jednodušší implementovat. Asymptotická složitost generování grafu je $\mathcal{O}(n^2)$, kde n je počet vrcholů generovaného grafu.

Metoda generateDirected generuje silně souvislé orientované grafy. Stejně jako generovaní neorientovaných grafů i tato metoda začíná s nalezením náhodného stromu. Hrany ve stromu musí být orientovány tak, aby do každého vrcholu existovala cesta z vrcholu 0. Další krok je vytvoření silně souvislého grafu. Tento krok se provede pomocí modifikovaného Tarjanova algoritmu, který je popsán v sekci 3.5. Poté se stejným způsobem jako u neorientované varianty se přidají dodatečné hrany, které ještě nebyly přidány. Modifikovaný Tarjanův algoritmus běží lineárně dlouho vzhledem k počtu vrcholů. Celková složitost generování orientovaného grafu je v nejhorším případě $\mathcal{O}(n^2)$, kde n je požadovaný počet vrcholů.

4.2.12 Třída Configuration

Třída Configuration je pomocnou třídou, která umožňuje měnit běh programu bez nutnosti kompilace. Díky této třídě je možné měnit implementace algoritmů, způsob získávání vstupního grafu nebo také kolik vláken bude použito při běhu programu.
Seznam veřejných členských proměnných:

- **graphType** určuje jaký typ grafu se bude zpracovávat a tedy jaký postup řešení se má zvolit,
- **pathFinderAlg** určuje jaký algoritmus se má použít pro hledání nejkratších cest,
- **matching** určuje jaký algoritmus se má použít na hledání párování v grafu nebo řešení Přiřazovacího problému v závislosti na variantě problému,
- **graphProvider** určuje jak se získá vstupní graf, buď se načte ze souboru nebo vygeneruje podle zadaných parametrů,
- **graphFilename** obsahuje cestu k souboru odkud se má případně načítat vstupní graf,
- **graphNumberOfVertices** obsahuje počet vrcholů grafu, který se má vygenerovat,
- **graphNumberOfEdges** obsahuje počet hran grafu, který se má vygenerovat,
- **graphDensity** obsahuje hustotu grafu, který se má vygenerovat,
- **seed** nastavuje *seed* pro náhodný generátor,
- **threadsCount** určuje počet vláken k použití.

Seznam metod:

- **getPathFinder** vytvoří instanci třídy PathFinder v závislosti na proměnné pathFinderAlg,
- **getMatching** vytvoří instanci třídy MatchingAlgorithm podle proměnné matching,
- **getBipartiteMatching** vytvoří instanci třídy BipartiteMatchingAlgorithm podle proměnné matching.

4.2.13 Třída ConfigurationLoader

Třída ConfigurationLoader poskytuje rozhraní k načítání konfigurace ze souboru nebo přímo z příkazové řádky.
Seznam metod:

- **loadFromCommandLineArgs** zjistí zda je předán soubor s konfigurací jako argument nebo je konfigurace zaznamenána přímo v argumentech příkazové řádky, podle toho zavolá metodu loadFromFile nebo loadFromArgs,
- **loadFromFile** načte konfiguraci ze souboru,
- **loadFromArgs** načte konfiguraci z argumentů příkazové řádky.

4.2.14 Soubor Profiler.hpp

Soubor Profiler.hpp obsahuje třídy používané k měření času a zapisování měření do souboru. K samotnému měření času se využívá `std::chrono` ze standardní knihovny C++. Tento soubor není funkční součástí řešení. Jedná se pouze o podpůrné třídy.

Základ souboru je převzat z [43], ale byl velmi silně modifikován a doplněn o další funkcionalitu. Třída ScopedTimer provádí měření času. Měření lze opakovaně zastavovat a spouštět. Třída Instrumentor se stará o zapisování výsledků měření do souboru.

4.3 Třídy specifické pro neorientovanou variantu

V neorientované variantě se řeší problém hledání minimálního párování v obecném grafu, který se u orientované varianty neřeší.

4.3.1 Třída UndirectedGraph

Třída UndiredctedGraph je potomkem třídy Graph. Jedná se o specializaci, která reprezentuje neorientovaný graf. Vrcholy v grafu jsou číslovány od 0 a hrany jsou uloženy jako seznam sousedů pomocí šablony `std::vector`. Váhy hran se navíc ukládají v matici pro rychlé dohledání váhy mezi jakýmikoli dvěma vrcholy. Třída reprezentuje správně i multigrafy bez smyček, ale musí platit, že váha hran mezi dvěma stejnými vrcholy je vždy stejná.
Seznam metod:

- *metody zděděné z třídy Graph*,
- **getDegree** vrátí stupeň daného vrcholu v grafu,
- **getDegrees** vrátí všechny stupně vrcholů v pořadí,
- **getEdgeCountMatrix** vrátí matici obsahující počty hran mezi dvojicemi vrcholů,
- **operátor<<** vypíše graf v jednoduchém formátu výčtu hran, hrany jsou vypisovány tak, že první vrchol má nižší číslo.

4.3.2 Abstraktní třída MatchingAlgorithm

Abstraktní třída MatchingAlgorithm definuje jak má vypadat rozhraní tříd, které implementují algoritmus na hledání minimálního perfektního párování.

Třída obsahuje jedinou metodu, getMatching, která bere jako argument matici typu $n \times n$, která obsahuje váhy hran mezi vrcholy. Při řešení UCPP se vždy hledá párování v úplném grafu se sudým počtem vrcholů, který je má kladné váhy hran. Toto předpokládá i samotná metoda, všechny prvky matice musí být definovány. Matice je jistě symetrická, protože graf je neorientovaný.
Seznam metod:

- **getMatching** nalezne minimální perfektní párování v úplném grafu a vrátí ho jako seznam dvojic.

„Ačkoli pro oba problémy (problém maximálního kardinálního párování a problém minimálního perfektního párování) existují velmi efektivní algoritmy, jejich implementace je velmi komplexní a vyžaduje značné množství práce. Z toho důvodu existuje malé množství komerčních a nekomerčních knihoven, které obsahují implementace algoritmů těchto problémů.“ [41]

Jak již bylo zmíněno v kapitole 2.2 Edmondsův květinový algoritmus je poměrně složitý a proto byly využity implementace dostupných knihoven. Byla využita knihovna LEMON a implementace Blossom V. Knihovna Boost sice nabízí funkci na hledání maximálního párování v obecném grafu, ale obsahuje nějakou chybu, protože při testování s grafy s 25 vrcholy funkce vracela špatné výsledky nebo vyhodila výjimku. Další volně přístupné knihovny jako například igraph, ngraph, Goblin nebo SNAP neobsahují implementaci algoritmu.

4.3.3 Třída NaiveMatching

Třída NaiveMatching je podtřídou MatchingAlgorithm a implementuje naivní hledání minimálního perfektního párování. V metodě getMatching se postupně generují všechny možné kombinace vrcholů a hledá se taková, která má nejnižší součet vah. Toto řešení je však velmi neefektivní, protože počet perfektních párování je roven

$$\frac{(2n)!}{2^n \cdot n!},$$

kde $2n$ je počet vrcholů úplného grafu.[†]
Seznam metod:

- **getMatching** vyzkouší všechny možné párování vrcholů a vrátí to s nejnižší váhou.

4.3.4 Třída Blossom5Matching

Třída Blossom5Matching dědí z třídy MatchingAlgorithm a využívá implementaci Edmondsova květinového algoritmu Blossom V. Celá implementace je ve složce *Algorithm/blossom5*.
Seznam metod:

- **getMatching** pomocí implementace Blossom V nalezne minimální perfektní párování, volá metodu třetí strany.

Metoda getMatching nejdříve vytvoří instanci třídy PerfectMatching, což je třída z implementace Blossom V. Postupně se přidají všechny hrany úplného grafu s korespondujícími vahami. Zavolá se metoda solve (metoda třetí strany z Blossom V) a poté se extrahuje nalezené řešení. Celková asymptotická složitost metody je $\mathcal{O}(n^3 \log n)$, kde n je počet vrcholů grafu, ve kterém se hledá párování [40].

4.3.5 Třída LemonMatching

Třída LemonMatching je podtřída abstraktní třídy MatchingAlgorithm. Minimální perfektní párování hledá pomocí algoritmu na hledání maximálního párování z knihovny LEMON.
Seznam metod:

- **getMatching** nalezne minimální perfektní párování v grafu s pomocí knihovny LEMON, volá metodu třetí strany.

Metoda getMatching hledá minimální perfektní párování, avšak knihovna LEMON poskytuje algoritmus na maximální párování. Jak bylo ukázáno v kapitole 2.2.1 problém minimálního perfektního párování pro kladně ohodnocený úplný graf lze převést na hledání maximálního párování v upraveném grafu. První se tedy provede úprava grafu a poté se zavolá metoda z knihovny LEMON, nakonec se extrahuje řešení.

Při implementaci byla pro reprezentaci grafu použita třída `lemon::FullGraph`. Použití této třídy snížilo čas strávený v metodě getMatching až o 25 % oproti použití `lemon::SmartGraph` nebo `lemon::ListGraph`. Celková asymptotická složitost metody je $\mathcal{O}(n^3 \log n)$, kde n je počet vrcholů grafu, ve kterém se hledá párování [26].

4.3.6 Třída BoostMatching

Třída BoostMatching dědí od třídy MatchingAlgorithm. Využívá knihovnu Boost, konkrétně BGL, která obsahuje funkci na hledání maximálního párování. Implementace je ovšem chybná. V některých případech vrací špatný výsledek a někdy vyhodí výjimku. Princip fungování metody getMatching je stejný jako u třídy LemonMatching.
Seznam metod:

- **getMatching metoda nevrací správné výsledky.**

[†]Posloupnost počtu perfektních párování v grafu K_{2n} je zaznamenána v encyklopedii posloupností OEIS a má označení A001147.

