# 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](https://cs.wikipedia.org/wiki/Graf_(teorie_grafů)) najděte nejkratší [hamiltonovskou kružnici](https://cs.wikipedia.org/wiki/Hamiltonovská_kružnice). To znamená, že na vstupu je graf, který obsahuje n vrcholů a všechny vrcholy jsou propojeny se všemi ostatními. Graf tedy obsahuje ((n-1)\*n)/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-froce prohledávání, které garantuje nalezení optimálního řešení.

## Brute-force

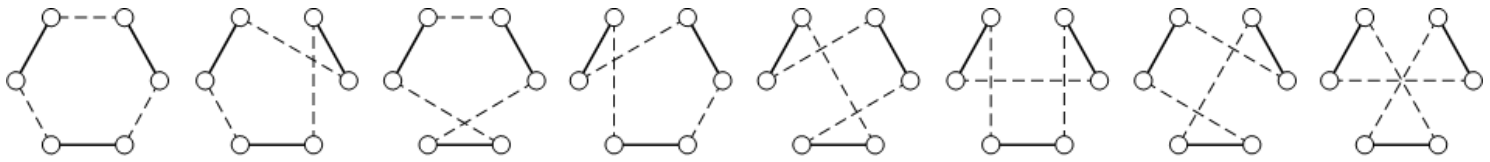
Algoritmus brute-force pracuje na principu prohledávání celého stavového prostoru. Funguje tedy tak, že postupně vygeneruje a vypočítá celkové ohodnocení všech možných permutací propojení vrcholů. U každé permutace si ověří, jestli není lepší než aktuálně nejlepší a pokud ano, nastaví ji jako novou nejlepší. 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 permutace všech propojení, které může 2\*k hran nabývat. Z tohoto seznamu odfiltrujeme rotace a reverzace. Poté ověříme, že zbylé permutace obsahují k fixních hran a splňují podmínku spojitosti. Výsledná množina hran reprezentuje všechny validní přepojení.

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 Eulorovský tah na Hamiltnovskou kružnici

První tři kroky jsou zřejmé nebo řešitelné algoritmy probranými na přednáskách(Primův nebo Kruskalův, Hledání eulorvského tahu).

4. Bod je velice jednoduchý začneme procházet eulerovský tah z bodu tři, uzel po uzlu, pokud jsme v daném uzlu ještě nebyli, tak leží na hamiltnovské kružnici, jinak jej přeskočíme. Výsledkem je neopakující se posloupnost uzlů, která obsahuje všechny uzly v grafu. Výsledné řešení TSP 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.

Složitost

# Implementace a výpočet teoretických složitostí

Aplikace je implementována v jazyce C++, pro načítání a ukládání grafů je použita knihovna OGDF. Naše řešení obsahuje tři algoritmy Brute force, k-OPT a Double tree, plus několik pomocných skriptů a generátor grafů – generuje úplné grafy, splňující trojúhelníkovou nerovnost, pomocí ODGF a úkládá je do .glm, případně i do .svg.

V – počet vrcholů

E – počet hran

## Ovládání

Aplikace nachystána na spouštění z příkazové řádky. V linuxu je možné zkompilovat aplikaci pomocí skriptu install.sh, který stáhne zdrojové kódy OGDF knihovny, zkompiluje ji, poté pomocí *cmaku* sestaví makefile pro projekt a zkompiluje projekt i s nalikovanou knihovnou OGDF. Na windosech bude třeba provést stejné kroky jen ručně.

Po zkopilování je možno aplikaci obsluhovat pomocí parametrů:

Generátor:

* -g/--generator path – nastavý režim generování graphů výsledný graph uloží jako .glm do path
* -c/--node-count x – nastavý počet uzlů grafu na x
* -s/--output-svg path – nepoviný – uloží výsledný graf i jako .svg do path

Řešení TSP:

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

Skripty:

./generateGraphs x y – vygeneruje grafy s x uzly až y uzly s růstem o 1 do složky Graphs

./runTest out.csv – pustí všechny tři algoritmy nad všemi grafy ve složce Graphs, pro k-OPT s k od 2-6

## Brute-force

Test

Časová složitost:

Při generování všech validních tras nezáleží na rotaci, tedy různé vrcholy startu trasy nemusíme brát v úvahu. Tedy počet permutací = (V-1)!. Protože ale nezáleží ani na směru trasy je toho číslo ještě nutné vydělit dvěma. Počet unikátních permutací v grafu s V vrcholy je výsledku roven ( (V-1)!/2).

Pro každou tuto permutaci je následně nutné vypočítat metriku postupným sečtením metrik jednotlivých hran, což zabere V kroků.

Výsledná časová složitost je tedy O(( (V-1)!/2)\*V)

prostorova asi V+E

## k-OPT

Test

## Double-tree

Implementace je rozdělena do čtyřech krocích:

1. Najít minimální kostru

* Využívá implementaci Primova algoritmu v OGDF → složitost
* Z výsledku se taví pole hran v minimální kostře → složitost

1. Zdvojit hrany v minimální kostře

* projde všechny hrany v minimální kostře → složitost

1. Najít Eulerovský tah ve zdvojené kostře

* inicializace polí pro uzly → složitost
* přiřazení hran vedoucích z uzlů → složitost
* projití všech uzlů s kontrolou zda jsem v uzlu již byl → složitost
  + jde o zdvojenou minimální kostru → složitost
  + prohledávaí pole navštívených uzlů → složitost

1. Převést Eulorovský tah na Hamiltnovskou kružnici

* Průchod Eulorovským tahem → složitost
* Mazání hran jenž nejsou na hamiltnově krožnici → složitost

Výsledná složitost

Prostorová:

* pole hran → složitost
* pole uzlů → složitost
* pole pro zdvojenou minimalní kostru → složitost
* 2 pole pro Eulerovský tah → složitost
* pole pro výsledek Eulorovského tahu a pole hamiltnovy kružnice → složitost

Výsledná složitost

# Měření a Výsledky

Meřili jsme to tak a tak ….

par grafů….

Meření nám potvrdilo teoretickou složitost odvozenou v kapitole implementace….

# Zdroje

zformatovat

# http://rtime.felk.cvut.cz/~hanzalek//KO/TSP\_e.pdf

https://cs.wikipedia.org/wiki/Úplný\_graf  
<https://cs.wikipedia.org/wiki/Problém_obchodního_cestujícího>

<https://www.algoritmy.net/article/5407/Obchodni-cestujici>

<https://pdfs.semanticscholar.org/ab7c/c83bb513a91b06f6c8bc3b9da7f60cbbaee5.pdf>

https://stackoverflow.com/questions/960557/how-to-generate-permutations-of-a-list-without-reverse-duplicates-in-python-us

https://www.geeksforgeeks.org/find-the-next-lexicographically-greater-word-than-a-given-word/