PŘÍRODOVĚDECKÁ FAKULTA UNIVERZITY PALACKÉHO
KATEDRA INFORMATIKY

BAKALÁŘSKÁ PRÁCE

Algoritmy pro problém Euklidovského obchodního cestujícího

2014 Petr Maléř

Anotace

Práce se zabývá Euklidovským problémem obchodního cestujícího a algoritmy jej řešící. Je rozdělena na tři hlavní části - představení daného problému, popis daných algoritmů a jejich testování na daných datech pomocí programu, který vznikl společně s bakalářskou prací a ve kterém jsou implementovány představené algoritmy. Práce by měla vyzdvihnout přednosti daných algoritmů pomocí dosažených výsledků z jejich testování.

Na tomto místě bych chtěl především poděkovat vedoucímu práce Mgr. Petru Osičkovi, za poskytnuté informace, připomínky, doporučení a především za velmi přátelský přístup při tvorbě bakalářské práce. Poděkování patří i rodině a přátelům za poskytnutou podporu po celou dobu studia.

Obsah

Seznam obrázků

Seznam tabulek

Předtím, než se začneme věnovat Problému obchodního cestujícího (dále už jen TSP[1]), je vhodné uvést několik základních pojmů z teorie grafů, které jsou v tomto problému klíčové.

0.1. Graf

Graf je základní pojem a také základní objekt v teorii grafů[2]. Grafem lze znázornit množinu objektů a vztahy mezi nimi. Objektům se přiřadí vrcholy a mezi vrcholy se vytvoří hrany. Graf lze použít například jako zjednodušený model nějaké skutečné sítě (například dopravní), který zdůrazňuje topologické vlastnosti objektů (např. měst) a zanedbává geometrické vlastnosti (např. tvar silnic nebo přesnou polohu objektů).

U grafu je nutné uvažovat, zda-li se jedná o *neorientovaný graf* nebo *orientovaný graf*.

Definice 0..1. *Neorientovaný graf je dvojice $G = \langle V, E \rangle$, kde V je neprázdná množina vrcholů a E je množina dvouprvkových množin vrcholů tzv. neorientovaných hran $E \subseteq \{\{u, v\} \mid u, v \in V, u \neq v\}$.*

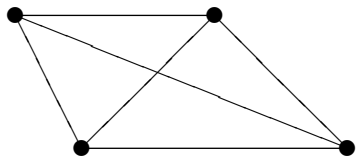
U neorientovaného grafu lze tedy říci, že pokud existuje hrana z vrcholu a do b, tak existuje hrana z vrcholu b do a. Na pořadí vrcholů tedy nezáleží.

Definice 0..2. *Orientovaný graf je dvojice $G = \langle V, E \rangle$, kde V je neprázdná množina vrcholů a E je množina uspořádaných dvojic vrcholů tzv. orientovaných hran.*

Jak již tedy z definice vyplývá, na rozdíl od neorientovaného grafu na pořadí hran záleží. Z existence hrany z a do b neplyne existence hrany b do a. Zpravidla v grafu orientované hrany značíme šipkami.

Jelikož instance Euklidovského TSP jsou pouze úplné grafy, uveďme si definici této vlastnosti.

Definice 0..3. *Graf $G = \langle V, E \rangle$ je úplný, pokud pro všechny vrcholy $u, v \in V, u \neq v$ existuje hrana $e \in E$, která u a v spojuje.*



Obrázek 1. Úplný graf

[1] Travelling salesman problem (zkratka z anglického názvu).

[2] Pojmem graf můžeme též rozumět i graf funkce – grafické znázornění funkce nebo také graf jako diagram. Tyto definice pro nás nejsou ale příliš důležité.

Pro úplný graf rovněž platí, že počet hran, které ho tvoří, je roven $\binom{n}{2}$, kde n je počet vrcholů v grafu.

V praktických aplikacích teorie grafů, jako je právě například TSP, slouží právě graf k popisu nějaké struktury. V Euklidovském TSP se jedná o mapu míst, které musí obchodní cestující navštívit. Jednotlivé prvky této struktury mohou mít přiřazeny nějaké důležité parametry. V případě TSP uvažujme graf, jehož vrcholy jsou města na mapě a hrany silnice spojující jednotlivá města. Při hledání nejkratší *Hamiltonovské kružnice* v grafu jsou klíčové ceny (vzdálenosti) mezi jednotlivými vrcholy, což vede k definici *ohodnocený graf*.

Definice 0..4. *Ohodnocený graf (G, w) je graf G spolu s funkcí* $w : E(G) \to \mathbb{Q}$. *Je-li e hrana grafu G, číslo w(e) se nazývá její ohodnocení nebo váha.*

Velice podobně lze samozřejmě definovat i ohodnocený neorientovaný graf. K neorientovanému případu lze velice snadno přejít přes symetrickou orientaci. Ohodnocení hran symetrické orientace *(G, w)* je přirozeně odvozeno z původního ohodnocení (obě protichůdné strany vzniklé z orientované hrany e dostanou ohodnocení *w(e)*).

0.1.1. Reprezentace grafu v PC

K tomu, aby se dalo v programu s grafy co nejsnáze manipulovat, je nutno zvolit správnou reprezentaci grafu. Graf lze v počítači reprezentovat mnoha způsoby, nejčastěji se však používají tyto dvě metody:

1. Matice sousednosti
2. Seznam sousedů

Matice sousednosti grafu s počtem vrcholů n je čtvercová matice o velikosti $n \times n$, ve které je znázorněno, mezi kterými dvěma vrcholy existuje hrana a mezi kterými ne. Mějme matici sousednosti A, která je definována jako $A = (a_{u,v})$, kde:

$$a_{u,v} = \begin{cases} 1 & \text{pokud } uv \in E, \\ 0 & \text{pokud } uv \notin E. \end{cases} \tag{1}$$

Pokud je tedy v matici sousednosti na souřadnicích (u, v) uložena hodnota 1, znamená to, že mezi vrcholy u a v existuje hrana. Pokud je zde uložena 0, hrana mezi vrcholy neexistuje.

V matici sousednosti lze v konstantním čase určit, zda mezi vrcholy u a v existuje hrana a hrany lze v konstantním čase přidávat i odebírat. Složitost úkonu projít všechny vrcholy sousedící s daným vrcholem je $O(n)$. Další informace o vrcholech (např stupeň vrcholu, pozice vrcholu apod.) musí být ale uchovány jinde než v matici sousednosti. Naproti tomu informace o hranách (např délka hrany,

barva hrany apod.) lze uložit přímo do matice sousednosti v podobě vhodně zvolené struktury. Matice sousednosti není vhodná pro reprezentaci řídkých grafů[3] z důvodu velké paměťové náročnosti. Velikost matice sousednosti je vždy $|V|^2$ [5].

Reprezentace pomocí *Seznamu sousedů* je reprezentace grafu založená na principu seznamu všech vrcholů V. Pro každý vrchol $v \in V$ je definován seznam S, ve kterém jsou uloženy všechny vrcholy z V, které sousední s v. Délka seznamu S, který je přiřazen vrcholu v je rovna stupni vrcholu v. V seznamu V lze uchovávat dodatečné informace o vrcholech a v seznamu S přiřazenému vrcholu v lze uchovávat informace o hranách.

Tato reprezentace se hodí zvlášť tehdy, pokud potřebujeme v algoritmech s grafy pracující, často procházet vrcholy, které s nějakým vrcholem sousedí a nepotřebujeme testovat, zda-li mezi dvěma vrcholy existuje hrana. Složitost takového testu je v této reprezentaci $O(n)$. Složitost přidání nebo odebrání hrany je lineární.

Nelze přesně určit, která z těchto dvou metod je efektivnější, záleží na úkonech, které s grafy chceme nejčastěji provádět. Lze však říci, že jednodušší na implementaci je reprezentace pomocí *Matice sousednosti*, která je používána častěji než *Seznam sousedů*.

Reprezentace grafu v přiloženém programu

Jelikož je součástí této práce také program pro experimentální porovnávání jednotlivých algoritmů pro řešení TSP, je vhodné poukázat, jaká byla zvolená reprezentace grafů. Jelikož se zde zabýváme Euklidovským TSP (viz později), grafy jsou vždy úplné, tj. mezi každou dvojicí vrcholů existuje hrana, byla by reprezentace pomocí matice sousednosti velmi neefektivní (tvořily by jí samé 1). Podobně neefektivní reprezentace grafu by byla i pomocí seznamů sousedů (každý soused má $n-1$ sousedů). Zde postačuje pouze obyčejné jednorozměrné pole *graphArray*, kde jsou uloženy samotné vrcholy a s nimi důležité informace, které daný vrchol charakterizují (např. souřadnice vrcholů). Tyto informace lze samozřejmě rozšiřovat v závislosti na potřebách jednotlivých algoritmů.

Není nutné, aby pole *graphArray* bylo dynamické, protože po načtení grafu do programu již není nutné žádné vrcholy do grafu přidávat nebo odebírat. Pokud je rozhodnuto algoritmy testovat na jiném grafu, je pouze nutné nový graf načíst z textového souboru (nebo náhodně vygenerovat pomocí vestavěné funkce) a posléze s ním pracovat. Pokud je potřeba používat v algoritmech dynamickou strukturu (např. seznam, zásobník apod.), je struktura vytvořena přímo pro daný algoritmus a vrcholy z *graphArray* jsou okopírovány. Lze vlastně i říci, že je vhodné, až nutné, zvolit *graphArray* jako strukturu, která je statická a to hlavně kvůli možnosti volat více algoritmů za sebou na stejný graf. Kdyby se pole *graphArray* při jednotlivém algoritmu nějak upravovalo (mazaly by se vr-

[3] Grafy s malým počtem hran

choly, přidávaly apod.), bylo by nutno při opětovném spuštění algoritmu nebo použití jiného algoritmu na stejný graf původní graf znovu načíst, což by bylo velmi neefektivní. Jako počet vrcholů v grafu, a tedy velikost pole *graphArray* existuje v programu důležitá konstanta *number*, kterou využívá prakticky každý algoritmus pro řešení TSP.

Jelikož v Euklidovském TSP pracujeme s úplnými grafy, je nutné znát vzdálenosti mezi jednotlivými dvojicemi vrcholů. V původní koncepci programu byla vytvořena matice o velikosti $number * number$, kde na příslušných pozicích $[i, j]$ byly uloženy vzdálenosti mezi vrcholy i a j. To se ale projevilo jako velmi neefektivní z důvodu velké paměťové náročnosti při načítání velkých grafů. Matice tedy byla zrušena a pokaždé, když nějaký algoritmus potřebuje zjistit vzdálenost mezi dvěma vrcholy i a j, použije se známý vzorec pro vypočítání vzdálenosti dvou vrcholů v dvojrozměrném prostoru:

$$\sqrt{(graphArray[i]_x - graphArray[j]_x)^2 + (graphArray[i]_y - graphArray[j]_y)^2}$$

Tato reprezentace grafu v Euklidovském prostoru je pro účel testování algoritmů řešící TSP zcela dostačující.

0.2. Cesta, kružnice

Pojmem *cesta* v teorii grafů rozumíme posloupnost vrcholů, pro které platí, že z každého vrcholu existuje hrana do jeho následníka v posloupnosti a žádný vrchol se v cestě neopakuje.