4.3.7 Třída UndirectedPipeline

Třída UndirectedPipeline je hlavní třída řešení neorientované varianty CPP. Třída v metodě run postupně vyhodnocuje kroky řešení UCPP, které jsou vidět na obrázku 4.1 vlevo. Zeleně označeny jsou kroky, které se řeší externě v jiných třídách. Rozhraní konstruktoru umožňuje použití různých implementací jednotlivých algoritmů při řešení.
Seznam metod:

- **run** vyřeší UCPP pro vstupní graf a vrátí nalezený tah.

Metoda run nalezne ve vstupním grafu vrcholy lichého stupně. Poté spočítá nejkratší cesty mezi všemi dvojicemi vrcholů lichého stupně. Získaná matice vzdáleností se stane vstupem do hledání minimálního perfektního párování. Podél nejkratších cest mezi vrcholy nalezeného párování se přidají hrany do grafu. Tím vznikne eulerovský graf. Posledním krokem je nalezení eulerovského tahu. Metoda tento tah vrátí. Pro jednotlivé kroky se používají třídy popsané dříve.

Nalezení vrcholů lichého stupně trvá lineárně dlouho vzhledem k počtu vrcholů vstupního grafu. Počet vrcholů lichého stupně může být až $|V|$. Přidání hran podél nejkratších cest trvá $\mathcal{O}(|E|)$, protože každá hrana může být přidána maximálně jednou [10]. Celková časová složitost metody run se mění v závislosti na použitých algoritmech hledání nejkratších cest, minimálního párování a eulerovského tahu.

4.4 Třídy specifické pro orientovanou variantu

V orientované variantě CPP se hledá minimální perfektní párování v grafu, který je bipartitní. Problém se dá převést na Přiřazovací problém.

4.4.1 Třída DirectedGraph

Třída DirectedGraph je podtřída třídy Graph. Jedná se o specializaci, která reprezentuje orientovaný graf. Stejně jako u třídy UndirectedGraph, hrany jsou uloženy jako seznam sousedů pomocí šablony `std::vector`. Dále se váhy hran ukládají do matice pro vyhledání váhy mezi dvěma vrcholy v konstantním čase. Navíc jsou přítomny dvě pole, které drží pro každý vrchol jeho vstupní a výstupní vrchol. Třída reprezentuje správně multigrafy bez smyček, kde pro dvě hrany, které mají stejné konce platí, že mají stejnou váhu. Toto omezení při řešení DCPP nijak nevadí.
Seznam metod:

- *metody zděděné z třídy Graph*,
- **getInDegree** vrátí vstupní stupeň daného vrcholu v grafu,
- **getOutDegree** vrátí výstupní stupeň daného vrcholu v grafu,
- **getDegree** vrátí stupeň daného vrcholu v grafu,
- **getDegreeDiff** vrátí rozdíl výstupního a vstupního stupně daného vrcholu v grafu,
- **getEdgeCountMatrix** vrátí matici obsahující počty hran mezi dvojicemi vrcholů,
- **operátor<<** vypíše graf v jednoduchém formátu pomocí výčtu hran.

4.4.2 Abstraktní třída BipartiteMatchingAlgorithm

Abstraktní třída BipartiteMatchingAlgorithm určuje rozhraní třídám, které implementují algoritmus na hledání minimální perfektní párování v bipartitním grafu. Třída obsahuje jedinou metodu getMatching, která bere matici vah mezi vrcholy grafu jako argument.
Seznam metod:

- **getMatching** nalezne minimální perfektní párování v úplném bipartitním grafu a vrátí ho jako seznam dvojic.

4.4.3 Třída NaiveBipartiteMatching

Třída NaiveBipartiteMatching dědí od abstraktní třídy BipartiteMatchingAlgorithm a implementuje naivní řešení hledání minimálního perfektního párování. Řešení hledá průchodem všech možných perfektních párování a nalezením takového, které je minimální. Každé perfektní párování v grafu $K_{n,n}$ odpovídá nějaké permutaci množiny o n prvcích.
Seznam metod:

- **getMatching** nalezne minimální perfektní párování v úplném bipartitním grafu a vrátí ho jako seznam dvojic.

Metoda getMatching využívá k průchodu všech permutací funkci `std::next_permutation`. Pro každou permutaci spočítá součet vah hran odpovídajícího párování a porovná to s dosud nalezeným minimem. Takovéto řešení je velmi neefektivní, protože počet permutací je roven $n!$ a pokaždé se musí sečíst n čísel.

4.4.4 Třída HungarianMethod

Třída HungarianMethod je podtřídou BipartiteMatchingAlgorithm. Hledání minimálního perfektního párování převádí na Přiřazovací problém a ten řeší pomocí Maďarské metody.
Seznam metod:

- **getMatching** nalezne optimální přiřazení a vrátí ho jako seznam dvojic.

Metoda getMatching odpovídá pseudokódu 6, který popisuje Munkresovu variantu Maďarské metody. Velikost vstupu odpovídá součtu $\sum_{v \in D^+} \delta(v)$ což může být v nejhorším případě $\mathcal{O}(|E|)$. Munkresova Maďarská metoda běží v čase $\mathcal{O}(n^3)$, kde n je počet řádků vstupní matice. To celkově dává asymptotickou složitost $\mathcal{O}(|E|^3)$ pro vstupní graf DCPP.

4.4.5 Třída MinCostFlowMatching

Třída MinCostFlowMatching dědí od třídy BipartiteMatchingAlgorithm a k hledání optimálního párování využívá algoritmus na hledání maximálního toku v síti při minimalizaci ceny.
Seznam metod:

- **getMatching** nalezne optimální párování pomocí nalezení maximálního toku při minimální ceně.

Metoda getMatching začíná vytvořením sítě z bipartitního grafu. Poté postupně vylepšuje tok, který odpovídá párování. Jakmile algoritmus nalezne maximální tok, vytvoří se výsledný seznam dvojic optimálního párování. Pro vstupní graf $G = (V, E)$ DCPP má, podle stejné úvahy jako u HungarianMethod, vzniklá síť $\mathcal{O}(|E|)$ vrcholů a $\mathcal{O}(|E|^2)$ hran. Tudíž celková složitost algoritmu je $\mathcal{O}(|E| \cdot |E|^2 + |E|^2 \log |E|)$, což po zjednodušení odpovídá $\mathcal{O}(|E|^3)$.

4.4.6 Třída DirectedPipeline

Třída DirectedPipeline je hlavní třídou řešení orientované varianty CPP. Třída v metodě run postupně vykonává kroky řešení DCPP, které jsou vidět na obrázku 4.1 vpravo. Zeleně jsou označeny kroky, které jsou implementovány v jiných třídách, které jsou pouze volány. Rozhraní konstruktoru umožňuje volbu různých implementací jednotlivých algoritmů při řešení.
Seznam metod:

- **run** vyřeší DCPP pro vstupní graf a vrátí nalezený tah.

Metoda run nalezne ve vstupním grafu nevyvážené vrcholy, které rozdělí podle toho jestli patří do množiny D^+ nebo D^-. Poté nalezne nejkratší cesty z vrcholů v D^- do vrcholů v D^+. Následně se matice vzdáleností rozšíří tak aby každý vrchol v byl v matici zastoupen právě $|\delta(v)|$krát. Po nalezení optimálního párování se do grafu přidají hrany podél nejkratších cest mezi vrcholy párování. Tím se stanou všechny vrcholy vyvážené a zbývá nalézt eulerovský tah. Po nalezení takového tahu ho metoda vrátí jako výsledek. Pro realizaci jednotlivých kroků se použijí třídy popsané výše.

Nalezení vrcholů patřících do množin D^+ a D^- zabere lineárně dlouho vzhledem k počtu vrcholů vstupního grafu. Jak již bylo popsáno výše počet vrcholů v bipartitním grafu, ve kterém se hledá minimální párování, má $\mathcal{O}(|E|)$ vrcholů vzhledem k vstupnímu grafu. Cest k přidání je tedy $\mathcal{O}(|E|)$ a mohou mít maximálně $|V|$ hran, tudíž přidání hran do grafu trvá $\mathcal{O}(|E||V|)$. Celková časová složitost metody run se mění v závislosti na použitých algoritmech hledání nejkratších cest, minimálního párování a eulerovského tahu.

4.5 Paralelní návrh

Součástí zadání této práce je paralelizace implementovaných algoritmů pomocí OpenMP.

4.5.1 Třída Dijkstra

Třída Dijkstra hledá nejkratší cesty mezi dvojicemi vrcholů v grafu. To provádí opakovaným voláním hledání nejkratších cest z jednotlivých vrcholů. Tyto volání jsou plně nezávislé na sobě a právě proto se jedná o vhodné místo pro použití direktivy `#pragma omp parallel for`. Protože jednotlivá volání se mohou vykonávat různě dlouho je tato direktiva doplněna o argument `schedule(dynamic)`. Tato paralelizace není efektivní když počet vláken je větší než počet vrcholů, ze kterých se nejkratší cesty. V tomto případě však bude hledání nejkratších cest poměrně rychlé.

Paralelizace při inicializaci a vytváření výstupní struktury nemá vliv na celkový běh programu, protože se v těchto fázích setrvává minimální dobu vzhledem k samotnému hledání nejkratších cest.

4.5.2 Třída FloydWarshall

Třída FloydWarshall používá Floydův–Warshallův algoritmus k nalezení nejkratších cest mezi všemi dvojicemi vrcholů grafu. Algoritmus v $k$té iteraci počítá nejkratší cesty s použitím nejvýše $k$tého vrcholu jako vnitřního vrcholu cesty. Při tom využívá nejkratší cesty vypočtené v $(k-1)$ním kroku. Z toho důvodu nelze použít paralelizaci ve vnějším cyklu.

