Porovnání - Problém obchodního cestujícího (Double-tree + k-OPT)

GAL – GRAFOVÉ ALGORITMY

TOMÁŠ JARŮŠEK - XJARUS02, PATRIK CHUKIR – XCHUKI00

Úvod

Cílem projektu je v programovacím jazyce C++ vytvořit konzolovou aplikaci, která implementuje heuristiky pro řešení problému obchodního cestujícího. Výsledky jednotlivých heuristik jsou následně porovnány mezi sebou z hlediska časové a prostorové složitosti. Pro instance problému s nízkým počtem vrcholů vstupního grafu je také provedeno porovnání kvality výsledné cesty s přesnou verzí algoritmu. V naší implementaci uvažujeme, že ohodnocení hran v grafu respektuje trojúhelníkovou nerovnost.

Problém obchodního cestujícího

Problém obchodního cestujícího je matematicky formulován takto: V daném ohodnoceném úplném grafu najděte nejkratší hamiltonovskou kružnici. To znamená, že na vstupu je graf, který obsahuje n vrcholů a všechny vrcholy jsou propojeny se všemi ostatními. Graf tedy obsahuje $\frac{n(n-1)}{2}$ hran. Všechny tyto hrany musí mít definované ohodnocení. V obecné variantě problému není vyžadováno, aby v grafu platila trojúhelníková nerovnost. Pokud ale platí, mluvíme pak o metrickém problému obchodního cestujícího. Cílem je poté v tomto grafu najít takovou trasu, která prochází všemi vrcholy grafu právě jednou a má co nejkratší ohodnocení.

Algoritmy

Aplikace implementuje tři metody řešení problému obchodního cestujícího. Jedná se o algoritmy *Double-tree* a *k-OPT*, ty k řešení používají heuristiku a tedy negarantují nalezení nejlepšího řešení, pouze se mu snaží co nejvíce přiblížit. Třetí algoritmus je *Brute-force* prohledávání, které garantuje nalezení optimálního řešení.

Brute-force

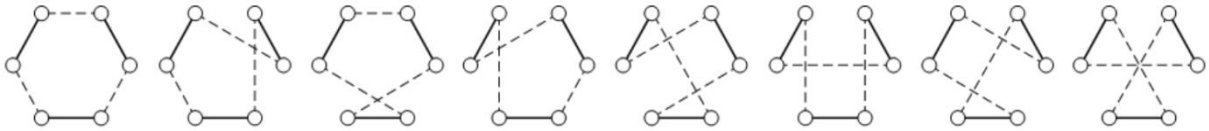
Algoritmus *Brute-force* pracuje na principu prohledávání celého stavového prostoru. Funguje tedy tak, že postupně vygeneruje a vypočítá celkové ohodnocení všech možných permutací propojení vrcholů. U každé permutace si ověří, jestli není lepší než aktuálně nejlepší a pokud ano, nastaví ji jako novou nejlepší. Po dokončení algoritmu je tedy zajištěno, že všechny možné trasy byly vyzkoušeny a výsledek je optimální.

k-OPT

Tato metoda pracuje na principu postupného přepojování k hran za podmínky, že celkové ohodnocení nové trasy s přepojenými hranami je menší a trasa se přitom nerozdělí na více částí.

První krok spočívá ve vytvoření validní počáteční trasy. Na jejím tvaru nezáleží, proto ji zvolíme náhodně, musí ale splňovat to, že se jedná o hamiltonovskou kružnici.

Před dalším krokem je nutné připravit si množinu všech validních přepojení k hran. Nejprve vygenerujeme všechny propojení, kterých může $2k$ hran nabývat. Z tohoto seznamu odfiltrujeme rotace a reverzace. Poté ověříme, že zbylé propojení obsahují k fixních hran a splňují podmínku spojitosti. Výsledná množina hran reprezentuje všechny validní přepojení.



