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:	31
Přílohy:	1 CD/DVD
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:	31
Supplements:	1 CD/DVD
Thesis language:	Czech

Anotace

Anotace - jeden odstavec

Synopsis

Anotace anglicky

Klíčová slova: problém obchodního cestujícího;

Keywords: travelling salesman problem;

Děkuji, děkuji, děkuji.

Místopřísežně prohlašuji, že jsem celou práci včetně příloh vypracoval/a samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

datum odevzdání práce podpis autora

Obsah

Seznam tabulek

1 Teorie

Problém obchodního cestujícího je úzce spjatý s *teorií grafů* a proto je potřeba si na úvod zavést některé z jejich základních pojmů.

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 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.

V některých situacích nás bude zajímat počet hran, kterým je jistý uzel koncovým. Tomuto číslu 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* definované následovně:

$$w : E \to D$$

kde w je *hranové ohodnocení*, E je množina hran příslušného grafu a D je nějaká množina hodnot.

Dále je vstupem tzv. *úplný* graf. V takovém grafu platí, 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 druhé. právě když mezi nimi existuje hrana (v případě orientovaných grafů musí mít ještě hrana 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 $\forall i \in \{0, \cdots, n\}\ v_i \in V$ a $\forall j \in \{1, \cdots, n\}\ e_j \in E$.

Sled nazýváme

- *uzavřený*, pokud $v_0 = v_n$
- *tah*, pokud $\forall k, l \in \{1, \cdots, n\}$, kde $k \neq l$, $e_k \neq e_l$ (neopakují se hrany)
- *cesta*, pokud $\forall k, l \in \{0, \cdots, n\}$, kde $k \neq l$, $v_k \neq v_l$ (neopakují se vrcholy)
- *kružnice*, pokud je *cestou*, ale $v_0 = v_n$

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 4
Pokud je neorientovaný graf souvislý a všechny jeho vrcholy mají sudý stupeň, pak v něm existuje uzavřený eulerovský tah.

Pro TSP je ještě klíčový 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 *souvislý graf* a *podgraf* grafu.

Definice 5 (Souvislý graf)
Neorientovaný graf $G = \langle V, E \rangle$ nazýváme *souvislý*, pokud $\forall u, v \in V$ existuje sled (viz později) z u do v.

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 nevyskytují se v něm 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 Problém

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, kde pro vstup existuje víc možných řešení a jejich úlohou je najít mezi nimi najít které bude mezi nimi buď minimální nebo maximální vzhledem k nějaké předem dané funkci.

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* toho řešení
- *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, které řeší takový problém zvažujeme jeho *správnost* a *optimalitu*. Řekneme, že algoritmus A pro problém P je *správný*, pokud $\forall x \in In, A(x) \in sol(x)$. *Optimální* je navíc pokud $\forall x \in In$ je $A(x)$ optimální řešení.

Některé optimální algoritmy bývají ale 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, 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 $\forall n \in \mathbb{N}$ 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, 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.

1.2.1 Složitost

Na začátku je třeba si zavést třídy **P** a **NP**. Algoritmy, které spadají do třídy **P** jsou řešitelné deterministickými stroji v polynomiálním čase a bereme je za praktický zvládnutelné. Na nedeterministických strojích můžeme řešit v polynomiálním čase **NP** problémy. U těchto strojů předpokládáme, že z možných kroků provede vždy ten žádaný a dostane se tak ke správnému výsledku bez nutnosti zkoušet všechny možnosti. Jistě platí $\mathbf{P} \subseteq \mathbf{NP}$, jelikož deterministické stoje jsou speciálním případem nedeterministických, které pouze nevyužívají možnosti nedeterministického výběru.

Některé rozhodovací problémy mají navíc tu vlastnost, že jsou na ně polynomiálně redukovatelné všechny rozhodovací problémy v **NP**. To znamená, že pro každý vstup libovolného problému P_1 můžeme v polynomiálním čase najít vstup pro P_2, který bude vracet stejnou odpověď, jakou by vrátil P_1 na původní vstup. Problémům s touto vlastností se říká ***NP**–těžké*.