Avšak jednotlivé iterace vnitřního cyklu jsou již nezávislé a tudíž lze použít direktivu překladu `#pragma omp parallel for`. Na rozdíl od Dijkstrova algoritmu lze zde předpokládat, že jednotlivé iterace trvají stejně dlouho a není potřeba využít dynamické přiřazování vláken.

4.5.3 Podtřídy MatchingAlgorithm

Podtřídy MatchingAlgorithm (krom NaiveMatching) využívají k implementaci knihovní funkce. Samotný Edmondsův algoritmus je velmi sekvenční. Postupně zlepšuje získané párování na základě předchozího nalezeného párování. Paralelizace implementací by vyžadovala udělat velmi pečlivé změny ve zdrojovém kódu, který má několik set řádků a paralelizace by nemusela být dostatečná. Jak je později vidět na obrázku 5.9, hledání minimálního perfektního párování v grafu trvá poměrně malou část celkové doby běhu. Z těchto důvodů se tato práce nezabývá paralelizací podtříd hledající minimální perfektní párování.

4.5.4 Třída HungarianMethod

Třída HungarianMethod implementuje Munkresovu variantu Maďarské metody. Implementace obsahuje mnoho vnořených cyklů a některé z nich se dají dobře paralelizovat. Paralelizovat je možné i hledání prvotního přiřazení, ale tato část se ve výsledků téměř vůbec neprojeví. Zásadní paralelizace jsou uvnitř hlavního cyklu v místě, kde se přepočítávají minima v matici.

Z důvodu paralelizace musely být provedeny určité změny v kódu. Šablona `std::vector` obsahuje specializaci pro primitivní datový typ `bool`. Díky této specializaci je efektivně využito místo v paměti, ale při paralelizaci nastávají problémy. Při zápisu hodnot na různé pozice blízko sebe vzniká hazard, race condition, protože se může zapisovat na stejný bajt v paměti. Kvůli tomuto hazardu algoritmus nemusí fungovat správně. Jednoduchá změna z `bool` na celočíselný typ `int` vyřeší takovýto problém.

4.5.5 Třída MinCostFlowMatching

Třída MinCostFlowMatching využívá algoritmus na hledání maximálního toku při minimalizaci ceny k nalezení minimálního perfektního párování. Nejvíce času se stráví v hlavním cyklu, kde se postupně hledá zlepšující cesta, která by vylepšila aktuální párování. Kroky smyčky tedy nemohou být spouštěny nezávisle. Uvnitř smyčky je hledá nejkratší cesta, přepočítávají se váhy hran a vylepšuje se párování podél cesty. Paralelizaci lze provést u přepočítávání vah hran pomocí direktivy `#pragma omp parallel for`. Takto se bude upravovat váha hran nezávisle na sobě, což částečně urychlí běh. Další možností paralelizace je využití paralelního algoritmu na hledání nejkratších cest, ale implementace takového algoritmu nelze dosáhnout pouhým přidáním direktiv překladu do stávajícího řešení.

4.5.6 Ostatní

Paralelizace by se dala využít i u dalších kroků řešení například hledání vrcholů lichého stupně / nevyvážených vrcholů, doplnění hran do grafu nebo hledání eulerovského tahu. Tyto oblasti zabírají velmi malou část celkové doby běhu a paralelizace by neměla velký vliv na celkový čas. Z tohoto důvodu se tato práce nezabývá paralelizací těchto částí.

4.6 Použití programu

Kompilací zdrojového kódu vznikne program na řešení obou variant Problému čínského listonoše. Pro úspěšnou kompilaci je třeba mít nainstalovány knihovny Boost a LEMON. Implementace Blossom V je již zahrnuta ve zdrojovém kódu. Kompilaci je možné provést pomocí utility *make*, ručně nebo vytvořit projekt ve Visual Studiu a provést kompilaci tam (vhodné pro Windows).

Program očekává, že dostane konfiguraci, která určí jaká varianta CPP se bude řešit, jaké algoritmy se použijí a jak se získá vstupní graf. Tato konfigurace může být předána jako soubor nebo pomocí argumentů příkazové řádky.

Pokud je konfigurace předána jako soubor, tento soubor musí obsahovat minimálně 4 řádky. První řádek určuje typ úlohy. Pro řešení neorientované varianty je třeba zvolit `Undirected`, pro orientovanou variantu je třeba zvolit `Directed`. Druhý řádek obsahuje název zvolené třídy na hledání nejkratších cest v grafu – `Dijkstra` / `FloydWarshall`. Třetí řádek určuje algoritmus hledání párování. Možné vstupy jsou: `Naive`, `Boost`, `Lemon`, `Blossom5`, `NaiveBiparite`, `Hungarian` a `MinCostFlow`. Čtvrtý řádek určuje jak se získá vstupní graf. Text `File <cesta k souboru>` zvolí načtení grafu ze souboru. Další možnost je `GeneratorNumOfEdges <n> <m>`, která bere počet vrcholů a počet hran jako argumenty. Obdoba je použití `GeneratorDensity`, které bere počet vrcholů a desetinné číslo udávající hustotu požadovaného grafu.

Podrobný popis práce s programem je v souboru `readme.txt` v příloze u zdrojového kódu.

Kapitola 5
Testování

Tato kapitola se věnuje testování implementací algoritmů, jejich porovnání a testování celého řešení. Implementace se testují odděleně na náhodných datech. Výběr algoritmů k použití ve výsledném celkovém testování se provádí na základě výsledků měření oddělených testů.

Pro kompilaci programu při testování byl použit překladač `g++` verze 7.5.0. Kompilace byla prováděna následujícími přepínači: `-Wall -pedantic -std=c++14 -O3 -fopenmp`.

Pro generování vstupních grafu byla použita třída GraphGenerator, která umí vytvářet náhodné orientované a neorientované grafy podle předaných parametrů (počet vrcholů a počet hran / hustota). Pro testování algoritmů na hledání párování v obecných grafech i těch bipartitních byly vytvářeny matice o daných rozměrech s náhodnými hodnotami z intervalu $\langle 1, 1\,000\,000 \rangle$. Hodnoty byly generovány pomocí `std::random_device` a bylo použito rovnoměrné rozdělení.

5.1 Testy správnosti

Testy správnosti probíhaly následovně. Nejprve byly testovány jednotlivé implementace algoritmů na několika instancích problémů, které byly připraveny ručně. Takto se testovaly implementace již při vývoji a díky těmto testům se odhalilo nejvíce chyb v kódu.

Po dokončení implementace se testovaly algoritmy mezi sebou po dvojicích. Byla vytvořena náhodná instance problému (graf nebo matice), která poté byla řešena oběma implementacemi. Takovéto testování se provádělo na řádově stovkách instancí problémů s různými parametry. Vstupní grafy se generovaly s různými počty vrcholů a různými hustotami. Matice se lišily nejen v hodnotách, ale i v rozměrech. Tyto testy neodhalily žádné chyby v implementaci algoritmů, ale v rozhraní, které třídy obsluhovalo.

Třídy UndirectedPipeline a DirectedPipeline byly testovány nejprve ručně na jednoduchých grafech, ve kterých nebylo těžké nalézt optimální řešení ručně. Tyto testy odhalily chyby v propojení využitých algoritmů a aritmetické chyby při hledání nevyvážených vrcholů.

Poslední se prováděly testy správnosti celého řešení DCPP a UCPP. K ověření správnosti výsledků byla použita knihovna JGraphT, která je napsána v Javě. Nejprve byla použita třída GraphGenerator, která vygenerovala náhodné grafy a uložila je do souborů. Tyto grafy se následně staly vstupem k porovnání. K porovnání výsledků řešení orientované varianty bylo také použito řešení z bakalářské práce Matěje Razáka [44]. Veškeré výsledky vyšly stejně, čímž byla ověřena správnost řešení.

parametr	možné hodnoty
n	500 1000 1500 2000 2500 3000
d	0,1 0,25 0,5 0,75 0,9

Tabulka 5.1 Parametry pro generování náhodných grafů

5.2 Měření času

Měření času běhu jednotlivých algoritmů se provedlo pomocí třídy ScopedTimer. Tato třída měří čas pomocí funkcí ze standardní knihovny C++. Třída je implementována tak, aby měření bylo co nejpřesnější a vedlejší operace, jako například zápis do souboru, nebyly zahrnuty v měření. Dále aby celkový dopad na dobu běhu byl co nejmenší, měří se pouze ucelené úseky kódu, které nejsou prováděny opakovaně v cyklu.

Výsledky měření se zapisují do souboru ve formátu JSON. Soubor může být dále zpracován k analýze běhu programu. Při programování této práce byl využit nástroj k vizualizaci takovýchto souborů, který je součástí webových prohlížečů na bázi softwaru Chromium. Nástroj lze nalézt například v prohlížeči Chrome pod url `chrome://tracing`.

Každé měření času, pro každou volbu parametrů vstupu, bylo provedeno vždy 10krát. Jako výsledný čas běhu se vzal průměr z naměřených časů. Tímto opakováním se zamezí zkreslení jednorázovými výchylkami časů, které mohou v průběhu měření vzniknout.

5.3 Třída Dijkstra

Ve třídě Dijkstra se počítají nejkratší cesty v grafu. Nejkratší cesty se nehledají mezi všemi dvojicemi vrcholů, ale pouze z vrcholů lichého stupně v případě UCPP a z vrcholů patřící do množiny D^- v případě DCPP. V obou případech může být takových vrcholů lineárně mnoho vzhledem k celkovému počtu vrcholů vstupního grafu. Floydův–Warshallův algoritmus musí vždy počítat nejkratší cesty mezi všemi vrcholy grafu i když to není nutné pro další postup řešení CPP. Aby se třídy lépe porovnávaly, tak se při testování počítaly cesty mezi všemi vrcholy grafu.

