Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

BAKALÁŘSKÁ PRÁCE

Algoritmy pro problém obchodního cestujícího

2024

Kateřina Sáňková

Vedoucí práce:
Mgr. Petr Osička, Ph.D.

Studijní program: Informatika,
Specializace: Programování a vývoj software

Bibliografické údaje

Autor:	Kateřina Sáňková
Název práce:	Algoritmy pro problém obchodního cestujícího
Typ práce:	bakalářská práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2024
Studijní program:	Informatika, Specializace: Programování a vývoj software
Vedoucí práce:	Mgr. Petr Osička, Ph.D.
Počet stran:	45
Přílohy:	elektronická data v úložišti katedry informatiky
Jazyk práce:	český

Bibliographic info

Author:	Kateřina Sáňková
Title:	Algorithms for the travelling salesman problem
Thesis type:	bachelor thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2024
Study program:	Computer Science, Specialization: Programming and Software Development
Supervisor:	Mgr. Petr Osička, Ph.D.
Page count:	45
Supplements:	electronic data in the storage of department of computer science
Thesis language:	Czech

Anotace

Tato bakalářská práce se zabývá problémem obchodního cestujícího a algoritmy používanými k jeho řešení. Z těchto znalostí pak vychází implementace přiložené knihovny. S jejím použitím byly provedeny experimenty a následné porovnání uvedených algoritmů.

Synopsis

This bachelor thesis focuses on the travelling salesman problem and algorithms used to solve it. Based on this knowledge, the attached library is implemented. Experiments were conducted using it, followed by a comparison of the mentioned algorithms.

Klíčová slova: problém obchodního cestujícího; NP problém; heuristika; aproximační faktor

Keywords: travelling salesman problem; NP problem; heuristic; aproximation factor

Ráda bych poděkovala Mgr. Petru Osičkovi za cennou pomoc a lidský přístup při vedení mé bakalářské práce a Lucii Meluzínové za gramatické úpravy.

Odevzdáním tohoto textu jeho autor/ka místopřísežně prohlašuje, že celou práci včetně příloh vypracoval/a samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

Obsah

1 Teorie **7**
- 1.1 Graf . . . 7
  - 1.1.1 Cestování v grafech . . . 7
- 1.2 Algoritmické problémy . . . 9
  - 1.2.1 Složitost . . . 10

2 Problém obchodního cestujícího **11**
- 2.1 Složitost . . . 11
- 2.2 Podproblémy TSP . . . 13
  - 2.2.1 Metrický problém obchodního cestujícího . . . 13
  - 2.2.2 Euklidovský problém obchodního cestujícího . . . 13

3 Algoritmy **14**
- 3.1 Nearest-addition algoritmus . . . 14
  - 3.1.1 Aproximační faktor . . . 15
  - 3.1.2 Těsný příklad . . . 17
- 3.2 Algoritmus double-tree . . . 19
  - 3.2.1 Aproximační faktor . . . 20
  - 3.2.2 Těsný příklad . . . 20
- 3.3 Christofidesův algoritmus . . . 20
  - 3.3.1 Aproximační faktor . . . 22
  - 3.3.2 Těsný příklad . . . 23
- 3.4 Kernighan–Lin algoritmus . . . 24
  - 3.4.1 Kernighan–Lin pro TSP . . . 26

4 Experimenty **32**
- 4.0.1 Výsledky pro náhodně generované grafy . . . 32
- 4.0.2 Výsledky pro vstupy se známými optimálními cestami . . . 37

5 Knihovna **39**
- 5.1 Struktury . . . 39
- 5.2 Algoritmy . . . 39
- 5.3 TSPLib . . . 40
- 5.4 Testovací aplikace . . . 40

Závěr **42**

Conclusions **43**

A Obsah elektronických dat **44**

Literatura **45**

Seznam obrázků

1 Příklad nearest-addition algoritmu . . . 15
2 Vztah okružní cesty v grafu a jeho kostry . . . 16
3 Těsný příklad nearest-addition algoritmu . . . 17
4 Těsný příklad nearest-addition algoritmu – cesty . . . 18
5 Těsný příklad nearest-addition algoritmu – chod algoritmu . . . 18
6 Příklad algoritmu double-tree . . . 19
7 Těsný příklad algoritmu double-tree – minimální kostra . . . 20
8 Příklad Christofidesova algoritmu . . . 22
9 Perfektní párování . . . 23
10 Těsný příklad Christofidesova algoritmu . . . 23
11 Těsný příklad Christofidesova algoritmu – cesta . . . 24
12 Sekvenční hledání hran . . . 27
13 Pár vyhovující podmínce proveditelnosti . . . 28
14 Pár nevyhovující podmínce proveditelnosti . . . 28
15 Problém alternativního x_2 . . . 30
16 Alternativní x_2 – první možnost cesty . . . 30
17 Alternativní x_2 – druhá možnost cesty . . . 31
18 Graf s délkami výsledných cest pro náhodně generované grafy . . . 35
19 Graf s časovými údaji pro náhodně generované grafy . . . 36
20 Ukázka testovací aplikace *TSPTestingApp* . . . 41

Seznam tabulek

1 Test náhodně generovaných grafů – nearest-addition algoritmus . . . 32
2 Test náhodně generovaných grafů – double-tree algoritmus . . . 33
3 Test náhodně generovaných grafů – Christofidesův algoritmus . . . 33
4 Test náhodně generovaných grafů – Kernighan–Lin algoritmus . . . 34
5 Test náhodně generovaných grafů – Kernighan–Lin algoritmus s omezeným backtrackingem . . . 35
6 Test na státy se známými optimálními cestami . . . 37

1 Teorie

Problém obchodního cestujícího (TSP – z anglického *travelling salesman problem*) je úzce spjatý s *teorií grafů* a proto je potřeba si na úvod zavést některé ze základních pojmů. Kapitola čerpá z [1].

1.1 Graf

Graf je jedna ze základních reprezentací prvků množiny objektů a jejich vzájemných propojení. Takovým objektům budeme říkat *vrcholy* (někdy také *uzly*) a propojením *hrany*. Uvažujeme-li navíc i orientaci hran, pak říkáme, že je graf *orientovaný*, jinak *neorientovaný*.

Definice 1 (Neorientovaný graf)
Neorientovaný graf je dvojice $\langle V, E\rangle$, kde V je neprázdná množina vrcholů a $E \subseteq \{\{u, v\} \mid u, v \in V,\ u \neq v\}$ je množina (*neorientovaných*) hran.
u, v nazýváme *koncové uzly* hrany u, v.

V některých situacích nás bude zajímat počet hran, kterým je jistý uzel koncovým. Tomuto číslu pro uzel u budeme říkat *stupeň vrcholu* u a budeme ho značit $deg(u)$. Důležité bude také následující tvrzení.

Věta 2
V každém grafu $G = \langle V, E\rangle$ *platí, že* $\sum_{v \in V} deg(v) = 2|E|$.

U problému obchodního cestujícího také chceme, aby hrany vstupních grafů měly nějakou váhu. Tu jim přiřazuje tzv. *hranové ohodnocení*, funkce

$$w : E \to D$$

kde D je množina hodnot, kterými hrany ohodnocujeme. V případě problému obchodního cestujícího se jedná o množinu reálných (nebo racionálních) čísel.

Dále požadujeme, aby vstupní graf TSP byl *úplný*. To znamená, že každé dva jeho vrcholy jsou spojeny hranou.

1.1.1 Cestování v grafech

Důležitou oblastí práce s grafy je cestování v nich. Vychází se z toho, že se z jednoho uzlu můžeme přemístit k druhému, právě když mezi nimi existuje hrana (v případě orientovaných grafů musí mít hrana ještě správný směr). Jednou z úloh o cestování je právě problém obchodního cestujícího.
Základním pojmem, od kterého budeme odvozovat další, je *sled*.

Definice 3
Sledem v grafu $G = \langle V, E\rangle$ rozumíme posloupnost $v_0, e_1, v_1, e_2, v_2, \cdots, e_n, v_n$,

kde $e_i = \{v_{i-1}, v_i\}$, pro každé $i \in \{0, \cdots, n\}$ máme $v_i \in V$ a pro každé $j \in \{1, \cdots, n\}$ máme $e_j \in E$.

Sled nazýváme

- *uzavřený*, pokud $v_0 = v_n$;
- *tah*, pokud pro každé $k, l \in \{1, \cdots, n\}$, kde $k \neq l$, platí $e_k \neq e_l$ (neopakují se hrany);
- *cesta*, pokud pro každé $k, l \in \{0, \cdots, n\}$, kde $k \neq l$, platí $v_k \neq v_l$ (neopakují se vrcholy);
- *kružnice*, pokud v definici cesty požadujeme $v_0 = v_n$.

Definice 4 (Souvislý graf)
Neorientovaný graf $G = \langle V, E \rangle$ nazýváme *souvislý*, pokud pro každé $u, v \in V$ existuje sled z u do v.

Speciální druh tahu je tzv. *eulerovský tah*. Pro něj platí, že vede přes všechny vrcholy a každá hrana se v něm vyskytuje právě jednou.

Věta 5
Pokud je neorientovaný graf souvislý a všechny jeho vrcholy mají sudý stupeň, pak v něm existuje uzavřený eulerovský tah.

Pro TSP (*travelling salesman problem*) je klíčový ještě termín *hamiltonovská kružnice*. Tou rozumíme takovou kružnici v grafu, která vede přes všechny jeho vrcholy.

Některé z algoritmů v knihovně jsou založené na hledání tzv. *minimální kostry grafu*. Před zavedením tohoto pojmu je ještě nutné si definovat, co je to *podgraf* grafu.

Definice 6 (Podgraf)
Graf $G_2 = \langle V_2, E_2 \rangle$ nazýváme *podgraf* grafu $G = \langle V, E \rangle$, právě když $V_2 \subseteq V$ a $E_2 \subseteq E$.

Definice 7 (Kostra grafu)
Kostra neorientovaného grafu je jeho souvislý podgraf, který obsahuje všechny jeho vrcholy a v němž se nevyskytují žádné kružnice.

Pokud mají hrany původního grafu přiřazené váhy příslušným hranovým ohodnocením w, potom můžeme uvažovat o *minimální kostře grafu*. Tou budeme rozumět právě takovou kostru $MSP = \langle V, E' \rangle$, která bude mít mezi ostatními minimální součet $\sum_{e \in E'} w(e)$.

1.2 Algoritmické problémy

Tato podkapitola vychází z [5].

Obecně lze problémy, u kterých je rozumné chtít pro řešení použít počítač, definovat pomocí množiny vstupů In, množiny možných výstupů Out a funkce $p : In \to Out$, která každému vstupu přiřazuje odpovídající výstup. Tedy $P = \langle In, Out, p \rangle$.

Optimalizační problémy jsou takové problémy, kdy pro vstup existuje víc možných řešení a jejich úlohou je mezi nimi najít to, které má mezi ostatními minimální nebo maximální hodnotu předem definované funkce.

Tyto problémy se dají charakterizovat

- množinou vstupů In;
- funkcí $sol : In \to 2^{Out}$, která každému vstupu přiřadí množinu *přípustných řešení*;
- funkcí $cost : In \times Out \to \mathbb{Q}$, která vstupu a jeho přípustnému řešení přiřazuje *cenu* tohoto řešení;
- hodnotou *goal*, které je buď min, nebo max.

Podle hodnoty *goal* se problému říká *minimalizační* nebo *maximalizační*.

Optimálním řešením pro vstup $x \in In$ pak označujeme takové řešení $y \in sol(x)$, pro které platí, že $cost(x, y) = goal\{cost(x, y') \mid y' \in sol(x)\}$. Cenu takového řešení značíme $OPT(x)$.

Při hledání algoritmu, který řeší takový problém zvažujeme jeho *správnost* a *optimalitu*. Řekneme, že algoritmus A pro problém P je *správný*, pokud pro každé $x \in In$ platí, $A(x) \in sol(x)$. *Optimální* je navíc pokud pro každé $x \in In$ je $A(x)$ optimální řešení.

Některé optimální algoritmy ale bývají časově náročné, často se proto hledají algoritmy, které nevydají pokaždé optimální řešení, ale pouze přibližné. Těmto algoritmům se říká *aproximační*.

Mějme takový algoritmus A. Pro každý vstup $x \in In$ můžeme definovat *aproximační faktor* jako

$$R_A(x) = \max\{\frac{cost(x, A(x))}{OPT(x)}, \frac{OPT(x)}{cost(x, A(x)}\}$$

Pro samotný algoritmus můžeme také definovat jeho aproximační faktor, vzhledem k nějaké funkci F mapující vstupy na přirozená čísla (typicky velikost vstupu, případně jiná jeho významná vlastnost), a to následovně

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

A můžeme označit za $f(n)$–aproximační algoritmus, pokud pro každé $n \in \mathbb{N}$ platí $R_A(n) \leq f(n)$. Jinými slovy, pokud u algoritmu známe jeho aproximační faktor, pak se můžeme spolehnout, že pro každý vstup velikosti n dostaneme

v nejhorším případě $f(n)$-krát horší výsledek vůči optimu.