Pokud jsou navíc **NP**, říká se jim ***NP**-úplné*. Tyto 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 $\mathbf{P} \neq \mathbf{NP}$ jde 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 polynomický algoritmus. Kdyby totiž takový algoritmus existoval, stačilo by ho využít pro nalezení optimálního řešení, porovnat jeho cenu s se zadaným číslem k pro P_D a na základě toho, jestli by bylo větší nebo menší v závislosti na povaze P_O vydat Ano nebo Ne. Tímto bychom tedy získali polynomiální algoritmus řešící také P_D. Jenže protože P_D je **NP**–těžký, tak všechny **NP** problémy jsou na něj polynomiálně redukovatelné a zvládli bychom je tedy i v polynomiálním čase řešit a tedy $\mathbf{P} = \mathbf{NP}$, což je spor.

2 Problém obchodního cestujícího

Problém obchodního cestujícího (TSP - *Travelling Salesman Problem*) je jedním z nejstudovanějších optimalizačních problémů. Podstatou je mezi zadanými městy nalézt cestu, která začíná i končí ve stejném místě a zbytek měst navštíví právě jedenkrát. Jelikož seznam měst můžeme brát jako množinu vrcholů grafu, můžeme o TSP uvažovat jako o úloze o cestování v grafu. Cestou, kterou hledáme je pak *hamiltonovská kružnice*.

Formálně bychom mohli TSP definovat následně:

Definice 8 (Problém obchodního cestujícího)

> NÁZEV: TSP
>
> VSTUP: Úplný neorientovaný graf $G = \langle V, E \rangle$,
>
> hranové ohodnocení $c : E \leftarrow R^+$
>
> 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

Jedním z důvodů, proč je TSP tak zkoumaným problémem, je jeho složitost. Samotný problém je jistě v **NP**. Nedeterministický stroj, který by měl za úkol ho vyřešit, by prostě vybral $|V|$ hran z E a jelikož by vždy volil správně, vrátil by nejkratší hamiltonovskou kružnici v grafu. Tohle by samozřejmě zvládnul provést v polynomiálním čase.

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?

Ví se, že tato verze problému 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 $\boldsymbol{P} = \boldsymbol{NP}$.

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

Definice 11 (Rozhodovací verze 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 jsme již zmínili, TSP má ve své klasické podobě na vstupu graf a jeho příslušné hranové ohodnocení. Vstup x 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 existuje tedy exponenciálně velká mezera mezi pozitivními a negativními instancemi HC. Pokud bychom měli 2^n–aproximační polynomiální algoritmus A pro TSP, pak pro pozitivní instance HC by našel cestu délky nejhůře $|V| \cdot 2^{|V|}$, pro negativní instance zase nejlépe délky $|V| \cdot 2^{|V|} + 1$.

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. Uvedli jsme navíc, že HC je **NP**–úplný. V kapitole o složitosti jsme řešili, že nalezení polynomiálního algoritmu pro kterýkoli z problémů této množiny by znamenalo, že **N** = **NP**.

2.2 Metrický problém obchodního cestujícího

TSP může být žádané také omezit jiným způsobem. Pokud přidáme sna hranové ohodnocení podmínku, že musí platit trojúhelníková nerovnost, tedy $\forall u, v, w \in V$ platí

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

pak dostaneme *metrický TSP*.

2.3 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

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

3 Algoritmy

Jak bylo již zmíněno, TSP je řešitelný v polynomiálním čase pouze nedeterministickými stroji. U deterministických (klasických) strojů je složitost exponenciální. Z toho ale vyplývá, že bychom se pro větší vstupy nemuseli v reálném čase dočkat odpovědi. Z tohoto důvodu se pro hledání řešení používají tzv. *heuristiky*. Při jejich použití se nemůžeme spolehnout, že najdeme optimální řešení. Jsme ale schopni nalézt přibližné řešení v reálném čase.