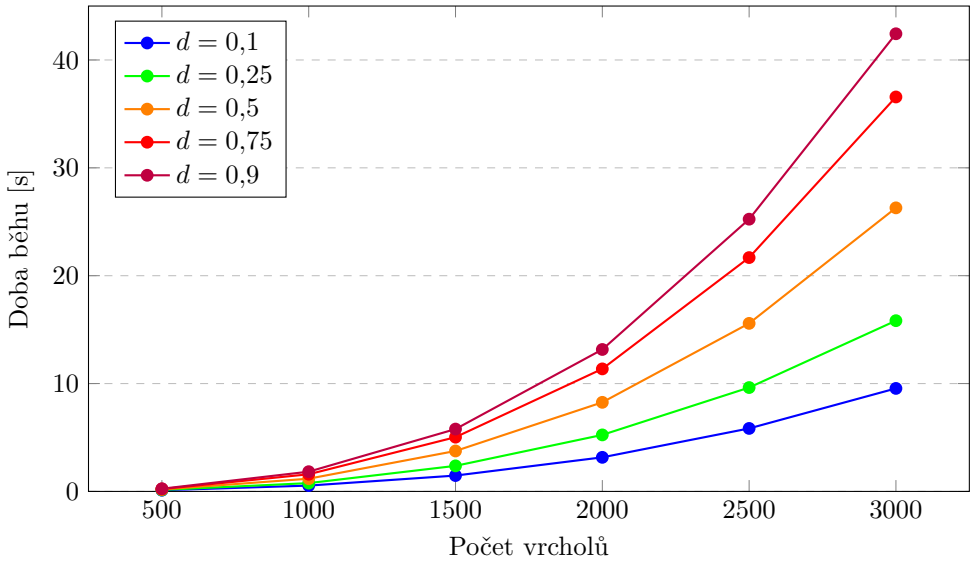
Testování probíhalo na náhodně generovaných grafech. Měření se provádělo na orientovaných a neorientovaných grafech nezávisle. Grafy byly generovány podle tabulky 5.1, kde n značí počet vrcholů a d požadovanou hustotu grafu.

Měří se běh metody getDistanceMatrix, která počítá nejkratší cesty mezi zadanými vrcholy. Měření probíhalo 10krát pro každou kombinaci parametrů n a d.

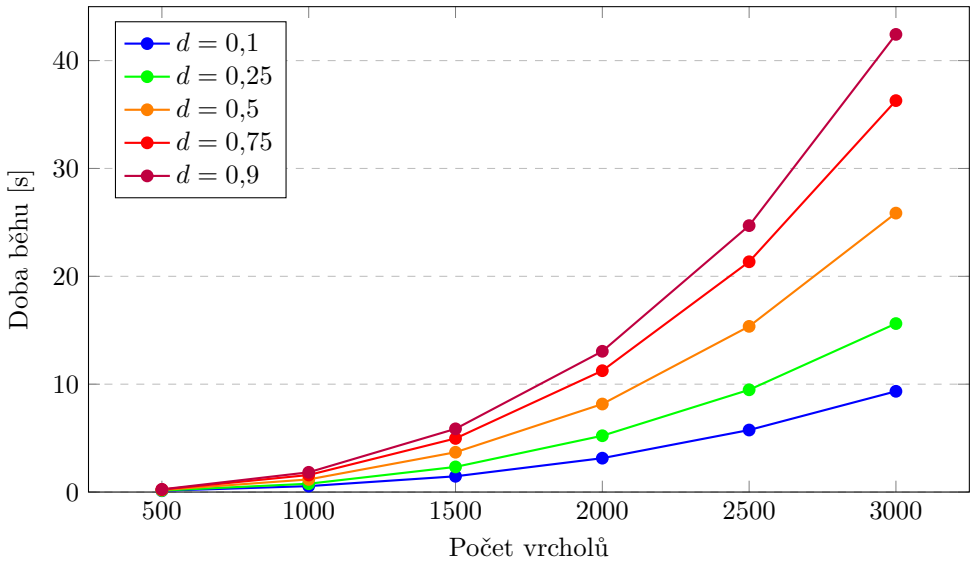
Na obrázku 5.1 je vidět graf závislosti doby běhu Dijkstrova algoritmu na počtu vrcholů v neorientovaném grafu. Podle asymptotické složitosti algoritmu je doba běhu závislá nejen na počtu vrcholů, ale také na počtu hran. Tato skutečnost je na obrázku zřejmá. Čím vyšší hustotu graf má tím déle trvá v něm nalézt nejkratší cesty.

Na obrázku 5.2 je vidět stejné měření avšak v tomto případě byl vstupem orientovaný graf. I přestože orientované grafy mohou mít dvojnásobně hran při stejném počtu vrcholů jak neorientované grafy, výsledné časy se mezi těmito variantami liší v řádu jednotek procent.

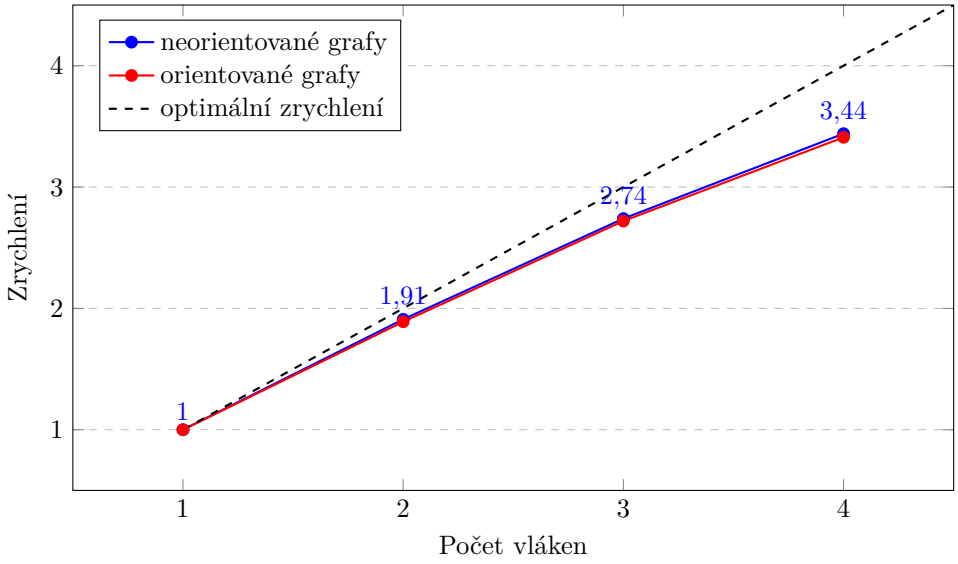
Testy potvrdily zrychlení běhu algoritmu při použití více vláken k výpočtu. Na obrázku 5.3 je vyobrazeno paralelní zrychlení na náhodných grafech, které mají 2000 vrcholů a jejich hustota je 0,5. Zrychlení vyšlo téměř stejně pro orientovanou i neorientovanou variantu. Při využití čtyř vláken došlo k 3,44násobnému zrychlení oproti sekvenčnímu běhu. Takovéto zrychlení se dá považovat za úspěšné. Z důvodu možné vyšší režie při použití více vláken se postupně koeficient zrychlení vzdaluje od optimální hodnoty.



Obrázek 5.1 Doba běhu Dijkstrova algoritmu na neorientovaných grafech



Obrázek 5.2 Doba běhu Dijkstrova algoritmu na orientovaných grafech



Obrázek 5.3 Paralelní zrychlení Dijkstrova algoritmu ($n = 2000, d = 0{,}5$)

5.4 Třída FloydWarshall

Ve třídě FloydWarshall se počítají nejkratší cesty v grafu. Nejkratší cesty se hledají mezi všemi vrcholy grafu nezávisle na variantě CPP a specifikaci vrcholů v argumentu metody getDistanceMatrix.

Stejně jako u testování třídy Dijkstra probíhalo testování na náhodně generovaných orientovaných a neorientovaných grafech nezávisle. Grafy byly generovány podle stejné tabulky parametrů 5.1, kde n značí počet vrcholů a d požadovanou hustotu grafu.

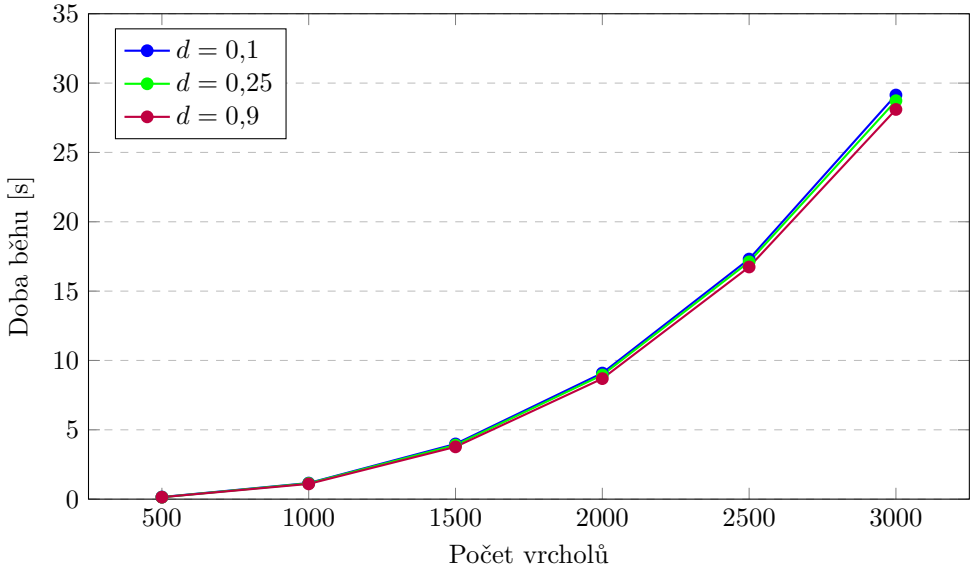
Měří se doba běhu metody getDistanceMatrix, která počítá nejkratší cesty mezi všemi vrcholy. Měření probíhalo 10krát pro každou kombinaci parametrů n a d.