Další významnou množinou problémů jsou *rozhodovací problémy*. Ty se vyznačují tím, že množina výstupů je omezena jen na Ano a Ne.

Každý optimalizační problém má svoji *rozhodovací verzi*. Místo toho, aby na vstupu byla pouze instance problému $x \in In$, přidá se navíc ještě číslo $k \in \mathbb{Q}$. Pokud je $k \geq OPT(x)$ u minimalizačního problému nebo $k \leq OPT(x)$ u maximalizačního, pak je výsledkem Ano, jinak Ne. [2]

1.2.1 Složitost

Tato část vychází z [2]. Na začátku je třeba si zavést třídy **P** a **NP**. Rozhodovací problémy, které spadají do třídy **P**, jsou řešitelné deterministickými stroji v polynomiálním čase a bereme je za prakticky zvládnutelné. **NP** problémy jsou také řešitelné v polynomiálním čase, ale nedeterministickými stroji. U těchto strojů předpokládáme, že z možných kroků provedou vždy ten žádaný, a dostane se tak ke správnému výsledku bez nutnosti zkoušet všechny možnosti. Jistě platí $\mathbf{P} \subseteq \mathbf{NP}$, jelikož deterministické stroje jsou speciálním případem nedeterministických, které pouze nemají možnost nedeterministického výběru.

Vysvětlíme si nyní termín *polynomiální redukovatelnost*. Říkáme, že problém P_1 je polynomiálně redukovatelný na P_2, pokud pro každý vstup A problému P_1 můžeme v polynomiálním čase najít vstup pro P_2, pro který P_2 vydá stejnou odpověď, jakou by vydal P_1 pro A.

Některé rozhodovací problémy mají tu vlastnost, že jsou na ně polynomiálně redukovatelné všechny rozhodovací problémy v **NP**. Problémům s touto vlastností se říká ***NP**-těžké*. Pokud navíc **NP**-těžký problém patří do NP, říkáme mu ***NP**-úplný*.

NP-těžké problémy jsou důležité, protože kdyby se podařilo najít algoritmus řešící kterýkoli z nich v polynomiálním čase, pak bychom jistě zvládli vyřešit i všechny **NP** problémy v polynomiálním čase, tedy $\mathbf{NP} \subseteq \mathbf{P}$ a z toho $\mathbf{NP} = \mathbf{P}$. Naopak, pokud bychom zjistili, že $\mathbf{NP} \neq \mathbf{P}$, pak pro žádný **NP**-úplný problém jistě nemůže existovat polynomiální algoritmus, který ho řeší.

Z $\mathbf{P} \neq \mathbf{NP}$ lze vyvodit ještě jedno důležité tvrzení. Nechť P_O je optimalizační problém a P_D je jeho rozhodovací verze. Pokud dokážeme, že P_D je **NP**-těžký, pak pro P_O neexistuje optimální polynomický algoritmus. Kdyby totiž takový algoritmus existoval, využili bychom ho k nalezení optimálního řešení instance x problému P_O. Pak bychom cenu tohoto řešení porovnaly s číslem k a rozhodli o tom, jestli je odpověď pro instanci (x, k) problému P_D Ano nebo Ne. Tímto bychom tedy získali polynomiální algoritmus řešící také P_D. Jenže protože P_D je **NP**-těžký, všechny **NP** problémy jsou na něj polynomiálně redukovatelné a zvládli bychom je tedy i v polynomiálním čase řešit, tudíž $\mathbf{P} = \mathbf{NP}$, což je spor.

2 Problém obchodního cestujícího

Kapitola primárně vychází z [3], definice pro probíraný problém pak z [2]. Problém obchodního cestujícího je jedním z nejstudovanějších optimalizačních problémů. Podstatou nalézt je mezi zadanými městy cestu, která začíná i končí ve stejném místě a zbytek měst navštíví právě jedenkrát. Jelikož seznam měst lze chápat jako množinu vrcholů grafu, můžeme o TSP uvažovat jako o úloze o cestování v grafu. Cestou, již hledáme, je pak *hamiltonovská kružnice*.

TSP definujeme následovně:

Definice 8 (Problém obchodního cestujícího)

NÁZEV: TSP

VSTUP: Úplný neorientovaný graf $G = \langle V, E \rangle$,

hranové ohodnocení $c : E \to R^+$

VÝSTUP: Hamiltonovská kružnice v G minimální délky

Ceny hran budeme nazývat také *délky* a pro jednoduchost budeme místo $c(\{u, v\})$ psát $c_{u,v}$.

2.1 Složitost

Zavedeme si nyní rozhodovací verzi k TSP.

Definice 9 (Rozhodovací verze TSP)

NÁZEV: TSP

VSTUP: Úplný neorientovaný graf $G = \langle V, E \rangle$,

hranové ohodnocení $c : E \leftarrow R^+$,
číslo k

OTÁZKA Existuje v G hamiltonovská kružnice délky nejvýš k?

Jedním z důvodů, proč je TSP tak zkoumaným problémem, je jeho složitost. Rozhodovací verze TSP je jistě v **NP**. Nedeterministický stroj, který by měl za úkol najít řešení pro (x, k), by zkrátka nedeterministicky vybral množinu $|V|$ hran z E a potom by ověřil, že vybraná množina hran je hamiltonovská kružnice a že má cenu větší než k. To by samozřejmě zvládl provést v polynomiálním čase.

Ví se, že rozhodovací verze TSP je **NP**-úplná. Tedy platí, že pokud nalezneme polynomiální algoritmus řešící TSP, pak dokážeme $\mathbf{P} = \mathbf{NP}$.

Nyní si dokážeme ještě zajímavější tvrzení.

Věta 10
Pokud by existoval 2^n-aproximační polynomiální algoritmus pro TSP, pak $\boldsymbol{P = NP}$.

Něž začneme s důkazem, musíme ji ještě zavést rozhodovací problém nalezení hamiltonovské kružnice (*HC - Hamiltonian Cycle*) v grafu.

Definice 11 (HC)

NÁZEV: HC

VSTUP: Neorientovaný graf $G = \langle V, E \rangle$

OTÁZKA Existuje v G hamiltonovská kružnice?

O tomto problému je známo, že je také **NP**-úplný. Navíc se dá snadno převést na TSP.

Jak již bylo zmíněno, TSP má ve své klasické podobě na vstupu graf a jeho příslušné hranové ohodnocení. Vstupní graf $x = \langle V, E \rangle$ pro HC bychom mohli redukovat na vstup $r(x)$ pro TSP tak, že bychom vytvořili úplný graf mezi uzly z původní instance a ohodnotili bychom jeho hrany následovně:

$$c_e = \begin{cases} 1 & \text{pro } e \in E \\ |V| \cdot 2^{|V|} + 1 & \text{pro } e \notin E \end{cases}$$

Snadno můžeme dojít k tomu, že

$$OPT_{TSP}(r(x)) = \begin{cases} = |V| & \text{pro } x \in HC \\ > |V| \cdot 2^{|V|} & \text{pro } x \notin HC \end{cases}$$

Uvědomme si, že hrany s cenou $|V| \cdot 2^{|V|}$ budou obsaženy v optimálním řešení pouze v případě, kdy nelze sestavit hamiltonovskou kružnici z hran původního grafu. Pokud musela být vybrána pouze jedna taková hrana, pak bude cena nalezené cesty

$$(|V| - 1) + (|V| \cdot 2^{|V|} + 1) > |V| \cdot 2^{|V|}$$

Díky takovému ohodnocení tedy existuje exponenciálně velká mezera mezi pozitivními a negativními instancemi HC. Pokud bychom měli 2^n-aproximační polynomiální algoritmus A pro TSP, pak by pro pozitivní instance HC našel cestu délky nejhůře $|V| \cdot 2^{|V|}$, pro negativní instance zase nejlépe délky $(2^{|V|} + 1) \cdot |V|$.

HC bychom mohli vyřešit tak, že bychom spustili A na $r(x)$, a pokud by délka vrácené cesty byla menší nebo rovna $|V| \cdot 2^{|V|}$, pak bychom vrátili ANO, v opačném případě NE. Vzhledem k tomu, že převod na $r(x)$, A i porovnání na konci běží v polynomiálním čase, pak bychom celý HC mohli rozhodnout v polynomiálním čase. Protože HC je NP-úplný, znamenalo by to, jak jsme diskutovali v kapitole 1.2.1, že P = NP.

2.2 Podproblémy TSP

Z předchozí podkapitoly vyplývá, že TSP je v obecném případě těžké v polynomickém čase i aproximovat. Přidáním jistých omezení na problém můžeme tuto obtížnost výrazně snížit a často nám takové typy problémů pro reálné použití dostačují.

2.2.1 Metrický problém obchodního cestujícího

Prvním takovým omezením je, že pro délky hran musí platit trojúhelníková nerovnost. Tedy pro každé $u, v, w \in V$ platí

$$c_{u,v} + c_{v,w} \geq c_{w,u}$$

Takovému problému se říká *metrický TSP*.

2.2.2 Euklidovský problém obchodního cestujícího

Přísnější podmínkou pak může být definice délky jako euklidovské vzdálenosti. Ta je pro vrcholy $u, v \in V$ se souřadnicemi $u = (u_1, u_2, \cdots, u_n)$, $v = (v_1, v_2, \cdots, v_n)$ definována

$$c_{u,v} = \sqrt{\sum_{i=1}^{n} (u_i - v_i)^2}$$

Jelikož při takovém ohodnocení bude jistě platiti i trojúhelníková nerovnost, pak je euklidovský TSP podproblémem metrického TSP.

3 Algoritmy

Pro TSP není známý žádný optimální polynomický algoritmus a platí-li $\mathbf{P} \neq \mathbf{NP}$, pak ani takový algoritmus existovat nemůže. Z tohoto důvodu se pro hledání řešení používají tzv. *heuristiky*. U nich se sice nemůžeme spolehnout, že najdeme optimální řešení, ale jsme schopni nalézt přibližné řešení v reálném čase.

U některých heuristik se navíc můžeme bavit o *aproximačním faktoru*. Připomeňme, že pro libovolný vstup problému I, pro nějž je cena optimálního řešení OPT, a heuristiku s aproximačním faktorem α platí, že se můžeme spolehnout, že cena řešení, které dostaneme použitím dané heuristiky na I, bude přinejhorším $\alpha * OPT$.

První tři algoritmy jsou popsány v [3], poslední pak v článku [4].

3.1 Nearest-addition algoritmus

Tento algoritmus spadá do skupiny hladových algoritmů a je poměrně intuitivní. Vychází z myšlenky, že k vytvářené cestě vždy přidáme nejbližší nenavštívené město. Nejprve najdeme taková dvě města i, j, jež jsou si v grafu nejblíže a vytvoříme cestu *path*, která povede z i do j a zpět. Následuje cyklus, v němž budeme hledat takovou hranu (k, l), že $k \in path$, $l \notin path$ a c_{kl} je všemi takovými hranami minimální. Předpokládejme, že město m je aktuálně v *path* za k. Vložíme l do *path* mezi k a m a následuje další iterace. Cyklus skončí v momentu, kdy jsme navštívili všechna města.

Algorithm 1: Nearest-addition algoritmus

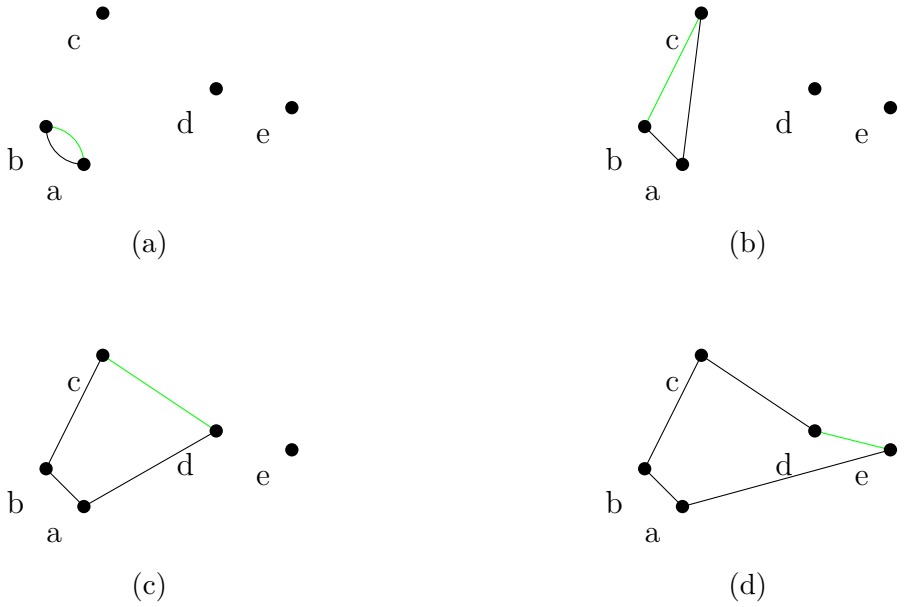
```
new list path;
new list unvisited;
unvisited ← nodes;
(i, j) ← shortest edge in graph;
path.Add(i, j, i);
unvisited.Remove(i, j);
while unvisited.Count > 0 do
    (k, l) ← shortest edge in graph, where k ∈ path and l ∈ unvisited;
    path.AddAfter(k, l);
    unvisited.Remove(l);
return path
```

Procedury *Add*, *AddAfter* a *Remove* použité v kódu pracují se seznamy, a to tak, že *list.Add(a, b, ...)* přidá na konec seznamu *list* všechny předané argumenty v pořadí od prvního k poslednímu, *list.AddAfter(a, b)* najde, kde se v *list* nachází prvek a a vloží za něj b a *list.Remove(a)* *list* první výskyt a.