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

3.1 Nearest addition

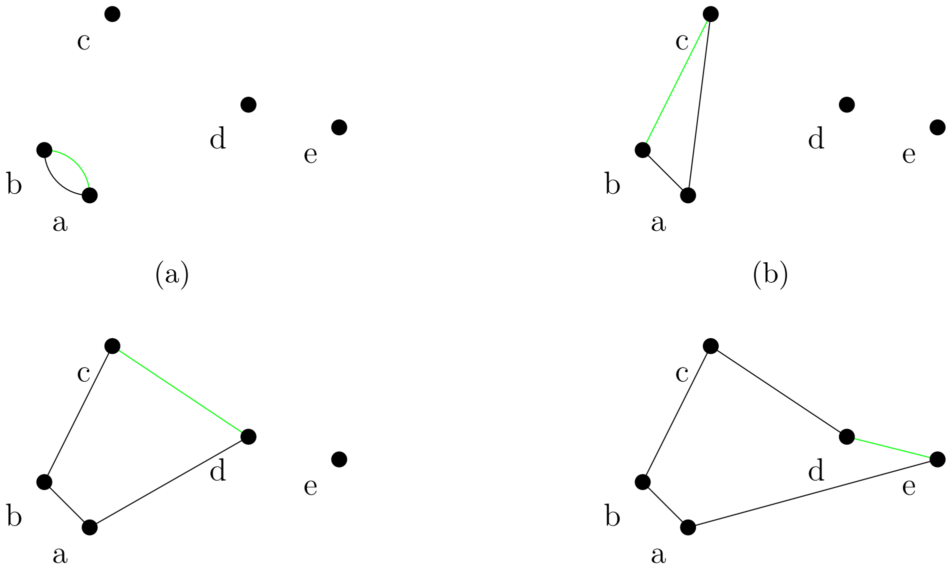
Tento algoritmus spadá do skupiny hladových algoritmů a je poměrně intuitivní. Vychází z myšlenky, že k vytvářené cestě vždy přidáme nejbližší nenavštívené město. Nejprve najdeme taková dvě města i, j, která jsou si v grafu nejblíže a vytvoříme cestu $path$, která povede z i do j a zpět. Následuje cyklus, ve kterém 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 všechny města.

Algorithm 1: Nearest addition algoritmus

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

Ukažme si jednoduchý příklad chodu algoritmu.

Příklad 12



(a) (b) (c) (d)

Obrázek 1: Nearest addition příklad

3.1.1 Aproximační faktor

Věta 13
Nearest addition je 2-aproximační algoritmus.

Abychom si tvrzení dokázali, musíme si nejdříve ukázat vztah k Primově algoritmu, který hledá minimální kostru grafu.

Algorithm 2: Primův algoritmus

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

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

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

Lemma 14

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 15

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 14 plyne

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

Po najití 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, tak 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 narest addition algoritmem bude tedy maximálně $2\sum_{l=1}^{n-1} c_{i_l j_l}$.

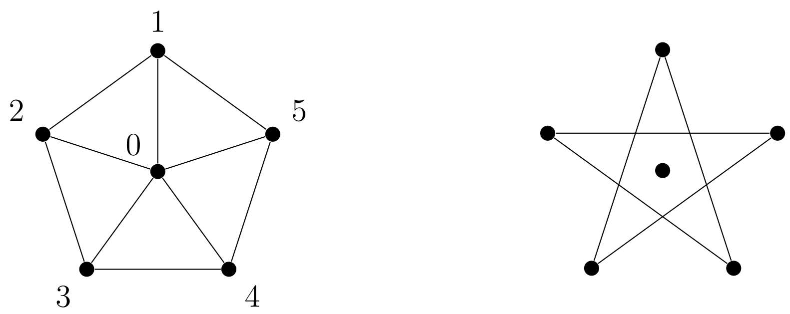
Upravíme li poslední nerovnost, dostaneme $\sum_{l=2}^{n} c_{i_l j_l} \leq 2OPT$.

To znamená, že cesta nalezená algoritmem v nejhorším případě stát bude stát nejvýše dvakrát tolik, co optimální cesta pro stejný vstup, což je přesně to, co jsme chtěli dokázat. □

3.1.2 Tight example

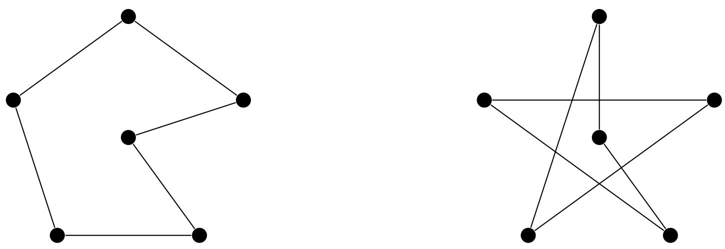
Můžeme si ukázat jeho tzv. *tight example*, což je pro algoritmus s aproximačním faktorem α příklad, u kterého nalezne řešení s cenou $\alpha * OPT$ (můžeme povolit určitou konstantní odchylku). Tedy budeme chtít najít vstup pro metrický TSP, pro který nám tento algoritmus vrátí cestu s délkou skoro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

Vezměme si jako vstup $(n-1)$-cípou hvězdu s jedním vrcholem ve svém středu. Hrany busou ohodnoceny následovně: hrany vlevo na obrázku budou mít cenu jedna a hrany vpravo dva.