Na obrázku 5.4 je zobrazena doba běhu Floydova–Warshallova algoritmu v závislosti na počtu vrcholů a hustotě grafu. Na rozdíl od Dijkstrova algoritmu je Floydův–Warshallův závislý pouze na počtu vrcholů vstupního grafu. Podle asymptotické složitosti doba běhu není závislá na počtu hran, avšak na obrázku je vidět mírné zrychlení s rostoucí hustotou. To může být způsobeno tím, že v řídkém grafu se častěji objeví nová cesta, která se uloží jako aktuálně nejkratší i když bude později znovu přepsána. Pro lepší přehlednost byly vykresleny pouze 3 různé hustoty.

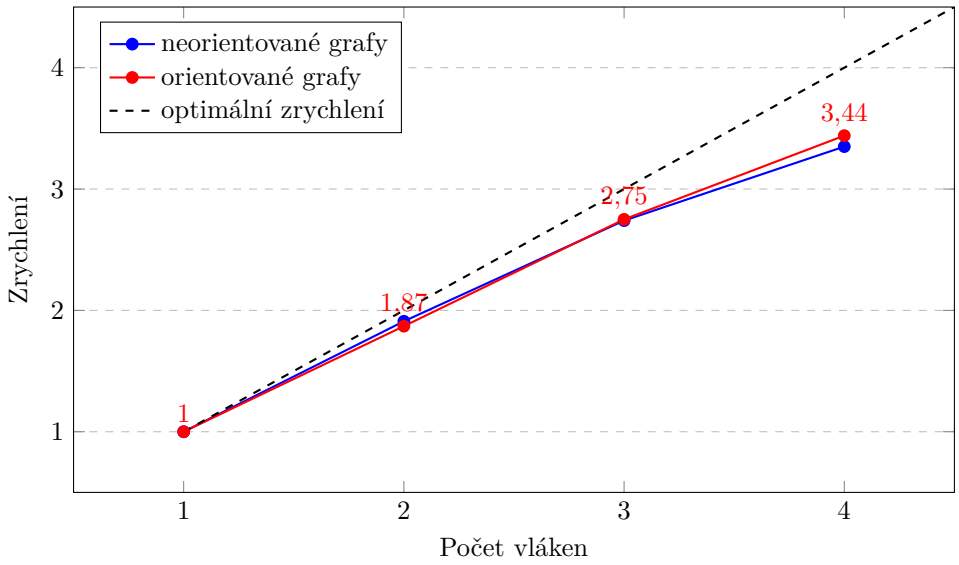
Pokud se algoritmus spouští na orientovaných grafech, celkové časy běhů jsou téměř totožné. Z pseudokódu algoritmu je zřejmé, že mezi rychlostí výpočtu nejkratších cest v orientovaném a neorientovaném grafu nebude téměř žádný rozdíl, protože vstupní matice mám v obou případech stejné rozměry. V orientované není vstupní matice symetrická. Tento minimální rozdíl potvrzují data z měření, kdy se rozdíly pohybovaly v rámci 1 %.

Paralelní zrychlení Floydova–Warshallova algoritmu, které je zobrazeno na obrázku 5.5, je velmi podobné zrychlení Dijkstrova algoritmu. Z této podobnosti lze usuzovat, že paralelizace obou algoritmů byla úspěšná a pravděpodobně naráží na určitý limit jakého koeficientu zrychlení lze dosáhnout při daném počtu vláken.

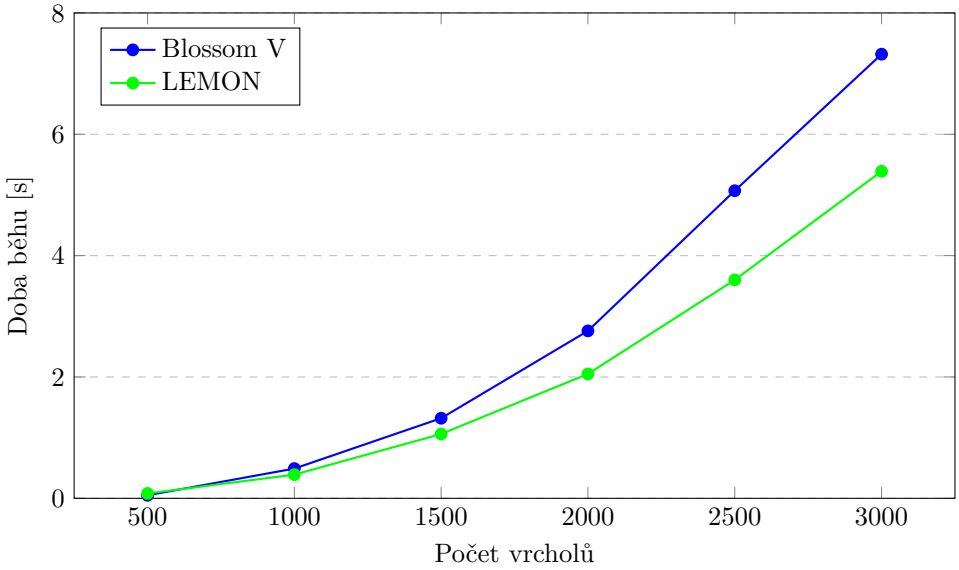
Z tříd řešících hledání nejkratších cest v grafu vychází pro použití při řešení CPP lépe třída Dijkstra. Sice pro grafy s hustotou vyšší než 0,5 vyšla tato třída jako pomalejší, ale je nutné si uvědomit, že se jednalo o hledání nejkratších cest mezi všemi vrcholy. V řešení UCPP a DCPP se hledají cesty pouze z podmnožiny všech vrcholů. U neorientované varianty jsou to vrcholy lichého stupně a u orientované vrcholy množiny D^-. Tato podmnožina vrcholů je jen velmi zřídka stejně velká jako množina vrcholů.



Obrázek 5.4 Doba běhu Floydova–Warshallova algoritmu na neorientovaných grafech



Obrázek 5.5 Paralelní zrychlení Floydova–Warshallova algoritmu ($n = 2000, d = 0{,}5$)



Obrázek 5.6 Doba běhu algoritmů na hledání minimálního perfektního párování

5.5 Porovnání BlossomVMatching a LemonMatching

Třídy BlossomVMatching a LemonMatching implementují algoritmus na hledání minimálního perfektního párování. Algoritmy očekávají na vstupu úplný neorientovaný graf s $2n$ vrcholy. Tento graf je reprezentovaný maticí $2n \times 2n$. Pro testování byly generovány náhodné matice s počtem řádků od 500 od 3000 po 500. Na těchto maticích byly spouštěny oba algoritmy a jejich časy zaznamenány.

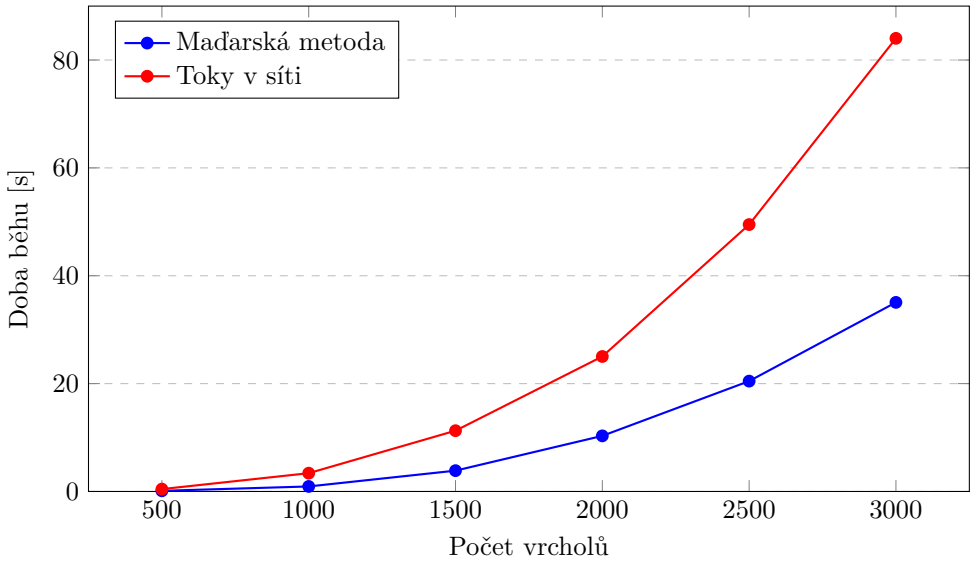
Na obrázku 5.6 je vidět porovnání obou algoritmů. Z grafu je zjevné, že algoritmus knihovny LEMON řeší problém rychleji než implementace Blossom V. O Blossom V se mluví jako o nejefektivnější implementaci Edmondsova algoritmu. V porovnání, kdy vstupem jsou obecné grafy, je Blossom V rychlejší než implementace knihovny LEMON [41]. Avšak v tomto případě se jedná vždy o úplné grafy. Možným důvodem, že knihovna LEMON vyšla v tomto testu lépe, je fakt, že třída `lemon::FullGraph` efektivně reprezentuje úplný graf. Tato třída obsahuje pouze jedno číslo, počet vrcholů, které definuje celý stav objektu. Blossom V neobsahuje žádnou takovou specializaci.