Ukažme si jednoduchý příklad chodu algoritmu. Na obrázku níže jsou zobrazeny vrcholy vstupního grafu. Za délky hran mezi nimi uvažujme euklidovskou

vzdálenost. Dále je na každém obrázku ukázaná aktuální cesta *path* a zelená hrana reprezentuje poslední nalezenou nejkratší hranu.

Příklad 12



Obrázek 1: Příklad nearest-addition algoritmu

3.1.1 Aproximační faktor

Budeme chtít dokázat, že nearest-addition je 2-aproximační algoritmus. Proto si musíme nejdříve ukázat vztah k Primově algoritmu, který hledá minimální kostru grafu.

Algorithm 2: Primův algoritmus

new list $spanningTree$;
new list $unvisited$;
$unvisited \leftarrow nodes$;
$(i, j) \leftarrow$ shortest edge in graph;
$spanningTree.Add((i, j))$;
$unvisited.Remove(i, j)$;
while $unvisited.Count > 0$ **do**
    $(k, l) \leftarrow$ shortest edge in graph, where $k \notin unvisited$ and $l \in unvisited$;
    $spanningTree.Add((k, l))$;
    $unvisited.Remove(l)$;
return $spanningTree$

Ze pseudokódu je jasné, že hrany nalezené Primovým algoritmem i nearest addition algoritmem budou totožné.

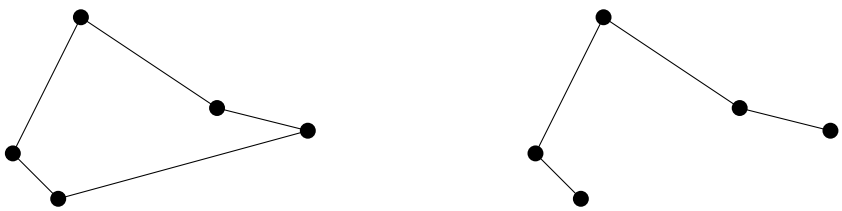
Zůstaňme ještě chvíli u kostry grafu a jejím vztahu k cestě nalezené TSP algoritmy.

Lemma 13

Pro libovolný vstup TSP je cena jeho optimální cesty alespoň tak vysoká, jako je cena minimální kostry pro tentýž vstup.

Důkaz

Vezměme optimální cestu pro libovolný vstup s více než jedním vrcholem a odstraňme některou z hran. Takto vzniklý graf jistě nebude mít vyšší cenu než optimální cesta. Navíc bude také kostrou původního vstupu. To si můžeme snadno ověřit. Graf stále obsahuje všechny vrcholy, jeho souvislost jsme nijak neporušili a jedinou kružnici, kterou obsahoval, jsme otevřeli odstraněním jedné z hran.



Obrázek 2: Vztah okružní cesty v grafu a jeho kostry

To, že tato kostra nemůže mít nižší cenu než minimální kostra, je triviální a lemma jsme dokázali. □

Věta 14

Nearest-addition algoritmus pro metrický problém obchodního cestujícího je 2-aproximační algoritmus.

Důkaz

Uvažujme podmnožiny $S_1, S_2, \ldots, S_{n-1}$ hran grafu, kde S_i je množina hran identifikovaných i-tou iterací. Vrátíme-li se k příkladu chodu algoritmu, pak ve stavu na obrázku 1a bude $S_1 = \{(a, b)\}$, pro stav na 1b bude $S_2 = \{(a, b), (b, c)\}$, atd. Dále mějme množinu $F = \{(i_1, j_1), (i_2, j_2), \cdots, (i_{n-1}, j_{n-1})\}$ reprezentující nejkratší hrany určené každým průchodem cyklu (na obrázku zvýrazněné zeleně). Jak bylo zmíněno výše, tato množina bude jistě množinou hran nejkratší kostry vstupního grafu. Tedy z lemma 13 plyne

$$OPT \geq \sum_{l=1}^{n-1} c_{i_l j_l}$$

Po nalezení první nejkratší hrany máme výslednou délku $c_1 = 2c_{i_1,j_1}$, jelikož ji vkládáme do cesty dvakrát. Dále, když přidáváme nové město j mezi i a k, rušíme hranu (i, k) a přidáváme hrany (i, j) a (j, k). Cesta se tedy prodlužuje o $c_{ij} + c_{jk} - c_{ik}$. Z trojúhelníkové nerovnosti snadno odvodíme, že $c_{jk} - c_{ik} \leq c_{ij}$. Rozdíl cen hran c_{jk} a c_{ik} je tedy shora ohraničený cenou c_{ij}.

Dosadíme-li tuto hodnotu do předchozí rovnice, zjistíme, že celková cesta se může v nejhorším případě prodloužit o $2c_{ij}$. Celková cesta nalezená nearest-addition algoritmem bude tedy maximálně

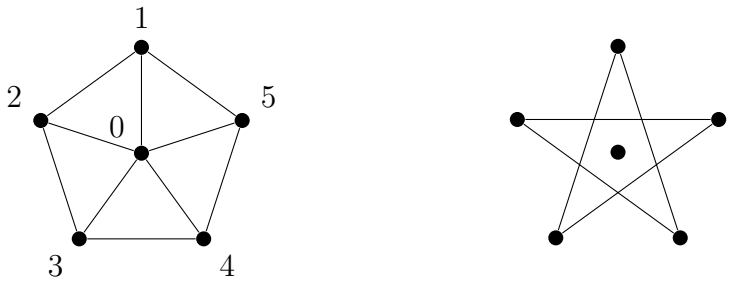
$$2 \sum_{l=1}^{n-1} c_{i_l j_l}$$

Upravíme li poslední nerovnost, dostaneme $\sum_{l=2}^{n} c_{i_l j_l} \leq 2OPT$.

3.1.2 Těsný příklad

Můžeme si ukázat jeho tzv. *těsný příklad* (tight example), což je pro algoritmus s aproximačním faktorem α příklad, u kterého nalezne řešení s cenou $\alpha * OPT$ (můžeme povolit určitou konstantní odchylku). Tedy budeme chtít najít vstup pro metrický TSP, pro nějž nám tento algoritmus vrátí cestu s délkou skoro $2 * OPT$.

V [6] je popsán právě takový vstup. Uvažujme $(n-1)$-cípou hvězdu s jedním vrcholem ve svém středu. Hrany budou ohodnoceny následovně: hrany vlevo na obrázku budou mít cenu jedna a hrany vpravo dva.

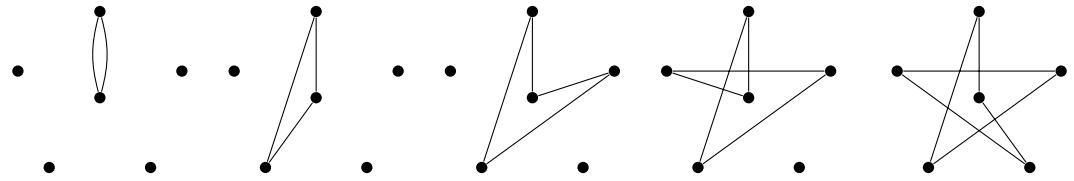


Obrázek 3: Těsný příklad nearest-addition algoritmu

Cena hran napravo je shora omezená právě dva, protože musí platit trojúhelníková nerovnost.

Obrázek 4: Těsný příklad nearest-addition algoritmu – cesty

Je snadno vidět, že optimální cesta (na obrázku výše vlevo) bude mít celkovou cenu n. Algoritmem ale může být nalezena i cesta jako na obrázku nahoře vpravo (příklad chodu algoritmu je zobrazen níže). Taková cesta obsahuje $n-2$ hran s vahou dva a dvě hrany s vahou jedna, tedy $2(n-2)+2 = 2n-2 = 2*OPT-O(1)$.



Obrázek 5: Těsný příklad nearest-addition algoritmu – chod algoritmu

3.2 Algoritmus double-tree

Jádrem algoritmu je zdvojení hran minimální kostry vstupního grafu a následné nalezení uzavřeného eulerovského tahu. To, že se v takovém grafu eulerovský tah nachází, je snadné dokázat. Z Věty 5 víme, že stačí, aby byl graf spojitý a všechny jeho uzly měly sudý stupeň. Zdvojíme-li každou hranu, vycházející z libovolného uzlu, pak uzel jistě bude mít sudý stupeň. Jelikož vycházíme z kostry grafu, spojitost je zřejmá.

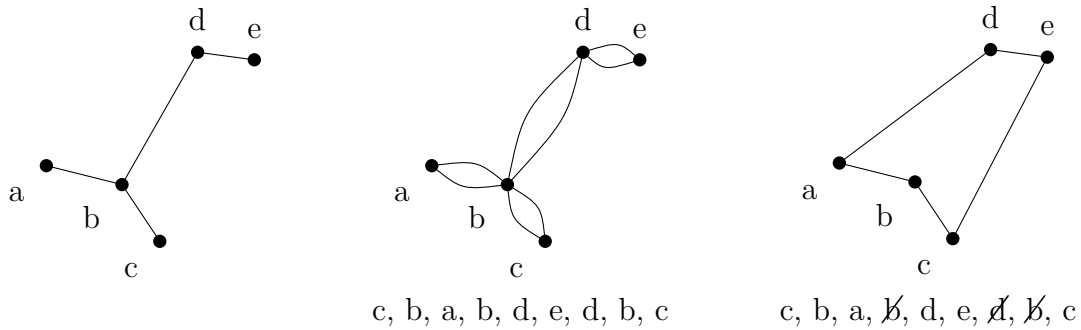
K vytvoření cesty už potřebujeme pouze zaručit, že nenavštívíme žádný uzel vícekrát. Toho dosáhneme pomocí zkracování. Máme-li eulerovský průchod $(i_0, i_1),\ (i_1, i_2), \cdots, (i_{k-1}, i_k),\ (i_k, i_0)$, pak vezměme posloupnost $i_0, i_1, \cdots, i_k$ a ponechejme pro každý uzel pouze jeho první výskyt. Nakonec ještě vraťme i_0 na konec cesty.

Algorithm 3: Double-tree algoritmus

```
Input: G
doubleTree ← FindMinimalSpanningTree(G);
foreach edge ∈ doubleTree do
    doubleTree.Add(edge);
eulerCircuit ← FindEulerCircuit(doubleTree);
create list path;
foreach node ∈ eulerCircuit do
    if node ∉ path then
        path.Add(node);
return path
```

Na následujícím obrázku je vidět jednoduchý chod algoritmu. Vlevo je nalezená minimální kostra, uprostřed pak graf se zdvojenými hranami a svým eulerovským průchodem, vpravo pak nakonec výsledná cesta vzniklá zkracováním.

Příklad 15



Obrázek 6: Příklad algoritmu double-tree

3.2.1 Aproximační faktor

Věta 16
Double-tree je 2-aproximační algoritmus.

Z Lemma 13 víme, že optimální cesta stojí alespoň tolik, co minimální kostra. Jelikož každou její hranu zdvojujeme, získáme dvakrát tak delší cestu.

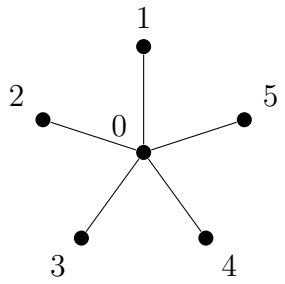
Při hledání eulerovského průchodu hrany pouze seřazujeme.