Obrázek 1: Ukázka všech validních přepojení pro 3-opt.

S touto množinou vypočtenou je možno začít přepojovat hrany. Algoritmus postupně postupuje grafem a generuje všechny unikátní k-tice hran, které se v grafu mohou vyskytnout. Pro každou k-tici se vyzkouší všechny její možné přepojení a vypočítá se jejich celkové ohodnocení. Nyní existuje více možností jak postupovat dále. Některé verze algoritmu tímto způsobem vyzkouší všechny k-tice se všemi možnými propojeními a uloží si tu s nejlepším snížením ohodnocením, tu poté přepojí. Naše verze funguje tak, že po nalezení prvního propojení s nižším ohodnocením okamžitě přepojíme a cyklus resetujeme. Tohle se opakuje, dokud nedojdeme do stavu, kdy jsme prošli celým grafem a nenašli žádné přepojení, které by vylepšilo ohodnocení. Zbylá trasa je vrácena jako výsledek.

Double-tree

Algoritmus *Double-tree* pracuje nad úplným grafem, ve kterém platí trojúhelníková nerovnost. Probíhá ve čtyřech krocích:

1. Najít minimální kostru
2. Zdvojit hrany v minimální kostře
3. Najít eulerovský tah ve zdvojené kostře
4. Převést eulerovský tah na hamiltonovskou kružnici

První tři kroky jsou zřejmé nebo řešitelné algoritmy probranými na přednáškách (Primův nebo Kruskalův, hledání eulerovského tahu).

Čtvrtý bod je velice jednoduchý, začneme procházet eulerovský tah z bodu tři, uzel po uzlu, pokud jsme v daném uzlu ještě nebyli, tak leží na hamiltonovské kružnici, jinak jej přeskočíme. Výsledkem je neopakující se posloupnost uzlů, která obsahuje všechny uzly v grafu. Výsledné řešení problému obchodního cestujícího jsou nejkratší hrany spojující uzly v posloupnosti. V rámci toho bodu jsme právě potřebovali, aby vstupní graf byl úplný – cesta mezi kterýmikoliv dvěma uzly existuje – a platila v něm trojúhelníková nerovnost – cesta A → B je vždy kratší než A → C → B, pro libovolné C.

Implementace a výpočet teoretických složitostí

Aplikace je implementována v jazyce C++ a spouští se z příkazové řádky, pro načítání a ukládání grafů je použita knihovna *OGDF*. Vývoj probíhal na školním serveru *Merlin.* Celý projekt, včetně knihovny OGDF, lze přeložit pomocí skriptu *install.sh*. Ten stáhne zdrojové kódy *OGDF*, zkompiluje ji, následně spustí *CMake* a *make* nad projektem. Výstupem skriptu je tedy přímo spustitelné řešení.

Obsahuje dva režimy, jeden pro generování náhodných úplných grafů a druhý pro samotné řešení problému obchodního cestujícího. V režimu generování je výstupem graf uložený ve formátu *.gml*,

jehož *edgeLabel* reprezentuje ohodnocení hrany. V režimu řešení je nutné zvolit algoritmus a jeho parametry spolu se vstupním *.gml* souborem. Výstupní graf může být uložen buď jako *.gml* soubor nebo vykreslen ve formátu *.svg*. K tomu je na standardní výstup vypsáno výsledné ohodnocení a čas běhu. Spuštění programu se provádí následujícím způsobem:

- **./tsp**

Generátor:

- **-g/--generator *path*** – nastaví režim generování grafů a výsledný graf uloží jako *.gml* do *path*
- **-c/--node-count *x*** – nastaví počet uzlů grafu na *x*
- **-s/--output-svg *path*** – nepovinný – uloží výsledný graf jako *.svg* do *path*

Řešení problému obchodního cestujícího:

- **-a/--algorithm *[1,2,3]*** – přijímá hodnoty 1 – 3, 1 → *k-OPT*, 2 → *Double-tree*, 3 → *Brute-force*
- **-k – nepovinný** – pouze pro *k-OPT* – hodnota *k*
- **-i/--input *path*** – vstupní soubor *.gml*
- **-o/--output** – nepovinný – výstupní soubor *.gml*
- **-s/--output-svg** – nepovinný – výstupní soubor *.svg*

Skripty:

- **./generateGraphs *x y*** – vygeneruje grafy s *x* uzly až *y* uzly s růstem o 1 do složky *Graphs*
- **./runTest *out.csv*** – spustí všechny tři algoritmy nad všemi grafy ve složce *Graphs*, pro *k-OPT* s *k* od 2-6

Implementace jednotlivých algoritmů jsou popsány níže, pro výpočet složitostí uvažujeme, že

$n = |V|$ – počet vrcholů

$m = |E|$ – počet hran

Brute-force

Při generování všech validních tras nezáleží na rotaci, tedy různé vrcholy startu trasy nemusíme brát v úvahu. Směr trasy také nehraje roli, ale je efektivnější otestovat i tyto případy, než generovat permutace bez reverzací (faktor navýšení je 2 oproti $n - 1$). Složitost kroku: $O\big((n-1)!\big)$

V naší implementaci provádíme výpočet ohodnocení jednotlivých tras projitím všech jejich hran. Toto by bylo možné udělat v konstantním čase úpravou ohodnocení předchozí trasy, pro účely této aplikace ale není toto urychlení nutné a omezovalo by znovupoužití kódu z jiných algoritmů. Složitost kroku: $O(n)$

Celková časová složitost: $O\big((n-1)!\,n\big) = O(n!\,n)$

Vstupní úplný graf je uložen v matici sousednosti: $\Theta(m+n)$

Aktuální trasa je uložena jako uspořádané pole vrcholů: $\Theta(n)$

Celková prostorová složitost: $\Theta(2n+m) = \Theta(m+n)$

k-OPT

První krok, který musíme provést je výpočet množiny všech validních přepojení k hran. Počet těchto přepojení závislý na k je $2^k(k-1)!$. Generace se skládá ze dvou částí: vygenerování všech možných přepojení bez reverzací (odstranění jedné reverzace je realizováno v lineárním čase) a rotací: $O\big((2k-1)!\,(2k-1)\big) = O(k!\,k)$ a testování, zda přepojení obsahuje všechny fixní spojení: $O(k^2)$. Složitost kroku: $O(k!\,k^3)$

Vygenerování počáteční náhodné trasy: $O(n)$

Samotný běh jednoho kroku algoritmu spočívá ve vygenerování všech k-tic hran, které graf obsahuje. K tomu je potřeba *k* vnořených cyklů o délce *n*. Složitost kroku: $O\big(n^k\big)$

Poté pro každou k-tici otestujeme všechny možné přepojení a spočítáme v lineárním čase nové ohodnocení. Složitost kroku: $O\big(2^k(k-1)!\,k\big)$

Každý krok končí jedním přepojením *k* hran, díky vázanému seznamu je přepojení jedné hrany realizováno v konstantním čase. Složitost kroku: $O(k)$

Kroky se provádí tak dlouho, dokud existuje přepojení, které sníží ohodnocení trasy. To může v nejhorším případě mít až exponenciální složitost, praxi se však provede mnohem méně přepojení. Složitost kroku: $O(k^n)$

Celková časová složitost: $O\big(k!\,k^3 + n + \big(n^k\big(2^k(k-1)!\,k\big) + k\big)k^n\big)$

Vstupní úplný graf je uložen v matici sousednosti:$\Theta(m+n)$

Aktuální trasa je uložena jako obousměrný vázaný seznam: $\Theta(n)$

Celková prostorová složitost: $\Theta(2n+m) = \Theta(m+n)$

Double-tree

Implementace je rozdělena do čtyřech kroků:

1. Najít minimální kostru
   - Využívá implementaci Primova algoritmu v *OGDF* → složitost $O\big(mlog(n)\big)$
   - Z výsledku se staví pole hran v minimální kostře → složitost $O(m)$
2. Zdvojit hrany v minimální kostře
   - projde všechny hrany v minimální kostře → složitost $O(n-1)$
3. Najít Eulerovský tah ve zdvojené kostře
   - inicializace polí pro uzly → složitost $O(n)$
   - přiřazení hran vedoucích z uzlů → složitost $O(m)$
   - projití všech uzlů s kontrolou zda jsem v uzlu již byl → složitost $O(2(n-1)n)$
     - jde o zdvojenou minimální kostru → složitost $O\big(2(n-1)\big)$
     - prohledávání pole navštívených uzlů → složitost $O(n)$

4. Převést eulerovský tah na hamiltonovskou kružnici
    - Průchod eulerovským tahem → složitost $O(2(n-1)+1)$
    - Mazání hran, jenž nejsou na hamiltonově kružnici → složitost $O(m)$

Celková časová složitost: $O\left(2n^2 + m\left(3 + log(n)\right) + 2n\right)$

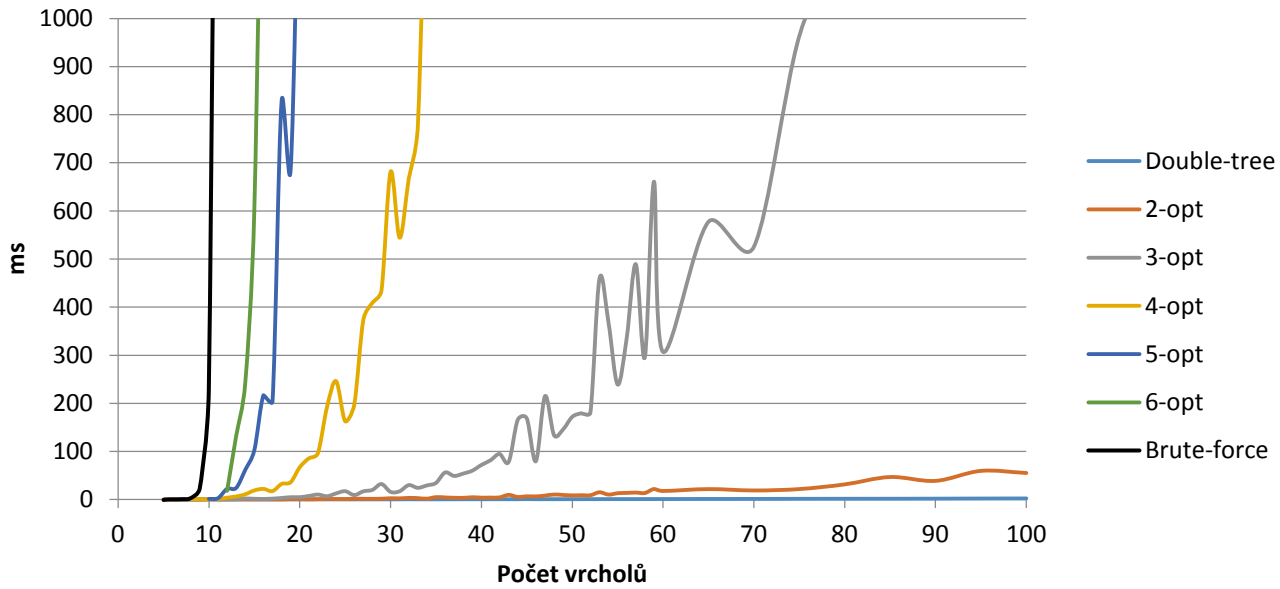
Prostorová:

- pole hran → složitost $\Theta(m)$
- pole uzlů → složitost $\Theta(n)$
- pole pro zdvojenou minimální kostru → složitost $\Theta\left(2(n-1)\right)$
- 2 pole pro eulerovský tah → složitost $\Theta(3n-1)$
- pole pro výsledek eulerovského tahu a pole hamiltonovy kružnice → složitost $\Theta(2n-1+n)$

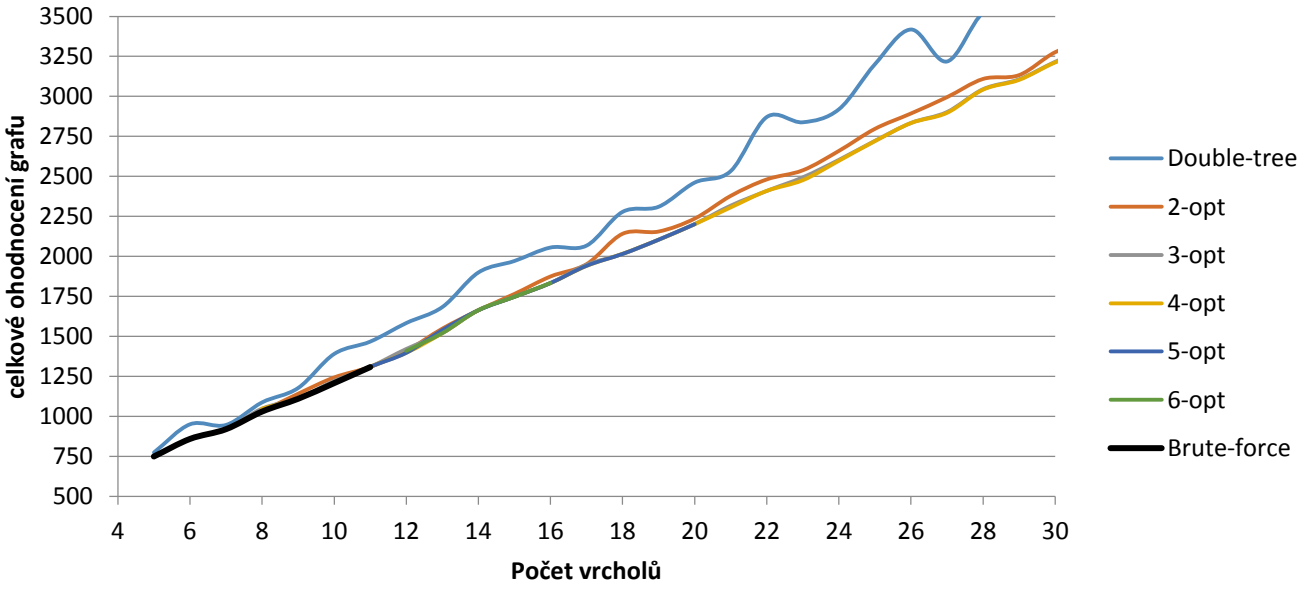
Celková prostorová složitost: $\Theta(m+9n-4)$

Měření a Výsledky

K měření jsme vygenerovali 73 grafů od 5 do 120 uzlů. Následně se nad nimi spustili *Double-tree a k-OPT* algoritmy, s tím že *k-OPT* se testoval pro *k* od 2 do 6. *Brute-force* jsme testovali jen na grafech s 5 – 11 uzly. U všech algoritmů jsme zaznamenávaly do *.csv* souboru čas a cenu výsledné trasy. Testování proběhlo na serveru Merlin.



Graf 1: Délky běhu algoritmů závislé na počtu vrcholů.



Graf 2: Výsledné ohodnocení trasy závislé na počtu vrcholů.