Definice 0..5. *Cesta v grafu* $G = \langle V, E \rangle$ *je posloupnost* $P = (v_0, e_1, v_1, e_2, v_2 \ldots\ldots e_n, v_n)$*, pro kterou platí* $e_i = \{v_i - 1, v_i\}$ *a zároveň pro všechna* i, j *platí, že* $v_i \neq v_j$ *pro* $i \neq j$

Pojmem *kružnice* (též cyklus), označujeme cestu $P = (v_0, e_1, v_1, e_2, v_2 \ldots e_n, v_n)$, kde platí $v_0 = v_n$. Graf, který jako podgraf obsahuje alespoň jednu kružnici, nazýváme *cyklický*. V opačném případě se jedná o graf *acyklický* (též strom, kostra).

Hamiltonovská cesta je cesta $P = (v_0, e_1, v_1, e_2, v_2, \ldots en, vn)$, která obsahuje všechny vrcholy v grafu právě jednou. Jako Hamiltonovskou cestu lze uvažovat i *Hamiltonovskou kružnici* s posloupností $P = (v_0, e_1, v_1, e_2, v_2 \ldots\ldots e_n, v_n)$, pro kterou tedy musí platit, že obsahuje všechny vrcholy tvořící graf a zároveň platí, že $v_0 = v_n$[4]. Pokud graf obsahuje Hamiltonovskou cestu, jedná se o *Hamiltonovský graf*.

[4]V TSP rozumíme pojmem nejkratší cesta právě nejkratší Hamiltonovskou kružnici.

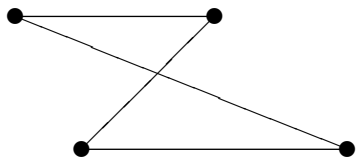
Je zřejmé, že[5]

- Hamiltonovská cesta je kostrou grafu.
- Odstraníme-li ze zadaného grafu hranu, která neleží na nejkratší Hamiltonovské cestě, řešení úlohy (tj. nejkratší Hamiltonovská cesta) se nezmění.

Rozhodnout, zda-li je graf Hamiltonovský, může být velice obtížné (viz později v kapitole o složitosti). Nebyla totiž dosud objevena žádná jednoduchá dostačující podmínka k tomu, aby byl graf Hamiltonovský. Uveďme si ale několik známých případů, kdy podmínka známá je.

Označme si u jako počet uzlů v grafu a $u \geq 3$.

1. Má-li každý uzel stupeň[5] alespoň $\frac{1}{2}u$, graf je Hamiltonovský (tzv. Diracova podmínka).
2. Je-li pro každou dvojici uzlů, které nejsou spojeny hranou, součet jejich stupňů alespoň u, pak je graf Hamiltonovský (tzv. Oreho podmínka).
3. Jestliže pro každé číslo $k \in \mathbb{N}$, pro které platí že $k < \frac{1}{2}u$, je počet uzlů, jejichž stupeň nepřevyšuje k, menší než k, pak graf je Hamiltonovský (tzv. Posova podmínka).



Obrázek 2. Hamiltonovská kružnice

[5]Stupeň vrcholu je počet hran, které do daného vrcholu zasahují.

1. Problém obchodního cestujícího

Na TSP poprvé upozornil v roce 1859 irský matematik R.W. Hamilton v souvislosti s hrou, jejíž cílem bylo spojit dvanáct vrcholů v grafu právě tak, aby byl každý vrchol navštíven právě jednou [4]. Od té doby se TSP řadí k nejvíce studovaným problémům diskrétní optimalizace v teorii grafů. Od 18. století, kdy byl tento problém poprvé definován, se jím zabývá celá řada fyziků, matematiků, informatiků a dalších odborníků, ale přesto se dosud nepodařilo nalézt efektivní matematický algoritmus, který by došel k jednoznačně nejlepšímu řešení v polynomiálně omezeném čase[6]. Existuje však několik algoritmů, které se v polynomiálním čase dokáží k optimálnímu řešení velice přiblížit a kterými se v této práci zaobíráme. TSP se řadí mezi optimalizační problémy, cože vede k definici *optimalizační problém*.

Definice 1..1. *Optimalizační problém je problém o nalezení minimálního nebo maximálního řešení ze všech přípustných řešení. Je definován pomocí*

1. *množiny instancí L*
2. *funkce sol přiřazující instanci $x \in L$ množinu přípustných řešení $sol(x)$*
3. *cenové funkce cost, která $y \in sol(x)$ přiřazuje racionální cenu cost*
4. *podmínky goal, která je bud maximem nebo minimem*

V případě, že goal je maximem, chceme nalézt takové $y \in sol(x)$, které má vzhledem ke všem prvkům $sol(x)$ maximální cenu. Hovoříme pak o maximalizačním problému. Podobně pak pro goal, který je minimem.

Nyní již definice samotného TSP.

Definice 1..2. *TSP je optimalizační problém definovaný následovně*

- $L = \{\langle G, c\rangle$, *G je neorientovaný úplný graf,* $c : E \to \mathbb{Q}$ *je ohodnocovací funkce*$\}$
- $sol = \{T, T$ *je Hamiltonovská kružnice* $\}$*(množina přípustných řešení)*
- $cost(T) = \sum_{e \in T} cost(e)$ *(suma ohodnocení hran v T)*
- *goal: minimum*

Cílem je nalézt Hamiltonovskou kružnici T takovou, že $cost(T)$ je nejmenší možná.

Speciální případ TSP je *metrický TSP* nebo *Euklidovský TSP*. Před samotnou definicí *metrického TSP* je nutno uvést definici pojmu *metrika*.

[6]Tento fakt je příznačný tomu, že TSP spadá mezi NP-úplné úlohy, které definujeme později.

Definice 1..3. *Funkce* $\rho : X \times X \to \mathbb{R}$ *se nazývá metrika, právě když splňuje následující vlastnosti*

- ρ *je symetrická:* $\rho(x, y) = \rho(y, x)$
- ρ *je nulová právě tehdy, když* $x = y : \rho(x, y) = 0 \Leftrightarrow x = y$
- ρ *splňuje trojúhelníkovou nerovnost* $\rho(x, y) \leq \rho(x, z) + \rho(z, y), \forall u, v, w \in V$

Definice 1..4. *O metrickém TSP hovoříme tehdy, pokud ohodnocovací funkce v grafu G je metrika.*

Nyní se se dostáváme k definici Euklidovského TSP.

Definice 1..5. *O Euklidovském TSP hovoříme tehdy, pokud všechny vrcholy v G odpovídají pozici v n-dimenzionálním prostoru a vzdálenost mezi dvěma body* $x = (x_1, x_2, x_3...x_n)$, $y = (y_1, y_2, y_3...y_n)$, $x, y \in V$ *je*

$$\left(\sum_{i=1}^{n} (x_i - y_i)^2\right)^{\frac{1}{2}}$$

Lze také říci, jsou-li vrcholy v grafu G zadány jako souřadnice v Euklidovském prostoru a c je Euklidovská vzdálenost mezi těmito vrcholy, hovoříme o Euklidovském TSP.

Jako TSP z praktického života lze uvést např. problém řidiče dodávky pošty, který má za úkol rozvést poštovní zásilky po městě a musí spotřebovat co nejmenší množství pohonných hmot. Musí tedy zvolit takový okruh, aby navštívil všechny příjemce zásilek, ujel co nejmenší vzdálenost a opět se vrátil na poštu. Samozřejmě pokud je adresátů malé množství, je nalezení optimálního řešení velmi triviální. Stačí použít algoritmus *hrubé síly*, tj. projít všechna možná řešení a vybrat to nejlepší. S rostoucím počtem vrcholů v grafu se počet řešení markantně zvyšuje a tedy projít všechna řešení, neboli projít všechny možné cesty a vybrat tu nejkratší, by trvalo extrémně dlouhou dobu, a proto je toto řešení prakticky nepoužitelné.

Počet vrcholů	Počet různých cest
3	1
5	12
7	360
10	181440
15	$4.35891456 \times 10^{10}$
20	$6.0822550204416 \times 10^{16}$
25	$3.1022420086662 \times 10^{23}$
30	$4.42088099686985 \times 10^{30}$
35	$1.47616399519802 \times 10^{38}$
40	$1.01989410405987 \times 10^{46}$

Tabulka 1. Počet různých cest v úplném grafu v závislosti na počtu vrcholů.

Jak je z tabulky patrné i při relativně malém počtu vrcholů je takřka nemyslitelné procházet všechny cesty v grafu a na základě toho vybrat tu nejkratší. Například kdyby projití jedné cesty trvalo $1s$, nalezení právě optimálního řešení v grafu o 25 vrcholech by trvalo $9.83 * 10^{15}$ let[7]. Počet možných cest je roven $\left(\frac{n!}{n}\right)/2$, kde n je počet vrcholů v grafu.

1.1. Důležité milníky TSP v historii

Následující historické údaje jsou z velké části převzaty ze zdroje [3].

- 1954 – G. Dantzing, R. Fulkerson a S. Johnson řešili příklad s 49 městy v efektivním čase.
- 1962 – Firma Procter&Gamble vyhlásila soutěž, ve které požadovala nalezení optimální cesty pomocí TSP při specifikovaných 33 městech. Vítězem soutěže se stal profesor G. Thompson z Carnegie Mellon University.
- 1977 – Groetschel nalezl optimální cestu pro 120 měst, ze kterých se později stalo Západní Německo.
- 1987 – Padberg a Rinaldi nalezli optimální cestu pro 532 AT&T přepínacích míst v USA.
- 1987 – Groetschel a Holland našli optimální cestu pro 666 zajímavých míst na světě.

[7] Téměř 8 biliard let.

- 1987 – Padberg a Rinaldi nalezli optimální cestu skrz strukturu 2 392 bodů, kterou obdrželi od Tektronics Incorporated.
- 1994 – Applegate, Bixby, Chvátal, a Cook našli optimální cestu pro 7 397 měst. TSP vznikl v programovatelném hradlovém poli2 aplikace v AT&T Bell laboratořích.
- 1998 – Applegate, Bixby, Chvátal, a Cook našli optimální cestu pro 13 509 měst v USA s populací větší než 500.
- 2000 - Clayův matematický ústav vypsal odměnu 1 milion dolarů za objasnění vztahu P a NP úloh, do kterých TSP spadá
- 2001 – Applegate, Bixby, Chvátal, a Cook našli optimální cestu pro 15 112 měst v Německu.
- Applegate, Bixby, Chvátal, Cook, a Helsgaun našli optimální cestu pro 24 978 měst ve Švédsku.

1.2. Složitost TSP

Než se začneme věnovat samotné složitosti TSP, je nutné uvést definici *rozhodovací problém*.

Definice 1..6. *Rozhodovací problém přiřazený optimalizačnímu problému je problém, jehož instance tvoří dvojice $\langle x, k\rangle$, kde x je instance optimalizačního problému a $k \in \mathbb{Q}$. Odpověď pro $\langle x, k\rangle$ je* 1*, pokud cena optimálního řešení x je menší nebo rovna (pro minimalizační problémy) nebo větší nebo rovna (pro maximalizační problémy) než k.*

Pro TSP je rozhodovací problém dán dvojicemi $\langle G, k\rangle$, kde G je graf, $k \in \mathbb{Q}$ a odpověď je 1, právě když G obsahuje Hamiltonovskou kružnici T, pro kterou platí že $cost(T) \leq k$.

Definice 1..7. *Optimalizační problém je NP-těžký* [8]*, právě když jeho přiřazený rozhodovací problém je NP-těžký.*

TSP je NP-těžký. To lze dokázat zredukováním NP-těžkého problému nalezení Hamiltonovské kružnice HT na rozhodovací problém přiřazený TSP.

[8] Nedeterministicky polynomiální.