Zbývá už jen zkracování. Zde opět vycházíme z trojúhelníkové nerovnosti. Uvažujme, že i, j, k jsou po sobě jdoucí města a j jsme již navštívili, měli bychom je tedy přeskočit. Je zřejmé, že $c_{ij} + c_{jk} \geq c_{ik}$, což dokazuje, že cena tohoto úseku zůstane přinejhorším stejná, a horní ohraničení se tedy nemění.

3.2.2 Těsný příklad

Jako příklad vstupu, pro který double-tree algoritmus najde cestu délky $2OPT$, můžeme vzít vstup z kapitoly 3.1.2 o nearest-addition algoritmu. Jelikož oba algoritmy mají stejný aproximační faktor, stačí ukázat, že se double-tree algoritmem můžeme dostat ke stejné cestě jako v již zmíněné předchozí kapitole.

U takového vstupu může být nalezena následující minimální kostra.



Obrázek 7: Těsný příklad algoritmu double-tree – minimální kostra

V grafu se zdvojenými hranami, pak můžeme uvažovat eulerovský průchod 0, 4, 0, 2, 0, 5, 0, 3, 0, 1, 0, ze kterého zkrácením dostaneme cestu 0, 4, 2, 5, 3, 1, 0, což je stejná cesta jakou jsme nalezli algoritmem nearest-addition.

3.3 Christofidesův algoritmus

Christofidesův algoritmus je velice podobný double-tree algoritmu. Opět začneme s minimální kostrou vstupního grafu. Existenci eulerovského průchodu ale zajistíme o něco chytřeji. Připomeňme si, že eulerovský průchod se v grafu nachází právě tehdy, když mají všechny jeho uzly sudý stupeň. V double-tree algoritmu jsme tuto podmínku splnili tak, že jsme zdvojili každou hranu kostry. To je ovšem redundantní. Postačí, když spárujeme takové její uzly, které mají lichý stupeň.

Mějme úplný graf, jehož množinou vrcholů budou právě ty vrcholy jeho minimální kostry s lichým stupněm, označme ji O. Z množiny jeho hran se budeme snažit vybrat takovou podmnožinu, ve níž bude pro každé $v \in O$ právě jedna hrana, pro kterou je v koncový uzel. Taková množina se nazývá *perfektní párování*. V grafu jich samozřejmě může být více. Pro naše účely bude jistě nejlepší hledat takové z nich, jež budou mít minimální součet délek všech svých hran, tedy *minimální perfektní párování*. K jeho nalezení se v programu používá C++ knihovna Vladimira Kolmogorova, která implementuje *Blossom V* algoritmus. Ve zdrojovém kódu knihovny byly provedeny mírné změny, aby nebylo nutné předávat funkci soubor, ale přímo pole s uzly.

Uvědomme si ale, že aby takové párování pro O existovalo, musí být $|O|$ sudé. Víme, že součet stupňů všech uzlů v neorientovaném grafu je sudé číslo (viz Věta 2). Označme si množinu všech vrcholů grafu V a množinu vrcholů se sudým stupněm E. Jistě platí

$$\sum_{v \in V} deg(v) = \sum_{v \in E} deg(v) + \sum_{v \in O} deg(v)$$

$\sum_{v \in V} deg(v)$ je ale určitě sudá a to samé platí pro $\sum_{v \in E} deg(v)$, tedy $\sum_{v \in O} deg(v)$ musí být také sudá. Vzhledem k tomu, že každý vrchol přispívá do součtu lichým číslem, pak musí být sudý jejich počet.

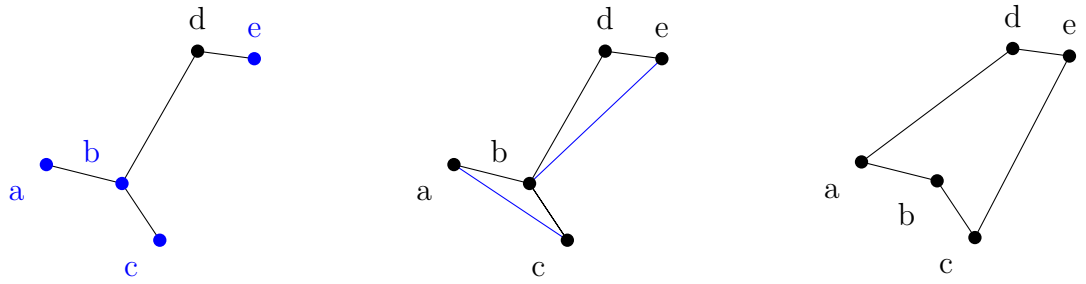
Přidáním minimálního perfektního párování k minimální kostře zvýšíme stupeň každého vrcholu s lichým stupněm právě o jedna, tudíž zajistíme, že všechny uzly budou mít sudý stupeň. V takovém grafu pak existuje uzavřený eulerovský tah. Po jeho nalezení už jen zkracujeme cestu stejným způsobem jako u double-tree algoritmu.

Algorithm 4: Christofidesův algoritmus

```
Input: G
spanningTree ← FindMinimalSpanningTree(G);
oddDegreeNodes ← FindNodesWithOddDegrees(spanningTree);
perfectMathing ← FindPerfectMatching(oddDegreeNodes);
eulerCircuit ← FindEulerCircuit(doubleTree + perfectMathing);
create list path;
foreach node ∈ eulerCircuit do
    if node ∉ path then
        path.Add(node);
return path
```

Podobně jako u algoritmu double-tree si ukážeme jednoduchý příklad. Vlevo je opět vidět minimální kostra, kde jsou modrou barvou zobrazeny vrcholy s lichým stupněm. Na dalším obrázku je k hranám přidáno minimální párování modrých vrcholů s eulerovským průchodem takového grafu. Nakonec je pak opět vidět výsledná zkrácená cesta.

Příklad 17



Obrázek 8: Příklad Christofidesova algoritmu

3.3.1 Aproximační faktor

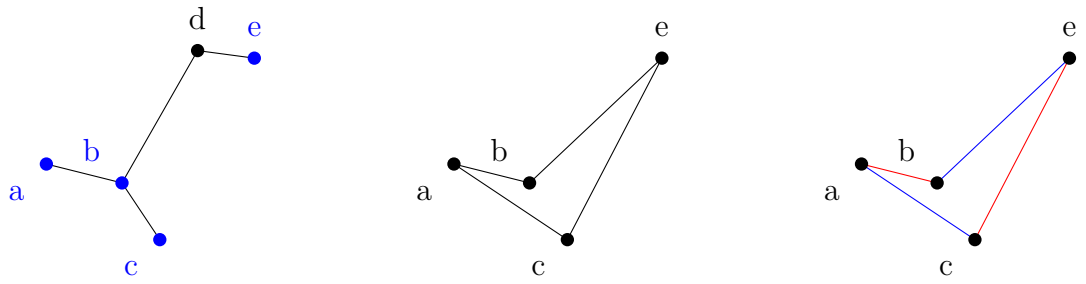
Věta 18

Christofidesův algoritmus je 3/2-aproximační algoritmus.

Nejdříve si musíme rozmyslet, co vlastně chceme dokázat. Potřebujeme, aby eulerovský průchod měl přinejhorším délku $\frac{3}{2}OPT$. Z Lemmatu 13 víme, že minimální kostra, která je v něm obsažená, má v nejhorším případě cenu OPT. Stačí tedy dokázat, že perfektní párování má maximálně cenu $\frac{1}{2}OPT$.

Pracujeme s množinou lichých uzlů v MST O. Nyní v ní budeme hledat cestu. Vyjdeme-li z nejkratší hamiltonovské kružnice nad všemi vrcholy vstupu, pak je zřejmé, že její délka bude nejvýše OPT. Tuto kružnici teď budeme pouze zkracovat. Uvažujeme-li dva vrcholy i a j, pro které platí, že na cestě mezi nimi jsou pouze vrcholy nenáležící do O, pak je jistě můžeme vynechat, a z trojúhelníkové nerovnosti víme, že v nejhorším případě bude hrana (i, j) stejně dlouhá jako původní cesta. V nejhorším případě tedy po tomto zkracování zůstane délku kružnice rovna OPT.

Nyní začněme střídavě obarvovat hrany nalezené cesty, řekněme modrou a červenou. Množina červených i množina modrých hran je perfektním párováním na O, platí totiž, že pokrývají všechny vrcholy a žádné dvě hrany nesdílí vrchol. Víme, že dohromady množiny dávají OPT, pak jistě jedna z množin bude mít délka menší nebo rovnu $\frac{1}{2}OPT$, což jsme chtěli dokázat.



Obrázek 9: Perfektní párování

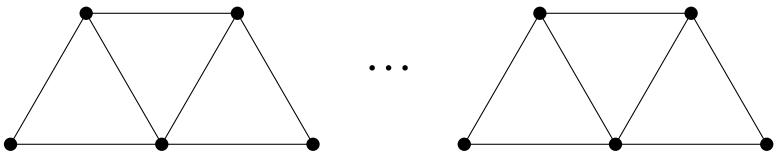
Z analýz aproximačních faktorů Christofidesova a double-tree algoritmů ještě vyplývá následující tvrzení.

Lemma 19

Sestrojíme-li pro vstup metrického TSP eulerovský podgraf délky $\alpha * OPT$, pak můžeme odvodit jeho α-aproximační algoritmus.

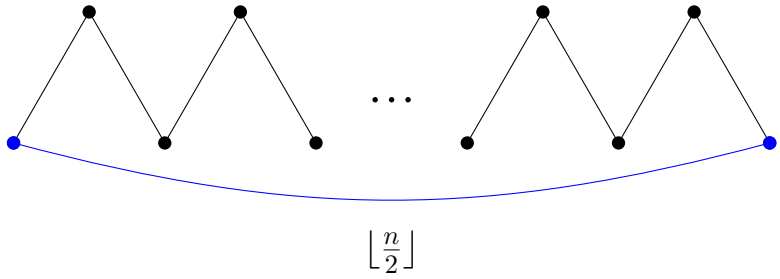
3.3.2 Těsný příklad

Vezměme si následující vstup (popsán v [6]), kde mají vyznačené hrany délku 1. Každá nevyznačená hrana má pak ohodnocení rovno délce nejkratší cesty mezi jejími koncovými uzly v grafu obsahujícím pouze hrany s ohodnocením 1 (takový graf je zobrazen na obrázku níže).



Obrázek 10: Těsný příklad Christofidesova algoritmu

Bude-li nalezena minimální kostra grafu zobrazená černými hranami na obrázku níže, pak jediné uzly s lichým stupněm budou krajní zobrazené modře. Jejich spojením pak dostaneme cestu (zkracování zde jistě nepovede k žádnému zlepšení).



Obrázek 11: Těsný příklad Christofidesova algoritmu – cesta

Podívejme se na délku takto nalezené cesty. Kostra bude mít jistě délku $n-1$ a poslední přidaná hrana $\lfloor \frac{n}{2} \rfloor$. Je také snadno vidět, že optimální cesta bude délky n. Pak už snadno

$$(n-1) + \lfloor \frac{n}{2} \rfloor = n - O(1) = OPT - O(1)$$

3.4 Kernighan–Lin algoritmus

Tento heuristický algoritmus není specifický pro TSP, ale je možné jím řešit obecně problémy, kdy máme nějakou podmínku C, již musejí splňovat jejich řešení, a funkci f, která takovým řešením přiřazuje určitou hodnotu. Zadanou pak máme množinu S, v níž hledáme podmnožinu T, která bude nejen splňovat podmínku C, ale její hodnota f bude mezi všemi vyhovujícími podmnožinami minimální. Takové problémy mívají často exponenciální složitost, proto se zpravidla spokojíme s hledáním takových podmnožin, které splňují C a jejich hodnota f je dostatečně malá. Jedním z takových problémů, kde se Kernighan–Lin využívá, je například *dělení grafů*.

Základní myšlenkou je postupné zlepšování úvodního řešení. Hledá se vždy lokální optimum, které se, pokud přinese nějaký zisk (zmenšení hodnoty f), použije pro další iteraci.

Na začátku se zvolí pseudonáhodné přípustné řešení $T \subseteq S$ a následně začne cyklus, kdy se snažíme T transformovat na jiné platné řešení T'. Pokud by platilo $f(T') < f(T)$, pak se T' použije pro další iteraci. V momentě, kdy už nenacházíme řešení, která by přinášela nějaký zisk, bylo nalezeno lokálně optimální řešení. V tuto chvíli se buď může skončit, nebo se vygeneruje nové počáteční řešení a proces se provede znovu.

Samotná transformace probíhá tak, že se hledají takové dvě množiny $\{x_1, x_2, \cdots, x_k\} \in T$ a $\{y_1, y_2, \cdots, y_k\} \in S - T$, pro které platí, že když nahradíme prvky první množiny v T prvky druhé množiny, pak dostaneme platné řešení T'. O tom, jak přijdeme k číslu k, ještě bude řeč později.