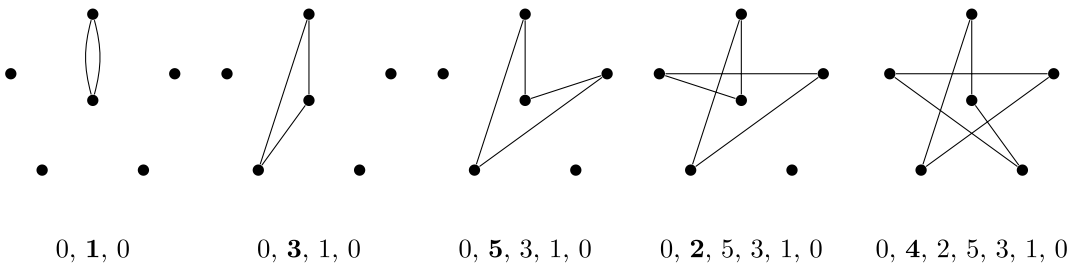
Obrázek 3: Nearest addition tight example

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



Obrázek 4: Nearest addition tight example - 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: Nearest addition tight example - chod algoritmu

3.2 Double-tree

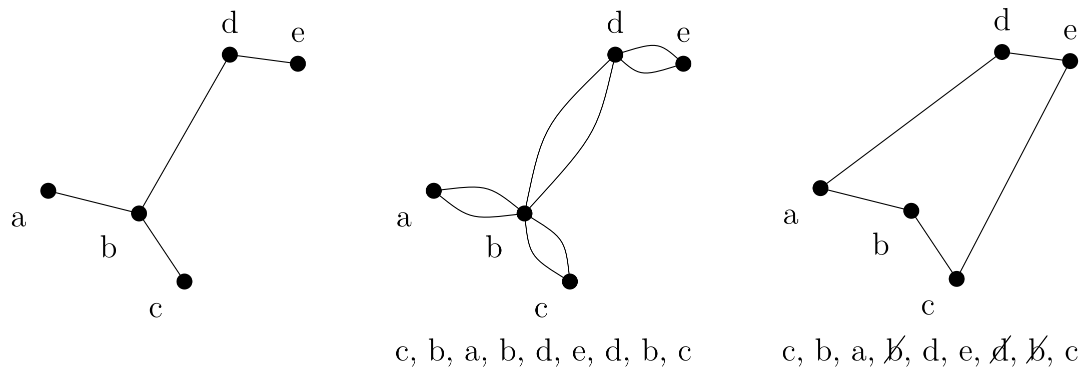
Jádrem algoritmu je zdvojení hran minimální kostry vstupního grafu a následné nalezení uzavřeného eulerovského tahu. To, že se v takovémto grafu takový tah nachází je snadné dokázat. Z Věty 4 víme, že stačí, aby byl graf spojitý a všechny jeho uzly měly sudý stupeň. Zdvojíme-li každou hranu, vycházející z libovolného uzlu, pak bude mít jistě uzel 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 $i_0, i_1, ...i_k$ a ponechejme pro každý uzel pouze jeho první výskyt. Nakonec ještě vraťme i_0 na konec cesty.

Algorithm 3: Double-tree algoritmus

Input: G
$doubleTree \leftarrow FindMinimalSpanningTree(G)$;
foreach $edge \in doubleTree$ **do**
    $doubleTree.Add(edge)$;
$eulerCircuit \leftarrow FindEulerCircuit(doubleTree)$;
vytvoř seznam $path$;
foreach $node \in eulerCircuit$ **do**
    if $node \notin path$ **then**
        $path.Add(node)$;
return $path$

Příklad 16



Obrázek 6: Double-tree příklad

3.2.1 Aproximační faktor

Věta 17

Double-tree je 2-aproximační algoritmus.

Z Lemma 14 víme, že optimální cesta stojí alespoň tolik, co minimální kostra. Jelikož každou její hranu zdvojujeme, získáme dvakrát tak delší cestu.

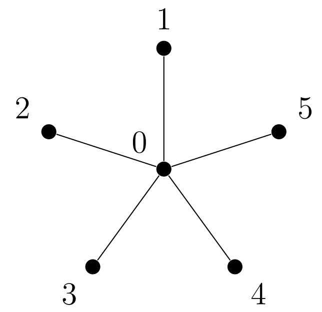
Při hledání eulerovského průchodu pouze hrany seřazujeme.

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

3.2.2 Tight example

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

U takového vstupu může být nalezena následná 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.