Redukce

Mějme graf $G = \langle V, E \rangle$, jehož instance je HT. Graf G zobrazíme na G' tak, že jej zúplněníme (tj. přidáme do množiny hran E hrany takové, aby se graf G stal úplným). Ceny hran v G' nastavíme následovně:

$$c(e) = \begin{cases} 1 & \text{pokud } e \in E \\ 100 & \text{pokud } e \notin E \end{cases} \tag{2}$$

Hodnotu rozhodovacího problému k nastavíme na $|V|$. Pokud existuje v G Hamiltonovská kružnice, pak platí, že cena optimálního řešení TSP $cost(HT) = |V|$ a tedy $cost(HT) \leq k$. Lze říci i naopak, pokud je cena optimálního řešení TSP $cost(HT) \leq k$, pak $cost(HT) = k$ a nalezená kružnice je Hamiltonovskou kružnicí v G.

2. Algoritmy řešící TSP

Jak už bylo řečeno, algoritmus který by našel opravdu tu nejkratší cestu polynomiálním čase doposud není znám. Je však známo několik poměrně jednoduše implementovatelných algoritmů bežících v přijatelném čase, které se dokáži k nejoptimálnějšímu řešení velice přiblížit. Tyto algoritmy se nazývají *aproximační* (neposkytující optimální řešení). Při aplikaci aproximačních problémů je žádoucí, aby se aproximační faktor daného algoritmu shora co nejvíce přiblížil hodnotě 1. Pokud je tato hodnota dosažena, znamená to, že je nalezeno právě optimální řešení[4].

Definice 2..1. *Pro aproximační algoritmus A řešící optimalizační problém definujme Aproximační faktor jako funkci $F : \mathbb{N} \to \mathbb{Q}^+$. Pro $x \in L$ optimalizačního problému máme*

$$R_a(x) = max\left\{\frac{OPT(x)}{cost(A(x))}, \frac{cost(A(x))}{OPT(x)}\right\}$$

kde $OPT(x)$ je cena optimálního řešení instance x a $cost(A(x))$ je cena řešení, které spočítá algoritmus A. Funkce F je poté definována jako

$$F(n) = max\{R_A(x) \mid |x| = n, x \in L\}$$

Funkce F vyjadřuje, kolikrát horší je v nejhorším případě řešení vypočítané algoritmem A než optimální řešení pro vstupy o velikosti n. Pokud funkce F je konstanta, pak aproximační faktor nezávisí na velikosti instance x. Je-li například $F(n) = 2$, pak víme, že optimalizační algoritmus vrátí řešení s nejhůře 2-krát větší cenou, než je cena optimální (pro maximalizační problém s 2-krát menší cenou).

2.1. Algoritmus Nearest neighbour (Nejbližší soused)

Jedná se o greedy (žravý) algoritmus, který tvoří *hamiltonovskou kružnici* tak, že v každém kroku vybere nejbližší vrchol již zpracované cesty. Poté, co zpracuje všechny vrcholy v grafu, cestu uzavře. Algoritmus nemá pro metrický TSP konstantní aproximační faktor. Aproximační faktor je logická funkce závislá na počtu vrcholů.

2.1.1. Pseudokód algoritmu

Vstup algoritmu Nearest neighbour (dále už jen NN) je *G, n*, kde G je ohodnocený neorientovaný graf (musí být zvolena vhodná struktura, ve které se uchovává, zdali vrcholy již spadají do cesty)[9] a $n \in \mathbb{N}, n > 1$, které označuje počet vrcholů v grafu. Výstup je pak seznam vrcholů s, který reprezentuje výslednou Hamiltonoskou kružnici.

1: **NN(G[ ],n)**
2: vytvoř seznam s
3: přidej do seznamu s bod $G[1]$ a označ ho jako navštívený
4: $x \leftarrow$ bod v G, který je nejblíže k bodu v seznamu s a je nenavštívený
5: $x.navstiven \leftarrow$ **true**
6: $s.konec \leftarrow x$
7: **while** počet prvků v $s < n$ **do**
8: $x \leftarrow$ bod v G, který je nejblíže k bodu na začátku seznamu s a je nenavštívený
9: $y \leftarrow$ bod v G, který je nejblíže k bodu na konci seznamu s a je nenavštívený
10: **if** vzdálenost bodu x od bodu na začátku seznamu < vzdálenost bodu y od bodu na konci seznamu **then**
11: $s.zacatek \leftarrow x$
12: $x.navstiven \leftarrow$ **true**
13: **else**
14: $s.konec \leftarrow y$
15: $y.navstiveny \leftarrow$ **true**
16: **end if**
17: **end while**
18: **return** s

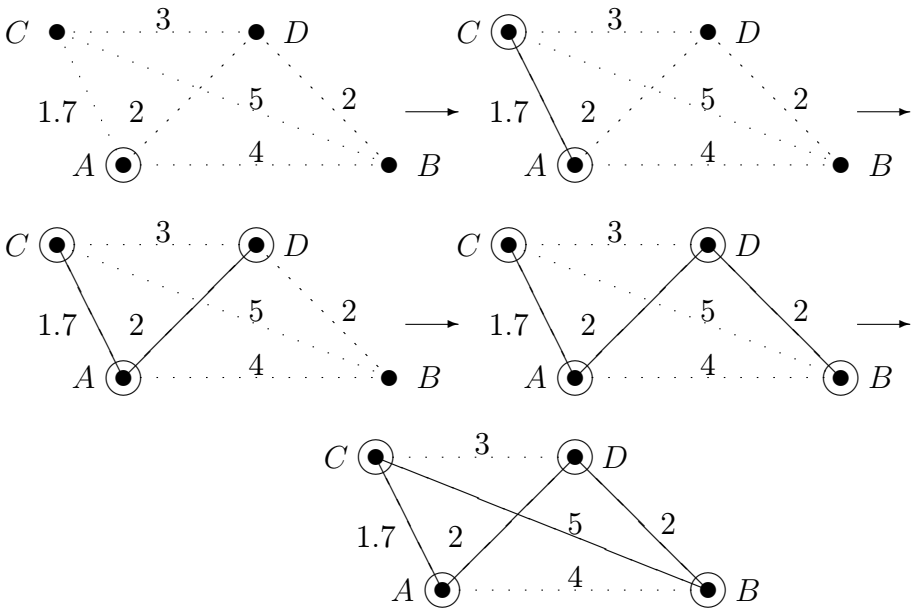
2.1.2. Příklad použití algoritmu

Jako první bod lze použít libovolný bod v grafu. Poté se vybírá vždy nejbližší bod nejprve od onoho prvního bodu, poté od okrajů aktuální cesty. Po navštívení

[9]Pokud není uvedeno jinak, vstup algoritmů pro řešení TSP vždy obsahuje klíčovou složku G, tedy pole, ve kterém jsou uloženy vrcholy grafu ve vhodné struktuře.

všech bodů je nutno samozřejmě spojit první a poslední bod cesty, aby vznikla Hamiltonovská kružnice.

Před samotným vykonáváním algoritmu nemusíme vrcholy ani hrany nijak upravovat (např. třídit). Na začátku algoritmus musí procházet všechny nenavštívené vrcholy. Např. pokud je již zpracovaná cesta tvořena právě 2 vrcholy, musí z $n-2$ vrcholů a vybrat právě ten nejbližší. Algoritmus tedy prochází $n \times n$ vrcholů a tedy složitost je $O(n^2)$.



Obrázek 3. Příklad běhu algoritmu NN

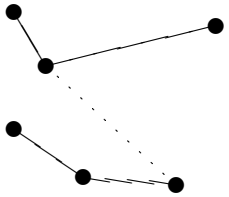
1. Jako první je vybrán vrchol A.
2. Jako nejbližší vrchol k vrcholu A je vybrán vrchol C.
3. Hledá se nejkratší hrana od krajních vrcholů cesty, tj. od vrcholů C a A. To je hrana z A do D. Hrana je tedy přidána do cesty.
4. Opět se hledá nejkratší hrana z krajních vrcholu, tj. C a D. Nejkratší je z D do B. Hrana je přidána do cesty.
5. Nyní jsou navštíveny již všechny vrcholy v grafu, je tedy nalezena Hamiltonovská cesta. Nyní jsou tedy spojeny krajní vrcholy cesty a vznikne Hamiltonovská kružnice.

2.2. Algoritmus Greedy heuristic (Žravě heuristický)

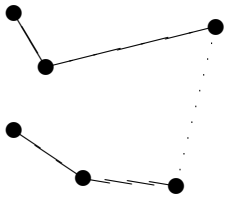
Tento algoritmus, jak už z názvu vyplývá, se snaží ve svém průběhu každým krokem dostat "žravě" co nejblíže k optimálnímu řešení. Princip algoritmu je velmi podobný jako při hledání minimální kostry grafu[10]. Postupně tedy spojuje hrany s nejmenším ohodnocením (s nejkratší délkou), dokud nepropojí všechny vrcholy. Každá hrana je přidána do cesty v závislosti na dvou podmínkách:

1. Hrana je přidána, pokud po jejím přidání nevznikne kružnice.
2. Hrana je přidána, pokud v grafu vznikne množina disjunktních cest (cesty vznikající v grafu se nesmí navzájem protínat).

Po propojení všech cest v grafu které byly nalezeny v průběhu algoritmu vznikne v grafu právě jedna Hamiltonovská cesta. Po přidání hrany mezi prvním a posledním vrcholem této cesty vznikne *Hamiltonovská kružnice*.



Přidáním hrany nevznikne množina disjunktních cest, hrana nemůže být přidána.



Přidáním hrany vnikne množina disjunktních cest, hrana může být přidána.

Obrázek 4. Disjunktní cesty

Po propojení všech vrcholů algoritmus spojí první a poslední vrchol cesty a vznikne *Hamiltonovská kružnice*.

[10]Kruskalův algoritmus.

2.2.1. Pseudokód algoritmu

Vstupem algoritmu Greedy heuristic (dále už jen GH) je pole hran *edges*, počet vrcholů $n \in \mathbb{N}, n > 1$ a počet všech hran q pro které platí $q = \binom{n}{2}$. Prvky v poli *edges* musí mít vhodnou strukturu pro uchování informací o hraně (jestli hrana spadá do cesty, délka hrany apod.). Výstupem je pak rovněž pole *edges* s příznakem v každé hraně, jestli do cesty spadá. Na začátku algoritmu žádná hrana do cesty nespadá.

Nyní samotný algoritmus GH. Před samotným tvořením cesty je nutné hrany v grafu vzestupně setřídit. V pseudokódu je použitý známý třídící algoritmus Quick-sort. V algoritmu jsou klíčové dvě podmínky *pod1* a *pod2*.

pod1 Přidáním hrany do cesty nevznikne kružnice.

pod2 Přidáním hrany do cesty nevznikne disjunktní cesta.

```
1: GH (edges[ ], n, q)
2: Quick-sort(edges, 0, q)
3: lengthWay ← 0
4: i ← 0
5: while lengthWay < n − 1 do
6:    if pod1 and pod2 then
7:       edges[i].cesta ← true
8:       lengthWay + +
9:    end if
10:   i + +
11: end while
12: return edges
```

Příznak hrany *edges.cesta* určuje, zda-li hrana spadá do cesty nebo ne. I když se z pseudokódu může zdát, že algoritmus GH je značně jednoduší než NN, je jeho implementace značně složitější a to právě kvůli ověřování podmínek *pod1* a *pod2*.