5.6 Porovnání HungarianMethod a MinCostMatching

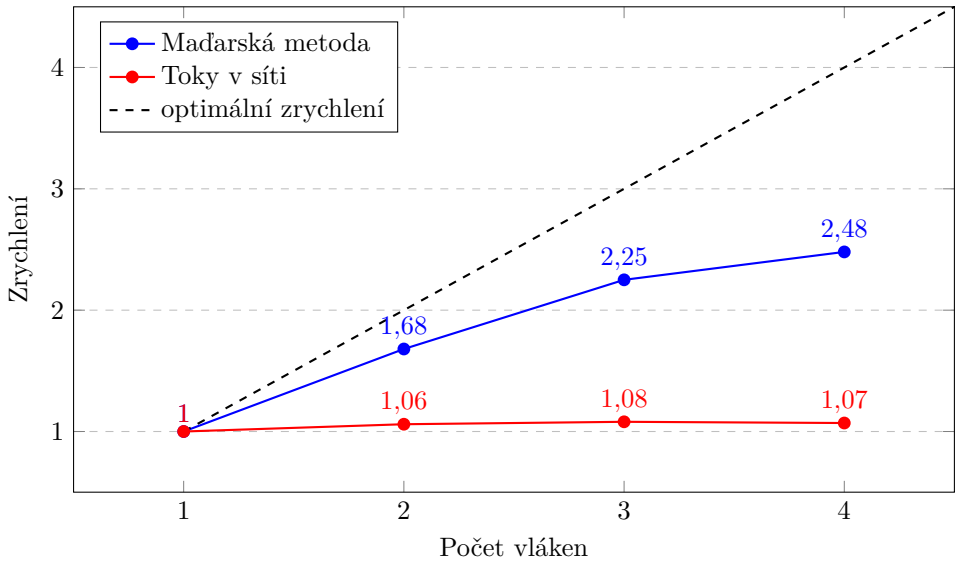
Třída HungarianMethod implementuje Munkresovu variantu Maďarské metody. Třída MinCostMatching řeší Přiřazovací problém pomocí toků v síti. Testování spočívalo ve vygenerování matice reprezentující úplný bipartitní graf, která se stala vstupem metody getMatching.

Na obrázku 5.7 je srovnání doby běhu obou tříd. Munkresova Maďarská metoda je při spuštění na stejných datech výrazně rychlejší. Jeden z důvodů může být to, že Maďarská metoda začíná nalezením nějakého (neprázdného) minimálního párování, které pak zvětšuje. Nalezením výchozího párování získá náskok oproti metodě využití toků, které začíná s prázdným párováním.

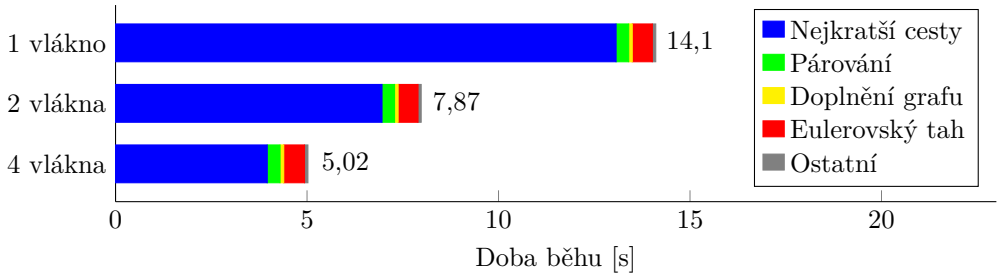
Dále se Munkresova varianta Maďarské metody ukázala jako vcelku dobře paralelizovatelná. Jak je na grafu z obrázku 5.8, při spuštění na 4 vláknech se dosáhlo téměř 2,5násobného zrychlení. Naproti tomu se paralelizace u třídy MinCostFlow ukázala jako velmi neefektivní / s velmi malým zrychlením.



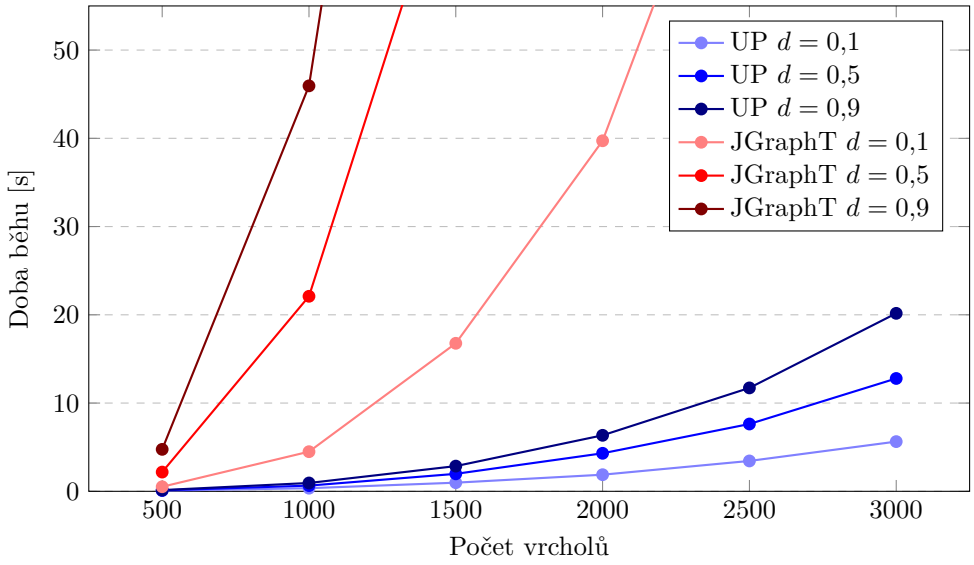
Obrázek 5.7 Doba běhu algoritmů řešící Přiřazovací problém



Obrázek 5.8 Paralelní zrychlení algoritmů řešící Přiřazovací problém ($n = 3000$)



Obrázek 5.9 Vliv paralelizace na celkovou dobu běhu UndirectedPipeline ($n = 3000, d = 0{,}5$)



Obrázek 5.10 Srovnání času běhu řešení neorientovaného CPP (UP – UndirectedPipeline)

5.7 Testování UndirectedPipeline

Třída UndirectedPipeline je zodpovědná za celý běh řešení UCPP. Řeší jednotlivé kroky a výstupy algoritmů zpracovává tak, aby se mohly stát vstupy v dalších krocích. Protože UndirectedPipeline využívá jiných tříd k řešení, je třeba si zvolit třídy, které budou použity při testování. Pro hledání nejkratších cest byl zvolen Dijkstrův algoritmus, který v testování vyšel jako rychlejší pro řídké grafy a běží rychleji pokud počet vrcholů lichého stupně je menší než celkový počet vrcholů. Perfektní minimální párování se najde pomocí třídy LemonMatching, která vyšla v testování jako rychlejší varianta. Pro nalezení eulerovského tahu se využije třída Hierholzer.

Na obrázku 5.9 je zobrazené rozdělení času podle jednotlivých kroků při využití různých počtů vláken. Měření bylo provedeno na náhodných grafech s 3000 vrcholy a hustotou 0,5. Při sekvenčním provedení se 92 % času stráví v hledání nejkratších cest v grafu. Díky paralelizaci této části celková doba běhu při použití 4 vláken 2,8násobně snížila.

Pro porovnání s existující implementací byla vybrána implementace knihovny JGraphT, která je napsána v jazyce Java. Nalezené implementace řešení CPP v jazyce C++ byly nevyhovující pro porovnání. Některé vracely nesprávné výsledky, některé byly příliš pomalé a některé nefungovaly vůbec. Řešení v knihovně JGraphT využívá Dijkstrův algoritmus pro hledání nejkratších cest. Pro hledání minimálního perfektního párování využívá implementaci, která je založena na článku o implementaci Blossom V. K nalezení eulerovského tahu slouží Hierholzerův algoritmus.

Na obrázku 5.10 je graf, který porovnává dobu běhu UndirectedPipeline (UP) a implementace v knihovně JGraphT. Měřený čas zahrnuje pouze řešení UCPP. Načítání grafu a vytváření instancí objektů není započítáno do celkového času. Vstupní grafy byly totožné pro obě měření. Jak je vidět z grafu, třída UndirectedPipeline řeší UCPP několikanásobně rychleji. Při vstupu o 3000 vrcholech a hustotě 0,9 trvalo implementaci knihovny JGraphT přes 1000 vteřin než se dobrala k výsledku, zatím co třída UndirectedPipeline to zvládla za přibližně 20 vteřin. Knihovna JGraphT nenabízí paralelizaci řešení.

5.8 Testování DirectedPipeline

Třída DirectedPipeline zabaluje celé řešení orientované varianty CPP. Řeší jednotlivé kroky a výstupy jednotlivých algoritmů zpracovává tak, aby se mohly stát vstupy v dalších krocích.

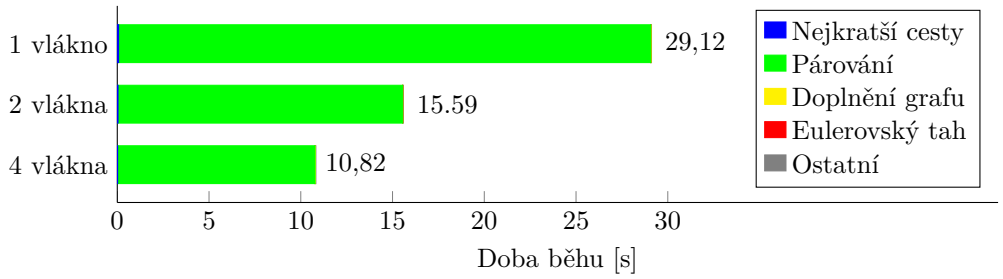
DirectedPipeline využívá rozhraní umožňující výměnu některých algoritmů. Pro testování byl použit Dijkstrův algoritmus pro nalezení nejkratších cest v grafu z vrcholů z D^-. Pro řešení Přiřazovacího problému byla zvolena třída HungarianMethod, protože vyšla jako rychlejší volba a více se projevila paralelizace. Nakonec, eulerovských tah se nalezne pomocí třídy Hierholzer.