Obrázek 7: Double tree tight example - minimální kostra

3.3 Christofides'

Christofidesův algoritmus je velice podobný double-tree algoritmu. Opět začneme s minimální kostrou vstupního grafu. Existenci eulerovského průchodu ale zajistíme 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 které bude $\forall v \in O$ právě jedna hrana, pro kterou je v koncový uzel. Taková množina se nazývá *perfektní párování.* Na množině jich samozřejmě může být více. Pro naše účely bude jistě nejlepší hledat takovou z nich, která budeme mít minimální součet délek všech svých hran, tedy *minimální perfektní párování.*

Uvědomme si ale, že aby takové párování pro O existovalo, tak 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

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

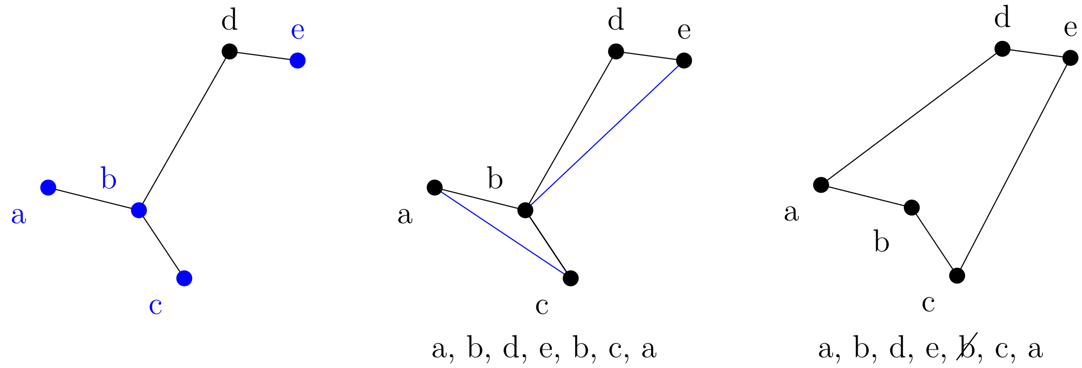
$\sum_{v \in V} deg(v)$ je ale určitě sudé a to samé platí pro $\sum ?_{v \in E}$, tedy $\sum_{v \in O} deg(v)$ musí být také sudý. Vzhledem k tomu, že každý vrchol přispívá do součtu lichým číslem, pak musí být sudý jejich počet.

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
$spanningTree \leftarrow FindMinimalSpanningTree(G)$;
$oddDegreeNodes \leftarrow FindNodesWithOddDegrees(spanningTree)$;
$perfectMathing \leftarrow FindPerfectMatching(oddDegreeNodes)$;
$eulerCircuit \leftarrow FindEulerCircuit(doubleTree + perfectMathing)$;
vytvoř seznam *path*;
foreach *node* $\in$ *eulerCircuit* **do**
    if *node* $\notin$ *path* **then**
        $path.Add(node)$;
return *path*

Příklad 18



Obrázek 8: Christofides' příklad

3.3.1 Aproximační faktor

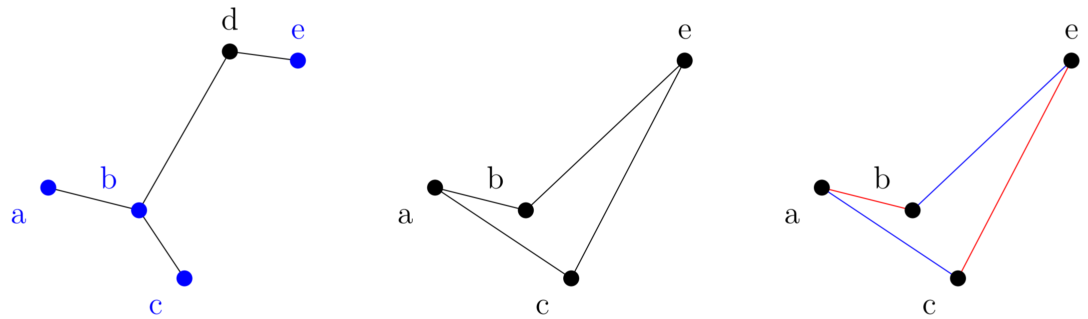
Věta 19

Christofidesův algoritmus je 3/2-aproximační algoritmus.