Idea algoritmu tedy spočívá v tom, že pro pseudonáhodné počáteční řešení budeme nacházet lokální optima a mezi nimi by se snad mělo objevit i optimum globální (nebo alespoň řešení se schůdnou hodnotou pro f). Přirozeně budeme

považovat algoritmus za tím kvalitnější, čím menší počet různých lokálních optim dostaneme a čím větší počet náhodných počátečních řešení nás dovede k dostatečně dobrému výsledku.

Obecná podoba heuristiky by se dala zapsat následovně:

Algorithm 5: Kernighan–Lin algoritmus – obecně

```
1 T ← pseudorandom solution;
2 T′ ← T;
3 while T′ do
4     T′ ← transformation of T such that f(T′) < f(T);
5 return T′ or start over if desired;
```

Vraťme se ještě ke zmíněnému koeficientu k. Jednou z možností, jak nad k uvažovat, je jako na předem zvolené číslo. Toho pro řešení TSP využil A. Croes s $k = 2$ a později S. Lin s $k = 3$. Tento přístup s sebou nese jistá úskalí, a to hlavně správnou volbu k. S narůstající hodnotou se výrazně zvyšuje i výpočetní čas, ale přirozeně dostáváme lepší řešení. Zjistit pak, jaké číslo nám zajistí dostatečně dobré výsledky ve schůdném čase, je náročné.

Tomuto problému se můžeme vyhnout tím, že k nebude mít žádnou pevně danou hodnotu. Takového způsobu se bude využívat v naší podobě algoritmu. Jelikož k nebude známé, nemůžeme již jednoduše uvažovat všechny podmnožiny T o velikosti k, jako u předchozí varianty. Na místo toho budeme iterativně hledat vždy nejvýhodnější x_i a y_i pro výměnu, budeme si zaznamenávat, pro jaké i byl zisk nejvyšší, a až když nebude možné nalézt žádné zlepšení, vyměníme příslušné množiny.

Algorithm 6: Kernighan - Lin algoritmus - variabilní k

```
1  T ← pseudorandom solution;
2  do
3      i ← 0;
4      T ← T′;
5      while improvement can be found do
6          i ← i + 1;
7          (x_i, y_i) ← best pair for exchange;
8          G_i ← total gain for {x_1, x_2, ··· , x_k} and {y_1, y_2, ··· , y_k};
9      k ← i, where G_i is max; \\definition of G_i is below
10     T′ ← (T − {x_1, x_2, ··· , x_k}) ∪ {y_1, y_2, ··· , y_k};
11 while f(T′) > f(T);
12 return T′;
```

Pro takový přístup vyvstává pár pravidel a otázek, které budeme muset zodpovědět.

Prvním pravidlem je *pravidlo disjunkce*. Budeme požadovat, aby množiny $\{x_1, x_2, ..., x_k\}$ a $\{y_1, y_2, \cdots, y_k\}$ nesdílely žádné prvky a zbytečně jsme opakovaně neprozkoumávali záměny, které nevedou ke zlepšení.

Druhé pravidlo vyplývá z variability k. Jelikož netušíme, pro jaké i budeme chtít výměnu provést, musíme vědět, že po každé výměně $\{x_1, x_2, \cdots, x_i\}$ za $\{y_1, y_2, \cdots, y_i\}$ dostaneme platné řešení. Podmínce, která musí být splněna, aby pravidlo platilo budeme říkat *podmínka proveditelnosti*.

Z pseudokódu uvedeného výše není jasné ještě pár věcí. Nejzřejmější je, jak identifikovat pár s největším ziskem. Potřebujeme najít takové *pravidlo výběru*, které bude rychlé a zároveň co nejefektivnější.

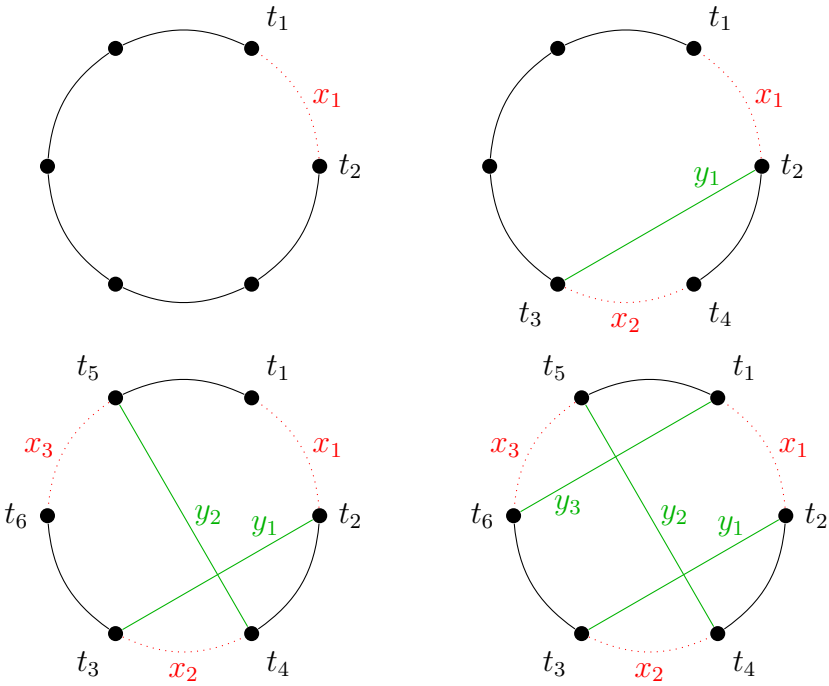
Další neznámou je funkce G_i. Tato funkce udává celkový zisk pro aktuálně nalezené prvky. Určíme-li g_i jako zisk pro (x_i, y_i), pak nejsnazším řešením je za G_i uvažovat $g_1 + g_2 + \cdots + g_i$. Tímto se také vyhneme tomu, že bychom přestali hledat v momentu, kdy bychom nalezli pár se zápornou hodnotou g_i. U této funkce budeme požadovat, aby dávala vždy kladné hodnoty, tedy abychom se nevydávali po cestách, které nám nedávají žádný zisk. Tomuto požadavku budeme říkat *pravidlo zisku*.

Nakonec ještě musíme nalézt odpovídající *ukončovací podmínku*, která bude udávat, zda má smysl pokračovat v hledání, nebo jsme již došli do bodu, kdy není možné nalézt další zlepšení. Nejtěžší úkol je pak najít dobrou rovnováhu mezi schůdným výpočetním časem a prozkoumáváním dostatku možností.

3.4.1 Kernighan–Lin pro TSP

Nejprve si uvědomme, že tuto heuristiku skutečně lze využít pro řešení TSP. Za množinu S můžeme brát všechny hrany úplného grafu nad vstupními městy. Podmínka C kladená na $T \subseteq S$, aby T byla řešením problému, jistě bude, že hrany, které obsahuje, musí tvořit hamiltonovskou kružnici mezi vstupními vrcholy. Funkce f pak bude takovým cestám přiřazovat jejich délku.

Nyní už se podíváme na to, jak konkrétně bude takový algoritmus vypadat pro TSP. Jeho jádrem je sekvenční hledání párů hran, kde první v původním řešení zrušíme a druhou do nového přidáme. Co je myšleno sekvenčním hledáním, si nejprve ilustrujeme následujícím obrázkem.



Obrázek 12: Sekvenční hledání hran

Sekvenčnost tedy znamená, že zrušíme-li hranu $x_i = (t_i, t_{i+1})$, pak přidaná hrana s ní bude druhý bod sdílet, tedy $y_i = (t_{i+1}, t_{i+2})$, a následně zrušená hrana bude mít zase tvar $x_{i+1} = (t_{i+2}, t_{i+3})$. Poslední přidaná hrana potom bude $y_k = (t_{2i}, t_1)$.

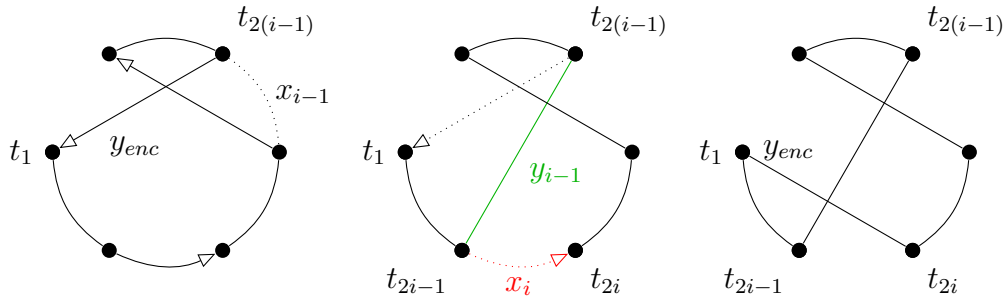
Další podmínky kladené na vybírané hrany jsou již zmíněné *pravidlo disjunkce*, kde $\{x_1, x_2, \cdots, x_k\} \cap \{y_1, y_2, \cdots, y_k\} = \emptyset$, *podmínka proveditelnosti* a námi určené *pravidlo výběru*.

Abychom mohli s hledáním kdykoli skončit, musíme pro každý pár ověřit, že po přidání hrany $y_k = (t_{2i}, t_1)$ nám znovu vznikne cesta. V implementaci knihovny je toto řešeno následovně.

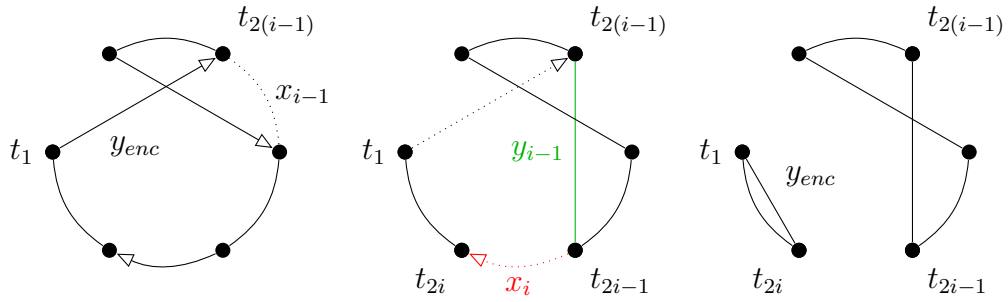
Při každé iteraci je k dispozici cesta, která vznikla záměnou $\{x_1, x_2, \cdots, x_{i-1}\}$ za $\{y_1, y_2, \cdots, y_{enc}\}$. Navíc se ještě udržuje hodnota t_1 (ta se v průběhu nemění) a $t_{2(i-1)}$.

Odstraněním $y_{enc} = (t_1, t_{2(i-1)})$ kružnici otevřeme. Nyní další rušená hrana $x_i = (t_{2i-1}, t_{2i})$ musí mít stejný směr jako hrana $(t_{2(i-1)}, t_1)$. Při obrácené orientaci by došlo k rozdělení grafu na dvě komponenty.

Jelikož klademe podmínku, že hrany jsou hledané sekvenčně, přidaná hrana, pak bude mít tvar $y_{i-1} = (t_{2(i-1)}, t_{2i-1})$. Nakonec cestu opět uzavřeme přidáním hrany $y_{enc} = (t_1, t_{2i})$.



Obrázek 13: Pár vyhovující podmínce proveditelnosti



Obrázek 14: Pár nevyhovující podmínce proveditelnosti

Nyní se podíváme na to, jak identifikovat aktuálně nejvýhodnější pár k výměně. Přirozeně se naskýtá možnost vybrat takové hrany x_i, y_i, pro které je $|x_i| - |y_i|$ mezi ostatními páry maximální. Právě tohoto *pravidla výběru* se využívá.

Problémem ale může být to, že v momentě, kdy vybereme takový pár, není brána v úvahu délku x_{i+1}. To není ideální, protože x_{i+1} může být výhodná hrana. Navíc se začneme omezovat na lokání informaci o nejbližším vrcholu k t_{2i}. Dobrým vylepšením algoritmu je tedy neposuzovat páry jen na základě rozdílu jejich délek, ale i délky x_{i+1}.

Zbývá nám ještě definovat ziskovou funkci G_i. Snadno se nabízí následná definice

$$G_i = \sum_{j=1}^{i} g_j = \sum_{j=1}^{i} (|x_j| - |y_j|)$$

Nakonec ještě musíme určit, za jakých podmínek se zkracováním cesty již přestaneme. Hledání přirozeně skončí, dojdou-li páry, které by vyhovovaly předešle definovaným podmínkám a pravidlům. Pro naši *ukončovací podmínku* si zavedeme proměnnou G^*, jež bude vyjadřovat největší nalezené zlepšení cesty. Uvědomme si, že $G^* \neq G_k$. Nejedná se totiž o zisk z vyměněných párů, ale o skutečné zkrácení cesty. Tuto hodnotu zjistíme pro i-tou iteraci následovně $G_{i-1} + |y_{enc}|$,

kde $y_{enc} = (t_{2i}, t_1)$. Hrana y_{enc} tedy uzavírá cestu, kde je poslední odstraněnou hranou x_i. Samotná ukončovací podmínka pak bude $G_i \leq G^*$.