Každá samostatná cesta má přiřazený unikátní klíč $k \in \mathbf{N}$. Hodnotu klíče k nese každý vrchol $v \in V$, který do dané cesty spadá, jako hodnotu v_k. Pokud vrchol v zatím do žádné cesty nespadá, je hodnota $v_k = 0$. Pro každý vrchol $v \in V$ uvažujme stupeň vrcholu v_d. Pokud vrchol v zatím nespadá do žádné cesty, je hodnota $v_d = 0$. Při každém přidáním hrany $e(x, y)$ do cesty se hodnota x_d a y_d inkrementuje. Při pokusu o přidání hrany $e(x, y)$ mohou nastat následující případy:

($x_k, y_k = 0$) **Hrana je přidána** Ani jeden vrchol prozatím nespadá do žádné cesty. Vytvoří se nová cesta s unikátním klíčem k a přiřadí se $x_k, y_k = k$.

($x_k = 0; y_k > 0; y_d < 2$) **Hrana je přidána** Vrchol x prozatím nespadá do žádné cesty, zatímco y již spadá do cesty s indexem k a stupeň $y_d < 2$ a tedy

přidáním hrany $e(x, y)$ vznikne podgraf, který je množinou disjunktních cest. Cesta s indexem k se rozšíří o vrchol x tak, že se přiřadí $x_k = y_k$. Obdobně pak pro případ $(y_k = 0, x_k > 0, x_d < 2)$.

$(x_k, y_k > 0; x_k \neq y_k; y_d, x_d < 2)$ **Hrana je přidána** Vrchol x i y již spadají každý do jiné cesty. Tyto dvě cesty se propojí přidávanou hranou $e(x, y)$ tím, že pro všechny vrcholy $v \in V$ pro které platí $v_k = v_x$ se přiřadí $v_k = v_y$.

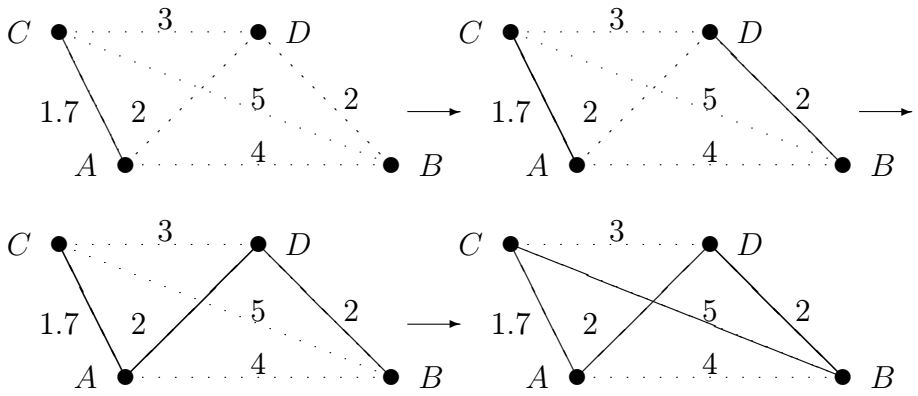
$(x_k = 0; y_k > 0; y_d = 2)$ **Hrana není přidána** Ačkoli vrchol x ještě do žádné cesty nespadá, vrchol y již do cesty spadá ale $y_d = 2$ a tedy přidáním hrany $e(x, y)$ nevznikne podgraf, který není množinou disjunktních cest. Obdobně pak pro případ $(y_k = 0, x_k > 0, x_d = 2)$.

$(x_k, y_k > 0; x_k \neq y_k; x_d = 2; y_d = 1)$ **Hrana není přidána** Ačkoli vrchol x i y spadá každý do jiné cesty a stupeň $y_d = 1$, hrana $e(x, y)$ nemůže být přidána, z důvodu $x_d = 2$. Nevznikne tedy podgraf, který není množinou disjunktních cest. Obdobně pak pro případ $(x_k, y_k > 0; x_k \neq y_k; y_d = 2; x_d = 1)$.

$(x_k = y_k; x_k, y_k \neq 0)$ **Hrana není přidána** Vrchol x i y spadají do stejné cesty a tedy přidáním hrany $e(x, y)$ by v cestě vznikla kružnice.

2.2.2. Příklad použití algoritmu

Předpokládejme, že hrany v poli *edges* jsou již vzestupně seřazené.



Obrázek 5. Příklad běhu algoritmu GH

1. Do cesty je přidána nejkratší hrana z A do B.
2. Dále je přidána druhá nejkratší hrana z A do C. Hrana může být přidána protože se jejím přidáním nezapříčiní vznik podgrafu, který není množinou disjunktních cest, ani kružnice.

3. Stejně tak další hrana z A do D.

4. Jelikož už jsou navštíveny všechny vrcholy v grafu, stačí pouze propojit první a poslední vrchol cesty a algoritmus je u konce.

Z principu algoritmu se může zdát, že algoritmus opravdu najde tu nejkratší cestu, jelikož využívá opravdu jen ty nejkratší hrany. Ovšem právě na Obrázku 5 si lze povšimnout, že algoritmus k uzavření cesty musel použít hranu z A do D, která je v grafu úplně nejdelší, což není optimální řešení.

2.3. Algoritmus Replace the edges (Prohození hran)

Jedná se o algoritmus, který upravuje již existující Hamiltonovskou kružnici tím, že se ji snaží neustále zkracovat. Princip zkracování kružnice je ten, že algoritmus rozdělí kružnici na dvě části a ověří, jestli prohozením hran, které tyto dvě části spojují, nevznikne kratší celková kružnice. Pokud ano, hrany prohodí a cesta je zkrácena.

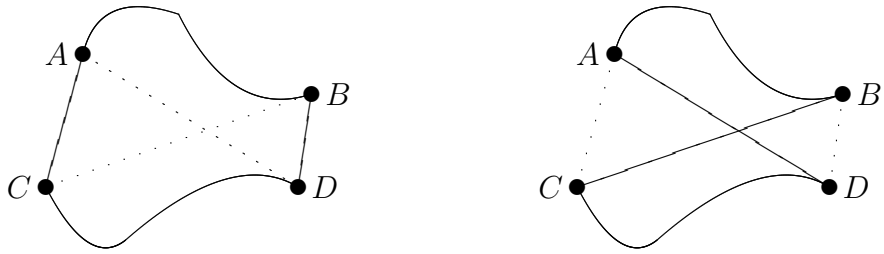
2.3.1. Pseudokód algoritmu

Jelikož pro pokus o zkrácení musíme projít všechny hrany, které jsou tvořeny vrcholy a reprezentace cesty je pomocí pole vrcholů ve kterém se předpokládá že cesta vede postupně v poli jako $v_1, v_2 ... v_n$, musí se vždy projít všechny dvojice hran $\langle a, b\rangle$ a $\langle c, d\rangle$, kde platí $\forall a, b, c, d \mid a \neq c, b \neq d$ a případně hrany prohodit, složitost algoritmu Replace the edges (dále už jen RtE) je $O(e^2)$, kde e je počet hran.

```
1: RtE(G[ ], n)
2: for i ← 0 to n − 1 do
3:    Ledge1 ← délka hrany ⟨G[i], G[(i + 1) mod n]⟩
4:    for j ← (i + 1) mod n to (j + 2) mod n = i, j++ mod n do
5:       Ledge2 ← délka hrany ⟨G[j], G[(j + 1) mod n]⟩
6:       Lalt ← délka hrany ⟨G[i], G[j]⟩+ délka hrany ⟨G[(i+1) mod n], G[(j+1)
         mod n]⟩
7:       if Lalt < Ledge1 + Ledge2 then
8:          Odstraň hranu ⟨G[i], G[(i + 1) mod n]⟩
9:          Odstraň hranu ⟨G[j], G[(j + 1) mod n]⟩
10:         Vytvoř hranu ⟨G[i], G[j]⟩
11:         Vytvoř hranu ⟨G[(i + 1) mod n], G[(j + 1) mod n]⟩
12:      end if
13:   end for
14: end for
15: return G
```

*Ledge*1 a *Ledge*2 jsou délky dvou hran, které v Hamiltonovské kružnici existují. Algoritmus zkouší prohazovat hrany tak, že namísto existujících hran vytvoří alternativní hrany o celkové délce *Lalt*. K prohození dojde pokud délka alternativních hran *Lalt* je menší než součet délek hran *Ledge*1 a *Ledge*2.

2.3.2. Příklad použití algoritmu



Nedojde k prohození hran. Dojde k prohození.

Obrázek 6. Prohození hran v RtE

V Obrázku 6 je znázorněn princip porovnávání hran a jejich následné prohození. Tedy k prohození hran dojde právě tehdy, pokud dojde ke zkrácení Existující Hamiltonovské kružnice. Tento algoritmus se od ostatních algoritmů liší hlavně tím, že cestu nevytváří, ale upravuje již existující cestu. Algoritmus lze využít tedy k samotnému vytvoření Hamiltonovské kružnice tím, že algoritmus pracuje s náhodnou posloupností vrcholů nebo se pokusí optimalizovat již existující kružnici vytvořenou jiným algoritmem. Této optimalizaci budě věnována pozornost později.

Je nutné také podotknout, že algoritmus RtE se rozhodně nehodí jako první implementace algoritmu pro řešení Euklidovského TSP. Je totiž poměrně složitější, na rozdíl od jiných algoritmů, si představit samotné prohazování hran. Reprezentace Hamiltonovské kružnice v nejednodušší implementaci, která byla použita i v přiloženém programu, je seznam vrcholů v tom pořadí, jak jdou v cestě za sebou. Tedy pokud cesta vede přes vrcholy *A, B, C, D, E, F*, tak přímo takto vypadá i seznam, který danou cestu reprezentuje. Pokud je třeba danou cestu optimalizovat prohozením dvou hran např. *A*-*B* a *E*-*F*, nestačí pouze vyměnit dva vrcholy *B* a *E*, aby vznikla cesta *A, E, C, D, B, F*, ale je nutné rotovat i celou posloupnost, která se mezi těmito vrcholy nachází. Pro řádné vyměnění hran tedy musí vzniknout posloupnost *A, E, D, C, B, F*.

2.4. Inserting Vertices (Vkládání vrcholů)

Princip tohoto algoritmu Inserting Vertices (dále už jen InV) je ten, že se snaží již zpracovanou kružnici rozšiřovat tím, že do ní vkládá dosud nenavštívené vrcholy. Je vždy přidán právě ten vrchol, který kružnici rozšíří o nejmenší

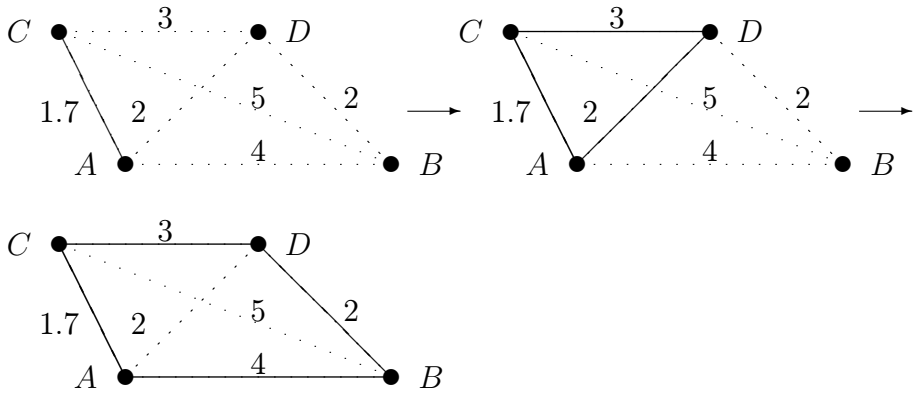
vzdálenost. Aby bylo možné rozšiřovat kružnici, je nutné na začátku algoritmu vybrat první hranu, ze které pak vznikne kružnice o třech vrcholech. Tato hrana může být zvolena libovolně ale pro nalezení co nejkratší Hamiltonovské kružnice je vhodnější, vybrat pravě tu nejkratší.