Nejdřív si musíme rozmyslet, co vlastně potřebujeme dokázat. Potřebujeme, aby eulerovský průchod měl při nejhorším délku $\frac{3}{2}OPT$. Z lemmatu 14 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. Budeme v ní teď hledat cestu. Vyjdeme-li z nejkratší cesty nad všemi vrcholy vstupu, pak je zřejmé, že její délka bude nejvýše OPT. Tuto cestu teď budeme pouze zkracovat. Uvažujeme-li dvě města i a j, pro která platí, že na cestě mezi nimi jsou pouze města 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 délka cesty 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

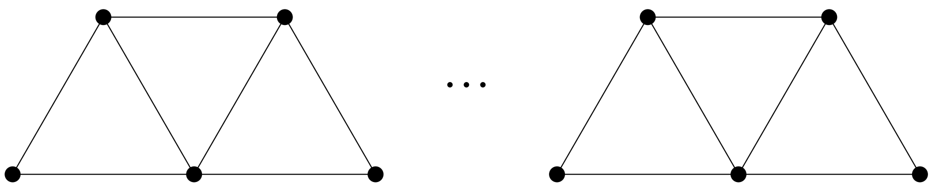
Z analýz aproximačních faktorů Christofidesova a double-tree algoritmů ještě vyplývá následující tvrzení.

Lemma 20

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

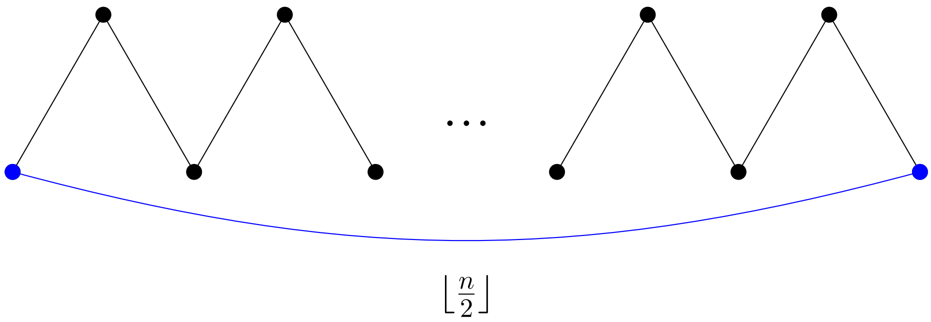
3.3.2 Tight example

Vezměme si následující vstup, kde mají vyznačené hrany délku 1.



Obrázek 10: Christofides' tight example

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 dostanem cestu (zkracování zde jistě nepovede k žádnému zlepšení).



Obrázek 11: Christofides' tight example - 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 Keringhan - Lin

Tento heuristický algoritmus není specifický pro TSP, ale je možné jím řešit obecně problémy, kde máme nějakou podmínku C, kterou musejí splňovat jejich řešení, a funkci f, která takovým řešením přiřazuje určitou hodnotu. Zadanou pak máme množinu S, ve které 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, kde 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, které by přinášely nějaký zisk, tak bylo nalezeno lokálně optimální řešení. V tuto chvíli se buď může skončit nebo se vygeneruje nové počáteční řešení a proces provést znovu.

Samotná transformace probíhá tak, že se hledají takové dvě množiny $\{x_1, x_2, ..., x_k\} \in T$ a $\{y_1, y_2, ..., y_k\} \in S - T$, 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 bude ještě řeč později.

Idea algoritmu tedy spočívá v tom, že pro pseudonáhodné počáteční řešení najdeme 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: Keringhan - Lin algoritmus - obecně

$T \leftarrow$ pseudonáhodné řešení;
$T' \leftarrow T$;
while T' **do**
    $T' \leftarrow$ tranformace T taková, že $f(T') < f(T)$;
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.

Tomu to problému se můžeme vyhnout tím, že k nebude mít žádnou fixní hodnotu. Takového způsobu využijeme my. Jelikož k nebude známé, nemůžeme již jednoduše uvažovat všechny podmnožiny T o velikosti k, jako u předchozí varianty, ale budeme iterativně hledat vždy nejvýhodnější x_i a y_i pro výměnu.