Nyní už si můžeme ukázat upravenou verzi Kernighan–Lin pro TSP.

Algorithm 7: Kernighan–Lin algoritmus pro TSP

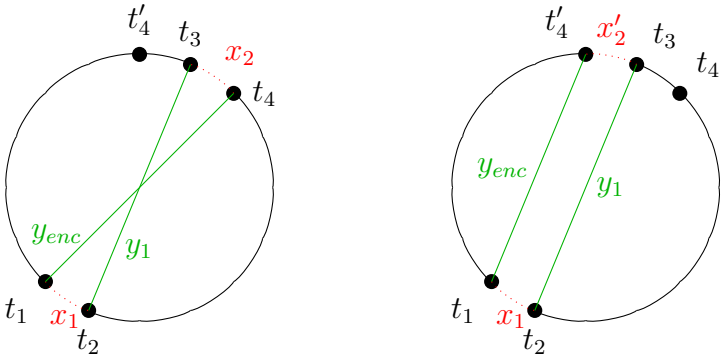
```
1  T ← pseudorandom solution;
2  T' ← T;
3  k = 0;
4  i = 0;
5  G* = 0; \\best improvement
6  do
7      i ← i + 1;
8      find pair x_i = (t_{2i-1}, t_{2i}), y_i = (t_{2i}, t_{2i+1}) for which:
           a) x_i ∉ {y_1, y_2, ...y_i} and y_i ∉ {x_1, x_2, ...x_i}
           b) T − {x_1, x_2, ...x_i} ∪ {y_1, y_2, ...y_i} is a tour
           c) |x_i| − |y_i| is maximal between such pairs
9      if no x_i, y_i were found then
10         break;
11     if G_{i-1} + c_{t_{2i},t_1} > G* then
12         k ← i;
13         G* = G_{i-1} + c_{t_{2i},t_1};
14     if G_i ≤ G* then
15         break;
16 while G* > 0;
17 T' ← (T − {x_1, x_2, ..., x_k}) ∪ {y_1, y_2, ..., y_k};
18 return T';
```

Co pseudokód neukazuje, je omezený backtracking, který se provádí v případě, že $G^* = 0$. Pak se na první a druhé úrovni zkoušejí ostatní možnosti za jednotlivé hrany. Tedy nejdříve budeme zkoušet alternativní y_2 (postupně podle sestupné hodnoty $|x_2| - |y_2|$), ve chvíli, kdy jsou všechny možnosti vyčerpány, se zkusí použít alternativní x_2 (tento případ je složitější). Takto se pokračuje i se všemi y_1, x_1 a t_1. V momentě, kdy nám žádná ze záměn nepřinese zlepšení, algoritmus končí a vrátí upravenou cestu.

Kdybychom prováděli takovýto backtracking na každé úrovni, zřejmě bychom byli schopni nalézt optimální řešení. To by ale trvalo nesmírně dlouho. Najít zlatou střední cestu mezi optimalitou a rychlostí je klíčové. Experimenty ukazují, že nezkoušíme-li všechna y_1, y_2, za mírné zhoršení efektivity dostaneme kratší výpočetní čas. V přiložené knihovně je i implementace s omezeným backtrackingem, který zkouší pouze dvě hodnoty pro y_1 a y_2 .

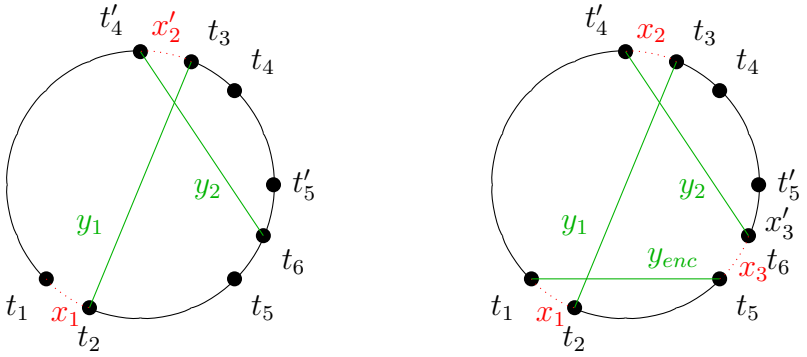
Vraťme se nyní ještě k problému s alternativní volbou x_2. Pro tuto hranu

máme jistě dvě volby. Abychom mohli cestu uzavřít, musí se t_4 nacházet mezi t_2 a t_3. V případě, že to tak není, bychom cestu přidáním y_{enc} místo uzavření rozdělili na dvě komponenty.



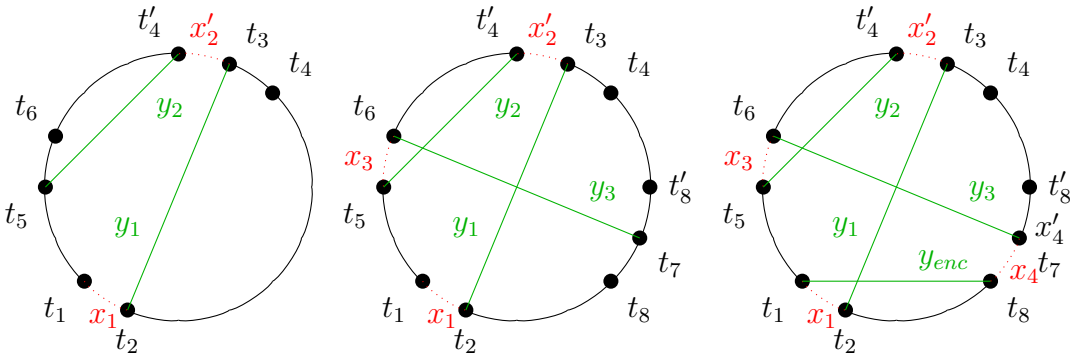
Obrázek 15: Problém alternativního x_2

Abychom i přes to mohli použít druhou volbu x_2, svolíme na chvíli z podmínky proveditelnosti a budeme pokračovat v hledání hran následovně. První možností je vybrat y_2 tak, že t_5 leží mezi t_3 a t_2. t_6 pak může být na libovolné straně od t_5. V tuto chvíli je již možné cestu řádně uzavřít.



Obrázek 16: Alternativní x_2 – první možnost cesty

Druhou možností je zvolit t_5 tak, že leží mezi t'_4 a t_1. t_6 se pak musí nacházet mezi t_6 a t_1. Nakonec t_7 musí být mezi t_3 a t_2 s tím, že volba t_8 závisí na tom, zda je delší x_4, nebo x'_4.



Obrázek 17: Alternativní x_2 – druhá možnost cesty

4 Experimenty

Nad probranými algoritmy byly provedeny experimenty pro porovnání jejich efektivity. K testování byl použit počítač s procesorem *Intel Core i7-8550U* a velikostí RAM 16 GB. Nejdříve jimi byly zpracovávány náhodně generované euklidovské grafy. Pro množství vrcholů od 5 do 500 bylo testováno 1 000 vzorků, výše pak 10 s rozpětím souřadnic (-1 000 – 1 000). U měření byla zaznamenána vždy průměrná a nejkratší délka cesty, stejně tak průměrný a nejkratší čas. Následně pak byly provedeny testy na datech, u kterých je známá optimální cesta nebo alespoň cesta, jejíž délka se té optimální značně blíží. Díky této informaci pak bylo možné zjistit u výsledků algoritmů jejich aproximační faktor.

4.0.1 Výsledky pro náhodně generované grafy

Tabulka 1: Test náhodně generovaných grafů – nearest-addition algoritmus

Nodes	**Length**		**Time**	
	average	**shortest**	**average**	**shortest**
5	4,535.18	1,823.87	0.01 ms	0.01 ms
10	6,924.88	3,953.35	0.05 ms	0.04 ms
15	8,557.2	4,810.07	0.12 ms	0.1 ms
25	11,045.27	7,643.03	0.44 ms	0.38 ms
50	15,487.12	12,341.39	2.93 ms	2.52 ms
100	21,589.98	18,600.28	20.54 ms	18.48 ms
250	33,688.58	30,896.78	307.27 ms	278.7 ms
500	47,170.87	44,236.8	2 s 225 ms	2 s 157 ms
1,000	66,362.79	65,393.66	17 s 802 ms	17 s 189 ms
2,000	93,887.51	93,332.49	2 m 19 s 18 ms	2 m 17 s 309 ms
3,000	114,278.71	113,302.76	7 m 57 s 999 ms	7 m 48 s 349 ms

Tabulka 2: Test náhodně generovaných grafů – double-tree algoritmus

Nodes	Length		Time	
	average	shortest	average	shortest
5	4,466.56	1,823.87	0.04 ms	0.02 ms
10	6,722.22	3,953.35	0.14 ms	0.08 ms
15	8,272.35	5,144.13	0.2 ms	0.16 ms
25	10,666.78	7,051.07	0.58 ms	0.46 ms
50	14,862.38	11,762.88	3.21 ms	2.71 ms
100	20,664.70	17,151.54	21.14 ms	18.87 ms
250	32,247.16	28,819.29	309.17 ms	280.73 ms
500	45,436.52	42,748.88	2 s 203 ms	2 s 153 ms
1,000	63,497.49	62,532.93	17 s 692 ms	17 s 463 ms
2,000	89,785.16	88,675.80	2 m 21 s 489 ms	2 m 19 s 334 ms
3,000	109,268.37	107,018.64	7 m 55 s 2 ms	7 m 50 s 709 ms

Podíváme-li se na první výsledky, můžeme vidět, že algoritmy nearest-addition a double-tree jsou na tom, co se efektivity týče, velice podobně. Double-tree mírně vede, co se délky výsledné cesty týče. Časově je pak nearest-addition o něco rychlejší pro malé vstupy, ale se zvyšujícím počtem uzlů se tento rozdíl srovnává.

Tabulka 3: Test náhodně generovaných grafů – Christofidesův algoritmus

Nodes	Length		Time	
	average	shortest	average	shortest
5	4,311.98	1,823.87	0.04 m	0.03 ms
10	6,058.89	3,953.35	0.17 ms	0.08 ms
15	7,259.48	4,726.46	0.29 ms	0.18 ms
25	9,225.93	6,860.62	0.96 ms	0.66 ms
50	12,594.04	10,650.56	5.89 ms	4.54 ms
100	17,280.01	15,277.42	39.56 ms	31.9 ms
250	26,585.91	24,564.74	576.95 ms	500.34 ms
500	37,018.6	35,454.21	4 s 106 ms	3 s 780 ms
1,000	52,217.02	51,295.77	32 s 874 ms	31 s 53 ms
2,000	72,835.8	71,801.88	4 m 16 s 376 ms	4 m 8 s 759 ms
3,000	88,637.24	88,107.63	14 m 32 s 880 ms	14 m 19 s 192 ms

Co se týká double-tree a Christofidesova algoritmu, které pracují na podobném principu, je zřejmé, že Christofidesův algoritmus vrací o dost kratší cesty, je ale také o to pomalejší. Velkou roli v tom jistě bude hrát způsob, jímž je implementované hledání eulerovského průchodu. Pro double-tree algoritmus toto hledání mohlo být značně zjednodušeno a díky tomu se časová složitost $O(|E|^2)$ Fleuryho algoritmu, který je použit pro hledání průchodu pro Christofidesův algoritmus, dala snížit až na $O(|E|)$, kde $|E|$ je počet hran ve vstupním grafu.