2.4.1. Pseudokód algoritmu

Vstup algoritmu je $n \in \mathbb{N}, n > 1$, které označuje počet vrcholů v grafu. Výstupem je pak seznam hran s, který obsahuje hrany tvořící výslednou cestu. Algoritmus postupně vkládá do kružnice nové vrcholy, dokud nejsou do kružnice vloženy všechny vrcholy v grafu a tím vznikne Hamiltonovská kružnice. Na začátku rozšiřovaná kružnice s neobsahuje žádný vrchol. Každá hrana v grafu je reprezentována jako $e(x, y)$, kde x a y jsou vrcholy, které hrana spojuje. Hrana, od které se kružnice rozšiřuje, je odstraněna z kružnice právě tehdy, když již kružnice s v grafu existuje (pokud jí tvoří minimálně 3 vrcholy). Není nezbytně nutné a ani vždy vhodné začínat od nejkratší hrany, ale ve většině případů je vhodné do výsledné cesty s přidat nejdříve právě nejkratší hranu.

```
1: InV(n)
2: cesta.add(nejkratsihrana.xnejkratsihrana.y)
3: for i ← 2 to n do
4:     mv ← vrchol nejbližší k cestě s
5:     mk ← hrana v cestě s, která je nejblíže k mv
6:     s.add(e(mk.x, mv))
7:     s.add(e(mk.y, mv))
8:     if i > 2 then
9:         s.remove(mk)
10:    end if
11: end for
12: return s
```

2.4.2. Příklad použití algoritmu



Obrázek 7. Příklad běhu algoritmu InV

1. Optimálně je do kružnice nejdříve přidána nejkratší hrana z A do C.

2. Nyní se musí kružnice rozšířit a to tak, aby se přidáním vrcholu do kružnice délka kružnice prodloužila o co nejméně. Pokud bychom přidali vrchol B, kružnice se prodlouží o 9. "Levnější" možnost je přidat vrchol D. Tím jr kružnice prodloužena pouze o 5.

3. Jako poslední je třeba do kružnice přidat vrchol D. Po projití všech možností rozšíření kružnice, je kružnice rozšířena z vrcholů A a D. Kružnice se tím prodlouží pouze o 4 (zrušená hrana A do D již do cesty nepatří a proto je její délka 2 z délky kružnice odečtena).

2.5. Algoritmus Nearest neighbor and Replace the edges (Nejbližší soused a Prohození hran)

Jak již bylo zmíněno v kapitole výše, algoritmus RtE lze použít ke zkrácení již existující Hamiltonovské kružnice, kterou vytvořil jiný algoritmus. Je lépe tento algoritmus využít ke zkrácení již nalezené Hamiltonovské kružnice, pokud chceme dosáhnout optimálnějšího výsledku než první, samotný algoritmus. K vytvoření Hamiltonovské kružnice a tedy k samotnému vyřešení TSP je použito jiného algoritmu, v našem případě NN. Po nalezení Hamiltonovské kružnice je cesta ještě zkrácena pomocí algoritmu RtE a tedy by mělo být dosaženo lepšího výsledku než dosáhl algoritmus NN (tento mechanismus bude dále označován jako algoritmus NNaRTE), což se snad ukáže v pozdější kapitole *Experimentálního porovnávání algoritmů*.

2.5.1. Pseudokód algoritmu

Jak lze předpokládat, pseudokód je velmi jednoduchý. Stačí použít nejdříve algoritmus NN pro vytvoření vstupní Hamiltonovské kružnice, se kterou pak dále bude pracovat RtE. Vstup algoritmu je tedy graf G a $n \in \mathbb{N}, n > 1$, které označuje počet vrcholů v grafu. Výstup algoritmu je pak upravený graf G, ve kterém jsou vrcholy seřazené v tom pořadí, které tvoří cestu.

```
1: NNaRTE(G[ ], n)
2: s ← NN(G[ ], n)
3: for i ← 0 to n − 1 do
4:     G[i] ← s[i]
5: end for
6: RTE(G[ ], n)
7: return G
```

2.6. Algoritmus Minimum spanning tree (Minimální kostry grafu)

Algoritmus Minimum spanning tree (dále už jen MST) patří mezi heuristické algoritmy založené na principu nalezení minimální kostry grafu. Funguje tedy nejen pro *Euklidovský* TSP, ale i pro *Metrický* TSP. Aproximační faktor algoritmu MST je 2, tzn. výsledná Hamiltonovská kružnice algoritmu TSP je v nejhorším případě právě dvakrát delší, než nejkratší Hamiltonovská kružnice, kterou graf obsahuje. Aproximační faktor tedy nepřesáhne hodnotu 2 a to z důvodu využití právě minimální kostry grafu jejíž ohodnocení je dolní mez optimální Hamiltonovské kružnice a její procházení do hloubky za předpokladu, že platí trojúhelníková nerovnost. Při procházení grafu do hloubky je každá hrana minimální kostry navštívena právě dvakrát a to při dosažení vrcholů a vracení se zpět na začátek procházení grafu [1]. Průběh algoritmu je rozdělen do 3 etap:

1. Nalezení minimální kostry grafu.
2. Projití nalezené kostry do hloubky a vytvoření posloupnosti vrcholů takové, ve které byly vrcholy navštíveny.
3. Odstranění duplicit z posloupnosti.

2.6.1. Pseudokód algoritmu

K nalezení minimální kostry grafu lze použít mnoho algoritmů (např. *Kruskalův algoritmus*[11]). Vstup tohoto algoritmu je tedy pole G, kde jsou uloženy vrcholy grafu a $n \in \mathbb{N}, n > 1$, které označuje jejich počet. Stejný vstup má i

[11] Poprvé publikován Josephem B. Kruskalem v roce 1956

samotný algoritmus MST. Výstup je pak podgraf grafu G graf F, který je minimální kostrou grafu G. Nyní je nutné projít F do hloubky. Princip procházení kostry $F = \langle V, E \rangle$ do hloubky je následující:

Mějme zásobník S a libovolný vrchol $v \in V$

1. Vlož v do S
2. Odeber prvek z S, označ ho jako t a označ ho jako navštívený.
3. Do S ulož všechny nenavštívené vrcholy, které sousedí s vrcholem t.
4. Pokud je S prázdný skonči, jinak pokračuj bodem 2.

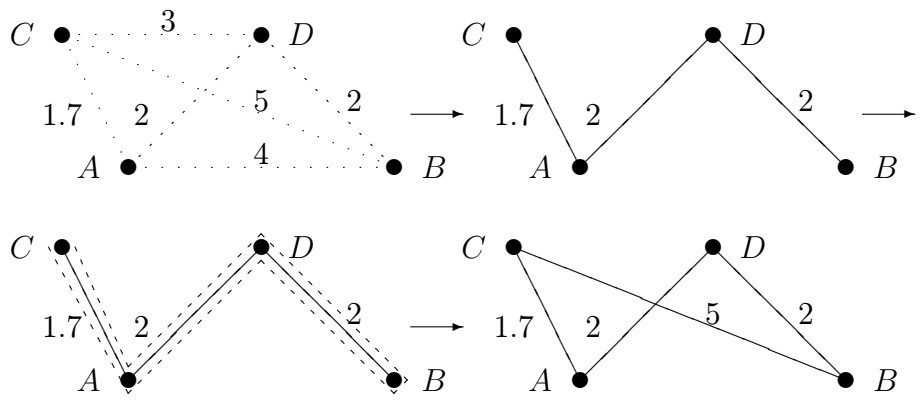
Pro úplnost je třeba dodat, že při procházení grafu do šířky se namísto zásobníku S použije fronta S.

Je také nutné zvolit si vhodnou strukturu, například seznam, kde se ukládá pořadí vrcholů, v jakém byly odebrány se zásobníku. Tuto posloupnost si označme Se. Po projití grafu do hloubky se projde posloupnost Se následovně: Posloupnost Se se prochází zleva doprava. Označme si právě kontrolovaný vrchol v_i. Pokud v pravé části Se od vrcholu v_i není nalezen žádný vrchol v_j, pro který platí $v_i = v_j$, je přidán vrchol v_i do seznamu way, který reprezentuje výslednou cestu algoritmu MST.

```
1: MST(G[ ], n)
2: F ← Kruskal(G, n)
3: Se ← Posloupnost vrcholů při projití kostry F do hloubky
4: toway ← true
5: for i ← 1 to Se.count − 2 do
6:    for j ← i + 1 to Se.count − 1 do
7:       if Se[i] = Se[j] then
8:          toway ← false
9:       end if
10:   end for
11:   if toway = true then
12:      way.add(S[i])
13:   else
14:      toway ← true
15:   end if
16: end for
17: return way
```

Naproti předešlým algoritmům je u tohoto algoritmu předpoklad, že jeho běh je výrazně delší a to z důvodu více fází algoritmu. Naproti tomu výrazná přednost algoritmu je právě jeho aproximační faktor.

2.6.2. Příklad použití algoritmu



Obrázek 8. Příklad běhu algoritmu MST

1. Graf, na který aplikujeme algoritmus MST.
2. Nalezena minimální kostra grafu,
3. Nyní graf projdeme do hloubky. Začneme ve vrcholu C. Tím vznikne posloupnost $Se = (C, A, D, B, D, A, C)$.
4. Z Se odstraníme duplicitní vrcholy od vrcholu Se_1. Vznikne tím cesta $way = (C, B, D, A, C)$, která je výstupní *Hamiltonovskou kružnicí* algoritmu.

2.7. Algoritmus TSP Backtracing (TSP pomocí zpětného vyhledávání)

Jelikož později v práci jsou zmíněné algoritmy porovnány a jedním z hlavních kriterií je, jak je jejich řešení blízko k právě tomu optimálnímu, je nutné implementovat optimální algoritmus, tedy algoritmus, který vrátí právě tu nejkratší Hamiltonovskou kružnici v grafu. Jak již bylo zmíněno, jediný zatím známý algoritmus, který by vždy takové řešení nalezl, je algoritmus hrubé síly, tedy naivní algoritmus, který prochází všechna možná řešení problému a poté vrátí to nejoptimálnější čili optimální.

Nelze tedy použít jiný algoritmus, než algoritmus hrubé síly. Lze ale zlepšit jeho efektivitu pomocí tzv. *zpětného vyhledávání* (dále jen BT - z alglického Backtracing), který hledání optimálního řešení výrazně urychlí. BT je vylepšení algoritmů založené na principu hrubé síly. Předpokládejme, že již máme zpracovanou část cesty $way = (v_0, v_1, v_2, v_3 ... v_n)$. Zda-li má smysl pokračovat v hledání

řešení v aktuální posloupnosti doplňováním dosud nenavštívených vrcholů je dáno dvěma podmínkami:

1. Hledání pokračuje pokud je posloupnost *way* prefixem řešení tj. má-li ještě smysl posloupnost rozšiřovat o další vrcholy v.

2. Hledání pokračuje pokud je *way* již částečným řešením tj. že řešení je dosaženo přidáním posledního vrcholu v_{n+1}. Jinak řečeno cesta *way* tvoří Hamiltonovskou cestu.

I když metoda BT velmi zlepší efektivitu naivního algoritmu, je jeho složitost stále $O(n!)$ a tedy je vhodný pouze pro grafy s malým počtem vrcholů (max. 15). Neřadí se ale mezi *aproximační* algoritmy a tedy výstupní řešení TSP je vždy to *minimální*.