Počet vrcholů	20		40		60		80		100		120	
Čas[ms]	Cena	Čas	Cena	Čas	Cena	Čas	Cena	Čas	Cena	Čas	Cena	Čas
Double-tree	2461	0,314	4785	0,622	7183	1,055	9299	1,612	12431	2,288	14563	2,735
2-OPT	2236	0,869	4261	3,964	6318	21,786	8333	31.566	10376	55,237	12344	123,602
3-OPT	2202	5,172	4187	72,062	6228	307,16	8174	1165,3	10192	2998,5		
4-OPT	2201	72,578	4180	1989,98	6201	21317,7	8161	65625,6	10158	225535		
5-OPT	2201	1860,2	4193	74640								
6-OPT	2201	68505										

Tabulka 1: Vybrané naměřené hodnoty.

Z grafů je patrné, že z pohledu ceny *Double-tree* vrací nejméně optimální řešení. Naopak čím vyšší stupeň *k-OPT* algoritmu, tím se více výsledek blíží optimálnímu řešení. Tohle je však vykoupeno rapidním nárůstem časové složitosti.

Složitost algoritmu *Double-tree* je kvadratická, stejně jako jeden krok algoritmu *2-opt*. Proto ve výsledcích vidíme, že časy běhu se podobají, *2-opt* však zaostává kvůli tomu, že se tento krok musí provést vícekrát. Počet kroků může být v nejhorším případě až exponenciální, v naše testování se ale jevil jako lineární. Algoritmus *k-OPT* poskytuje znatelně lepší výsledky, pokud ale vezmeme v úvahu poměr ohodnocení výsledné trasy k času, tak se *Double-tree* jeví jako nejefektivnější řešení pro grafy s vysokým počtem vrcholů.

Navyšování *k* u *k-OPT* vede ke zlepšení výsledku (není to však garantováno, jak můžeme vidět v tabulce 1), avšak za velký nárůst času běhu. To je dáno tím, že složitost kroku je $O(n^k)$. Z toho vyplývá, že navyšování *k* dává smysl jen tehdy, pokud je naším cílem nalezení nejkratší trasy bez ohledu na čas.

Časy běhu *k-OPT* byly měřeny bez prvotního nalezení množiny přepojení, ta ale nemalým dílem přispívá k době běhu při navýšení *k*, jak můžeme vidět v tabulce 2. Protože se však dá předpočítat a znovupoužít pro různé grafy, nebyl její výpočet do času běhu započten.

k	2	3	4	5	6
Čas[ms]	0,011346	0,134357	5,546548	398,3772	48318,26

Tabulka 2: Časy potřebné pro nalezení množiny validních přepojení.

Algoritmus *Brute-force* dle očekávání poskytuje optimální výsledky, díky však jeho složitosti $O(n!\, n)$ je jeho použití nepraktické už pro grafy s 12 uzly.

Pokud porovnáme výsledky *k-OPT* a *Double-Tree* s přesným *Brute-force* tak vidíme, že v našem testování *Double-tree* nikdy nevrátil přesný výsledek, naopak *k-OPT* vrátil stejný nebo velice blízký. Tuto závislost by bylo potřeba otestovat pro grafy s více vrcholy, to však kvůli *Brute-force* nebylo možné. Je ale zřejmé, že *k-OPT* se více blíží optimálnímu řešení než *Double-tree.*

Zdroje

Pro vypracování projektu byly použity tyto zdroje:

1. http://rtime.felk.cvut.cz/~hanzalek//KO/TSP_e.pdf
   https://cs.wikipedia.org/wiki/Problém_obchodního_cestujícího
2. https://www.algoritmy.net/article/5407/Obchodni-cestujici
3. https://pdfs.semanticscholar.org/ab7c/c83bb513a91b06f6c8bc3b9da7f60cbbaee5.pdf
4. https://stackoverflow.com/questions/960557/how-to-generate-permutations-of-a-list-without-reverse-duplicates-in-python-us
5. https://www.reddit.com/r/compsci/comments/3wafau/tsp_2opt_complexity/
6. https://en.wikipedia.org/wiki/2-opt
7. https://en.wikipedia.org/wiki/3-opt