Tabulka 4: Test náhodně generovaných grafů – Kernighan–Lin algoritmus

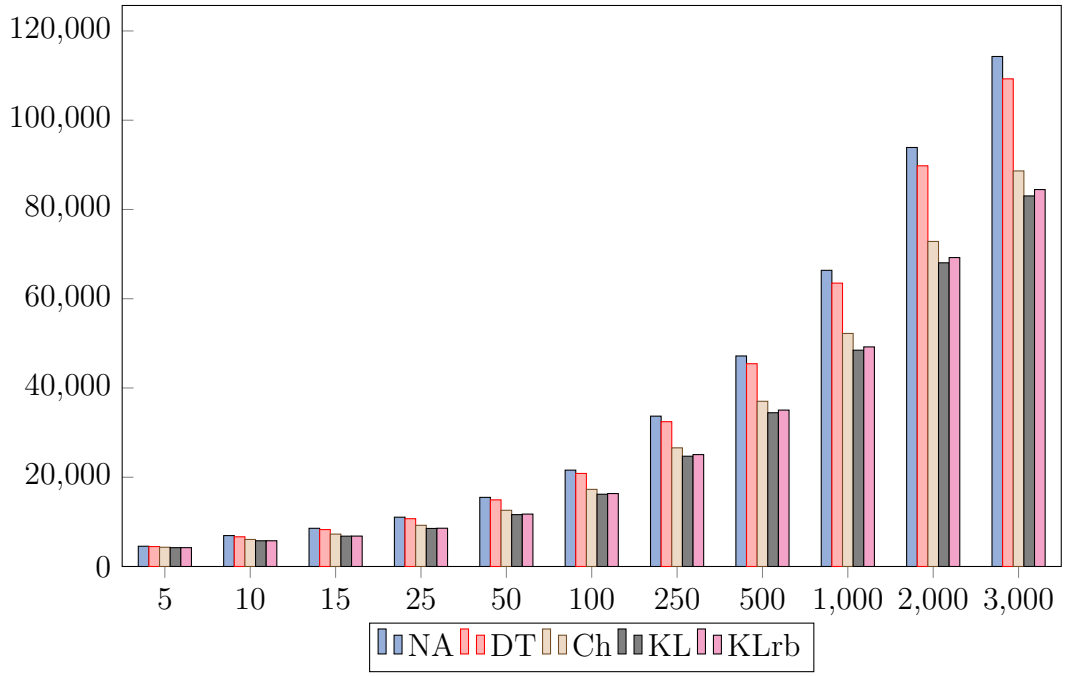
Nodes	**Length**		**Time**	
	average	**shortest**	**average**	**shortest**
5	4,220.89	1,816.18	0.08 ms	0.02 ms
10	5,762.91	3,768.08	0.56 ms	0.11 ms
15	6,796.69	4,417.87	1.03 ms	0.36 ms
25	8,524.89	6,338.58	3.36 ms	1.36 ms
50	11,623.58	9,971.3	17.69 ms	6.83 ms
100	16,194.06	14,228.2	82.59 ms	45.14 ms
250	24,709.99	23,283.12	673.13 ms	419.97 ms
500	34,452.72	32,942.05	2 s 812 ms	1 s 941 ms
1,000	48,468.76	47,801.85	13 s 289 ms	10 s 803 ms
2,000	68,046.66	67,488.66	1 m 6 s 296 ms	54 s 748 ms
3,000	83,033.12	82,391.27	2 m 47 s 44 ms	2 m 22 s 448 ms

Porovnáme-li výsledky Kernighan–Lin algoritmu s ostatními, přesvědčíme se, že cesty, které hledá jsou mezi ostatními nejkratší. Pro vstupy s malým počtem vrcholů toto přichází s horší časovou složitostí. Ta se ovšem posléze srovná a pro velké vstupy se stává i časově mnohem efektivnější.

Tabulka 5: Test náhodně generovaných grafů – Kernighan–Lin algoritmus s omezeným backtrackingem

Nodes	Length		Time	
	average	shortest	average	shortest
5	4,220.76	1,816.18	0.07 ms	0.02 ms
10	5,766.36	3,768.08	0.42 ms	0.11 ms
15	6,811.56	4,417.87	0.76 ms	0.3 ms
25	8,573.7	6,447.74	2.34 ms	0.92 ms
50	11,739.94	9,998.61	11.86 ms	4.89 ms
100	16,345.56	14,336.08	55.18 ms	27.6 ms
250	25,072.9	23,064.13	453.65 ms	268.31 ms
500	35,044.28	33,314.86	1 s 966 ms	1 s 316 ms
1,000	49,203.67	48,481.74	9 s 87 ms	7 s 680 ms
2,000	69,218.56	68,175.6	45 s 30 ms	37 s 687 ms
3,000	84,460.58	83,714.59	1 m 49 s 761 ms	1 m 31 s 171 ms

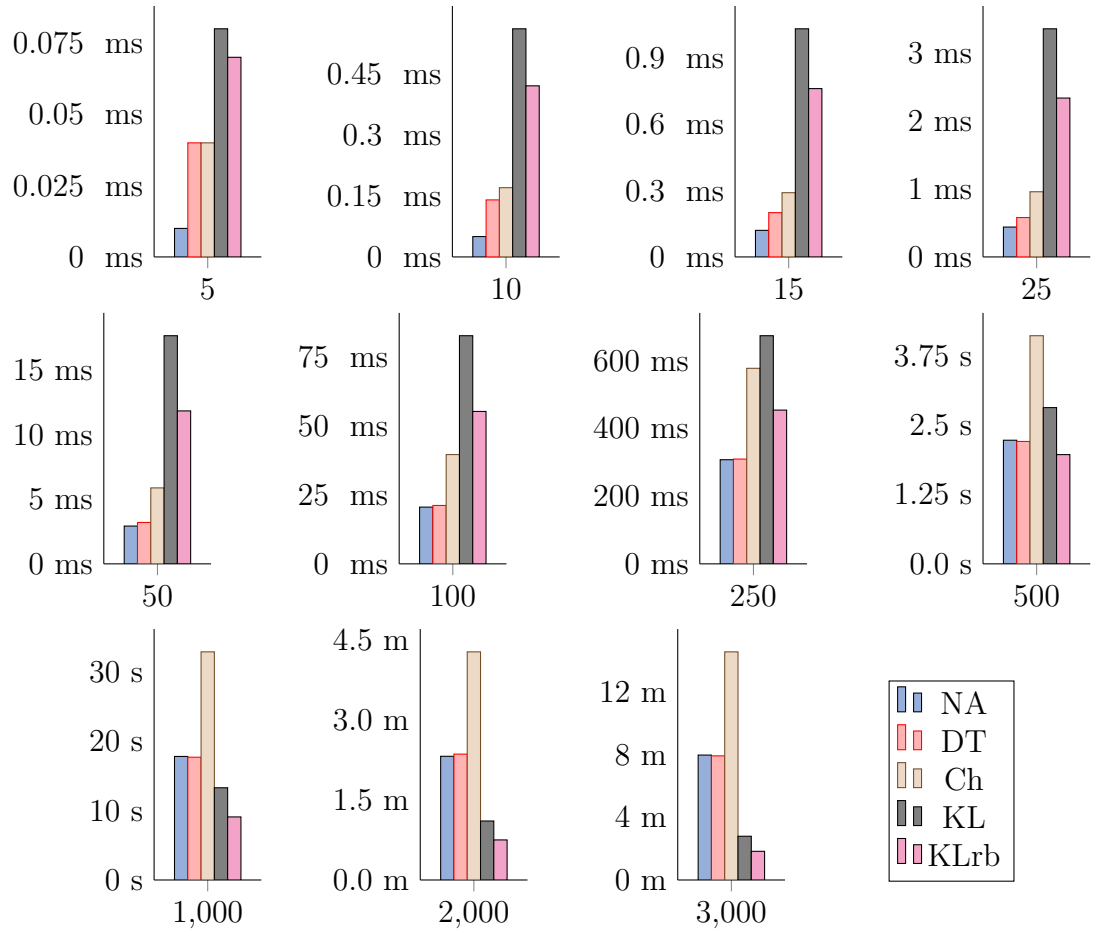
Poslední hodnoty byly naměřeny pro verzi Kernighan–Lin algoritmu s omezeným backtrackingem. Ukázalo se, že toto omezení vede k rozumné záměně efektivity za rychlost. Výsledné cesty nejsou o tolik horší jako u klasické verze algoritmu a zrychlení je znatelné.



Obrázek 18: Graf s délkami výsledných cest pro náhodně generované grafy

Na předchozím grafu jsou vizualizované délky nalezených cest. Z 2-aproximačních algoritmů je double-tree o něco efektivnější než nearest-addition. Podíváme-li se na efektivnější algoritmy, pak Kernighan–Lin i jeho omezená verze jsou o něco lepší než Christofidesův. Je ale dobré pamatovat na to, že nemáme zaručenou hodnotu aproximačního faktoru.

Na grafech níže jsou algoritmy porovnány z časového hlediska.



Obrázek 19: Graf s časovými údaji pro náhodně generované grafy

U malých vstupů mají 2-aproximační algoritmy nejlepší výsledky, nearest-addition ještě o něco lepší než double-tree. Nejhůře si pak vede Kernighan–Lin. Jakmile se ale vstupy začnou zvětšovat, začne být zřejmá pomalost Christofidesova algoritmu a projeví se rychlost algoritmu Kernighan–Lin pro velká data.

Implementace algoritmů v knihovně sice není časově nejefektivnější, ovšem nenaráží na problémy s pamětí, jelikož si neukládají hrany (kterých je u úplných grafů značné množství), ale počítají si jejich délky za běhu. Jelikož Kernighan–Lin algoritmus pracuje vždy s konkrétní cestou, množství hran, o nichž musí uvažovat, je značně omezeno a pro velké množství vrcholů se projeví vyšší časová efektivita.

4.0.2 Výsledky pro vstupy se známými optimálními cestami

Díky tomu, že je TSP tolik řešeným problémem, je k dispozici spousta grafů, kde délky optimálních cest známe nebo se jim alespoň prokazatelně blížíme. Ty se jistě hodí ke zjištění, jak si na tom algoritmy vedou, co se aproximačního faktoru jeho řešení týče.

Pro tento test byla data čerpána ze zdroje https://www.math.uwaterloo.ca/tsp/world/countries.html. Na stránce je 27 zemí, pro které jsou daná města v TSPlib formátu, mezi nimiž se má cesta hledat. Rozsah dat je značný – nejnižší počet měst má Západní Sahara (29) a nejvíce Čína (71 009). Soubory s těmito vstupními daty jsou přiloženy k bakalářské práci ve složce *data*.

Následující tabulka vždy ukazuje název země, počet měst a pak aproximační faktory pro řešení nalezené algoritmy (v některých buňkách se místo tohoto údaje nachází znak 'X' značící, že výpočet nebyl dokončen kvůli dlouhé délce trvání). Pro každý algoritmus je pak na konci uveden průměrný aproximační faktor ze všech výsledků. U některých zemí je připsána hvězdička. Ta značí, že neznáme přesnou délku optimální cesty, jedná se ovšem o odchylky v rámci setin procenta.

Tabulka 6: Test na státy se známými optimálními cestami

Country	**Nodes**	**NA**	**DT**	**CH**	**KL**	**KlrB**
Argentina	9,152	1.56	1.39	1.12	1.06	1.09
Burma *	33,708	1.40	1.38	1.11	1.05	1.07
Canada	4,663	1.42	1.38	1.10	1.04	1.07
China *	71,009	X	X	X	1.05	1.06
Djibouti	38	1.28	1.41	1.04	1.00	1.00
Egypt	7,146	1.40	1.36	1.10	1.13	1.20
Ireland	8,246	1.42	1.38	1.12	1.05	1.06
Finland	10,639	1.44	1.38	1.11	1.06	1.07
Greece	9,882	1.39	1.37	1.11	1.08	1.11
Honduras	14,473	1.56	1.39	1.14	1.05	1.10
Italy	16,862	1.42	1.38	1.11	1.11	1.19
Japan	9,847	1.40	1.35	1.10	1.18	1.23
Kazakhstan	9,976	1.44	1.38	1.11	1.05	1.10
Luxembourg	980	1.59	1.36	1.15	1.04	1.05
Morocco	14,185	1.42	1.37	1.11	1.05	1.07
Oman	1,979	1.43	1.36	1.10	1.13	1.28

Nicaragua	3,496	1.54	1.38	1.13	1.06	1.08
Panama	8,079	1.52	1.38	1.12	1.05	1.14
Qatar	194	1.34	1.34	1.13	1.05	1.06
Rwanda	1,621	1.65	1.43	1.17	1.05	1.06
Sweden	24,978	1.41	1.36	1.11	1.07	1.07
Tanzania	6,117	1.43	1.36	1.12	1.05	1.09
Uruguay	734	1.42	1.38	1.12	1.05	1.07
Vietnam	22,775	1.38	1.37	1.10	1.05	1.07
Western Sahara	29	1.23	1.29	1.15	1.06	1.04
Yemen	7,663	1.40	1.36	1.10	1.12	1.14
Zimbabwe	929	1.41	1.38	1.13	1.07	1.16
Average approx. factor		**1,44**	**1,37**	**1,12**	**1,07**	**1,10**

Z dat lze vidět, že Kernighan–Lin algoritmus dosahuje velice dobrých výsledků. Jeho rychlejší alternativa je pak srovnatelně dobrá s o mnoho pomalejším Christofidesovým algoritmem. Nearest-addition pak dopadl o něco hůře než double-tree, oba však dávaly výsledky s aproximačním faktorem výrazně nižším než je jejich horní hranice.

Obecně byly experimenty časově vysoce náročné, ale algoritmy byly schopné zpracovat i velice rozsáhlé vstupy. Na problém narazily až u Číny s přes 70 000 městy. Ani tam ale neměly problémy s pamětí, pouze byl výpočet kvůli dlouhému trvání ukončen. Jediný algoritmus, který byl schopen i tento vstup zpracovat ve schůdném čase byl Kernighan–Lin, což opět ukazuje jeho efektivitu.