2.7.1. Pseudokód algoritmu

K tomu, aby bylo již při startu hledání nejkratší cesty využito BT, je nutné znát délku *bL* (border length), která ohraničuje nejdelší délku, kterou by mohlo mít řešení v nejhorším případě. K získání *bLength* je využit algoritmus MST, a to z důvodu jeho aproximačního faktoru 2. Vstup algoritmu je tedy graf G a $n \in \mathbb{N}, n > 1$, které označuje počet vrcholů v grafu. Výstup je pak seznam vrcholů, který je výstupem algoritmu bT, který tvoří optimální Hamiltonovskou kružnici v grafu G.

1: **TSPBt (G[ ], n)**
2: bL ← MST(G[ ], n).length
3: M ← G
4: vytvoř prázdnou množinu *way*
5: **return** bT(M, way, 0)

Na začátku algoritmu se nejdříve do *bL* uloží délka Hamiltonovské kružnice, která je výstupem algoritmu MST a která je tedy v nejhorším případě dvakrát delší než nejkratší Hamiltonovská kružnice v grafu G. Do množiny M se uloží všechny vrcholy v grafu G. Tato množina obsahuje vrcholy, které lze ještě přidat do vytvářené cesty (nedochází tedy v cestě k duplicitám). Poté je ještě vytvořena prázdná množina *way*, ve které se postupně budou ukládat již zpracované vrcholy cesty. Nyní se dostáváme k podstatě samotného algoritmu, funkci BT. Vstup funkce BT je množina M, ve které jsou uloženy vrcholy, které ještě lze do cesty *way* přidat, množina *way*, která obsahuje vrcholy v právě zpracovávané cestě (je nutné, aby množina *way* nesla společně s posloupností vrcholů také číslo *length*, které udává, jak je aktuální cesta tvořená zpracovanými vrcholy dlouhá) a číslo *index*, které udává pozici vrcholu, který je potenciálně v následujícím zavolání *bt* odstraněn z množiny M a přidán do cesty *way*.

```
1: BT (M, way, index)
2: if M = ∅ then
3:    way.add(way[0])
4:    if way.length ≤ bL then
5:       bL ← way.length
6:       return  way
7:    end if
8: else
9:    way.add(M)
10:   if way.length < bL then
11:      M ← M − M[index]
12:      for i ← 0 to M.count do
13:         bT(M, way, i)
14:      end for
15:   end if
16: end if
```

Hlavní princip backtrackingu je využit na 4. řádku, kde se kontroluje, zdali ještě má smysl pokračovat v hledání řešení v aktuální cestě *way*, tedy pokud ještě zpracovaná cesta nepřesáhla délku Hamiltonovské kružnice, která byla již nalezena dříve (jestli je zpracovaná cesta *way* prefixem řešení). Druhý případ, kdy algoritmus pokračuje v hledání cesty, je pokud byly již zpracovány všechny vrcholy v grafu, tedy pokud množina M je prázdná. Pokud ano, znamená to, že je nalezena Hamiltonovská cesta a je nutno udělat poslední krok - uzavřít Hamiltonovskou kružnici. Pokud délka Hamiltonovské kružnice je stále menší nebo rovno než bL, podařilo se nalézt prozatím nejkratší Hamiltonovskou kružnici a tedy kružnici lze vrátit jako výsledek aktuální rekurze a obnovit bL, protože pokud se podaří najít v budoucnu optimálnější řešení než je *way*, musí být délka potenciální Hamiltonovské kružnice menší než bL.

3. Experimentální porovnání algoritmů

Nyní, po popsání několika algoritmů pro řešení Euklidovského TSP, je třeba algoritmy navzájem porovnat a zjistit, který z těchto algoritmů je nejefektivnější. Kvalita algoritmů je porovnána na základě dvou důležitých faktorů:

- Výsledná délka Hamiltonovské kružnice pro stejný graf.
- Čas, který algoritmus potřeboval k nalezení řešení.

V první fázi testování jsou algoritmy aplikovány na náhodné grafy s různým počtem vrcholů. Je celkem použito 1000 náhodných grafů (pokud není uvedeno jinak) se stejným souřadnicovým rozsahem (0 - 2 000) a jako výsledek měření je použita průměrná hodnota (jak délky Hamiltonovské kružnice tak času). Souřadnice vrcholů v každém grafu byli vygenerovány ryze náhodně. Generování náhodných grafů bylo implementováno v testovacím programu. Vzhledem k tomu, že v práci bylo implementováno i řešení Euklidovského TSP pomocí Backtrackingu, který nalezne právě nejkratší Hamiltonovskou kružnici, je možné i ověřit, jak blízko se jednotlivé algoritmy přiblíží k optimálnímu řešení. Samozřejmě tento faktor lze zkoumat jen pro grafy s nízkým počtem vrcholů (maximálně 10), a to z toho důvodu, jak již bylo zmíněno, pro grafy s větším počtem vrcholů je nalezení nejkratší Hamiltonovské kružnice extrémně časově náročné.

3.1. Testování algoritmu NN

Počet vrcholů	**Opt. řešení**	**Algoritmus**	**Čas [ms]**
5	4 395,14726	4 251,4968	0,06
10	5 689,4363	6 236,4668	0,11
15	x	7 606,03	0,25
20	x	8 886,872	0,39
50	x	13 786,5815	2
100	x	19 183,012	8
250	x	29 617,5962	48
500	x	41 324,405	186
1000	x	57 688.0362	777
1500	x	70 283,504	1 880
3000	x	98 441,1606	7 464

Tabulka 2. Výkon algoritmu NN na náhodných grafech

Z tabulky je patrné, že ani při malém množství vrcholů algoritmus nedosáhne právě té nejkratší Hamiltonovské kružnice. Nicméně rozdíl délek nejkratší Hamiltonovské kružnice a kružnice, kterou algoritmus nalezl není nijak markantní, a proto usuzuji, že algoritmus by mohl být použit i v praktických úlohách, kdy je třeba nalézt nejkratší cestu přes všechny vrcholy (města) v grafu. Další přednost algoritmu je jeho snadná implementace. Bylo by vhodné ho použít i jako ukázku možného řešení Euklidovského TSP pro laiky.

Je nutno také podotknout, že na rozdíl od většiny ostatních algoritmů, algoritmus NN nepracuje s hranami grafu, ale pouze z jeho vrcholy. Není tedy nutné ukládat do vhodné datové struktury velké množství hran, ale postačí nám při implementaci poměrně snadným způsobem pracovat pouze s vrcholy. Tento algoritmus se tedy hodí i pro grafy s extrémním počtem vrcholů pro jeho malou paměťovou složitost.

3.2. Testování algoritmu GH

Počet vrcholů	**Opt. cesta**	**Algoritmus**	**Čas [ms]**
5	4 304,0686	4 362,267	0,26
10	5 757,3598	6 146,8811	0,37
15	x	7 516,9037	0,52
20	x	8 628,4734	0,57
50	x	13 247,0182	1
100	x	18 400,3986	3
250	x	28 036, 4164	15
500	x	39 952,157	65
1000	x	54 173,8735	359
1500	x	68 810,669	771
3000	x	91 902,0344	3 222

Tabulka 3. Výkon algoritmu GH na náhodných grafech

Stejně jako předchozí algoritmus NN, je algoritmus GH také žravý algoritmus a proto délky výstupních kružnic jsou velmi podobné. Lze si ale povšimnout, že časová náročnost algoritmu při velkém počtu vrcholů prudce stoupá a z toho důvodu, že algoritmus GH musí před tvořením výstupní cesty utřídit $\frac{n(n-1)}{2}$ hran. Dále pak už setříděné hrany sekvenčně prochází a rozhoduje, zdali je možné příslušnou hranu do výsledné Hamiltonovské kružnice přidat nebo ne. Lze tedy také říci, že tento algoritmus není pro extrémně velké úplné grafy příliš vhodný. Jak již bylo zmíněno, na rozdíl od algoritmu NN, pracuje tento algoritmus ne

s vrcholy, ale s hranami. Jelikož na začátku není známo, které hrany budou ve výsledné Hamiltonovské kružnici použity, je nutno uložit a setřídit všechny hrany v úplném grafu, což může být paměťově velmi náročné, pracujeme-li s množstvím hran čítající řádově milióny. Vzhledem k tomu, že oba algoritmy jsou stejného typu, je vhodné porovnat přímo tyto dva algoritmy.

Počet vrcholů	**Optimálnější řešení**	**Rychlejší algoritmus**
5	NN	NN
10	GH	NN
15	GH	NN
20	GH	NN
50	GH	GH
100	GH	GH
250	GH	GH
500	GH	GH
1000	GH	GH
1500	GH	GH

Tabulka 4. Porovnání algoritmů NN a GH

Měření poměrně jasně ukázalo kvality těchto dvou algoritmů. Pro grafy s větším počtem vrcholů je výhodnější použít algoritmus GH, pokud důležitějším faktorem oproti času hledání Hamiltonovské kružnice je její samotná délka. Nicméně časové rozdíly algoritmů jsou natolik malé, že lze říct, že obecně lepší algoritmus je jednoznačně algoritmus GH, pomineme-li jeho větší paměťovou náročnost.

3.3. Testování algoritmu RtE

Další testovaný algoritmus je RtE. Tento algoritmus se od předchozích dvou liší tím, že se snaží zkrátit již existující Hamiltonovskou kružnici.

Počet vrcholů	Nejkratší cesta	Algoritmus	Čas [ms]
5	4 251,6061	4 309,3467	0,02
10	5 750, 9013	5 960,4267	0,07
15	x	7 152,6674	0,16
20	x	8 217,2752	0,3
50	x	12 900,303	2
100	x	18 172,7084	8
250	x	28 716,9202	47
500	x	40 669,8777	230,2
1000	x	57 578,4828	869
1500	x	70 532,6408	2 050
3000	x	99 891, 2288	8 011

Tabulka 5. Výkon algoritmu RtE na náhodných grafech

Z průběhu měření je patrné, jak se algoritmus od ostatních liší. Při pozorování bylo zjištěno, že ačkoli má algoritmus na svém vstupu cestu, která tvoří doslova zmatenou "pavučinu" hran, algoritmu se ji daří rozplétat. I při prvním pohledu na Hamiltonovskou kružnici v úplném grafu lze říct, zda-li se vůbec může jednat o nejkratší Hamiltonovskou kružnici - pokud se v cestě hodně hran kříží, rozhodně se nejedná o nejoptimálnější řešení Euklidovského TSP, ale pokud v Hamiltonovské kružnici není na první pohled žádné překřížení hran, lze předpokládat, že se může jednat o téměř nebo úplně optimální řešení. Algoritmus RtE, jak lze vidět i z vizuální kontroly běhu algoritmu, se těchto překřížení snaží zbavovat a tím se co nejvíce přiblížit k optimálnímu řešení. Při větším počtu vrcholů nějaké překřížení hran vždy zůstává. Existují i jiné varianty tohoto algoritmu, kterými se v této práci nezabýváme pro složitost jejich implementace, které se jistým způsobem dokáží k optimální Hamiltonovské kružnici ještě více přiblížit, ale to je již rozsáhlejší problematika. Ačkoli je tento algoritmus jiného typu než předchozí dva algoritmy, délky výsledných Hamiltonovských kružnic jsou dosti podobné.

3.4. Testování algoritmu Inserting Vertices

Tento algoritmus byl testován dvěma způsoby:

1. Rozšiřování od nejkratší hrany (jako první byla do řešení přidána nejkratší hrana a od té se pak dále rozšiřovalo).
2. Rozšiřování od nejdelší hrany (jako první byla do řešení přidána nejdelší hrana a od té se pak dále rozšiřovalo).

Mezi těmito způsoby lze přepnout pouze malou modifikací zdrojového kódu (převrácení znaménka). Vyšlo ale najevo, po rozšiřování od nejdelší hrany byly výstupní Hamiltonovské kružnice značně delší, než při rozšiřování od nejkratší hrany, proto je zde uvedena pouze jedna tabulka, která ukazuje testování prvního způsobu startu algoritmu.

Počet vrcholů	**Nejkratší cesta**	**Algoritmus**	**Čas [ms]**
5	4 395,1472	4 509,6488	0,04
10	5 823,36924	6 921,8809	0,13
15	x	8 563,0033	0,33
20	x	9 940,6006	0,7
50	x	15 502,0935	10
100	x	21 610,9048	64
250	x	33 657,9856	1 092
500	x	47 124,6691	8 525
1000	x	65 963,98598	67 666
1500	x	80 969,92878	233 231
3000	x	114 005,1702	1 880 732

Tabulka 6. Výkon algoritmu InV na náhodných grafech

Z testování tohoto algoritmu vzešlo najevo, že délkám Hamiltonovských kružnic, nalezenými předchozími algoritmy, jsou délky výstupních kružnic dosti podobné, ne však kratší. To algoritmus dělá spíše méně efektivním algoritmem. Co však je na algoritmu zarážející, je jeho časová složitost, která je oproti časové složitosti předchozích algoritmů velmi značná a pro grafy s větším počtem vrcholů až extrémní. Tento jev je způsoben tím, že tento algoritmus nepřidává hrany na základě nějaké posloupnosti, jako GH a na rozdíl od algoritmu NN nehledá nejbližší vrchol pouze od krajů cesty (koncových vrcholů), ale v každém kroku projde všechny hrany (dvojice vrcholů v cestě) a pro každou z nich hledá právě ten vrchol, který je k cestě nejblíže. Každé přidání hrany do Hamiltonovské kružnice je tedy velmi časově náročné a to se projeví zvlášť u grafů s vysokým počtem

vrcholů. Při implementaci algoritmu nebyl nalezen způsob, jak pracovat pouze s vrcholy grafu podobně jako u algoritmu NN. Je tedy nutná práce s hranami, což podobně jako algoritmus GH, dělá tento algoritmus neefektivní pro práci s extrémně velkým počtem vrcholů, a to z důvodu vysoké paměťové náročnosti. Nebyl tedy shledán žádný aspekt algoritmu, který by ukazoval nějaké výhody v potenciálním praktickém použití.

Z důvodu velké časové náročnosti bylo u grafů o 1000, 1500 a 3000 vrcholech použitý průměrný výsledek pouze z 5 měření a ne z 1000 jako obvykle.

3.5. Testování algoritmu NNaRtE

Počet vrcholů	**Opt. cesta**	**Algoritmus**	**Čas [ms]**
5	4 232,186	4 248,708	3,4
10	5 730,1540	5 812,1541	15
15	x	6 954,7473	15
20	x	7 916,105	16
50	x	12 094,6706	22
100	x	16 742,0506	29
250	x	25 858,5627	123
500	x	36 108,3215	421
1000	x	50 550,3753	1 955
1500	x	61 610,0514	3 796
3000	x	86 619,0603	14 073

Tabulka 7. Výkon algoritmu NNaRtE na náhodných grafech

Měření prokázalo, že použít algoritmus RtE na Hamiltonovskou kružnici utvořenou jiným algoritmem, v našem případě NN, je dobrou volbou. Ke konci běhu samotný algoritmus NN vytváří dlouhé hrany, aby se dostal k dosud nenavštíveným vrcholům. Délky těchto hran pak algoritmus RtE velmi efektivně eliminuje, což se projeví především u grafů s velkým počtem vrcholů. Zde je dále uvedena tabulka, která znázorňuje, o kolik se podařilo algoritmu RtE zkrátit Hamiltonovskou kružnici, kterou vytvořil algoritmus NN.

Počet vrcholů	Zkrácení
5	2,788
10	4 243,3127
15	651,2827
20	970,767
50	1 691,9109
100	2 440,9614
250	3 759,0292
500	5 216,0835
1000	7 166,25442
1500	7 015,98942
3000	13 082,03806

Tabulka 8. Optimalizace řešení od algorimtu NN algoritmem RtE

Velikost zkrácení prokázala, až na malé výkyvy, že tuto optimalizaci je vhodné používat zejména při hledání Hamiltonovské kružnice v grafech s velkým počtem vrcholů, kde dojde po použití algoritmu RtE ke značné optimalizaci výsledného řešení.

3.6. Testování algoritmu MST

Počet vrcholů	Nejkratší cesta	Algoritmus	Čas [ms]
5	4 254,3884	4 449,6293	5
10	5 709,3821	6 611,9244	5
15	x	8 255,6114	5
20	x	9 593,5655	5
50	x	15 136,0495	6
100	x	21 091,2228	8
250	x	33 012,5369	21
500	x	46 276,1103	98
1000	x	65 014,6728	302
1500	x	79 402,1756	748
3000	x	111 398,2987	2 950

Tabulka 9. Výkon algoritmu MST na náhodných grafech

I přes značně složitější implementaci algoritmu MST oproti předchozím algoritmům, nejsou výsledky testování nějak uspokojivější než u předchozích algoritmů a to jak v délce výstupní kružnice, tak v časové složitosti.

Na druhou stranu, oproti předchozím algoritmům existuje důkaz, že aproximační faktor tohoto algoritmu je v nejhorším případě 2. Výsledná kružnice tedy je při jakémkoli vstupu v nejhorším případě maximálně dvakrát delší než kružnice nejkratší. Tato vlastnost dělá algoritmus velmi spolehlivým pro použití v praxi. Méně uspokojivé, jak již bylo zmíněno, je naměřená časová složitost. Tento jev je způsoben značným rozdělením algoritmu do více fází. Vezměme první fázi algoritmu MST, tj. nalezení minimální kostry grafu. Tato část je realizována Kruskalovým algoritmem[12], který je velmi podobný algoritmu GH (to nese i tu nevýhodu, že algoritmus tedy není vhodný z důvodu paměťové náročnosti pro grafy s extrémně velkým počtem vrcholů, jak bylo vysvětleno v kapitole 4.2.). Zatímco algoritmus GH by po této fázi již vrátil optimální řešení, algoritmus MST musí po této fázi procházet graf do hloubky a poté z posloupnosti navštívených vrcholů vymazat duplicity.

3.7. Vzájemné porovnání algoritmů

Nyní, když byly otestovány všechny algoritmy, je nutné výsledky porovnat.

[12]Vzestupné setřídění hran podle velikosti a následné vybírání takových hran, aby nevznikla v grafu kružnice.

Počet vrcholů	Nejkratší HK	Nejkratší čas
5	NNaRtE	RtE
10	NNaRtE	NN
15	NNaRtE	RtE
20	NNaRtE	RtE
50	NNaRtE	GH
100	NNaRtE	GH
250	NNaRtE	GH
500	NNaRtE	GH
1000	NNaRtE	GH
1500	NNaRtE	GH
3000	NNaRtE	GH

Tabulka 10. Porovnání všech algoritmů

Z tabulky je jasně patrné, který algoritmus jednoznačně pro většinu grafů poskytne řešení, které je nejblíže k tomu optimálnímu. Algoritmus NNaRtE je pro řešení TSP ideální volbou a to z několika důvodů:

- Optimalizace již existující cesty.
- Kombinace dvou rozdílných druhů algoritmů.
- Využívání algoritmu s poměrně malou paměťovou náročností.
- Možnost použít pro optimalizovanou cestu i jiný algoritmus (například GH).

Určitě by se daly nalézt ještě další přednosti. Jak již bylo zmíněno dříve, algoritmus RtE, který optimalizuje cestu utvořenou algoritmem NN, je jen jedním z mnoha algoritmů podobného druhu a také nejjednodušší. Tento algoritmus prohazuje vždy jen 2 hrany, složitější algoritmy jich prohazují i více najednou v závislosti na složitějších podmínkách, než pouze zkrácení vzdálenosti. Algoritmus RtE je efektivní sám o sobě a je též velmi efektivní, pokud se použije pouze jako optimalizace již zpracované Hamiltonovské kružnice (jeho efektivnost zůstává stejná a je pouze přičtena k efektivnosti algoritmu NN). Je velmi pravděpodobné, že dosažení ještě optimálnější Hamiltonovské kružnice by bylo dosaženo použitím GH místo NN z důvodu menší časové náročnosti a jeho větší efektivnosti, což je znázorněno v Tabulce 4. Tím se ovšem již v této práci nezabýváme, jelikož i aplikací algoritmu NN bylo dosaženo velmi dobrých výsledků.

Nyní je vhodné uvést i tabulku, která znázorní, který algoritmus byl ze všech použitých algoritmů ten nejméně uspokojivý.

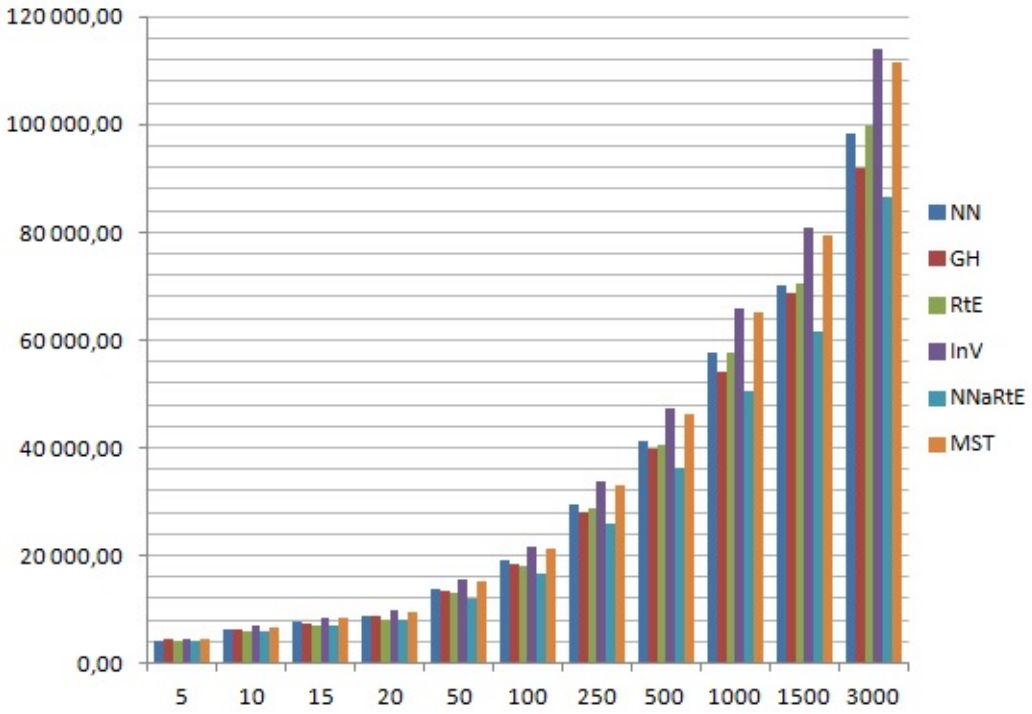
Počet vrcholů	**Nejdelší HK**	**Nejdelší čas**
5	InV	MST
10	InV	MST
15	InV	NNaRtE
20	InV	NNaRtE
50	InV	NNaRtE
100	InV	InV
250	InV	InV
500	InV	InV
1000	InV	InV
1500	InV	InV
3000	InV	InV

Tabulka 11. Porovnání všech algoritmů

Testování, který algoritmus je nejméně vhodný pro řešení TSP ukázalo, že Inserting Ver. není příliš efektivní, jak z hlediska nalezení nejkratší Hamiltonovské kružnice, tak z hlediska časové náročnosti, která je při aplikaci na velké grafy opravdu extrémní. V popisu tohoto algoritmu bylo i podotknuto, že si tento algoritmus, při extrémně velkém počtu vrcholů, dělá velké nároky i na paměť počítače. Je tedy zřejmé, že tento algoritmus se nehodí pro běžné používání na řešení Euklidovského TSP. Jak lze vidět z výsledných hodnot testování algoritmu MST v Tabulce 9, průměrná hodnota všech délek Hamiltonovských kružnic se velmi blíží k délkám kružnic nalezených algoritmem InV. Proto ani algoritmus MST není nejvhodnější volbou k řešení Euklidovského TSP, pokud pro nás není důležitým aspektem aproximační faktor 2.