Algorithm 6: Keringhan - Lin algoritmus - variabilní k

$T \leftarrow$ pseudorandom solution;
do
    $i \leftarrow 0$;
    $T \leftarrow T'$;
    while *improvement can be found* **do**
        $i \leftarrow i + 1$;
        $(x_i, y_i) \leftarrow$ best pair for exchange;
        $G_i \leftarrow$ total gain for $\{x_1, x_2, ..., x_k\}$ and $\{y_1, y_2, ..., y_k\}$;
    $k \leftarrow i$, where G_i is max `// (definition of` G_i `is below)`;
    $T' \leftarrow (T - \{x_1, x_2, ..., x_k\}) \cup \{y_1, y_2, ..., y_k\}$;
while $f(T') > f(T)$;
return T';

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

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

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

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

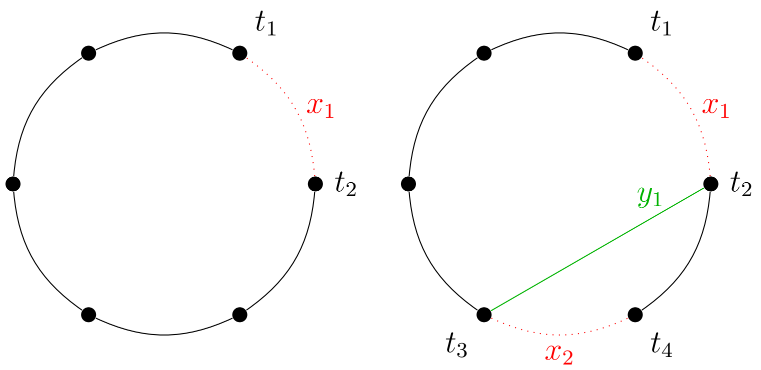
Další neznámou je funkce G_i. Tato funkce udává celkový zisk pro aktuálně nalezené prvky. Určíme-li g_i jako zisk pro (x_i, y_i), pak nejsnazším řešením je za G_i uvažovat $g_1 + g_2 + ... + g_i$. Tímto se také vyhneme tomu, že bychom přestali hledat v momentu, kdy bychom nalezly pár se zápornou hodnotou g_i.

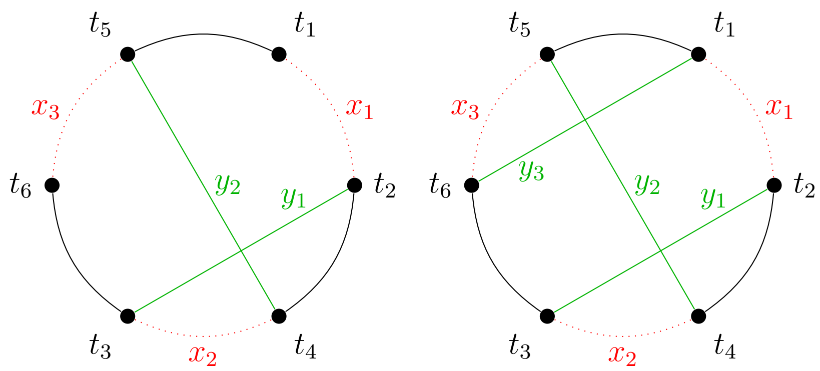
Nakonec ještě musíme nalézt odpovídající *ukončovací podmínku*, která bude udávat, zda-li má smysl pokračovat v hledání nebo jsme již došli do bodu, kdy k dalšímu zlepšení nedojdeme. Nejtěžší úkol je pak najít dobrý balanc mezi schůdným výpočetním časem a prozkoumáváním dostatku příležitostí.

3.4.1 Keringhan - Lin pro TSP

Nejprve si uvědomme že skutečně lze tuto heuristiku 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 okružní cestu mezi vstupními městy. Funkce f pak bude přiřazovat takovým cestám jejich délku.

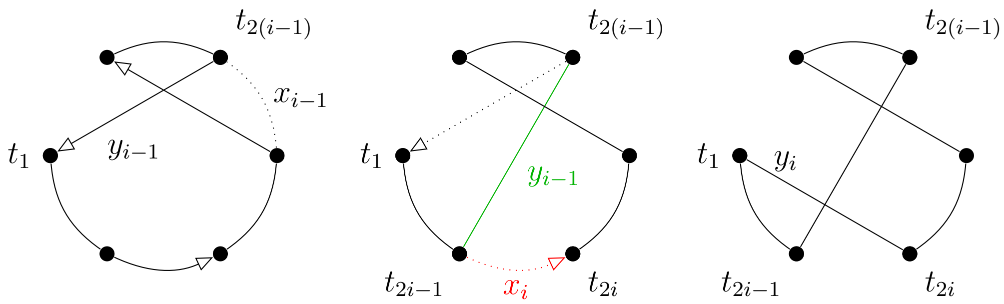
Nyní už se podíváme na to, jak konkrétně bude takový algoritmus vypadat pro TSP. Jeho jádrem je sekvenční hledání párů hran, kde první v původním řešení zrušíme a druhou do nového přidáme. Co je myšleno sekvenčním hledání si nejprve ilustrujeme následujícím obrázkem.