K práci je přiložený soubor *results.ods* obsahující délky všech nalezených cest.

5 Knihovna

Součástí příloh je C# knihovna *TSPAlgorithmsLibrary* implementující popsané algoritmy pro euklidovský TSP a WPF aplikace *TSPTestingApp*, která knihovnu používá pro jejich testování.

5.1 Struktury

Základní poskytovanou třídou je `Node` s vlastnostmi `Id` a souřadnicemi `X` a `Y` typu `double`. Jedinou veřejnou metodou je `double Distance(Node node)` udávající euklidovskou vzdálenost k předanému vrcholu.

Druhou důležitou třídou je `Path` reprezentující výslednou hamiltonovskou kružnici. Je možné nastavit aktuální pozici v cestě, kterou udává vlastnost `CurrentIndex`, a směr cesty změnou `Direction`. Pro směr je zaveden výčtový typ stejného názvu s hodnotami `Forwards` a `Backwards`.

Tyto vlastnosti jsou buď nastavitelné přímo, nebo je možné využít metod
`SetCurrentIndex(Node node)`

`SetDirection(Node fromNode, Node toNode)`
- `fromNode` se v cestě musí nacházet přímo před nebo přímo za `toNode`

```
SetStartingPoint(Node fromNode,
                 Node toNode,
                 Direction direction = Direction.Forward)
```
- `CurrentIndex` nastaví na index `fromNode` a `Direction` od `fromNode` k `toNode`, není-li hodnota `direction` `Backwards`.

Cestu jde pak cyklicky procházet metodami `Next`, `Prev`, `PeekNext`, `PeekPrev`, `PeekAfter` a `PeekBefore`. Také lze přistupovat k vrcholům pomocí indexu, případně získat index vrcholů metodou `IndexOf`, nebo získat `List` s vrcholy zavoláním `ToList`.

Nakonec si ještě zmíníme vlastnost `Length`, která, jak název napovídá, vrátí celkovou délku cesty.

5.2 Algoritmy

Knihovna obsahuje pro každý algoritmus třídu, tedy `NearestAddition`, `DoubleTree`, `Christofides` a `KernighanLin`, navíc pak ještě třídu `KernighanLinReducedBacktracking`, která implementuje Kernighan–Lin algoritmus s omezeným backtrackingem, jak bylo zmíněno v kapitole 3.4.1.

Společným rozhraním pro tyto třídy je `ITSPAlgorithm`. Rozhraní předepisuje následující metody.

```
Path FindShortestPath(List<Node> nodes);

Path FindShortestPath(List<Node> nodes,
                      string outputFolderPath,
```

```
                        string outputFileName);

Path FindShortestPath(string inputFilePath);

Path FindShortestPath(string inputFilePath,
                        string outputFolderPath,
                        string outputFileName);
```

Hodnotou parametru `inputFilePath` by měla být cesta k souboru ve formátu TSPLib, ze kterého budou načtena vstupní data. Jsou-li předány parametry `outputFolderPath` a `outputFileName`, pak bude do `outputFolderPath` uložena nalezená cesta opět ve formátu TSPLib do souboru *`outputFileName`.tour*.

5.3 TSPLib

Součástí knihovny jsou navíc statické třídy pracující s již zmíněným TSPLib formátem. Jejich názvy s veřejnými funkcemi jsou popsány níže.

TSPDeserializer

```
List<Node> DeserializeNodes(string filePath);
```

TourDeserializer

```
Path DeserializePath(string filePath, List<Node> nodes);
```

TSPSerializer

```
void SerializeNodes(List<Node> nodes, string folderPath, string
fileName);
```

TourSerializer

```
void SerializePath(Path path, string folderPath, string fileName);
```

5.4 Testovací aplikace

Jak bylo zmíněno, součástí příloh je také testovací desktopová aplikace *TSPTestingApp*. Ta využívá jak metody `FindShortestPath` pro hledání hamiltonovských kružnic, tak třídy z předchozí kapitoly.

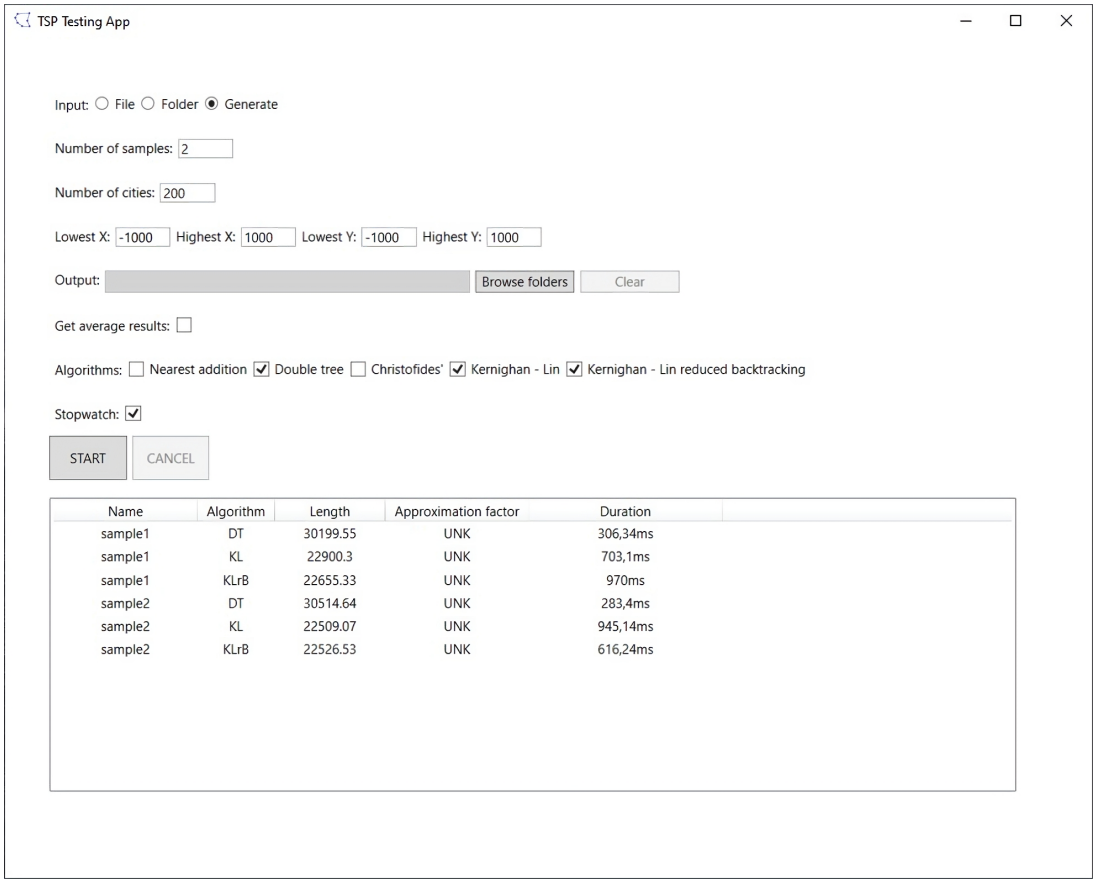
Aplikace umožňuje uživateli vybrat si ze tří módů – *File*, *Folder* a *Generate*. Tato volba udává typ vstupních dat – *.tsp* soubor, složka, ze které se zpracují všechny *.tsp* soubory, nebo náhodně generovaný euklidovský graf.

U grafu lze zvolit rozpětí souřadnic, počtu vrcholů a počtu vzorků. Navíc je ještě možnost zobrazení pouze průměrných výsledků pro každý algoritmus.

U *File* a *Folder* módů je možné zvolit cestu k *.opt.tour* souboru/složce odkud se čerpá optimální výsledek pro vstup. V tomto případě bude u výsledků zobrazena i hodnota *Approximation factor*.

Vždy je také možné přidat cestu ke složce, do níž se uloží výsledky ve formátu *.tsp* (u grafů zároveň vznikne *.tsp* soubor), a zvolit zda-li chce uživatel měřit délku trvání hledání cesty. Samozřejmě je zde i volba toho, které algoritmy budou testovány.

Pro množství výsledku do pěti zobrazí aplikace grafy, jinak jsou data zobrazena tabulkou.



Obrázek 20: Ukázka testovací aplikace *TSPTestingApp*

Závěr

Práce se věnuje problému obchodního cestujícího. Na začátku je problém popsán, je vysvětlena jeho složitost a podproblémy, jichž se týkají další kapitoly. Následně jsou popsány vybrané algoritmy. Jsou u nich uvedeny pseudokódy, a je-li znám, tak aproximační faktor a těsný příklad.

Na těchto algoritmech byly provedeny experimenty, jejichž výsledky jsou zpracovány v odpovídající kapitole. Z těchto testů vyplývá, že algoritmus Kernighan–Lin se ukázal být nejefektivnější, co se délek nalezených cest týče. Stejně tak se ukázal jako nejrychlejší pro grafy s velkým množstvím uzlů. Pro malé vstupy je časově nejefektivnější algoritmus nearest-addition, který měl ovšem také nejhorší průměrný aproximační faktor. Nejhorší výsledky, pokud jde o čas, měl Christofidesův algoritmus.

Součástí práce je také přiložená knihovna *TSPAlgorithmsLibrary* psaná v jazyce C#, která implementuje popsané algoritmy a s jejíž pomocí byly prováděny zmíněné experimenty. Dále je přiložena testovací WPF aplikace *TSPTestingApp*, jež knihovnu využívá.

Conclusions

This thesis focuses on the travelling salesman problem. It begins by describing the problem, explaining its complexity and it's sub-problems addressed in subsequent chapters. Selected algorithms are then described, accompanied by pseudocode and, when known, an approximation factor and a tight example.

Experiments were conducted on these algorithms. Their results are analyzed in the corresponding chapter. These tests showed that the Kernighan–Lin algorithm proved to be the most efficient in terms of the lengths of the paths found. Similarly, it was also the most efficient in terms of time for graphs with a large number of nodes. For small inputs, the nearest-addition algorithm is the fastest, although it also had the worst average approximation factor. Christofides' algorithm performed the worst in terms of time.

Part of this thesis is also an attached library called *TSPAlgorithmsLibrary*, written in C#, which implements the described algorithms and was used to conduct the mentioned experiments. Additionally, a testing WPF application, *TSPTestingApp*, utilizing the library, is provided as well.

A Obsah elektronických dat

Součástí práce jsou následující elektronická data v systému katedry informatiky:

`text/`
: Adresář s textem práce ve formátu PDF, vytvořený s použitím závazného stylu KI PřF UP v Olomouci pro závěrečné práce, včetně všech (textových) příloh, a všechny soubory potřebné pro bezproblémové vytvoření PDF dokumentu textu (případně v ZIP archivu), tj. zdrojový text textu a příloh, vložené obrázky, apod.

`README.md`
: Textový soubor s požadavky na systém a informacemi o spuštění testovací aplikace a použití knihovny.

`src/`
: Adresáře se zdrojovým kódem implementované knihovny, testovací aplikace a upravené knihovny použité k hledání perfektního párování.

`data/`
: Adresář s daty k experimentům na skutečných zemích.

`results.ods`
: Soubor s výsledky experimentů.

Literatura

[1] BĚLOHLÁVEK Radim a VYCHODIL, Vilém. Diskrétní matematika pro informatiky II. Olomouc 2006. Dostupný z: http://belohlavek.inf.upol.cz/vyuka/dm2.pdf

[2] JANČAR, Petr. Teoretická informatika – učební text. Ostrava: Ediční středisko VŠB-TUO, 2007. ISBN 978-80-248-1487-2. Dostupný z: http://phoenix.inf.upol.cz/~jancarp/VAS/jancar-ti-vsb.pdf

[3] WILLIAMSON, David P. a SHMOYS, David Bernard. The design of approximation algorithms. New York, N.Y.: Cambridge University Press, 2011. ISBN 978-0-521-19527-0. Dostupné z: https://www.designofapproxalgs.com/book.pdf

[4] LIN, Shen a KERNIGHAN Brian W. An Effective Heuristic Algorithm for the Traveling-Salesman Problem. Operations Research, Vol. 21, No. 2, s. 498-516. INFORMS 1973. Dostupné z: https://www.jstor.org/stable/169020

[5] HROMKOVIČ, Juraj. Algorithmics for hard problems: introduction to combinatorial optimization, randomization, approximation, and heuristics. 2nd ed., corr. print. Texts in theoretical computer science. Berlin: Springer, 2004. ISBN 3-540-44134-4

[6] VAZIRANI, Vijay V. Approximation algorithms. Berlin: Springer, 2001. ISBN 3-540-65367-8.