Podíváme-li se na výsledky testování časových složitostí, je také zjevné, že algoritmus NNaRtE patří k méně rychlým algoritmům z důvodu optimalizace již nalezené cesty algoritmem NN, což se ale projevuje nejvíce v grafech s malým počtem vrcholů. Při velkém počtu vrcholů je časová složitost u algoritmu Inserting Ver. velmi značná, jak již bylo zmíněno dříve.

Pro úplnost je zde uveden graf, ve které jsou znázorněny délky Hamiltonovských kružnic nalezené všemi algoritmy.



Obrázek 9. Graf výkonosti algoritmů

3.8. Aplikace algoritmů na skutečné země

Jak již bylo zmíněno na začátku textu, TSP je velmi diskutovaným problémem v matematických i informatických kruzích, a proto lze na internetu nalézt různé grafy, které lze použít jako vstupy tohoto problémů (mapy skutečných států s městy). Jako zdroj těchto dat byly v této práci použity webové stránky *http://www.math.uwaterloo.ca/tsp/world/countries.html*, kde lze získat pro potřeby měření data 27 států s příslušnými souřadnicemi měst. Pro měření skutečných států bylo vybráno všech 27 dostupných zemí od Západní Sahary s 29 městy po Čínu s 71 009 městy. U měření velkých grafů, jako je například Vietnam s 22 775 městy nebo právě Čína s 71 009 městy, je velmi pravděpodobné, že se u některých algoritmů setkáme s problémem nedostatku paměti. Tato skutečnost je zaznamenána v záznamu o měření *mereni.xlsx*, který je přiložen k práci na CD. V samotném textu práce je znázorněna pouze tabulka s konečnými výsledky měření (viz. níže).

U většiny států je známé i optimální řešení TSP (nebo velmi blízké optimálnímu řešení s danou přesností), které je porovnáno s nejkratší Hamiltonovskou kružnicí nalezenou představenými algoritmy. Z těchto údajů lze tedy získat i aproximační faktor, tj. kolikrát delší je nalezená délka Hamiltonovské kružnice, než je délka optimální. Algoritmy, které zde byly představeny, patří mezi ty jednodušší a intuitivní postupy pro řešení Euklidovského TSP - složitější algoritmy jsou předmětem vyššího výzkumu a pravděpodobně ty byli použity k nalezení optimálního řešení, které je k dispozici na webovém zdroji[13]. Je velmi nepravděpodobné, že by autoři těchto webových stránek dosahovali optimálního řešení pomocí backtrackingu, a to z důvodu jeho velké časové náročnosti.

[13]Odkazuji se především na styl algoritmů, které jsou vysvětleny v knize [2] kapitola 10.2.

V následující tabulce je uveden pouze nejlepší výsledek, který byl dosažen představenými algoritmy. Kompletní výsledky z testování všech algoritmů jsou zaznamenány v dokumentu *mereni.xlsx* na přiloženém CD.

Země (počet měst)	**Nalezeno NNaRtE**	**Opt. délka**	**Apr. faktor**
Argentina (9 152)	1 051 490	837 479	1,25554193
Burna (33 708)	1 054 158	959 304	1,098877937
Čína (71 009)	4 990 995	4 566 563	1,092943424
Džibuti (38)	6 752	6 656	1,014423077
Egypt (7 146)	188 731	172 387	1,094809933
Finsko (10 639)	568 893	520 527	1,09291737
Honduras (14 473)	193 863	177 105	1,094627481
Irsko (8 246)	226 694	206 171	1,099543583
Itálie (16 862)	614 011	557 315	1,101730619
Japonsko (9 847)	544 140	491 924	1,106146478
Kanada (4 663)	1 417 284	1 290 319	1,098398148
Katar (193)	10 395	9 352	1,111526946
Kazachstán (9 976)	1 168 472	1 061 882	1,100378385
Lucembursko (980)	12 428	11 340	1,095943563
Maroko (14 185)	467 651	427 377	1,0942353
Nikaragua (3 496)	104 996	96 132	1,092206549
Omán (1 979)	95 247	86 891	1,096166461
Panama (8 079)	127 461	Neuvedeno	Neuvedeno
Rvanda (1621)	29 024	26 051	1,114122299
Řecko (9 882)	332 451	300 899	1,104859106
Švédsko (24 978)	938 662	855 597	1,097084258
Tanzanie (6 117)	435 580	394 718	1,103522008
Uruguay (734)	86 252	79 114	1,090224233
Vietnam (22 775)	627 302	569 288	1,101906241
Jemen (7 663)	264 270	238 314	1,108915129
Západní Sahara (29)	28 802	27 603	1,043437308
Zimbabwe (929)	103 135	95 345	1,081703288

Tabulka 12. TSP - Aplikace na skutečné země (s daným počtem měst)

Jak již bylo zjištěno v testování algoritmů v případě náhodných grafů, nejoptimálnějšího řešeni ve většině případů dosáhl NNaRtE. V případě zkoumaných zemí, podle předpokladů, dosáhl právě tento algoritmus všech nejoptimálnějšího řešení. Velmi uspokojivý je aproximační faktor dosažených výsledků, jehož průměrná hodnota je $1,1$. Jak je znázorněno v tabulce na straně 44, aproximační faktor je minimální i u grafů s velkým počtem vrcholů, což jej činí velmi efektivním i v praktickém použití. Naopak je nutné podotknout, že algoritmus InV svojí časovou náročností dalece předčil ostatní algoritmy, které zde byly představeny. Pro země s počtem měst přesahující 3000, algoritmus potřeboval pro nalezení kružnice řádově hodiny. Proto ho znovu nedoporučuji pro praktické používání, jelikož ani dosažené řešení není příliš uspokojivé, o čem se lze přesvědčit v přiloženém souboru *mereni.xlsx*.

Závěr

Výzkum algoritmů pro řešení Euklidovského TSP prokázal, že tento problém se dá řešit velmi rozličnými způsoby. Představené algoritmy určitě nejsou všechny, které byli v dosavadní době objeveny, ale i přes jejich poměrně jednoduchou implementaci je jejich funkčnost poměrně spolehlivá, což se projevilo především velmi malým aproximačním faktorem v Tabulce 12.

Z výsledků testování algoritmů lze vyvodit, že pro řešení Euklidovkského TSP v grafech s malým počtem vrcholů je vhodné použít algoritmus založený na Backtracingu, který nalezne vždy to optimální řešení v polynomiálním čase. Při aplikaci na grafy s větším počtem vrcholů je vhodné použít algoritmus, který využívá výhod NN (nebo GH) a RtE, a tím dosáhnout aproximačního faktoru, který je velmi blízký 1. Je tedy vhodné algoritmy rozumným způsobem kombinovat, což by mohlo být předmětem dalšího výzkumu.

Rozhodně není vhodné použít algoritmus InV pro řešení Euklidovského TSP na grafy s velkým počtem vrcholů, kvůli jeho vysoké časové náročnosti a ne příliš uspokojivých dosažených výsledků.

Reference

[1] Skiena, Steven. *The algorithm design manual*, 2nd edition. Springer, 2008.

[2] David P. Williamson, David B. Shmoys. *The design of approximation algorithms*. Cambridge University press, 2011

[3] Pokorná, Petra. *Problém obchodního cestujícího pomocí metody mravenčí kolonie*. Univerzita Pardubice, 2008

[4] Plesník, Jan. *Grafové algoritmy*, Vydavatelstvo Slovenskej akadémie vied. Bratislava, 1983.

[5] Šeda, Miloš. *Teorie grafů*, Vysoké učení technické v Brně. 2003

A. Přiložený testovací program

Zde bude popsán program, který vznikl společně s bakalářskou prací k experimentálnímu porovnání algoritmů.

Program je určen pouze k experimentálním účelům, jak bylo požadováno v zadání. Jeho ovládání je velice intuitivní. Vždy je možné testovat pouze jeden algoritmus. O průběhu spuštěného algoritmu informuje Progress bar v horní části okna (s vyjímkou algoritmu BT - Back tracking). O výsledcích měření informuje InfoBox v pravé horní části programu. Po načtení grafu je možno s grafem pohybovat pomocí kláves *A*, *W*, *D* a *S*. Je zde i možnost zapnout nebo vypnout zobrazování indexů vrcholů. U grafů s menším počtem vrcholů nehraje roli zapnutí této možnosti, avšak při velkých grafech zobrazení indexů vrcholů velmi snižuje rychlost aplikace a proto je zde možnost vypnutí. V záložce *Výpočet* je možnost vypnutí průběžného vykreslování běhu algoritmů, pokud uživatele nezajímá průběh algoritmu ale pouze jeho výsledek. Ve stejné záložce se nachází i možnost přerušení běhu algoritmů (tato funkce nefunguje pro algortimus Backtracking, na což je ale uživatel upozorněn ještě před zahájením běhu algoritmu), a uložení záznamu o měření posledního spuštěného algoritmu do textového souboru.

Vstupní grafy

Graf lze načíst pomocí možnosti *Načíst graf* v záložce *Graf*. Každý vstupní graf je ve formátu textového souboru, kde na prvním řádku je zapsáno celé číslo větší jak nula, které určuje, kolik vrcholů požaduje uživatel načíst. To se prokázalo jako velká výhoda při testování algoritmů na grafy, kdy při načítání grafů s rozdílným počtem vrcholů stačilo přespat pouze toto číslo a nemusely se tvořit nové textové soubory pro další grafy s jiným počtem vrcholů. Na dalším řádku začínají tři sloupce čísel, oddělené mezerou. První sloupec slouží jako obyčejná indexace vrcholů (indexace je od 1). Druhý a třetí sloupec jsou opět celá čísla, znázorňující souřadnice jednotlivých vrcholů, tedy druhý sloupec souřadnice x, třetí sloupec souřadnice y.

K testování algoritmů není nutné vždy načítat textové soubory s grafy, ale je možné použít vestavěnovu funkci pro generování náhodných grafů, která se nachází pod záložkou *Graf*. Tato možnost byla použita při testování algoritmů na náhodné grafy.

B. Obsah přiloženého CD

V samotném závěru práce je uveden stručný popis obsahu přiloženého CD.

`bin/`
: Spouštěcí spubor TSP.exe, který slouží k samtonéu spuštění programu.

`src/`
: Kompletní zdrojové texty programu TSP vytvořené v programu Visual studio 2012.

`data/`
: Testovací data použitá v práci - země.

`text/`
: Samotný text bakalářské práce ve formátu pdf + zdrojové kódy textu bakalářské práce se soubory potřebnými k překladu.

`mereni.xlsx`
: Kompletní záznam o testování algoritmů na skutečné země.