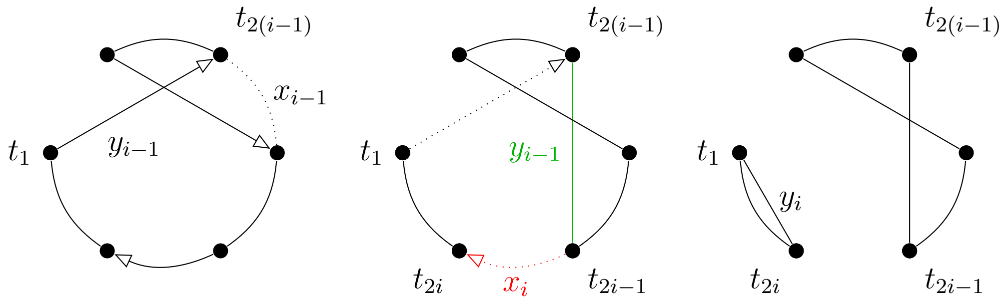


Obrázek 12: Sekvenční hledání hran

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 $\{x_1, x_2, ...x_k\} \cap \{y_1, y_2, ...y_k\} = \emptyset$, *podmínka proveditelnosti* a námi určené *pravidlo výběru*.
Abychom mohli kdykoli s hledáním skončit, musíme pro každý pár ověřit, že po přidání hrany $y_k = (t_{2i}, t_1)$ nám vznikne znovu cesta. V implementaci knihovny je toto řešeno následovně. Při každé iteraci je k dispozici cesta, která vznikla záměnou $\{x_1, x_2, ...x_{i-1}\}$ za $\{y_1, y_2, ...y_{i-1}\}$. Navíc se ještě udržuje hodnota t_1 (ta se v průběhu nemění) a $t_{2(i-1)}$. Cestu znovu otevřeme odstraněním hrany $y_{i-1} = (t_1, t_{2(i-1)})$. Nová hrana $x_i = (t_{2i-1}, t_{2i})$ pak 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, tak přidaná hrana, pak bude mít tvar $y_{i-1} = (t_{2(i-1)}, t_{2i-1})$. Nakonec cestu opět uzavřeme přidáním hrany $y_i = (t_1, t_{2i})$.



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, tak nebere v úvahu délku x_{i+1}. Tohle není ideální, protože x_{i+1} může být výhodná hrana. Navíc se začneme omezovat na lokání informaci o nejbližším vrcholu k t_{2i}. Dobrou vylepšením algoritmu je tedy neposuzovat páry jen na základě rozdílu jejich délek, ale i délky x_{i+1}.
Nakonec nám ještě zbývá určit, za jakých podmínek již se zkracováním cesty přestaneme. Jasnou podmínkou je, že nám dojdou páry, které by vyhovovaly *pravidlu disjunkce* a *podmínce proveditelnosti.* Abychom mohli podmínku zpřísnit, zavedeme si funkci udávající zisk z dosavadně vybraných párů.

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

Nyní už si můžeme ukázat upravenou verzi Keringhan - Lin pro TSP.

Algorithm 7: Keringhan - Lin algoritmus pro TSP

$T \leftarrow$ pseudorandom solution;
$T' \leftarrow T$;
$k = 0$;
$i = 0$;
$G^* = 0$ `// best improvement`;
do
  $i \leftarrow i + 1$;
  find pair $x_i = (t_{2i-1}, t_{2i}), y_i = (t_{2i}, t_{2i+1})$ for which:

  a) $x_i \notin \{y_1, y_2, ...y_i\}$ and $y_i \notin \{x_1, x_2, ...x_i\}$

  b) $T - \{x_1, x_2, ...x_i\} \cup \{y_1, y_2, ...y_i\}$ is a tour

  c) $|x_i| - |y_i|$ is maximal between such pairs

  if *no* x_i, y_i *were found* **then**
    break;
  if $G_{i-1} + c_{t_{2i},t_1} > G^*$ **then**
    $k \leftarrow i$;
    $G^* = G_{i-1} + c_{t_{2i},t_1}$;
  if $G_i \leq G^*$ **then**
    break;
while $G^* > 0$;
$T' \leftarrow (T - \{x_1, x_2, ..., x_k\}) \cup \{y_1, y_2, ..., y_k\}$;
return T';

3.4.2 Random info

Rychlost cca n^2

Závěr

Závěr práce v „českém“ jazyce.

Conclusions

Thesis conclusions in “English”.