Obrázek 5.11 obsahuje rozdělení času jednotlivých kroků řešení DCPP pro vstup obsahující 600 vrcholů a hustotu 0,1. Obrázek je téměř jednobarevný, protože hledání optimálního přiřazení má kubickou složitost a velikost vstupu do Maďarské metody je často o mnoho větší než počet vrcholů vstupního grafu. Obrázek 5.12 obsahuje přehled velikostí vstupů v závislosti na volbě parametrů vstupního grafu. Čas strávený při řešení Přiřazovacího problému zabírá přes 99 % celkového času. Při běhu na 4 vláknech se celková doba zkrátila téměř na polovinu.

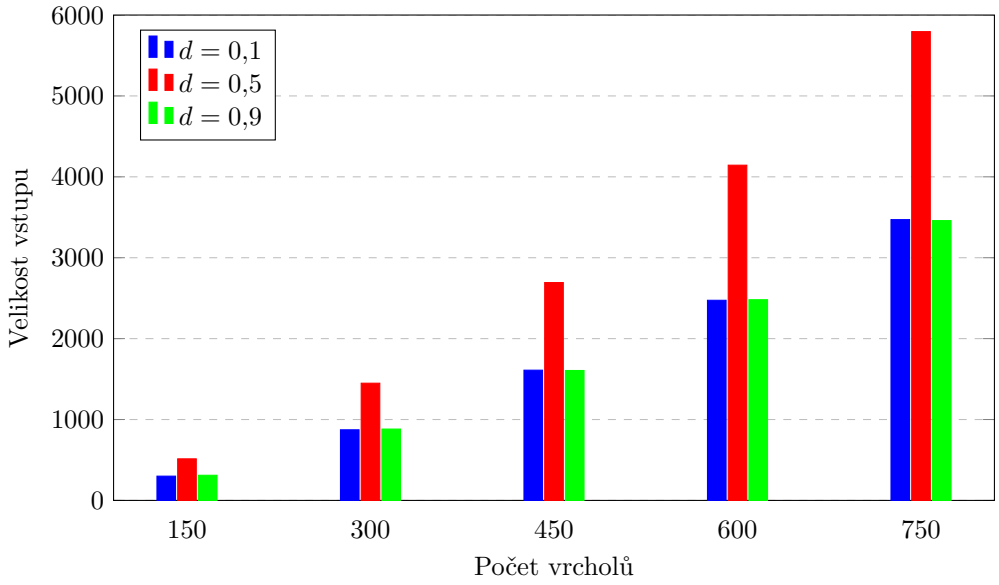
Pro porovnání DirectedPipeline s existujícími řešeními byla opět zvolena knihovna JGraphT a řešení Matěje Razáka, který se orientovanou variantou CPP zabýval ve své bakalářské práci [44]. Implementace Matěje Razáka je v jazyce C++ a autor ve své práci také vyhodnotil použití Dijkstrova algoritmu a Maďarské metody jako nejrychlejší. Měření času běhů jednotlivých řešení bylo měřeno na náhodně generovaných grafech, které byly pro všechny řešení stejné. Jak je možné vidět na obrázku 5.12, pro grafy o hustotě 0,1 a 0,9 je velikost vstupu do Maďarské metody velmi podobná. Z tohoto důvodu je v grafu 5.13 jsou vykresleny časy pouze pro hustoty 0,1 a 0,5.

Na grafu v obrázku 5.13 jsou vykresleny doby běhů jednotlivých řešení DCPP. Řešení z této práce, třída DirectedPipeline, je označeno jako DP, řešení Matěje Razáka je označené jako CD-CPP, což je autorovo pojmenování řešení. Výsledky knihovny JGraphT jsou označeny jako JGraphT. Řešení z této práce pro grafy o hustotě 0,1 je téměř 4krát rychlejší než řešení JGraphT. Vstup o 750 vrcholech a hustotě 0,5 trval knihovně JGraphT zpracovat více než 1000 vteřin zatím co ostatním implementacím to trvalo okolo 400 vteřin.

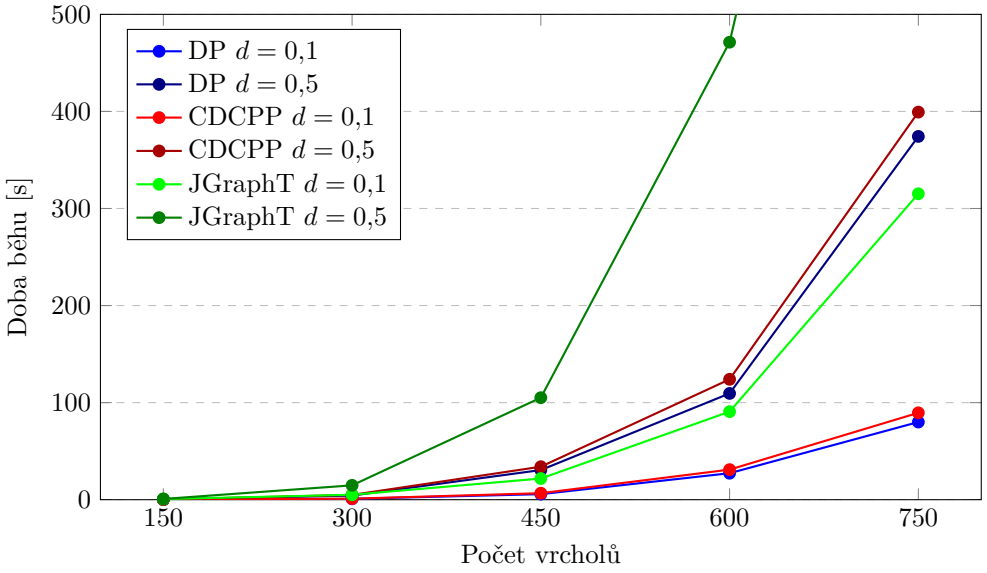
Implementace řešení Matěje Razáka CDCPP nabízí využití paralelizace. Při sekvenčním řešení je DirectedPipelinev v průměru o 5 % rychlejší než CDCPP. Avšak při využití 4 vláken se implementace CDCPP stala stejně rychlou a rozdíl v časech běhu byl menší než 1 %.



Obrázek 5.11 Vliv paralelizace na celkovou dobu běhu DirectedPipeline ($n = 600, d = 0.1$)



Obrázek 5.12 Porovnání velikostí vstupů do Maďarské metody v závislosti na parametrech vstupního grafu



Obrázek 5.13 Srovnání času běhu řešení orientovaného CPP, řešení této práce – DP, řešení Matěje Razáka – CDCPP, řešení knihovny JGraphT – JGraphT

Závěr

V této práci byl představen Problém čínského listonoše pro orientované a neorientované grafy. Cílem práce bylo nastudování, popis a implementace algoritmů pro řešení obou těchto variant. Posledním dílčím cílem bylo využití OpenMP pro paralelizaci řešení. Všechny cíle byly naplněny.

V prvních třech kapitolách byl rozebrán Problém čínského listonoše, definovány základní pojmy a popsány jednotlivé kroky řešení obou variant problému. Pro hledání nejkratších cest v grafu byl představen Dijkstrův a Floydův–Warshallův algoritmus. Dále byl popsán Edmondsův květinový algoritmus, který byl následně použit k hledání minimálního perfektního párování v úplném grafu. Pro hledání optimálního párování v bipartitním grafu byla představena Maďarská metoda a algoritmus na základě toků v síti. Na hledání eulerovského tahu byl využit Hierholzerův algoritmus.

V implementační části byl vytvořen fungující program, který je několikanásobně rychlejší než řešení knihovny JGraphT. V porovnání s prací Matěje Razáka, který popsal pouze orientovanou variantu, vyšlo, že obě implementace jsou obdobně rychlé. Paralelizace programu se u některých tříd ukázala jako efektivní.

Možností rozšíření této práce je zaměření se na tzv. *Transporation problem*, který by netrpěl problémy, které se objevily při řešení Přiřazovacího problému. Vhodná úprava tříd UndirectedPipeline a DirectedPipeline by umožnila vybírat si algoritmus hledání nejkratších cest v grafu v závislosti na parametrech vstupu. Tento automatizovaný výběr by mohl snížit dobu běhu pro velmi husté grafy.

Dalšími možnými rozšířeními je použití algoritmů, které jsou od základu paralelní místo paralelizace sekvenčních algoritmů. Bylo by například možné využít blokový Floydův–Warshallův algoritmus a vektorizaci pomocí SIMD instrukcí nebo ho implementovat na GPU. Při slevení na podmínce nalezení optimálního řešení CPP by se na některé kroky dalo využít aproximačních algoritmů a heuristik, například hledání párování.

Bibliografie

1. BIGGS, N.; LLOYD, E. K.; WILSON, R. J. *Graph theory 1736 – 1936*. Clarendon Press, 1986.
2. EISELT, H. A.; GENDREAU, M.; LAPORTE, G. Arc Routing Problems, Part I: The Chinese Postman Problem. *Operations Research*. 1995, roč. 43, č. 2, s. 231–242. Dostupné také z: http://www.jstor.org/stable/171832.
3. MALEK, M.; MOURAD, A. N.; PANDYA, M. Topological testing. *Proceedings. 'Meeting the Tests of Time', International Test Conference*. 1989, s. 103–110. Dostupné také z: https://doi.ieeecomputersociety.org/10.1109/TEST.1989.82283.
4. BARAHONA, F. On some applications of the chinese postman problem. *Algorithms and Combinatorics*. 1989, roč. 9. Dostupné také z: http://hdl.handle.net/10068/226018.
5. BARAHONA, F.; MAYNARD, R.; RAMMAL, R.; UHRY, J. P. Morphology of ground states of two-dimensional frustration model. *Journal of Physics A: Mathematical and General*. 1982, roč. 15, č. 2, s. 673–699. Dostupné také z: https://doi.org/10.1088/0305-4470/15/2/033.
6. MILIOTIS, P.; LAPORTE, G.; NOBERT, Y. Computational comparison of two methods for finding the shortest complete cycle or circuit in a graph. *RAIRO - Operations Research - Recherche Opérationnelle*. 1981, roč. 15, č. 3, s. 233–239. Dostupné také z: http://www.numdam.org/item/RO_1981__15_3_233_0/.
7. SUCHÝ, O.; VALLA, T. *BI-AG2: Přednášky č. 1–12* [online]. 2019 [cit. 2021-04-06]. Dostupné z: https://courses.fit.cvut.cz/BI-AG2/lectures.html. [Soubor přístupný po přihlášení do sítě ČVUT].
8. LU, L. *Math 777 Graph Theory I Lecture Note 3: Eulerian circuits and directed graphs* [online]. 2005 [cit. 2021-04-06]. Dostupné z: http://people.math.sc.edu/lu/teaching/2005fall_776/lecture3.pdf.
9. MALÍK, J.; SUCHÝ, O.; TVRDÍK, P.; VALLA, T. *Algoritmy a grafy 1: Přednášky č. 1–12* [online]. 2020 [cit. 2021-04-06]. Dostupné z: https://courses.fit.cvut.cz/BI-AG1/lectures/index.html. [Soubor přístupný po přihlášení do sítě ČVUT].
10. GUAN, M. Graphic programming using odd and even points. *Chinese Mathematics*. 1962, roč. 1, s. 237–277. Dostupné také z: https://www.math.uni-bielefeld.de/documenta/vol-ismp/16_groetschel-martin-yuan-ya-xiang.pdf.
11. EDMONDS, J.; JOHNSON, E. L. Matching, Euler tours and the Chinese postman. *Mathematical Programming*. 1973, roč. 5, č. 1, s. 88–124. Dostupné také z: https://doi.org/10.1007/BF01580113.

12. MINIEKA, E. The Chinese Postman Problem for Mixed Networks. *Management Science*. 1979, roč. 25, č. 7, s. 643–648. ISSN 00251909, ISSN 15265501. Dostupné také z: http://www.jstor.org/stable/2630397.
13. PAPADIMITRIOU, C. H. On the Complexity of Edge Traversing. *J. ACM*. 1976, roč. 23, č. 3, s. 544–554. Dostupné také z: https://doi.org/10.1145/321958.321974.
14. GUAN, M. On the windy postman problem. *Discrete Applied Mathematics*. 1984, roč. 9, č. 1, s. 41–46. Dostupné také z: https://www.sciencedirect.com/science/article/pii/0166218X84900891.
15. EISELT, H. A.; GENDREAU, M.; LAPORTE, G. Arc Routing Problems, Part II: The Rural Postman Problem. *Operations Research*. 1995, roč. 43, č. 3, s. 399–414. Dostupné také z: http://www.jstor.org/stable/171865.
16. PEARN, W. L.; LI, M. L. Algorithms for the Windy Postman Problem. *Computers & Operations Research*. 1994, roč. 21, č. 6, s. 641–651. Dostupné také z: https://www.sciencedirect.com/science/article/pii/0305054894900795.
17. FREDERICKSON, G. N. Approximation Algorithms for Some Postman Problems. *J. ACM*. 1979, roč. 26, č. 3, s. 538–554. ISSN 0004-5411. Dostupné z DOI: 10.1145/322139.322150.
18. EDMONDS, J. Maximum matching and a polyhedron with 0,1-vertices. *Journal of Research of the National Bureau of Standards Section B Mathematics and Mathematical Physics*. 1965, s. 125. Dostupné také z: https://nvlpubs.nist.gov/nistpubs/jres/69B/jresv69Bn1-2p125_A1b.pdf.
19. DIJKSTRA, E. W. A note on two problems in connexion with graphs. *Numerische mathematik*. 1959, roč. 1, s. 269–271. Dostupné také z: https://doi.org/10.1007/BF01386390.
20. MAREŠ, M.; VALLA, T. *Průvodce labyrintem algoritmů*. Praha: CZ.NIC, z.s.p.o, 2017. ISBN 978-80-88168-19-5.
21. FLOYD, R. W. Algorithm 97: Shortest Path. *Commun. ACM*. 1962, roč. 5, č. 6, s. 345. ISSN 0001-0782. Dostupné z DOI: 10.1145/367766.368168.
22. WARSHALL, S. A Theorem on Boolean Matrices. *J. ACM*. 1962, roč. 9, č. 1, s. 11–12. ISSN 0004-5411. Dostupné z DOI: 10.1145/321105.321107.
23. WEISSTEIN, E. W. *"Floyd-Warshall Algorithm."From MathWorld – A Wolfram Web Resource.* [Online]. 2021 [cit. 2021-04-15]. Dostupné z: https://mathworld.wolfram.com/Floyd-WarshallAlgorithm.html.
24. BERGE, C. Two theorems in graph theory. *Proceedings of the National Academy of Sciences*. 1957, roč. 43, č. 9, s. 842–844. ISSN 0027-8424. Dostupné z DOI: 10.1073/pnas.43.9.842.
25. DUAN, R. *Maximum Weighted Matching (II)* [online]. 2012 [cit. 2021-04-19]. Dostupné z: https://resources.mpi-inf.mpg.de/departments/d1/teaching/ss12/AdvancedGraphAlgorithms/Slides07.pdf.
26. EGERVÁRY RESEARCH GROUP ON COMBINATORIAL OPTIMIZATION. *LEMON 1.3.1 Documentation* [online]. 2014 [cit. 2021-04-14]. Dostupné z: http://lemon.cs.elte.hu/pub/doc/1.3.1/index.html.
27. KUHN, H. W. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*. 1955, roč. 2, č. 1-2, s. 83–97. Dostupné z DOI: https://doi.org/10.1002/nav.3800020109.
28. BRILLIANT.ORG. *Hungarian Maximum Matching Algorithm* [online]. 2021 [cit. 2021-04-27]. Dostupné z: https://brilliant.org/wiki/hungarian-matching/.
29. MUNKRES, J. Algorithms for the Assignment and Transportation Problems. *Journal of the Society for Industrial and Applied Mathematics*. 1957, roč. 5, č. 1, s. 32–38. ISSN 03684245. Dostupné také z: http://www.jstor.org/stable/2098689.

30. WONG, J. K. A new implementation of an algorithm for the optimal assignment problem: An improved version of Munkres' algorithm. *BIT Numerical Mathematics*. 1979, roč. 19, č. 3, s. 418–424. ISSN 1572-9125. Dostupné z DOI: 10.1007/BF01930994.

31. AHUJA, R. K.; MAGNANTI, T. L.; ORLIN, J. B. *Network Flows: Theory, Algorithms, and Applications*. USA: Prentice-Hall, Inc., 1993. ISBN 013617549X.

32. ZWICK, U. *Analysis of Algorithms: Minimum cost flow and weighted bipartite matching* [online]. 2006 [cit. 2021-04-28]. Dostupné z: http://www.cs.tau.ac.il/~zwick/grad-algo-06/min-cost-flow.pdf.

33. FORD, L. R.; FULKERSON, D. R. Maximal Flow Through a Network. *Canadian Journal of Mathematics*. 1956, roč. 8, s. 399–404. Dostupné z DOI: 10.4153/CJM-1956-045-5.

34. EDMONDS, J.; KARP, R. M. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *J. ACM*. 1972, roč. 19, č. 2, s. 248–264. ISSN 0004-5411. Dostupné z DOI: 10.1145/321694.321699.

35. KLEINBERG, R. *Introduction to Algorithms* [online]. Cornell University, Ithaca, NY, 2010 [cit. 2021-04-28]. Dostupné z: https://www.cs.cornell.edu/courses/cs4820/2012sp/handouts/edmondskarp.pdf.

36. HOPCROFT, J.; KARP, R. An n5/2 Algorithm for Maximum Matchings in Bipartite Graphs. *SIAM J. Comput.* 1973, roč. 2, s. 225–231.

37. 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. Dostupné také z: https://csce.ucmss.com/cr/books/2017/LFS/CSREA2017/MSV3359.pdf.

38. DAWES, B.; ABRAHAMS, D.; AL., et. *Boost 1.75.0 Library Documentation* [online]. 2020 [cit. 2021-05-01]. Dostupné z: https://www.boost.org/.

39. DEZSŐ, B.; JÜTTNER, A.; KOVÁCS, P. LEMON – an Open Source C++ Graph Template Library. *Electronic Notes in Theoretical Computer Science*. 2011, roč. 264, č. 5, s. 23–45. Dostupné také z: https://www.sciencedirect.com/science/article/pii/S1571066111000740.

40. KOLMOGOROV, V. Blossom V: a new implementation of a minimum cost perfect matching algorithm. *Mathematical Programming Computation*. 2009, roč. 1, č. 1, s. 43–67. ISSN 1867-2957. Dostupné z DOI: 10.1007/s12532-009-0002-8.

41. MICHAIL, D.; KINABLE, J.; NAVEH, B.; SICHI, J. V. JGraphT—A Java Library for Graph Data Structures and Algorithms. *ACM Trans. Math. Softw.* 2020, roč. 46, č. 2. Dostupné také z: https://dl.acm.org/doi/10.1145/3381449.

42. OPENMP. *OpenMP Specifications* [online]. 2020 [cit. 2021-05-01]. Dostupné z: https://www.openmp.org/specifications/.

43. CHERNIKOV, Y. *Basic instrumentation profiler* [online]. 2019 [cit. 2021-05-01]. Dostupné z: https://gist.github.com/TheCherno/31f135eea6ee729ab5f26a6908eb3a5e.

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

Obsah přiloženého média

- `readme.txt` ..... stručný popis obsahu média
- `exe` ..... adresář se spustitelnou formou implementace
- `src` ..... adresář se zdrojovými kódy
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF