VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA PODNIKATELSKÁ
ÚSTAV INFORMATIKY

FACULTY OF BUSINESS AND MANAGEMENT
INSTITUT OF INFORMATICS

EVOLUČNÍ ALGORITMY PŘI ŘEŠENÍ PROBLÉMU OBCHODNÍHO CESTUJÍCÍHO

EVOLUTIONARY ALGORITHMS FOR THE SOLUTION OF TRAVELLING SALESMAN PROBLEM

DIPLOMOVÁ PRÁCE
MASTER‘S THESIS

AUTOR PRÁCE
AUTHOR
Bc. LUKÁŠ JURČÍK

VEDOUCÍ PRÁCE
SUPERVISOR
prof. Ing. PETR DOSTÁL, CSc.

BRNO 2014

Vysoké učení technické v Brně
Fakulta podnikatelská

Akademický rok: 2013/2014
Ústav informatiky

ZADÁNÍ DIPLOMOVÉ PRÁCE

Jurčík Lukáš, Bc.

Informační management (6209T015)

Ředitel ústavu Vám v souladu se zákonem č.111/1998 o vysokých školách, Studijním a zkušebním řádem VUT v Brně a Směrnicí děkana pro realizaci bakalářských a magisterských studijních programů zadává diplomovou práci s názvem:

Evoluční algoritmy při řešení problému obchodního cestujícího

v anglickém jazyce:

Evolutionary Algorithms for the Solution of Travelling Salesman Problem

Pokyny pro vypracování:

Úvod
Vymezení problému a cíle práce
Teoretická východiska práce
Analýza problému a současné situace
Vlastní návrhy řešení, přínos návrhů řešení
Závěr
Seznam použité literatury
Přílohy

Seznam odborné literatury:

DOSTÁL, P. Pokročilé metody analýz a modelování v podnikatelství a veřejné správě. 1. vyd. Brno: CERM, 2008. 340 s. ISBN 978-80-7204-605-8.
DOSTÁL, P. Advanced Decision Making in Business and Public Services. Brno : CERM, 2011. 168 s., ISBN 978-80-7204-747-5.
DOSTÁL, P. The Use of Soft Computing in Management. In Vasant, P. Handbook of Research on Novel Soft Computing Intelligent Algorithms: Theory and Practical Applications USA: IGI Globe, 2013. ISBN13 9781466644502.
THE MATHWORKS. MATLAB – User's Guide. The MathWorks, Inc. 2013.

Vedoucí diplomové práce: prof. Ing. Petr Dostál, CSc.

Termín odevzdání diplomové práce je stanoven časovým plánem akademického roku 2013/2014.

L.S.

doc. RNDr. Bedřich Půža, CSc. Ředitel ústavu	doc. Ing. et Ing. Stanislav Škapa, Ph.D. Děkan fakulty

V Brně, dne 29.04.2014

Abstrakt

Diplomová práce se zabývá problematikou evolučních algoritmů na problému obchodního cestujícího (TSP). V první části jsou uvedeny teoretické základy z teorie grafů a složitosti algoritmů. Následuje část věnující se vybraným optimalizačním metodám. Cílem práce je vytvořit aplikaci, která řeší problém TSP s použitím evolučních algoritmů.

Abstract

This diploma thesis deals with evolutionary algorithms used for travelling salesman problem (TSP). In the first section, there are theoretical foundations of a graph theory and computational complexity theory. Next section contains a description of chosen optimization algorithms. The aim of the diploma thesis is to implement an application that solve TSP using evolutionary algorithms.

Klíčová slova

Problém obchodního cestujícího, evoluční algoritmy, Matlab, Hamiltonovská kružnice, optimalizace mravenčí kolonií, kukaččí algoritmus, Random Search, Exhaustive Search, hladové vyhledávání, horolezecký algoritmus, teorie složitosti, NP-úplné problémy.

Keywords

Travelling Salesman Problem, evolutionary algorithms, Matlab, Hamiltonian Cycle, Ant Colony Optimization, Cuckoo Search, Random Search, Exhaustive Search, Greedy Search, Hill Climbing, Computational complexity theory, NP-complete problems.

Bibliografická citace

JURČÍK, L. *Evoluční algoritmy při řešení problému obchodního cestujícího.* Brno: Vysoké učení technické v Brně, Fakulta podnikatelská, 2014. 79 s. Vedoucí diplomové práce prof. Ing. Petr Dostál, CSc.

Čestné prohlášení

Prohlašuji, že předložená diplomová práce je původní a zpracoval jsem ji samostatně. Prohlašuji, že citace použitých pramenů je úplná, že jsem ve své práci neporušil autorská práva (ve smyslu Zákona č. 121/2000 Sb., o právu autorském a o právech souvisejících s právem autorským).

V Brně dne 21. května 2014

..…….
podpis studenta

Poděkování

Chtěl bych poděkovat vedoucímu mé diplomové práce prof. Ing. Petru Dostálovi, CSc. za cenné připomínky, rady a čas, který mi věnoval při vzniku této práce. Dále bych chtěl poděkovat své rodině za podporu při mém studiu.

Obsah

Úvod

Tématem této práce je řešení problému obchodního cestujícího (TSP). Jedná se logistickou úlohu, která má efektivně navrhnout cestu skrz dané množství míst a znovu se navrátit do výchozího bodu. Typicky tento problém řeší řada firem a jejich obchodní cestující při návštěvách jednotlivých měst. Může se ale vztáhnout i do jiného prostředí. Nemusí jít o města v jednom státě, kraji nebo světadíle, ale např. o jednotlivé domy v rámci jedné ulice apod. Snahou je minimalizovat náklady, které obchodní cestující musí vynaložit na své cestě, peníze na benzín, čas strávený cestováním, který mohl být využit jinou prací atd.

Pro řešení lze použít řadu metod a postupů, od náhodného výběru přes různé matematické a statistické metody. V této práci se zaměřuji na metody založené na evolučních algoritmech, které jsou inspirovány jevy a procesy v přírodě.

Nejdříve bude problém TSP vysvětlen formálně s použitím názvosloví z teorie grafů. Další část se zaměřuje na problematiku složitosti úloh a zařazení problému obchodního cestujícího do této kategorie.

Následuje část věnována evolučním algoritmům, jejich původu, vysvětlení základních pojmů a rozdělení jednotlivých optimalizačních úloh s jejich popisem.

Poslední část se věnuje programu Matlab, ve kterém bude vytvořena aplikace využívajících optimalizačních metod představených v předchozí kapitole.

1 Vymezení problému a cíle práce

Problém obchodního cestujícího je stále aktuální problematika v řadě oborů a podnikatelské činnosti, jako je logistika, doprava, výroba polovodičových prvků atd. Z ekonomického hlediska nalezení nejkratší cesty znamená úsporu materiálu, času a peněz pro daný podnik. V této práci se zaměřím na řešení problému pomocí evolučních algoritmů, jejich vzájemnému porovnání mezi sebou i s dalšími typy optimalizačních metod.

1.1 Cíle

Cílem práce je vytvořit aplikaci, která bude řešit problém obchodního cestujícího pomocí evolučních algoritmů. Spolu s dalšími optimalizačními metodami pak bude provedena řada testů, na jejichž základě dojde k vyhodnocení použitelnosti evolučních algoritmů v praxi.

V teoretické části práce je cílem objasnit problematiku evolučních algoritmů a rozebrat přístupy několika nejběžnějších optimalizačních úloh.

1.2 Metodika práce

Aplikace, na jejíž vytvoření se práce zaměřuje, je vytvořena v programu Matlab metodou Switched board programming. Popis jednotlivých algoritmů, které program porovnává, je v kapitole 2.4.3.

Pro sběr testovacích dat byla zvolena technika analýzy dokumentů a experimentů. Část vstupních dat reprezentuje rozmístění největších měst České republiky. Dalším typem vstupních dat postavených na technice experimentů je specifické rozmístění bodů, náhodné rozmístění, uspořádání do pravidelné mřížky nebo do podoby kruhu.

2 Teoretická východiska práce

Obchodní cestující[1] má na starost nákup a prodej zboží a služeb koncovým zákazníkům nebo organizacím. S tím souvisí další jeho činnost, která často vyžaduje cestovat za zákazníky. Problém obchodního cestujícího – TSP (Travelling Salesman Problem) je typ úlohy, která řeší plánování navštívení nějakého množství *X* měst a návrat do výchozího města. Úkolem tedy je rozhodnout jaké město a v jakém pořadí navštívit, aby celková trasa byla co nejkratší.

Důležitost správného naplánování se projeví u většího množství měst nebo u větších vzdáleností mezi nimi. Z ekonomického hlediska tak může nastat situace, kdy špatně zvolenou cestou může celková uražená dráha být i o tisíce kilometrů delší než optimální trasa. To se pak projeví delším časem nutným k cestování, který mohl obchodní cestující strávit jinou prací. Delší cesta má také za následek vyšší spotřebu benzínu a vyšší opotřebení použitého dopravního prostředku, které se opět projeví vyšší výslednou cenou. Optimalizací cesty obchodního cestujícího tak můžeme snížit výdaje daného podniku.

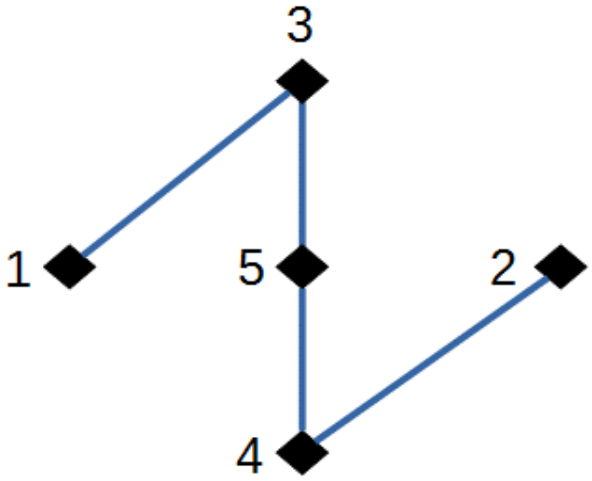
2.1 Základy teorie grafů

Hledání optimální cesty v problému obchodního cestujícího (TSP) je vlastně hledání nejkratší hamiltonovské kružnice v úplném grafu, kde jednotlivá města reprezentují uzly a cesty vedoucí mezi městy představují hrany. V následující části proto budou formálně představeny základní pojmy z teorie grafů používané při popisování problému obchodního cestujícího [7].

2.1.1 Graf

Graf je uspořádaná dvojice $G = (V, E)$, kde V je množina vrcholů a E je množina hran [8]. Příklad jednoduchého grafu můžeme vidět na obrázku 2-1. Je definován takto: $V = \{1,2,3,4,5\}$, $E = \{\{1,3\},\{2,4\},\{3,5\},\{4,5\}\}$.

[1] Charakteristika povolání obchodního cestujícího: http://www.infoabsolvent.cz/Povolani/Karta/102007



Obrázek 2-1: Ukázka jednoduchého grafu. Zdroj: vlastní.

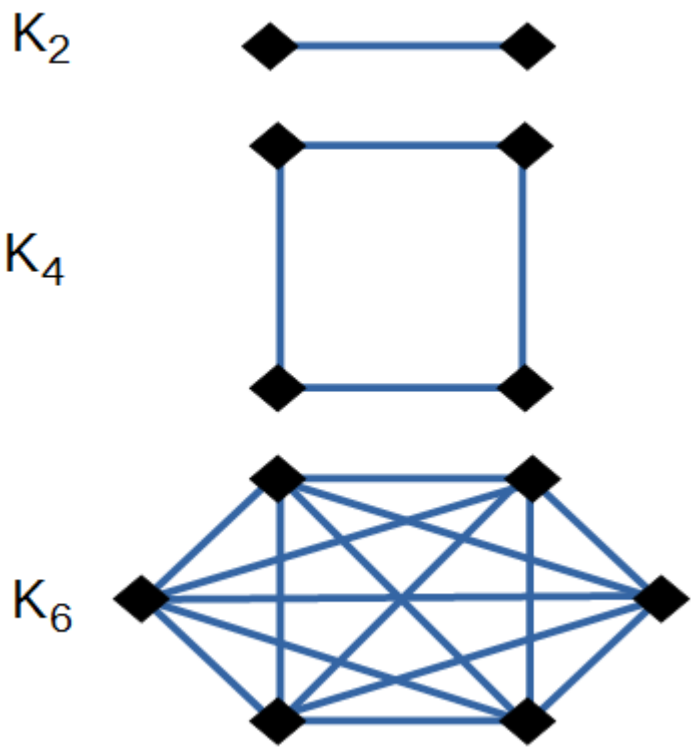
2.1.2 Délka cesty

Pro graf z obrázku 2-1 je délka cesty 4. Počet vrcholů má 5 a hran 4. Obecně vyjádřeno, délka cesty *n* obsahuje *n+1* vrcholů a *n* hran, jak je zobrazeno na obrázku 2-2 [8].

Obrázek 2-2: Zobrazení délky cesty grafu. Zdroj: upraveno na základě [8] str. 2.

2.1.3 Úplný graf

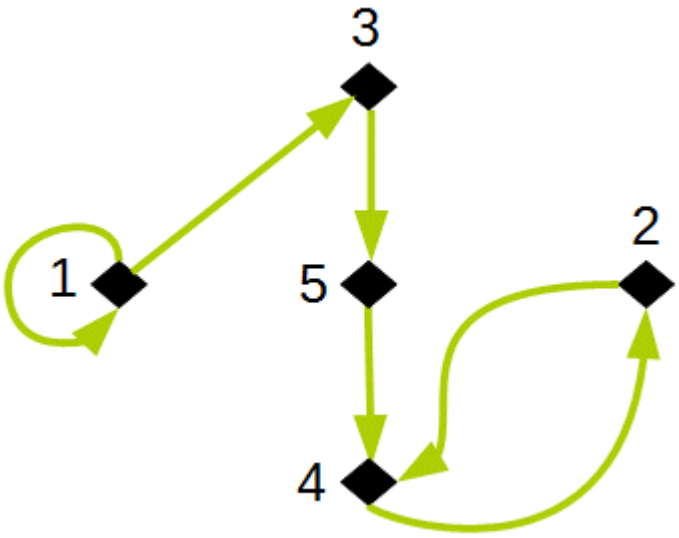
Úplný graf je takový, který má navzájem pospojované všechny hrany. Značí se K_n pro *n* vrcholů. Obsahuje $\binom{n}{2}$ hran. To platí pro grafy s $n \geq 1$ počtem hran. Na obrázku 2-3 je ukázka tří úplných grafů pro počet vrcholů 2,4 a 6 [8].



Obrázek 2-3: Zobrazení úplného grafu. Shora: úplný graf pro 2,4 a 6 vrcholů. Zdroj: vlastní.

2.1.4 Orientovaný graf

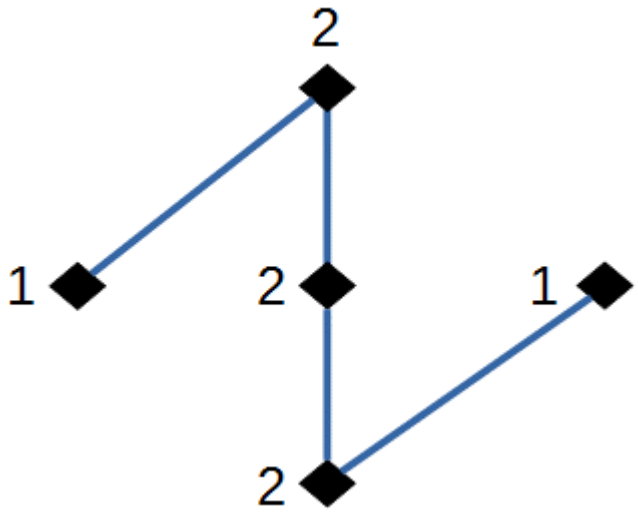
Problém obchodního cestujícího je popisován neorientovaným grafem. Přesto zde vysvětlím pro úplnost i pojem orientovaného grafu. Orientovaný graf obsahuje u hran šipky, které rozlišují směrování dané hrany z jednoho bodu do druhého. Formálně zapsáno, orientovaný graf je uspořádaná dvojice $D = (V, E)$, kde V je množina vrcholů a E je uspořádaná dvojice vrcholů $E \subseteq V \times V$. Hrana *(u,v)* se zobrazí jako šipka z vrcholu *u* do *v*. Není to tedy to samé jako hrana *(v,u)*! Označení *(u,u)* pak symbolizuje smyčku [8]. Příklad orientovaného grafu D je na obrázku 2-4. $V = \{1,2,3,4,5\}$, $E = \{\{1,1\},\{1,3\},\{3,5\},\{5,4\},\{4,2\},\{2,4\}\}$.



Obrázek 2-4: Ukázka orientovaného grafu včetně ukázky smyčky ve vrcholu 1 a rozdílu hrany (2,4) a (4,2). Zdroj: vlastní.

2.1.5 Stupeň vrcholu

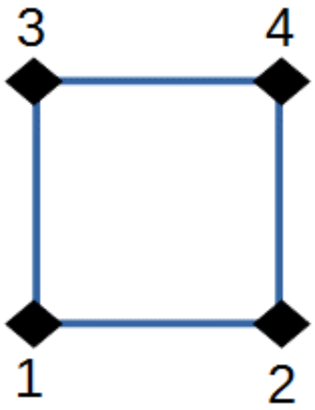
Mějme graf G. Stupeň vrcholu vyjadřuje počet hran vycházejících z vrcholu v. Značí se následovně: $d_G(v)$ [8]. Stupeň vrcholu se může zapsat přímo u daného vrcholu jako na obrázku 2-5, kde čísla neznačí název vrcholu, ale jeho stupeň, tedy kolik hran z něj vychází.



Obrázek 2-5: Ukázka grafu s vyznačenými stupni vrcholů místo názvů. Zdroj: vlastní.

2.1.6 Kružnice

Kružnicí v grafu G označíme takové spojení vrcholů, které vytvoří cyklus. Délku kružnice n pak vyjadřuje počet hran, které tvoří takový cyklus. V tomto případě musí být splněna podmínka, že graf obsahuje $n \geq 3$ vrcholů [8]. Jednoduchá kružnice v grafu je zobrazena na obrázku 2-6.



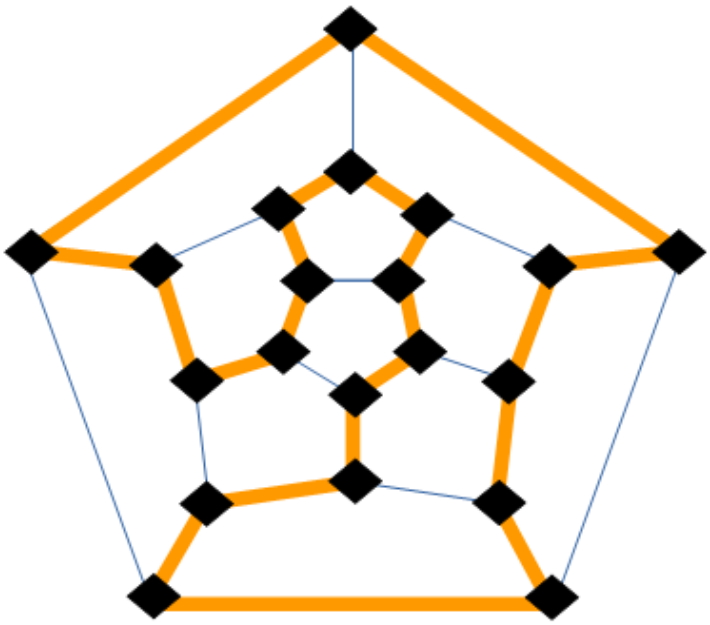
Obrázek 2-6: Ukázka kružnice v grafu. Délka kružnice je 4. Zdroj: vlastní.

2.1.7 Hamiltonovský graf a kružnice

Podle *Diracovy podmínky* v [8] je Hamiltonovský graf takový, který má $n \geq 3$ vrcholů a každý vrchol je minimálně stupně $\frac{n}{2}$. Jinak můžeme definovat Hamiltonovský graf jako takový, který obsahuje nějakou Hamiltonovskou kružnici.

Hamiltonovská kružnice je kružnicí v grafu, která prochází všemi vrcholy právě jednou. Podle této definice by se mohlo zdát, že je celkem jednoduché nalézt v grafu Hamiltonovskou kružnici, ale není tomu tak. Tato úloha patří mezi NP-úplné problémy, viz kapitola 2.2.3. Název je odvozen od matematika W. R. Hamiltona, který vymyslel hlavolam mnohostěnu o 20 vrcholech, kde cílem bylo nalézt právě Hamiltonovskou kružnici. Tomuto útvaru se říká dvanáctistěn. Podobu tohoto hlavolamu je možné vidět na obrázku 2-7.

Dále se můžeme setkat s pojmem Hamiltonovská cesta, která prochází všemi vrcholy grafu právě jednou, ale nevrací se zpět a netvoří tak kružnici v grafu [9].



Obrázek 2-7: Graf o 20 vrcholech (dvanáctistěn) se zobrazenou Hamiltonovskou kružnicí. Zdroj: upraveno podle [9] obr. 6.2, str. 73.

2.2 Složitost

Teorie složitosti zkoumá, zda jsou různé úlohy řešitelné. Zabývá se tématikou náročnosti na zdroje a čas. Jak je daný algoritmus efektivní. Pokud nám jde jen o odpověď na ano/ne pro splnění nějaké podmínky, pak se jedná o rozhodovací problémy. Pokud hledáme např. nejkratší cestu v grafu, jde o optimalizační problém nebo se můžeme zabývat vyhledávacími problémy, kde hledáme část splňující zadanou vlastnost.

Při výpočtu daného problému či úlohy, nebereme v potaz, na jakém hardwaru pracujeme a jaké přitom využíváme softwarové vybavení. Samotný výpočet spotřebovává nějaké zdroje a podle toho rozlišujeme nejčastěji *časovou složitost* a *paměťovou (prostorovou) složitost*.

V teorii složitosti nás tedy nezajímá konkrétní hardwarové ani softwarové zařízení. Časová složitost vyjadřuje počet operací (kroků) nutných k provedení dané činnosti. Prostorová (paměťová) složitost vyjadřuje počet bitů potřebných k uložení dat během výpočtu [10].

2.2.1 Asymptotická časová složitost

Asymptotická časová složitost patří mezi často používané kritérium pro hodnocení algoritmů. Je vyjádřena funkcí o *N* prvcích, se kterou je algoritmus porovnáván. Rozlišujeme tři složitosti: *horní hranici chování*, značí se velkým řeckým písmenem omikron O, *dolní hranici chování* Ω (omega) a *třídou chování* Θ (theta).

Horní hranice časového chování O(g(n)) značí množinu funkcí $f(n)$, pro kterou platí: $\{f(n): \exists(c > 0,\ n_0 > 0)\ takové, že\ \forall(n \geq n_0)\ platí: [0 \leq f(n) \leq c \cdot g(n)]\}$, kde c a n_0 jsou kladné konstanty. *„Pak zápis $f(n) = O(g(n))$, označuje, že funkce $f(n)$ roste maximálně tak rychle, jako funkce $g(n)$. Funkce $g(n)$ je horní hranicí množiny takových funkcí, určené zápisem $O(g(n))$“* [17].

Dolní hranice časového chování Ω(g(n)) značí množinu funkcí $f(n)$, pro kterou platí: $\{f(n): \exists(c > 0,\ n_0 > 0)\ takové, že\ \forall(n \geq n_0)\ platí: [0 \leq c \cdot g(n) \leq f(n)]\}$, kde c a n_0 jsou kladné konstanty. *„Pak zápis $f(n) = \Omega(g(n))$, označuje, že funkce $f(n)$ roste minimálně tak rychle, jako funkce $g(n)$. Funkce $g(n)$ je dolní hranicí množiny všech funkcí, určených zápisem $\Omega(g(n))$“* [17].

Třída časového chování Θ(g(n)) značí množinu funkcí $f(n)$, pro kterou platí: $\{f(n): \exists(c_1 > 0,\ c_2 > 0,\ n_0 > 0)\ takové, že\ \forall(n \geq n_0)\ platí: [0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)]\}$, kde c_1, c_2 a n_0 jsou kladné konstanty. *„Pak zápis $f(n) = \Theta(g(n))$, označuje, že funkce $f(n)$ roste tak rychle, jako funkce $g(n)$. Funkce $g(n)$ vyjadřuje horní a současně dolní hranicí množiny funkcí, označených zápisem $\Theta(g(n))$“* [17].

Nyní můžeme provést následující klasifikaci často se vyskytujících složitostí algoritmů [17]:

- $\Theta(1)$ – konstantní
- $\Theta(\mathrm{n})$ – lineární
- $\Theta(\log(\mathrm{n}))$ – logaritmická
- $\Theta(\mathrm{n} \cdot \log(\mathrm{n}))$ – lineárně-logaritmická
- $\Theta(n^2)$ – kvadratická
- $\Theta(n^3)$ – kubická
- $\Theta(k^n)$ – exponenciální, pro $k > 0$

Vliv řádu a kardinality úlohy je zobrazen v tabulce 2-1, kde *n* udává velikost vstupu. Výpočet jedné operace uvažujeme 1 *ns*. Počítač tak vykoná 1 miliardu operací za vteřinu. Z tabulky je patrné, že exponenciální a faktoriálové úlohy jsou pro vyšší počet vstupů nepoužitelné.

Úlohy můžeme ještě dále rozdělit na polynomiální a nepolynomiální. Mezi polynomiální patří lineární, logaritmické, kvadratické atd. Pro tyto typy úloh můžeme rostoucí složitost kompenzovat například použitím výkonnějšího počítače. U nepolynomiálních úloh (exponenciální, faktoriálové) roste složitost exponenciálně, jak dokládají údaje v tabulce 2-1. Těmto typům úloh se proto snažíme vyhýbat [18].

Tabulka 2-1: Příklad doby výpočtu funkce g(n). Výpočet jedné operace uvažujeme 1 ns, kde n udává velikost vstupů. Zdroj: upraveno podle [18].

$g(n)$	$n = 10$	$n = 20$	$n = 50$	$n = 100$
n	$10\ ns$	$20\ ns$	$50\ ns$	$100\ ns$
$n \cdot log_2(n)$	$33\ ns$	$86\ ns$	$282\ ns$	$664\ ns$
n^2	$100\ ns$	$400\ ns$	$900\ ns$	$100\ \mu s$
n^3	$1\ \mu s$	$8\ \mu s$	$27\ \mu s$	$1\ ms$
2^n	$1\ \mu s$	$1\ ms$	$10^{21}\ s \cong 3 \cdot 10^{13}\ let$	$10^{292}\ s$
$n!$	$3\ ms$	$10^9\ s \cong 31\ let$	$10^{23}\ s \cong 3 \cdot 10^{15}\ let$	$10^{2558}\ s$

2.2.2 Turingův stroj

Turingův Stroj (TS) si můžeme neformálně představit jako zařízení, které se skládá z řídící jednotky, nekonečné pásky, ze které se čtou/zapisují data a z hlavy, která se pohybuje nad páskou a provádí čtení/zápis [21].

Formálně je Turingův stroj sedmice $(Q, \Sigma, \Gamma, \delta, q_0, B, F)$, kde

- Q – konečná množina stavů
- Σ – konečná množina vstupních symbolů
- Γ – konečná množina páskových symbolů, $\Sigma \subset \Gamma$
- δ – přechodová funkce, tj. parciální zobrazení množiny $(Q \backslash F) \times \Gamma$ do množiny $Q \times \Gamma \times \{L, R\}$, kde L je pohyb hlavy o 1 pole doleva, R – doprava
- $q_0 \in Q$ – počáteční stav

- $F \subseteq Q$ – množina koncových stavů

Nedeterministický Turingův stroj (NTS) je takový, že v jedné situaci může provést několik různých kroků. NTS může být ve variantě s jednou nebo více páskami a hlavami [21].

2.2.3 Třídy složitosti

K tomu, abychom mohli problémy přiřazovat do určitých kategorií, byly zavedeny třídy složitosti. Ty vytvářejí určitou hierarchii problémů dle jejich složitosti. Naší snahou je daný problém zařadit co nejpřesněji do nejnižší třídy. Správným přiřazením problému můžeme odhalit podobnosti s jinými problémy a využít již známých postupů či nástrojů při jeho řešení nebo převoditelnosti na jiný typ, u něhož je řešitelnost známa [19].

Problém *p1* může být polynomiálně převeden na druhý (*p2*), pokud existuje Turingův stroj TS s polynomiální časovou složitostí, který pro libovolný vstup *y* prvního problému dokáže sestrojit vstup druhého problému *y'* a platí, že výsledek *p1* pro *y* je stejný jako *p2* pro *y'*.

Dále můžeme o problému *p1* patřícímu do třídy složitosti *X* říct, že je *X*-těžký, pokud pro každý problém $p2 \in X$ platí, že může být *p2* polynomiálně převeden na *p1*. Jestliže ještě $p1 \in X$, pak je navíc *p1 X*-úplný. Pro tyto problémy platí, že jsou nejtěžší v dané třídě složitosti. Mezi nejdůležitější patří třídy P a NP. Tříd složitostí existuje celá řada, my si přiblížíme pouze ty nejdůležitější z nich [10].

Třída P

Obsahuje problémy, které jsou řešeny pomocí deterministického Turingova stroje v polynomiálním čase. S možností použití neomezené velikosti paměti.

Příklady P problémů jsou např. test prvočíselnosti nebo hledání společného dělitele [10].

Formálně můžeme definovat: *„rozhodovací úloha $\mathcal{U}$ leží ve třídě $\mathcal{P}$, jestliže existuje deterministický TS, který rozhodne jazyk $L_{\mathcal{U}}$ a pracuje v polynomiálním čase, tj. $T(n)$ je $O(p(n))$ pro nějaký polynom $p(n)$"* [20].

Třída NP

Obsahuje problémy, které jsou řešeny pomocí nedeterministického Turingova stroje v polynomiálním čase. Výpočet se může větvit do několika cest. Každou cestu ověřujeme deterministickým TS. Pokud je cesta vyřešena, dostáváme řešení.

Mezi NP úplné problémy patří problém obchodního cestujícího (viz kapitola 2.3), problém batohu (je dán batoh o určitém objemu a řada předmětů různých velikostí, s cílem zaplnit batoh z *X* procent) nebo problém kliky (zda existuje v grafu klika určité velikosti; klika je tvořena vrcholy, které jsou vzájemně propojeny hranami) [10].

Formálně můžeme definovat třídu NP: *„rozhodovací úloha $\mathcal{U}$ leží ve třídě $\mathcal{NP}$, jestliže existuje nedeterministický TS, který rozhodne jazyk $L_{\mathcal{U}}$ a pracuje v polynomiálním čase“* [20].

Třídy PSPACE a NPSPACE

Tyto dvě třídy si jsou rovny. Obsahují problémy, které jsou řešeny pomocí deterministického/nedeterministického TS za použití polynomiální velikosti paměti.

Mezi tyto problémy patří například problém pravdivosti kvantifikovaných booleovských formulí, hra Sokoban nebo hra Mahjong [10].

Třídy EXPTIME a EXPSPACE

Tyto třídy mají exponenciální složitost. Jde o nezvládnutelné problémy, pro něž neexistuje polynomiální algoritmus. Příkladem těchto problémů jsou například hry šachy, dáma nebo GO [10].

2.3 Problém obchodního cestujícího

Problém obchodního cestujícího (TSP) spočívá ve snaze naleznout Hamiltonovskou kružnici v grafu, viz kapitola 2.1.7. Obecně je úloha definována množinou měst a vzdálenostmi mezi jednotlivými městy. Každé město musí obchodní cestující navštívit právě jednou a na konci se dostat do místa, ze kterého vyjel. Přitom se snaží celou trasu naplánovat tak, aby ujel co nejkratší vzdálenost. Tato úloha patří do třídy složitosti NP úplných.

Formálně může být TSP popsán následovně: mějme graf $G = (V, E)$, kde V je množina všech měst m, $V = \{v_1, \dots, v_m\}$ a E je množina hran, $E = \{(r, s): r, s \in V\}$.

Množina hran E je reprezentována vzdálenostmi mezi městy (vrcholy grafu), kterou můžeme definovat jako množinu D. Například označení vzdálenosti mezi vrcholy r a s zapíšeme jako, $D = (d_{r,s})$. Pokud platí, že $d_{r,s} = d_{s,r}$, pak se jedná o symetrickou úlohu TSP (STSP). Pokud daná rovnost neplatí, jde o asymetrickou variantu (ATSP).

Řešení problému může vypadat jednoduše. Projdeme množinu všech možných cest a z nich vybereme tu nejkratší. Formálně tuto myšlenku můžeme zapsat jako:

$$C(\pi) = \sum_{i=1}^{n-1} d_{\pi(i),\pi(i+1)} + d_{\pi(n),\pi(1)}$$

kde π značí permutace všech možností množiny V. Tento přístup ale s rostoucím počtem vrcholů grafu začíná být neřešitelným problémem. Vyjádření všech permutací představuje faktoriálovou časovou složitost, jak bylo ukázáno v tabulce 2-1, pro velikost vstupu 20 se výsledek pohybuje v řádech jednotek až desítek let. Proto se při řešení problému TSP přistupuje k řadě aproximačních technik využívající řadu heuristik, které se snaží postupně svůj výsledek stále zlepšovat, více o optimalizačních metodách v kapitole 2.4.3 [22].

2.4 Evoluční algoritmy

Evoluční algoritmy vycházejí z Darwinovy evoluční teorie. Základem je nějaká populace, která se dále vyvíjí a mění. Postupně tak dochází ke zlepšování určitých jejich vlastností. Na těchto základech pak vznikla řada algoritmů inspirujících se v přírodě nebo chováním některých živočišných druhů, např. netopýří algoritmus, mravenčí kolonie, kukaččí algoritmus, proudění vody atd. Tyto metody pracují s různými heuristikami, které mění nějakým způsobem danou populaci, pokud je nové řešení výhodnější, nahrazuje starší variantu. Důležitým aspektem je vnášení náhodných změn do navrhovaných řešení.

Rozvoj evolučních algoritmů nastal především v posledních desetiletích díky rozvoji a pokroku počítačů. Tyto algoritmy se používají pro složité komplexní optimalizační problémy, u kterých mohou mít klasické metody problémy. Jedním z těchto typů úloh je právě problém obchodního cestujícího, na který je tato práce zaměřena [11].

Principy evoluce se uplatňují v celé řadě aplikací, např. v genetických algoritmech nebo v umělých neuronových sítích.

Genetické algoritmy jsou založeny na evolučních procesech. Přizpůsobují se novým podmínkám a zlepšuje se řešení daného problému. Kompletní popis jedince je uložen v kyselině deoxyribonukleové (DNA). U genetických algoritmů se setkáváme s pojmy chromozóm (část DNA) a gen (jednotlivé části chromozómu).

Umělé neuronové sítě se snaží napodobit fungování a principy lidského mozku. Mozek je tvořen neurony, které jsou různě pospojovány. Snahou je simulovat tuto činnost pomocí výpočetní techniky [12].

2.4.1 Základní pojmy

Před podrobnějším rozborem tématu o evolučních algoritmech si vysvětlíme několik základních pojmů a názvosloví pro lepší pochopení dalšího textu. Následující pojmy jsou popsány na základě [11] a [13].

- **Populace:** množina jedinců, tedy jednotlivých prvků daného prostoru. Na začátku bývá populace generována náhodně.
- **Generace:** představuje aktuální populaci. Po každém cyklu selekce, křížení a mutace vzniká nová generace jedinců.
- **Genotyp:** soubor genetických informací jedince.
- **Fenotyp:** soubor pozorovaných vlastností jedince. Jak se daný gen projevuje, např. v přírodě barva očí, atd.
- **Operátory:** mutace, křížení, inverze atd. Tyto operátory slouží k modifikaci jednotlivých jedinců, díky čemuž se následně vyvíjí celá populace, která získává nové vlastnosti.
- **Fitness:** účelová funkce (fitness function) se používá ke stanovení kvality jedince při výpočtech využívajících evoluční algoritmy.

2.4.2 Princip evolučních algoritmů

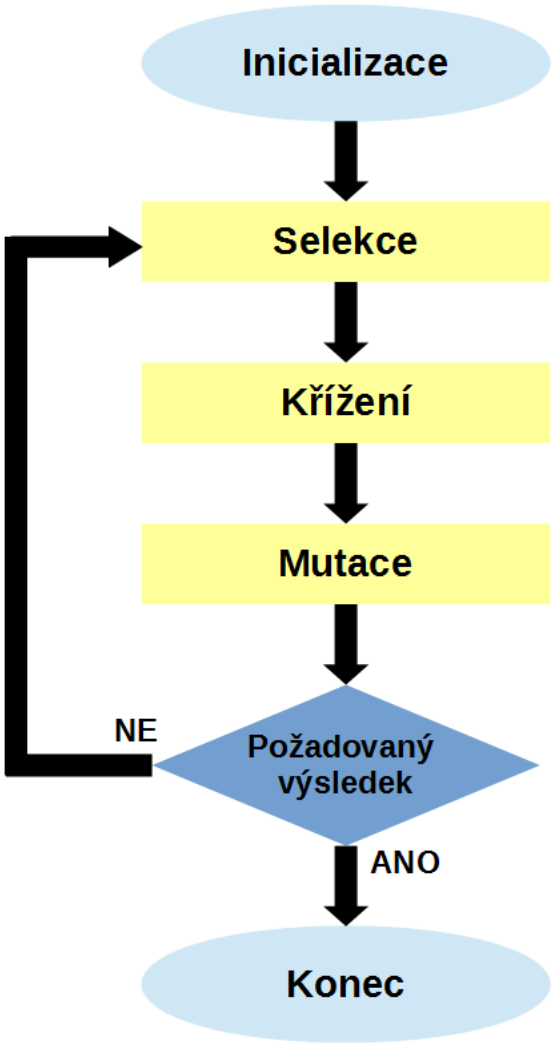
Obecně lze princip evolučních algoritmů popsat ve třech krocích, ve kterých jsou provedeny operátory selekce, křížení a mutace. Toto schéma je zachyceno na obrázku 2-8. Následuje podrobnější popis jednotlivých etap podle [2].

- **Selekce** (výběr potomků): podle *evoluční strategie* se do další generace dostává lepší potomek na úkor rodiče. Vygeneruje se předem daný počet potomků, kteří

se seřadí podle kvality (k tomuto posouzení se používá fitness funkce) a vybere se stanovené množství, se kterými se pracuje dál. U *evolučního programování* je možnost pokračovat ve výpočtech i pro slabší jedince, pokud jsou náhodně rozdělovány do skupin a z těchto skupin se vybírají nejsilnější jedinci. Pokud se v jedné skupině nahromadí více slabších jedinců, tak ti lepší se použijí v další generaci.

- **Křížení:** je definováno pro každý typ problému zvlášť. Pokud jsou jedinci popsáni diskrétně, je proces křížení výběrem dat jednoho z nich. Jsou-li ale např. definováni pomocí reálných čísel, může docházet ke křížení pomocí průměru z obou rodičů. Oproti genetickým algoritmům, které vybírají jedince pseudonáhodně, je zde výhoda ve sdílení informací, které může vést rychleji k dosažení optimální funkce.

- **Mutace:** u genetických algoritmů se mutace provádí pouze na některých jedincích. U evolučních algoritmů se naopak provede vždy. Samotná mutace se provede přičtením náhodného čísla, které má Gaussovo rozdělení. S vhodnou směrodatnou odchylkou je náhodné číslo většinou velmi malé, blízké nule. Díky tomu dosahujeme pouze drobných odchylek u jedince. Nedochází tak k velkým změnám jako u genetických algoritmů, kde dochází k překlopení některých bitů a odchylka od původního jedince tak může být velká.



Obrázek 2-8: Vývojový diagram obecného principu evolučních algoritmů. Zdroj: upraveno na základě [14], obr. 4.1, str. 86.

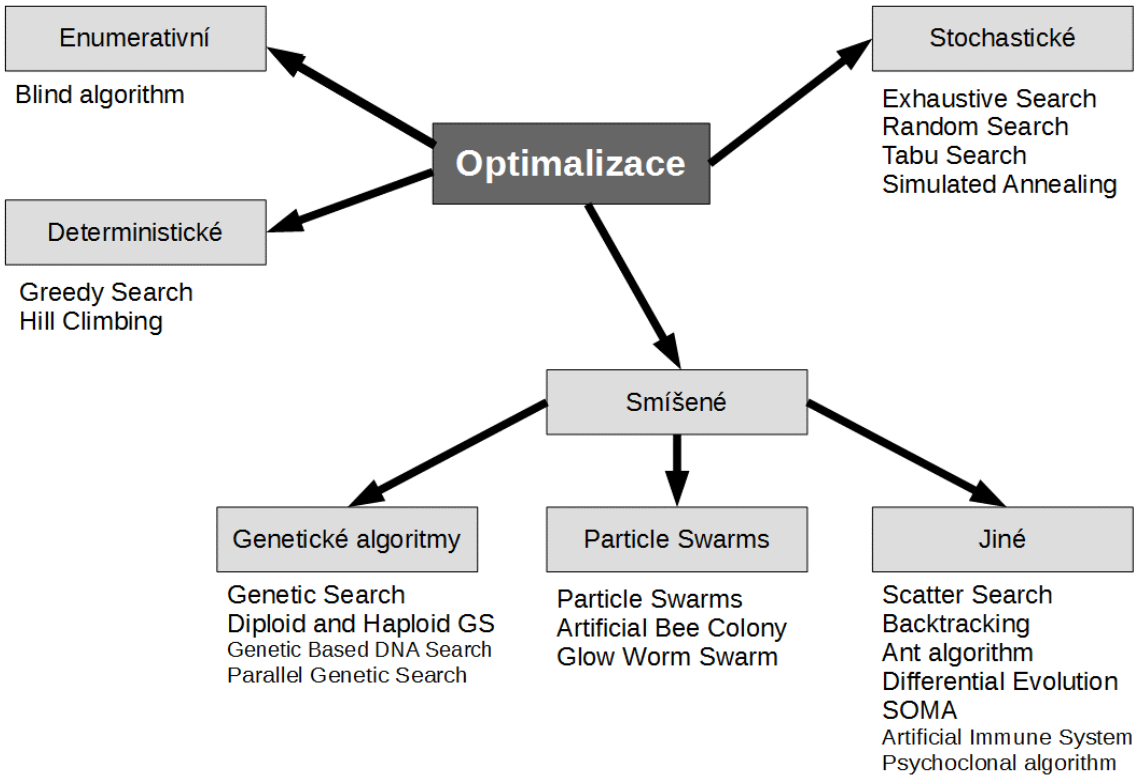
Kromě popisu vývojovým diagramem nebo slovním vyjádřením, lze obecný postup práce evolučního algoritmu popsat i pseudokódem (program 2-1), který může vypadat následovně:

```
Generování počáteční populace G(0) a nastavení i = 0;
REPEAT
   Ohodnocení každého jedince v populaci;
   Výběr rodičů z G(i) založený na jejich finess v G(i);
   Aplikace operátorů na vybrané rodiče a vytvoření potomků, které formují G(i + 1);
   i = i + 1;
UNTIL není splněna podmínka ukončení.
```

Program 2-1: Pseudokód obecného principu evolučního algoritmu. Zdroj: [11] str. 27.

2.4.3 Optimalizační metody

Existuje celá řada optimalizačních metod, které můžeme rozdělit do několika skupin, jak je ukázáno na obrázku 2-9. Názvy metod jsou uvedeny v angličtině, neboť do češtiny nebývají často překládány a tak jsou ponechány v originálním názvu pro snadnější dohledání v literatuře. Pouze u těch metod, pro které se vžil i český název, bude občas použit, jako např. u metody simulovaného žíhání (Simulated Annealing) apod. Dále následuje podrobnější popis některých metod.



Obrázek 2-9: Seznam vybraných optimalizačních metod. Zdroj: upraveno na základě [1], obr. 5.1, str. 180.

Exhaustive Search

Tato metoda patří mezi základní a nejjednodušší. Její princip spočívá v projití všech možných řešení a z nich vybere tu nejlepší. Z toho vyplývá její největší nedostatek, že musí procházet každou variantu. Pro n měst tak dostáváme $(n-1)!$ možností. Pro praktické použití se tak nedá použít pro vyšší počet měst jako je přibližně 12. Nespornou výhodou metody je fakt, že vždy najde nejkratší a nejoptimálnější řešení. Pro malý počet měst je tato metoda i velmi rychlá [1]. Postup výpočtu je znázorněn programem 2-2.

```
Vygeneruj všechny možné cesty pro n měst a ulož
je do seznamu CESTA[];
NEJ_C = CESTA[1];
for i=2:(n-1)!
      if CESTA[i] < NEJ_C
          NEJ_C = CESTA[i];
      end
end
return NEJ_C;
```

Program 2-2: Pseudokód algoritmu Exhaustive Search. Zdroj: upraveno podle [1], prog. 5.1, str. 180.

Random Search

Random Search patří mezi ty jednodušší algoritmy. Princip spočívá v náhodném výběru cest a vzájemným porovnáváním se vybere ta lepší. Pro výběr cesty se používá generátor náhodných čísel s rovnoměrným rozložením. Kvalitu výsledku tak ovlivňuje kvalita generování náhodných čísel. Na začátku se stanový počet iterací, který určuje počet vygenerovaných náhodných cest. Metoda je použitelná i pro velký počet měst, ale nenalezne vždy nejoptimálnější cestu [1]. Postup výpočtu je uveden v programu 2-3.

```
Vyber náhodně cestu a ulož do proměnné NEJ_C;
i = 1;
if i < max_iterace
   Náhodně vyber cestu C
   if c < NEJ_C
      NEJ_C = C;
   end
end
return NEJ_C;
```

Program 2-3: Pseudokód algoritmu Random Search. Zdroj: upraveno podle [1], prog. 5.2, str. 181.

Greedy Search

Hltavé prohledávání patří mezi deterministické optimalizační metody. Výpočet probíhá v jednotlivých iteracích, kdy se prohledávají všechny možnosti a z nich se vybere ta nejlepší. Na případu obchodního cestujícího je metoda aplikována následovně. Pro současný bod (město) porovnáváme všechny jeho následovníky a vybere se ten s nejkratší vzdáleností mezi sebou. V další iteraci se vezme tento bod jako aktuální a opět se prohledává v jeho sousedních bodech. Algoritmus nenalezne vždy nejlepší řešení, protože v jednotlivých krocích vybírá vždy nejvíce vyhovující cestu. Neumí tak zvolit horší řešení, které by v celku vyšlo jako lepší varianta. Postup výpočtu je znázorněn programem 2-4 [4].

```
Seznam BODY[] obsahuje všechny prvky kromě
prvního;
Vlož počáteční bod do seznamu CESTA[];
while BODY[] není prázdné
   min = inf;
   for i=1:(počet bodů v BODY[])
      Vypočítej vzdálenost mezi BODY[i] a
      CESTA[end] a ulož do proměnné n;

      if n < min
         min = n;
         index = i;
      end
   end
   CESTA[end+1] = bod ze seznamu BODY[index];
   Odstraň bod BODY[index];
end
return CESTA[];
```

Program 2-4: Pseudokód algoritmu Greedy Search. Zdroj: vlastní podle popisu v [3], kap. 11.4, str. 159.

Hill Climbing

Tento algoritmus pracuje podobně jako metoda lokálního hledání. Pro náhodně vygenerované řešení se určí okolí, označované také jako sousedství, ve kterém se hledá optimální fitness funkce. Po nalezeném optimu se pro toto místo generuje jeho okolí a algoritmus tak „šplhá“ po pomyslné křivce všech řešení ke svému vrcholu, kde se nalézá optimum funkce [4].

Na rozdíl od metody Greedy Search se zde při prohledávání sousedství může vybrat i zdánlivě horší řešení, které celkově vede k lepšímu výsledku. Výpočet končí po dosažení zadaného množství iterací. Kromě tohoto parametru lze kvalitu výpočtu ovlivnit i volbou velikosti generovaného okolí a způsobem jeho vytvoření.

Při aplikování metody na problém TSP se okolí generuje kolem náhodně zvoleného bodu a poté se tento bod zkouší prohazovat se všemi body ze sousedství. Pokud je některá z těchto výměn lepší než výchozí postavení, body se prohodí a prohozený bod se stává středem pro vygenerování nového sousedství. Pokud z daného okolí nelze vybrat lepšího jedince, zvolí se za střed náhodně jakýkoliv jiný bod z celé množiny a postup se opakuje do skončení počtu iterací.

Nevýhodou metody je možnost zacyklení ve smyslu, že se výpočet dostane do lokálního řešení, ve kterém již dříve jednou byl. Řešením je pak spouštět algoritmus opakovaně s různými počátečními řešeními.

Algoritmus není úplný, nemusí vždy nalézt nejoptimálnější řešení. Záleží i na zvoleném počtu iterací. S vyšším počtem, by se měl algoritmus stále zlepšovat [3]. Algoritmický zápis metody je zobrazen programem 2-5.

```
Zvolíme počet iterací iter_max;
Náhodně vygenerujeme počáteční řešení CESTA_0;
CESTA_x = CESTA_0;
for i=1:iter_max
   Vygeneruj okolí N pro CESTA_0
   Pro každé x ∈ N hledej takovou kombinaci
   CESTA_0, kde: f(CESTA_loc) < f(CESTA_0)
   %hledáme nejkratší cestu, proto nás zajímá
   %minimum funkce f

   if f(CESTA_loc) < f(CESTA_0)
      CESTA_x = CESTA_loc;
      CESTA_0 = CESTA_loc;
   end
end
return CESTA_x;
```

Program 2-5: Pseudokód algoritmu Hill Climbing. Zdroj: upraveno podle [3] str. 163 a 164.

Cuckoo Search

Kukaččí algoritmus je založen na principu snášení vajíček kukačky do cizích hnízd. Toto parazitické chování kukačky zvyšuje šanci na uchování jejich genů bez vyvinutí příliš velké energie (nemusí se starat o výstavbu hnízda, jeho ochranu a hlavně výchovu potomků a shánění potravy). Díky tomu může věnovat čas snášením dalších vajec. Někteří ptáci dokáží odhalit kukaččí vajíčko a poté ho z hnízda vyhodí. Tato vlastnost je v algoritmu taky zahrnuta.

Vejce představuje jedno řešení problému obchodního cestujícího. Kukačka snese vejce, pokud je toho řešení lepší než původní vajíčko, je v hnízdě nahrazeno. Dále se předpokládá, že *x* vajec bude vyhodnoceno jako cizí a budou vyhozena z řešení. To platí pro ta řešení s nejhorší fitness funkcí. Jejich počet je dán podílem p_a, který může nabývat hodnot (0,1). Vajíčka s nejlepším řešením přecházejí do další generace výpočtu.

Při vytváření nových řešení se u tohoto algoritmu často využívá Lévyho rozložení, které dává lepší výsledky než náhodné řešení dané normálním rozložením nebo řešení založené na Brownově pohybu. Lévyho rozložení v aplikaci algoritmu CS vyjadřuje, že kukačka nejčastěji snáší vejce do nejbližších hnízd, ale zároveň je schopna uletět i velkou vzdálenost [15].

```
Vygeneruj populaci n hnízd a urči jejich fitness;
while (t < maxGenerace)
   Vyber náhodně kukačku a generuj řešení pomocí
   Lévyho rozložení, poté vyhodnoť fitness Fi;

   Vyber náhodně hnízdo j;
   if (Fi > Fj)
      Nahraď j novým řešením;
   end
   Opustíme část nejhorších hnízd daných podílem
   pa a místo nich vytvoříme nová;

   Nejlepší řešení uchováme;
   Ohodnoť řešení a najdi současné nejlepší;
end
```

Program 2-6: Pseudokód základního algoritmu Cuckoo Search. Zdroj: upraveno podle [15].

Ant Colony

Optimalizace mravenčí kolonií. Můžeme se setkat i se zkratkou vzniklou z počátečních písmen anglického názvu – ACO. Jak již z názvu vyplývá, tento algoritmus se inspiruje chováním skutečných mravenců při hledání jejich potravy. Mravenci se při hledání potravy pohybují náhodně. Pokud mravenec nalezne nějakou potravu, začne na zpáteční cestě k mraveništi vypouštět feromon, podle kterého se řídí další mravenci. Ti už se nepohybují náhodně, ale vydají se cestou, kde zachytí feromon. V okamžiku, kdy další mravenci naleznou potravu, začnou také vypouštět feromony a daná cesta se stává silnější. Důležitou vlastností je postupné slábnutí feromonu. Tak vymizí slabé cesty, které jsou méně optimální a zůstávají pouze ty lepší. Díky této vlastnosti odpadá problém, že by cesta končila v některém z lokálních extrémů.

V ideálním případě, kdy mezi mraveništěm a potravou je přímá cesta bez překážek, se tak mravenci budou po nějaké době pohybovat touto přímou dráhou. V reálných podmínkách, ale na cestách bývají různé překážky. Mravenci nejdříve překážku obcházejí ze všech stran, ale postupně díky působení feromonů začnou používat pouze nejkratší cestu. Pseudokód aplikace mravenčí kolonie na problém obchodního cestujícího je znázorněn programem 2-7.

```
Všechny hrany ohodnoť výchozí hladinou feromonu
τ0;
Náhodně rozmísti mravence po městech;
while (i < max_iterace)
   while (mravenec ještě nedokončil cestu)
      for (j < max_ant)
         ant[j] se přesune k dalšímu městu na
         základě pravděpodobnostní funkce;
      end
   end
   for (k < max_ant)
      Vyprchání feromonů;
      Aplikování nových feromonů;
      if (délka cesty ant[k] < best)
         best = délka cesty ant[k];
      end
   end
end
```

Program 2-7: Pseudokód algoritmu Ant Colony Optimization (ACO) pro řešení problému TSP. Zdroj: upraveno podle [16].

Přesun mravenců mezi městy i a j se řídí podle následující pravděpodobnostní funkce:

$$p_{i,j} = \frac{[\tau_{i,j}]^{\alpha}[\eta_{i,j}]^{\beta}}{\Sigma[\tau_{i,j}]^{\alpha}[\eta_{i,j}]^{\beta}},$$

kde $\tau_{i,j}$ udává množství feromonů mezi uzly i,j. α a β jsou parametry vlivu na $\tau_{i,j}$ resp. $\eta_{i,j}$. $\eta_{i,j}$ značí požadavek na hranu i,j [3].

3 Analýza problému a současné situace

Problém obchodního cestujícího se objevuje v celé řadě odvětví, v logistice, dopravě nebo např. při návrhu desek plošných spojů. Jak je již obsaženo v názvu problému, řešením problematiky se nemusí zabývat pouze obchodní cestující, který jezdí po klientech a nabízí různé produkty. Může se jednat například o pekaře, který rozváží pečivo po okolních pekárnách a potřebuje minimalizovat náklady spojené s projetým benzínem a časem stráveným rozvážením zboží. Další použití může mít například na různých výstavách a veletrzích, například pro novináře, kteří chtějí navštívit *X* stanovišť a přitom se nechtějí zbytečně zdržovat přecházením z jednoho místa na druhé, ale potřebují svůj čas (absolvovanou cestu) naplánovat co nejefektivněji. Proto je tento problém stále aktuální a uplatnění najdou nástroje, které se věnují řešení TSP, zaměřující se na zvýšení produktivity a efektivity činnosti daných firem.

Pro řešení problému TSP vznikla řada nástrojů pracující na základě různých principů, přes genetické algoritmy, stochastické metody, deterministické metody, až k metodám inspirovaných chováním živočichů v přírodě a přírodními zákonitostmi obecně. Tyto metody jsou známé jako evoluční algoritmy, na které se tato práce zaměřuje. Úloha patří mezi NP těžké, takže klasické metody, které pracují hrubou silou a zkoumají všechna možná řešení, jsou použitelná pouze pro malou velikost vstupů. Pro větší množství vstupů, z časového hlediska, jde o neřešitelné problémy. Proto se v této práci zaměřujeme na evoluční algoritmy, které jsou vhodné právě pro tento typ úloh.

Pro vytvoření aplikace byl zvolen program Matlab, který umožňuje jednoduchou tvorbu grafického prostředí. Díky tomu je možné soustředit se na implementaci jednotlivých optimalizačních metod.

3.1 Vstupní data

Problému obchodního cestujícího se věnuje řada projektů. Velmi rozšířenou podobou formátu vstupních dat je například projekt TSPLIB[2]. Formát dat je v podobě textového souboru a nově také ve formátu xml.

Program Matlab dokáže načíst jak soubory v textové podobě (*.txt), tak i xml formát dat. Přesto v uvažované aplikaci nebude knihovna TSPLIB použita z důvodu doplnění vstupních dat o možnost ohraničení. Příkladem může být rozložení měst určitého státu a vyznačení státních hranic v grafu. K tomuto účelu je nutné ve vstupním souboru uchovávat dva typy dat: samotné body a body definující ohraničení. Z tohoto důvodu byl vybrán formát xls a novější xlsx, který data zobrazuje přehledně v tabulce.

[2] Webové stránky projektu: https://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/

4 Návrh a řešení aplikace

Cílem této práce je vytvořit aplikaci v programu Matlab[3], kde na problému obchodního cestujícího budou testovány a porovnávány jednotlivé metody evolučních algoritmů. V této kapitole představíme prostředí Matlabu a způsoby jakými v něm lze vytvořit aplikaci s grafickým uživatelským prostředím (GUI). Aplikace bude primárně vyvíjena ve verzi Matlabu R2008b s testováním i na novější verzi R2013a, tak aby byla zajištěna plná kompatibilita a funkčnost.

4.1 Matlab

Matlab je interaktivní prostředí pro numerické výpočty, vizualizaci a programování. Zároveň je to vysoko úrovňový programovací jazyk, ve kterém se dají snadno psát programy, které pracují rychleji než v tradičních programovacích jazycích jako je C/C++ nebo Java. Matlab se dá použít k široké řadě problémů od zpracování signálů, práci s obrázky, zvukem, k biologickým nebo k ekonomickým výpočtům. Obsahuje spoustu knihoven (Toolboxů), které dělají z tohoto programu univerzální nástroj pro většinu myslitelných úloh. Příkladem může být toolbox pro Fuzzy logiku, neuronové sítě, statistiku, optimalizaci atd. [5].

4.1.1 Práce v Matlabu

Tvorba programů v Matlabu se dá rozdělit do několika částí. Může se používat jako taková „chytrá kalkulačka“ přímým zadáváním příkazů do sekce s názvem *Command Window*. Po zadání příkazu tak hned uvidíme výsledek. Pokud zadáváme více příkazů a průběžné výsledky nás nezajímají, tak použitím středníku za každým příkazem, nedojde k vypsání výsledku. V případě, kdy ale potřebujeme některé hodnoty opakovaně vkládat nebo upravovat může být tento způsob zdlouhavý. Navíc po ukončení programu a pozdějším pokračování musíme opět zadávat příkazy znovu nebo je postupně vyvolávat z historie příkazů. Pro dávkové zpracování více příkazů nebo možnost uložit rozdělanou

[3] Webové stránky programu: http://www.mathworks.com/

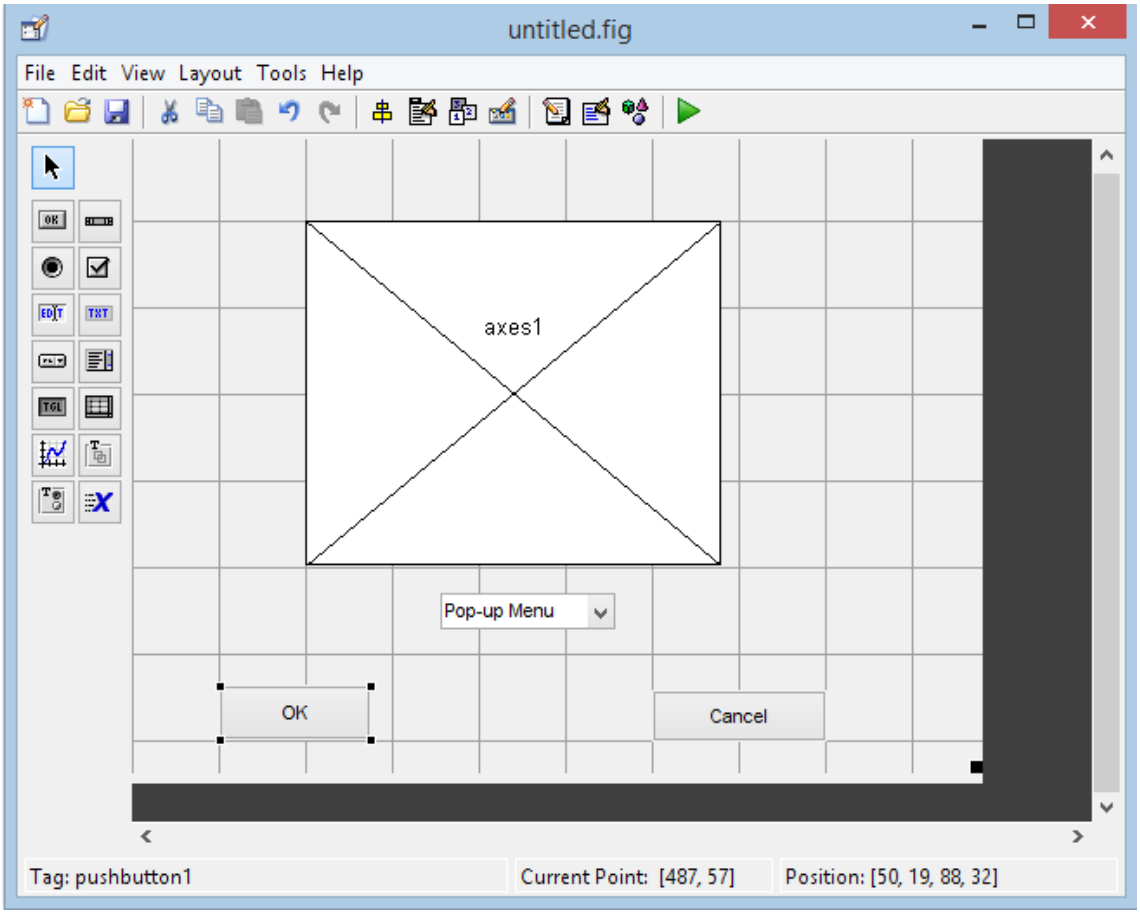
práci a znovu se k ní vrátit nabízí tento program možnost ukládat data do tzv. *m-souborů*. Spuštěním takového souboru pak dochází k provedení všech jeho příkazů naráz.

Matlab nabízí také tvorbu grafického uživatelského prostředí (GUI – Graphical User Interface). Zde jsou na výběr dvě možnosti. Použít vestavěnou grafickou aplikaci, která programátora navádí v jednotlivých krocích a usnadňuje mu práci nebo ruční napsání veškerého kódu. Oba přístupy mají své klady a zápory a je pouze na tvůrci konkrétní aplikace, pro kterou z možností se rozhodne [6].

GUIDE

Nástroj pro tvorbu grafického rozhraní se jmenuje *GUIDE*. Spouští se buď zadáním příkazu *guide* v *Command Window* nebo přes nabídkový panel $File \rightarrow New \rightarrow GUI$. Po výběru typu okna se nám zobrazí okno jako na obrázku 4-1, na kterém jsou pro ukázku vložena 2 tlačítka, pop-up menu a objekt typu *axes* pro vykreslení např. grafů. V levé části je možné vidět typy objektů, které se mohou vkládat do okna aplikace. Dvojím kliknutím na libovolný vložený objekt nebo přes dialogové okno se zobrazí panel nástrojů k danému prvku. Zde můžeme měnit jeho vlastnosti. Tady je hlavní výhoda tohoto typu tvorby GUI. Všechny vlastnosti daného prvku se nám zobrazí přehledně v jednom okně. Odpadá tak nutnost pamatovat si jednotlivé vlastnosti a nemusí se měnit ručně psaním příkazu.

To je hlavní přednost této metody, kterou ocení především začátečníci nebo příležitostní programátoři. Také to může být vhodné pro velké aplikace se spoustou prvků, kde by bylo ruční programování zdlouhavé a nepřehledné. Nevýhodou nástroje *GUIDE* může být dodatečné upravování aplikace. Výsledný kód je vygenerován automaticky a může být složitější se v něm zorientovat, pokud chce uživatel dělat další úpravy ručně bez použití nástroje *GUIDE* [6].



Obrázek 4-1: Ukázka pracovního prostředí pro tvorbu GUI pomocí nástroje GUIDE. Zdroj: vlastní printscreen.

Switched Board Programming

Pokud se snažíme o vytvoření jednoduchého čistého kódu je vhodnější místo různých nástrojů vytvořit si grafické prostředí ručně. Matlab je vysoko úrovňový jazyk a tvorba jednotlivých prvků je intuitivní a relativně jednoduchá, takže na rozdíl od jiných programovacích jazyků (např. C++/Java), kde ruční psaní GUI není doporučováno, tento problém odpadá.

Metoda Switched Board Programming využívá příkazů *switch* a *case* a dále využívá vlastnosti, že funkce může volat sama sebe. Při spuštění programu, se volá funkce bez vstupních parametrů a dojde tak k vykreslení okna a definovaných prvků. Pokud pak např. uživatel klikne na tlačítko, zavolá se znovu ta samá funkce, ale s parametrem, který identifikuje konkrétní prvek a provede adekvátní reakci. Tento základní postup je nastíněn pseudokódem v programu 3-1. Proměnná *nargin* obsahuje počet parametrů, se kterými je funkce spuštěna.

```
function muj_program(vstupni_parametry)
   if nargin == 0
      % funkce muj_program je spuštěna bez
      % parametrů
      % vykreslí se prvky grafického prostředí

      figure(...);
      uicontrol(...);
      ...
      uicontrol(...);
   else
      % funkce je spuštěna s nějakými parametry
      % provede se indentifikace argumentu
      % a provede se reakce na něj

      switch(vstupni_parametry)
         case('reakce_na_tlac_1')
               set(...); % provedení reakce
         case('reakce_na_tlac_2')
               set(...);
      end
   end
end
```

Program 4-1: Princip fungování metody Switch Board Programming pomocí pseudokódu. Zdroj: upraveno na základě [6] str. 148.

4.2 Vytvořená aplikace

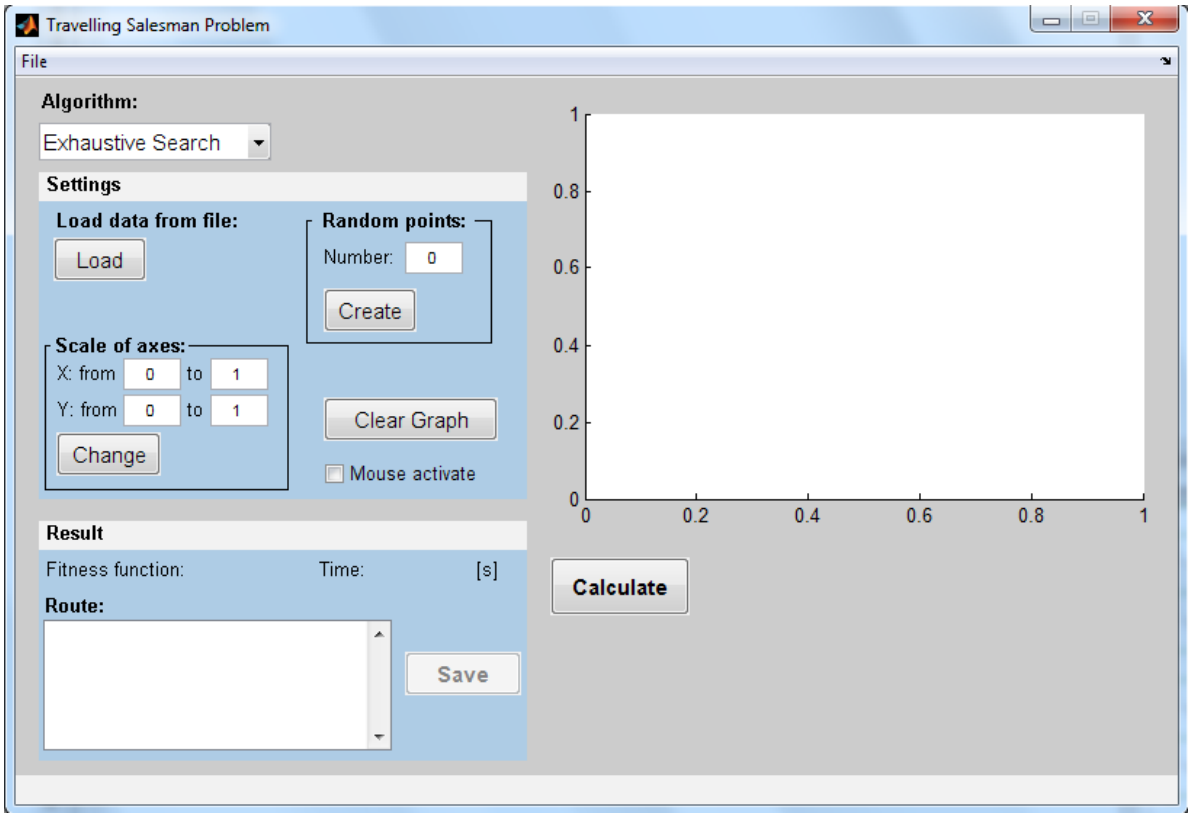
V předchozí kapitole 4.1 věnující se tvorbě grafických aplikací pod programem Matlab byla představena metoda Switched Board Programming, podle které byla vytvořena aplikace, která tvoří stěžejní část této práce. Znázorňuje několik optimalizačních algoritmů představených v kapitole 2.4.3. Tato kapitola ukáže možnosti a použití představené aplikace. V další části pak budou jednotlivé algoritmy měřeny a porovnávány.

4.2.1 Manuál vytvořené aplikace

Základní obsluha programu

Program se spouští z hlavní složky projektu souborem *main.m*. V otevřeném editoru s tímto souborem provedeme spuštění daným tlačítkem nebo klávesou *F5*. Další možností je napsáním příkazu *main* (bez přípony) v hlavním okně programu v sekci *Command Window*. Program Matlab umožňuje z *m* souborů vytvořit samostatně spustitelné soubory s příponou *exe*. K tomuto účelu slouží vestavěná aplikace, která se spustí příkazem *deploytool*. Stejného výsledku můžeme docílit i pomocí příkazu *mcc*. Díky tomu můžeme vytvořenou aplikaci vyzkoušet a otestovat i na počítačích bez nainstalovaného Matlabu.

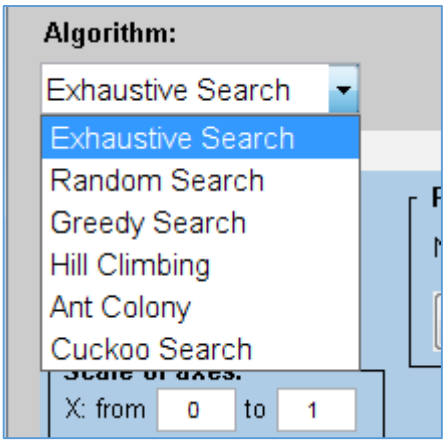
Ukončit aplikaci můžeme klasicky jako každou jinou křížkem v pravém horním rohu nebo přes nabídku menu *File/Exit*, případně klávesovou zkratkou *ctrl+Q*. Po spuštění aplikace se nám zobrazí její hlavní okno znázorněné na obrázku 4-2.



Obrázek 4-2: Screenshot vytvořené aplikace po jejím spuštění. Zdroj: vlastní.

Pracovní plocha okna je rozdělena do několika částí. Vlevo nahoře vybíráme algoritmus výpočtu. Celkem je implementováno 6 různých optimalizačních algoritmů (obrázek 4-3):

- **Exhaustive Search**
- **Random Search**
- **Greedy Search**
- **Hill Climbing**
- **Ant Colony**
- **Cuckoo Search**



Obrázek 4-3: Část okna aplikace s rozbaleným seznamem implementovaných algoritmů. Zdroj: vlastní.

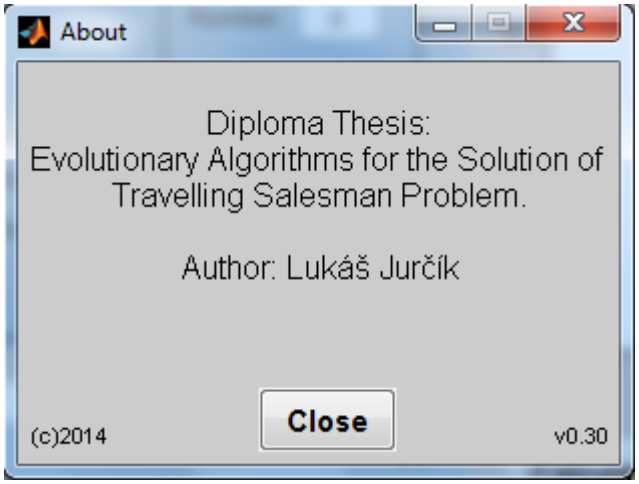
Pod výběrem algoritmu je část věnující se několika nastavením. Od načtení dat ze souboru, přes vytvoření vlastních náhodných bodů, změnu měřítek os grafu, vymazání celého grafu a nakonec obsahuje možnost vkládat data přímo tlačítkem myši.

V levé dolní části okna je část zobrazující výsledky výpočtu. Po dokončení výpočtu se zde zobrazí hodnota fitness funkce, čas výpočtu a nakonec výsledná cesta, která sestává z čísla uzlu, jeho souřadnic a jména uzlu, pokud bylo zadané (uzel je možné pojmenovat, pouze pokud jsou data načtena ze souboru). Můžeme zde naleznout také tlačítko *Save* sloužící k uložení výsledků do textového souboru.

Pravá část okna aplikace obsahuje v horní polovině graf zobrazující uzly a po výpočtu i cestu mezi nimi. V dolní části je pak prostor pro dodatečné nastavení konkrétní metody, například počtu iterací nebo velikost populace. Na obrázku 4-2 je tento prostor prázdný, protože ihned po spuštění programu je vybrána první metoda *Exhausted Search,* která nemá žádné dodatečné parametry k nastavení.

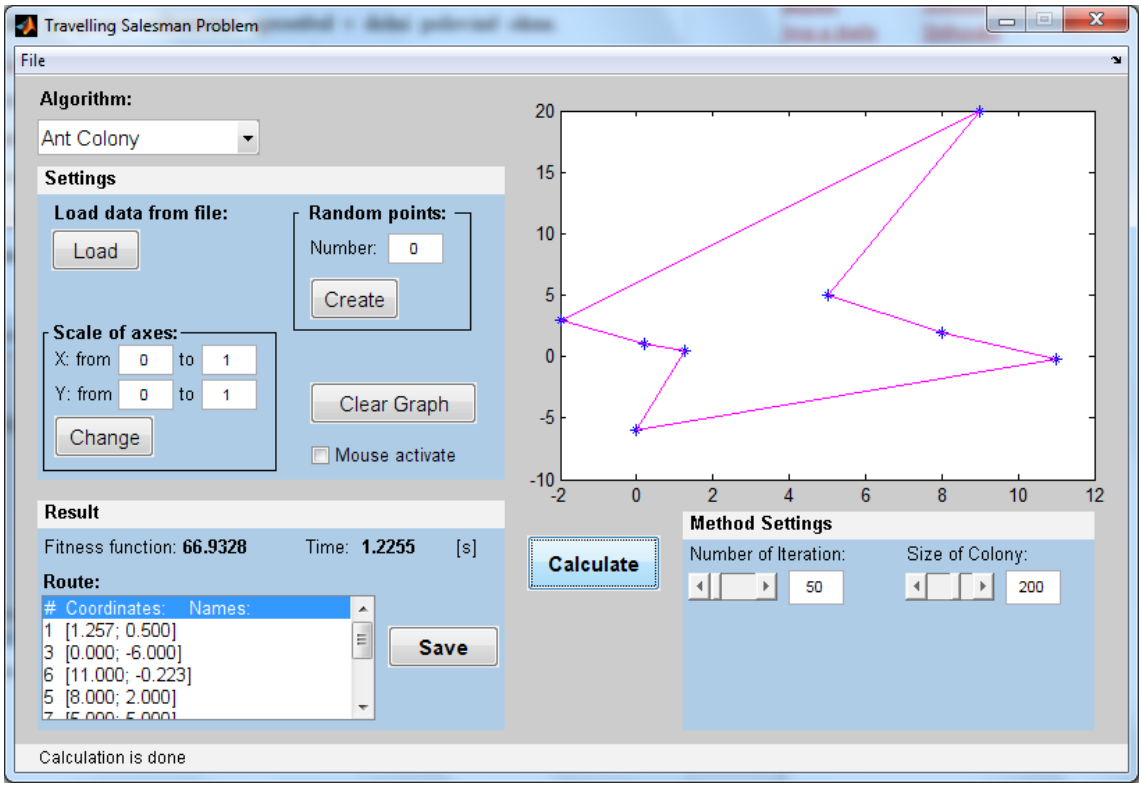
Nejdůležitější tlačítko programu nalezneme uprostřed v dolní polovině okna. Jedná se o tlačítko *Calculate*, kterým spouštíme výpočet. Úplně dole ještě nalezneme stavovou lištu, ve které se zobrazují základní informace během používání programu. Mezi zobrazované informace patří jméno vybraného algoritmu, při jeho změně, oznámení o spuštění výpočtu, jeho ukončení atd.

V položce menu *File* nalezneme kromě volby pro ukončení programu také položku zobrazující informační okno o aplikaci (obrázek 4-4).



Obrázek 4-4: Okno se základními údaji o aplikaci, které se zobrazí přes položku menu: File/About. Zdroj: vlastní.

Na obrázku 4-5 je zobrazeno okno aplikace s výsledkem výpočtu s použitím algoritmu Ant Colony. Na tomto obrázku můžeme vidět zobrazenou dodatečnou sekci pro nastavení konkrétní metody. V tomto případě počtu iterací a velikost populace (mravenců). V sekci výsledků je vidět část seznamu uzlů s jejich souřadnicemi. Položka se jménem je prázdná, protože nebyla vyplněna. Zdrojová data pocházejí ze souboru *data05.xlsx*. Dále na obrázku vidíme i použití dolní stavové lišty, která nám oznamuje, že výpočet byl dokončen.

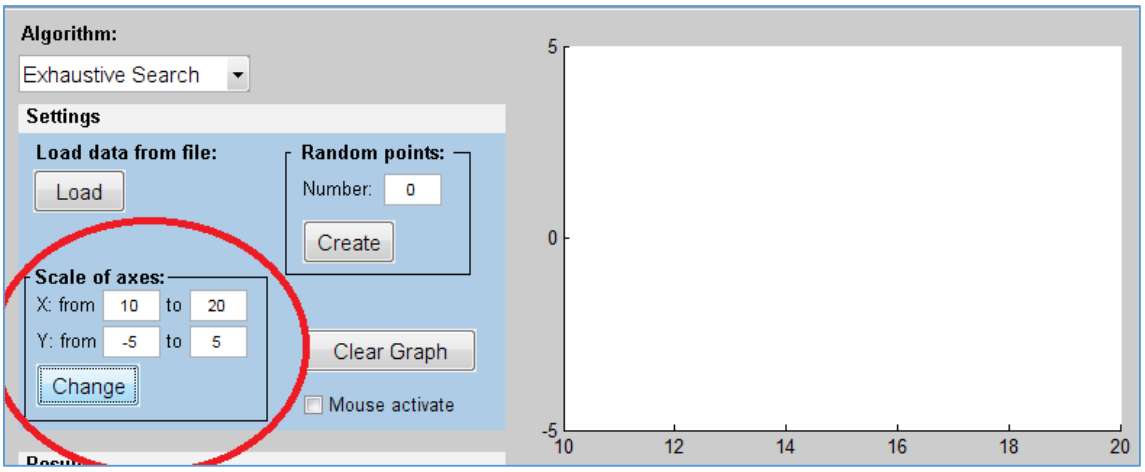


Obrázek 4-5: Screenshot vytvořené aplikace zobrazující výsledek výpočtu s využitím algoritmu Ant Colony. Zdroj: vlastní.

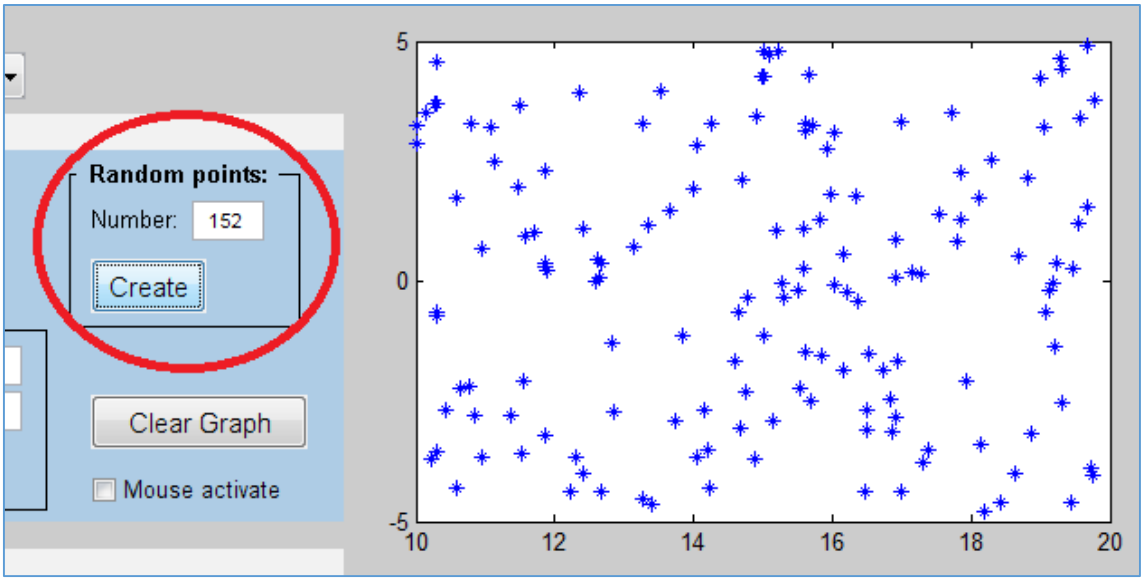
Načtení nebo vytvoření vstupních dat

V představené aplikaci jsou 3 způsoby vložení vstupních dat, načtením z externího souboru, automatickým vygenerováním nebo ručním vložením pomocí myši. Ve výchozím nastavení jsou v grafu nastavena měřítka os na hodnotu od 0 do 1. Pokud chceme provést změnu měřítka, slouží k tomuto účelu oblast ohraničená rámečkem *Scale of axes*. Změna v grafu se projeví až po potvrzení kliknutím na tlačítko *Change*. Na obrázku 4-6 je zmiňovaná oblast pro změnu meřítka os zvýrazněna červeně. Zároveň je již v grafu vidět provedená změna.

První možností vložení vstupních dat je vygenerování bodů automaticky. Tato metoda vytvoří body náhodně v celém prostoru grafu podle aktuálního nastavení měřítek os podle normálního rozložení. Obrázek 4-7 zachycuje část okna, kde je zvýrazněna sekce *Random points*, ve které se zadává požadovaný počet vytvářených bodů. Body se v grafu vytvoří až po stisknutí tlačítka *Create*.



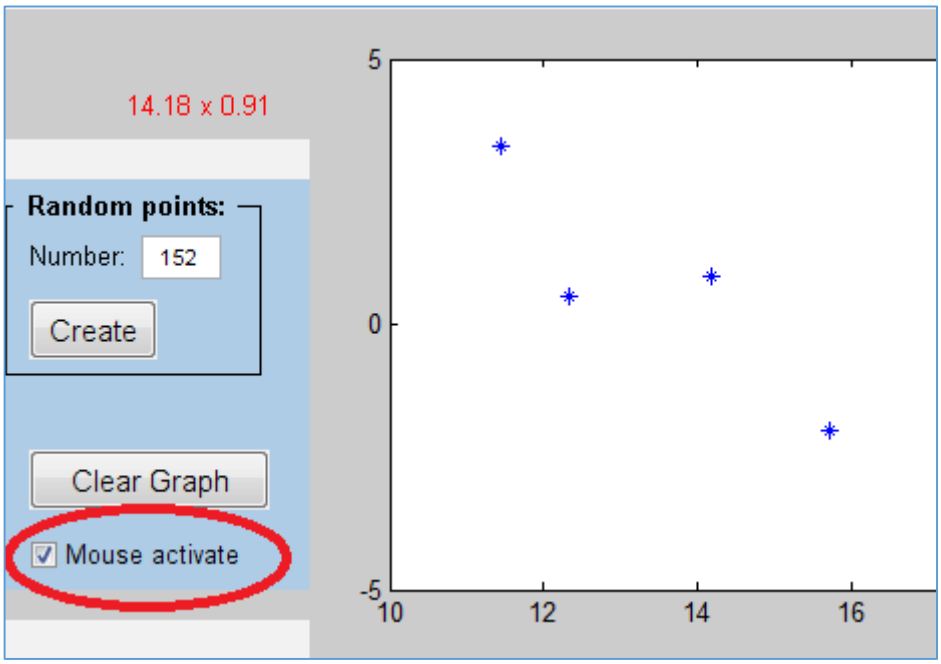
Obrázek 4-6: Část okna aplikace zobrazující oblast pro změnu měřítek os grafu. Tato oblast je zvýrazněna červeným oválem. Zdroj: vlastní.



Obrázek 4-7: Vytvoření vstupních bodů automaticky náhodně pomocí normálního rozložení. Červeně je zvýrazněna oblast programu, kde se nastavuje počet vytvářených bodů. Zdroj: vlastní.

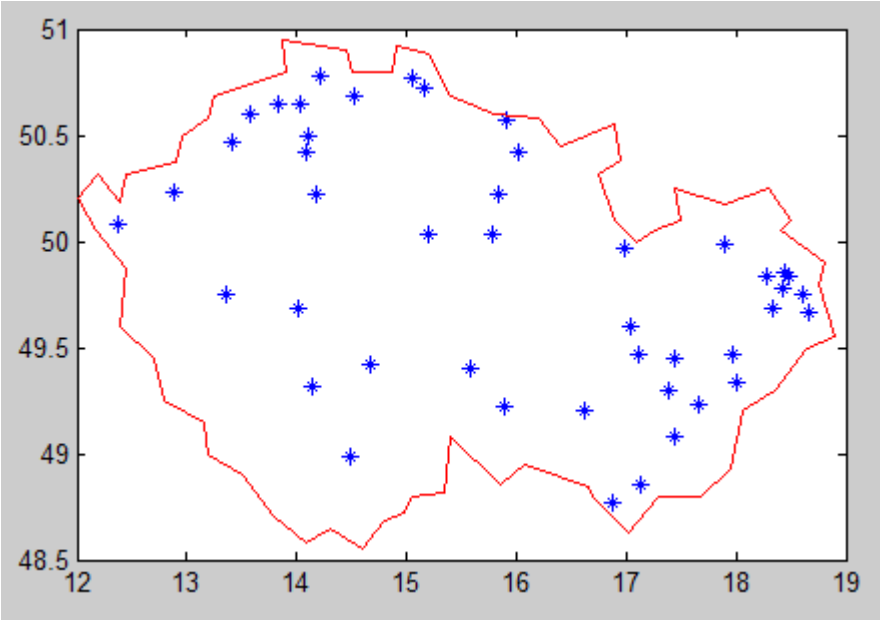
Dalším způsobem vložení dat je ručně pomocí myši. Abychom mohli použít tuto variantu, musíme nejdříve zatrhnout zatržítko *Mouse activate*. Tento krok je zde z důvodu, aby nedocházelo k náhodnému a nechtěnému vkládání bodů během přejíždění kurzoru myši přes graf. Při běžném umístění kurzoru myši do grafu se nalevo od grafu vypíší souřadnice kurzoru černou barvou. Pokud je povolena možnost vkládat body, souřadnice se zobrazují červenou barvou. Tento prvek slouží jako případné upozornění pro uživatele, že se nachází v módu vkládání bodů. Na obrázku 4-8 je v červeném oválu

zvýrazněno zatrhovací tlačítko pro povolení/zakázání vkládání bodů pomoci myši. Také zde vidíme, že vypisované souřadnice pozice kurzoru jsou zobrazeny červenou barvou.



Obrázek 4-8: Část okna aplikace zaměřená na vkládání bodů pomocí myši. Pro povolení vkládání je nutné zatrhnout zatržítko Mouse activate, které je červeně zvýrazněno. Při pohybu myši v grafu se následně vypisují vlevo nahoře od grafu souřadnice kurzoru myši červeně. Zdroj: vlastní.

Poslední možností je vstupní data nahrát z externího souboru. Program podporuje soubory s příponou *.xls* a *.xlsx*. Výhodou tohoto způsobu je především rychlost načtení, dále přesnost vložených bodů. Myší se nemusí podařit vložit bod s požadovanou přesností na dané souřadnice. Mezi další výhody patří znovu použitelnost stejných dat. Navíc je možné jednotlivé body v souboru pojmenovat. Poslední výhodou je možnost zahrnout do souboru další body, které v grafu vykreslí ohraničení, které může sloužit pro přehlednost zobrazovaných dat, např. pro vykreslení bodů představujících města nějakého státu můžeme vykreslit i hranice země. Ukázka vykreslení bodů i s ohraničením je znázorněna na obrázku 4-9.



Obrázek 4-9: Ukázka načtených dat z externího souboru a zobrazení v grafu. Kromě samotných bodů je vykresleno i ohraničení dat. Zdroj: vlastní.

Data v excelovém souboru musejí splňovat určitá kritéria. Na prvním řádku souboru v první buňce musí být číslo udávající celkový počet bodů k načtení a ve vedlejším poli počet bodů, které tvoří ohraničení. Ohraničení a pojmenování bodů je volitelná vlastnost, která nemusí být v souboru obsažena. Pokud soubor neobsahuje ohraničení, bude příslušné pole obsahovat nulu. Tento první řádek se dvěma čísly slouží ke snadnějšímu a rychlejšímu načtení a zpracování souboru.

Hodnoty ve sloupci *A* udávají x-ové souřadnice bodů. Sloupec *B* y-ové souřadnice. Tyto sloupce musejí obsahovat stejný počet záznamů, které by měly odpovídat číslu na prvním řádku. Pokud by soubor obsahoval více záznamů, ty které budou navíc, se do programu nenačtou. Pokud by jich bylo naopak méně, data se nenačtou žádná a do konzole bude vypsána chybová hláška. Souřadnice bodů mohou být jak celá čísla, tak i desetinná. I když Matlab pracuje s desetinou tečkou, data v Excelu zadáváme s desetinou čárkou. Sloupec *C* může obsahovat pojmenování jednotlivých bodů, které je volitelné.

V dalších dvou sloupcích se nachází souřadnice pro ohraničení. Sloupec *D* x-ové souřadnice a sloupec *E* y-ové souřadnice. Pro data platí stejná pravidla jako pro souřadnice bodů, tedy jejich počet musí odpovídat hodnotě v poli *B1*. Ohraničení se vykresluje jako přímka mezi 2 zadanými body. První bod je počátečním bodem a druhý

bod koncovým bodem. Při navazování ohraničení se další bod bere jako koncový a předchozí automaticky jako výchozí bod přímky. Na obrázku 4-10 je zobrazena část souboru v programu Excel, který obsahuje vstupní data pro aplikaci.

	A	B	C	D	E
1	15	63			
2	14,183	50,217	Praha	17,1	50
3	16,617	49,2	Brno	17,25	50,05
4	18,283	49,833	Ostrava	17,5	50,1
5	13,367	49,75	Plzeň	17,45	50,25
6	15,05	50,767	Liberec	17,9	50,17
7	17,033	49,6	Olomouc	18,3	50,25
8	14,033	50,65	Ústí nad La	18,5	50,1
9	15,833	50,217	Hradec Kra	18,4	50,05
10	14,483	48,983	České Buc	18,8	49,9
11	15,783	50,033	Pardubice	18,75	49,8
12	18,433	49,783	Havířov	18,9	49,55
13	17,667	49,233	Zlín	18,65	49,5
14	14,1	50,5	Kladno	18,35	49,3
15	16,017	50,417	Most	18,07	49,2
16	18,483	49,833	Karviná	17,95	48,93
17				17,68	48,8
18				17,3	48,8
19				17,015	48,63
20				16,7	48,8
21				16,65	48,84
22				16,07	48,95
23				15,85	48,85
24				15,4	49,08
25				15,35	48,82
26				15,05	48,8

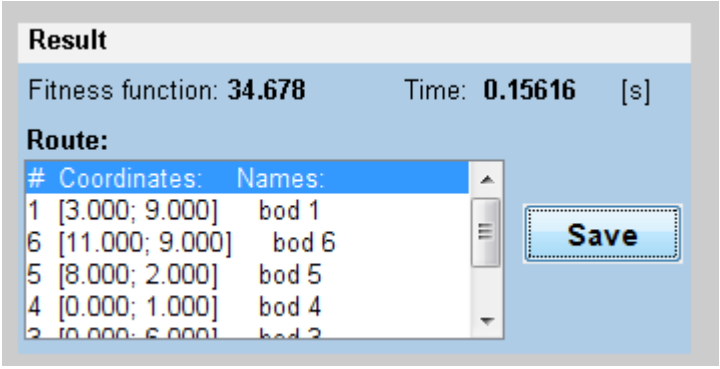
Obrázek 4-10: Ukázka struktury souboru pro načtení dat. Zdroj: vlastní.

Výsledek výpočtu

V této části se budeme zabývat možnostmi zobrazení výsledku výpočtu. Po zadání vstupních dat, v našem případě načtení souboru *data01.xlsx*, a spuštění výpočtu tlačítkem *Calculate*, se do grafu vykreslí cesta a v části *Result* (obrázek 4-11) se zobrazí tyto hodnoty:

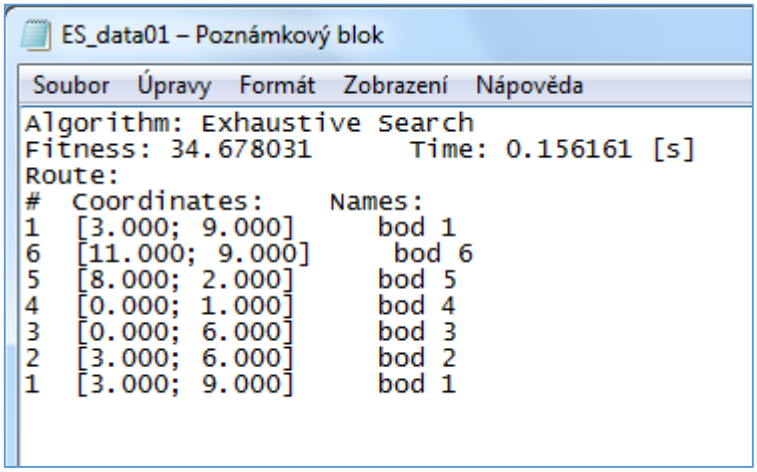
- **Fitness funkce**
- **Čas výpočtu** (hodnota je ve vteřinách)

- **Cesta** – tvoří ji seznam sestávající z čísla indexu bodu, jeho souřadnic a jména bodu, pokud je k dispozici.



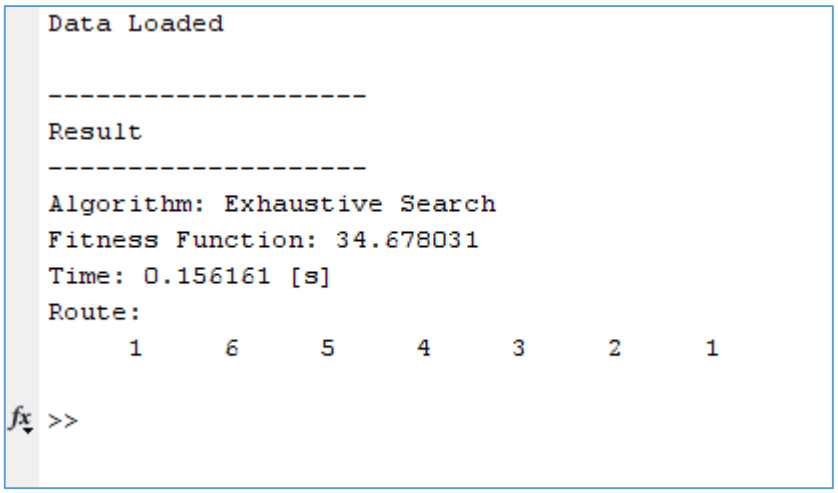
Obrázek 4-11: Ukázka zobrazení výsledku výpočtu pro algoritmus Exhaustive Search na datech ze souboru data01.xlsx. Zdroj: vlastní.

Po spuštění a dokončení výpočtu se zaktivuje v části *Result* tlačítko *Save*, které můžeme použít pro uložení výsledku do textového souboru. Zobrazení obsahu souboru je na obrázku 4-12. Na rozdíl od výsledku přímo v aplikaci je v uložené podobě navíc ještě položka informující o použitém algoritmu.



Obrázek 4-12: Zobrazení uloženého výsledku do textového souboru. Soubor obsahuje název algoritmu, fitness funkci, čas výpočtu a cestu, která sestává z indexu daného bodu, jeho souřadnic a případně jména bodu, pokud bylo zadáno. Zdroj: vlastní.

Kromě zobrazení výsledku výpočtu v okně aplikace a možnosti uložení do souboru, se výsledky ještě zobrazí přímo v Matlabu v okně *Command Window*, jak znázorňuje obrázek 4-13. V této podobě se zobrazují informace o výsledné cestě pouze ve tvaru indexů jednotlivých bodů.



Obrázek 4-13: Část okna Command Window programu Matlab se zobrazenými výsledky výpočtu. Zdroj: vlastní.

4.2.2 Testování a vyhodnocení

Jelikož ne všechny implementované optimalizační algoritmy jsou schopny pracovat s velkým množstvím vstupních dat, bude porovnávání algoritmů sestávat ze dvou částí. V první části pro malý vzorek dat, se bude testovat na datech od 4 do 10 bodů, protože metoda Exhaustive Search je schopna reálně pracovat pouze s 11 body. Pro více bodů je výpočet omezen z časových důvodů, viz kapitola 2.4.3 v části věnující se ES.

Tento algoritmus prohledává každou možnost a vždy nalezne nejkratší cestu. Jeho výsledky budou tudíž brány jako referenční a kontrolní hodnoty pro ostatní metody u malého vzorku dat.

Testování bude probíhat na počítači s procesorem Intel (2,4 GHz) s operační pamětí 4 GB a pod operačním systémem Windows 7. Konfigurace je zobrazena v tabulce 4-1. Sledovanými hodnotami je čas výpočtu a hodnota fitness funkce. Pro daný algoritmus a soubor dat bude výpočet spuštěn desetkrát a výsledný čas výpočtu vezmeme jako aritmetický průměr. Během provádění výpočtů se nebude na daném počítači provádět žádná další činnost, která by mohla zatížit procesor nebo snížit volné místo v paměti RAM a tím ovlivnit celkovou dobu výpočtu. Opakovaným měřením zároveň zkontrolujeme, zda nedošlo k ovlivnění některého měření. Pokud by se doba výpočtu radikálně lišila od ostatních, bude dané měření z tabulky vyškrtnuto a opakováno.

Paměťovou náročnost programu nebudeme sledovat, protože je jednak ovlivněno obecně celým programem Matlab, dále snahou o přesnější měření bychom ovlivnili

rychlost výpočtu algoritmu. Posledním argumentem je velikost operační paměti na běžném počítači, která se pohybuje v rozmezí 4 až 8 GB a není proto nutné příliš sledovat tuto charakteristiku. Pokud by program potřeboval příliš velké množství paměti, její nedostatek by se projevil na celkové době výpočtu, kterou sledujeme. Program pro svůj běh potřebuje přibližně asi 200 MB volného místa v paměti RAM.

Všechny testy budou provedeny na souborech, které jsou umístěny na přiloženém CD ve složce *data*.

Tabulka 4-1: Konfigurace testovacího počítače. Zdroj: vlastní.

Procesor	Intel Core 2 Duo P8600 (2,4 GHz)
Operační paměť	4 GB
Operační systém	Microsoft Windows 7 Professional (64 bit)

4.2.2.1 Malý vzorek dat

Pro malý vzorek dat budou k testování použity soubory s názvem *data_city_04.xlsx* až *data_city_10.xlsx*, kde číslo v názvu souboru označuje počet bodů, se kterými se bude pracovat.

Exhaustive Search

Tento algoritmus bude sloužit jako referenční hodnota fitness funkcí pro ostatní metody na testu pro malý vzorek dat, protože tato metoda nalezne vždy nejlepší řešení. Omezení na 10 bodů je z důvodu velké časové náročnosti metody pro větší počet bodů. Z tabulky 4-2 vidíme, jak hodnoty času od 9. bodu rostou exponenciálně. Pro zajímavost v programu je nastaven limit pro 11 bodů, ale tato hodnota v měření již není, její výpočet se ale pohybuje okolo 430 vteřin.

Tabulka 4-2: Výsledky algoritmu Exhaustive Search pro 4 až 10 bodů. Zdroj: vlastní.

Počet bodů	**4**	**5**	**6**	**7**	**8**	**9**	**10**
fitness funkce	10,13	10,41	10,48	10,93	11,00	11,20	11,43
průměrná doba výpočtu [s]	0,001	0,001	0,009	0,05	0,41	3,72	37,49

Random Search

Tato metoda určuje výslednou cestu zcela náhodně. Z tabulky 4-3 je možné vidět, že pro větší počet bodů, konkrétně od 8 již metoda nedokáže naleznout nejkratší možnou cestu. S větším počtem bodů pak odchylka od referenční hodnoty narůstá.

Tabulka 4-3: Výsledky algoritmu Random Search pro 4 až 10 bodů. Počet iterací byl nastaven na 200. Zdroj: vlastní.

Počet bodů		4	5	6	7	8	9	10
fitness funkce	nejlepší	10,13	10,41	10,48	10,93	11,00	11,39	12,86
	průměrná	10,13	10,41	10,48	11,00	11,32	12,83	14,35
	nejhorší	10,13	10,41	10,48	11,26	11,75	14,25	15,62
průměrná doba výpočtu [s]		0,04	0,06	0,08	0,10	0,12	0,15	0,17
referenční hodnota fitness		10,13	10,41	10,48	10,93	11,00	11,20	11,43
odchylka fitness [%]		0	0	0	0	0,02	1,62	12,55

Greedy Search

Metoda Greedy Search hledá vždy nejkratší cestu mezi uzly. Jelikož algoritmus bere jako výchozí bod vždy první vložený, dojde při každém spuštění ke stejnému výsledku, a proto v tabulce nejsou průměrné, nejlepší a nejhorší varianta fitness funkce.

V tabulce 4-4 vidíme, že tato metoda nedosahuje příliš dobrých výsledků a již od 5 bodů není schopna nalézt nejoptimálnější cestu grafem. Pro 10 bodů je odchylka od referenční hodnoty rovna 52,7%.

Tabulka 4-4: Výsledky algoritmu Greedy Search pro 4 až 10 bodů. Zdroj: vlastní.

Počet bodů	4	5	6	7	8	9	10
fitness funkce	10,13	11,02	11,54	13,71	15,21	17,33	17,44
průměrná doba výpočtu [s]	0,0003	0,00054	0,0006	0,0006	0,0007	0,0008	0,001
referenční hodnota fitness	10,13	10,41	10,48	10,93	11,00	11,20	11,43
odchylka fitness [%]	0	5,88	10,21	25,46	38,28	54,65	52,68

Hill Climbing

Optimalizační metoda Hill Climbing při dostatečném počtu iterací nebo počtu opakování výpočtu je schopna naleznout optimální cestu, viz tabulka 4-5. S rostoucím počtem bodů je vidět z průměrné a nejhorší fitness funkce, že k dosažení optimálního

výsledku je potřeba obecně provést více pokusů nebo zvýšit počet iterací. V našem testu na malém vzorku dat stačilo použít 200 iterací a 10 opakování pro každý test.

Tabulka 4-5: Výsledky algoritmu Hill Climbing pro 4 až 10 bodů. Počet iterací byl nastaven na 200. Zdroj: vlastní.

Počet bodů		4	5	6	7	8	9	10
fitness funkce	nejlepší	10,13	10,41	10,48	10,93	11,00	11,20	11,43
	průměrná	10,13	10,41	10,48	10,97	11,12	11,94	12,38
	nejhorší	10,13	10,41	10,48	11,11	11,23	12,91	14,52
průměrná doba výpočtu [s]		0,04	0,06	0,07	0,10	0,11	0,03	0,05
referenční hodnota fitness		10,13	10,41	10,48	10,93	11,00	11,20	11,43
odchylka fitness [%]		0	0	0	0	0	0	0

Ant Colony

Tento algoritmus nalezl pro malý vzorek dat sestávající ze 4 až 10 bodů vždy nejkratší cestu, jak je vidět z tabulky 4-6. Pro dané nastavení metody, které bylo 50 iterací a velikost populace 200, byla optimální cesta nalezena při každém pokusu.

Tabulka 4-6: Výsledky algoritmu Ant Colony pro 4 až 10 bodů. Počet iterací byl nastaven na 50 a velikost populace na 200. Zdroj: vlastní.

Počet bodů		4	5	6	7	8	9	10
fitness funkce	nejlepší	10,13	10,41	10,48	10,93	11,00	11,20	11,43
	průměrná	10,13	10,41	10,48	10,93	11,00	11,20	11,43
	nejhorší	10,13	10,41	10,48	10,93	11,00	11,20	11,43
průměrná doba výpočtu [s]		0,36	0,48	0,59	0,74	0,86	1,00	1,15
referenční hodnota fitness		10,13	10,41	10,48	10,93	11,00	11,20	11,43
odchylka fitness [%]		0	0	0	0	0	0	0

Cuckoo Search

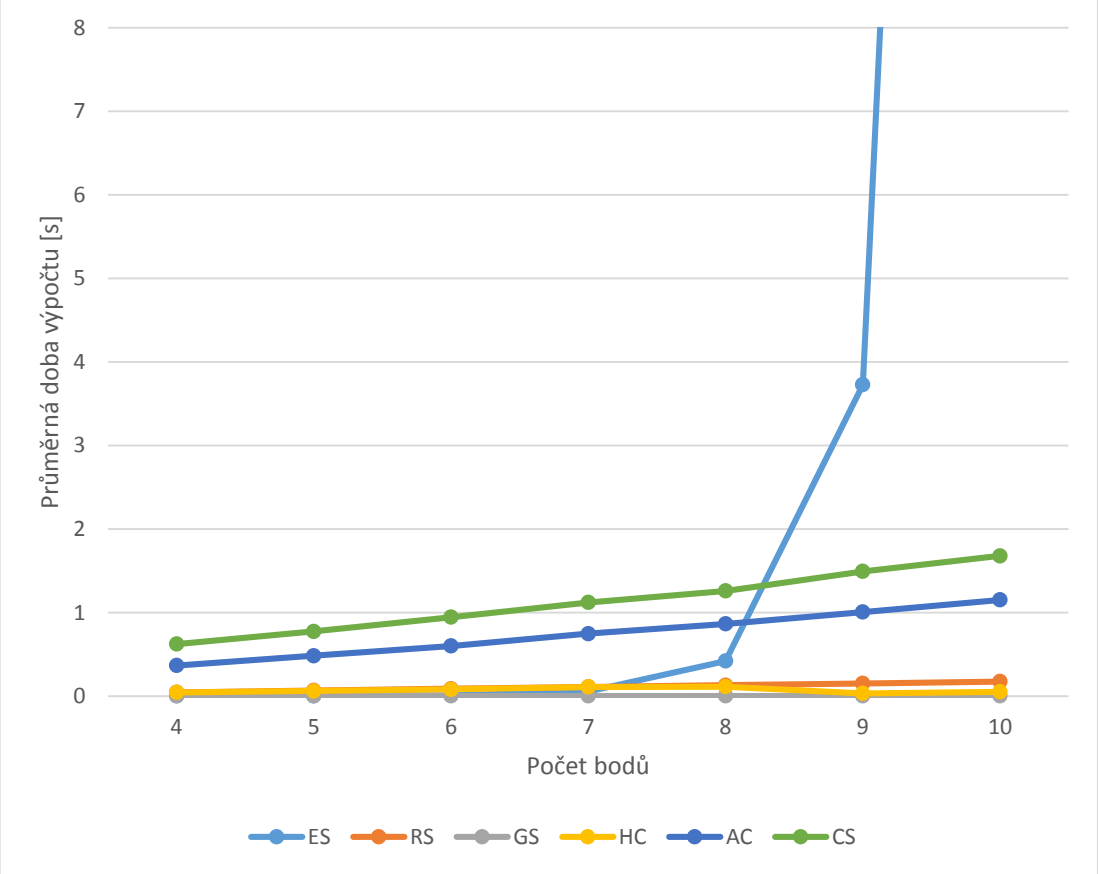
V tabulce 4-7 jsou zobrazeny výsledky kukaččího algoritmu. Pro nastavení počtu iterací na 50 a velikost populace na 50 jedinců se pro 9 a 10 vstupních bodů nepovedlo během 10 opakování testu nalézt nejoptimálnější cestu.

Tabulka 4-7: Výsledky algoritmu Cuckoo Search pro 4 až 10 bodů. Počet iterací byl nastaven na 50 a velikost populace také na 50. Zdroj: vlastní.

Počet bodů		4	5	6	7	8	9	10
fitness funkce	nejlepší	10,13	10,41	10,48	10,93	11,00	12,92	13,13
	průměrná	10,13	10,41	10,53	11,04	11,36	13,46	13,97
	nejhorší	10,13	10,41	10,61	11,07	11,75	14,49	14,47
průměrná doba výpočtu [s]		0,62	0,77	0,94	1,12	1,25	1,49	1,67
referenční hodnota fitness		10,13	10,41	10,48	10,93	11,00	11,20	11,43
odchylka fitness [%]		0	0	0	0	0	15,34	14,92

Vyhodnocení malého vzorku dat

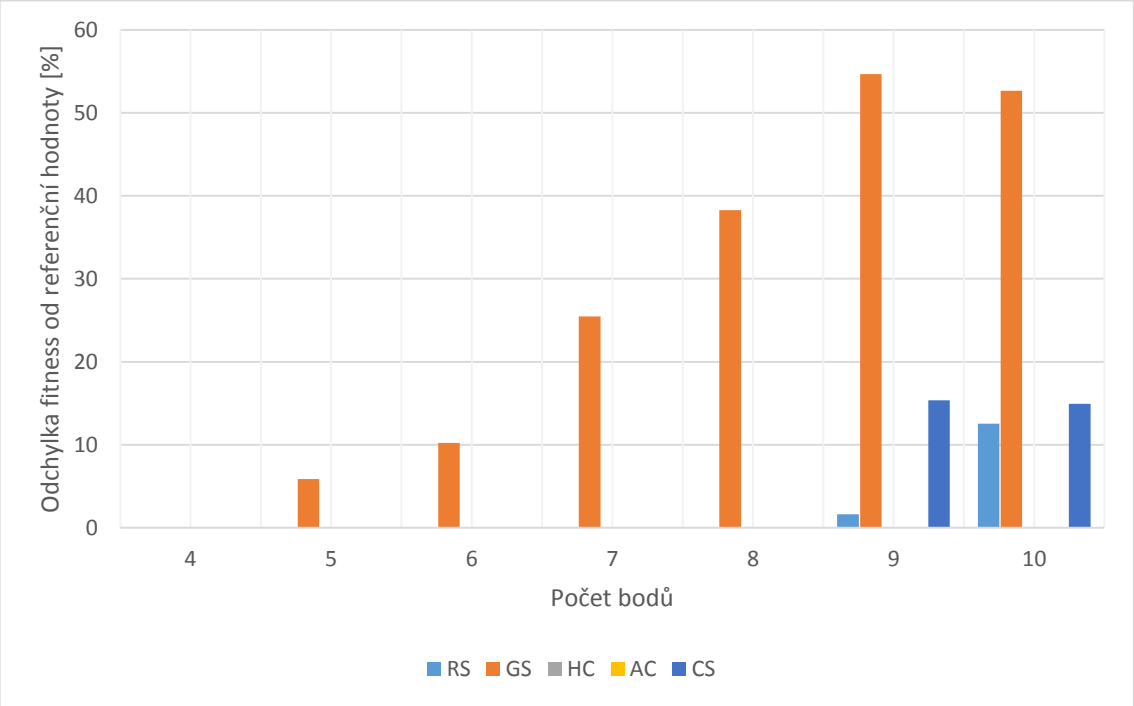
Časovou závislost doby výpočtu na počtu vstupních dat vyjadřuje graf na obrázku 4-14. Z něj je patrné, že u metody Exhaustive Search roste čas zpracování s větším počtem bodů exponenciálně a nedá se použít pro větší množství bodů než je 11. Ostatní metody pracovaly i pro větší množství bodů v časech do 2 vteřin.



Obrázek 4-14: Graf závislosti průměrné doby výpočtu na počtu vstupních bodů pro jednotlivé optimalizační algoritmy. Zdroj: vlastní.

Kromě časové závislosti jsme sledovali i hodnoty fitness funkce, konkrétně jsme měřili odchylku nejlepší dosažené hodnoty v rámci 10 opakování s referenční hodnotou, kterou byly výsledky metody Exhaustive Search.

Velikost odchylek zachycuje obrázek 4-15. Nejlépe dopadli metody Hill Climbing a Ant Colony, které pro všechny vstupy malého vzorku dat dosáhly nulové odchylky a nalezly tak vždy nejkratší cestu. Ostatní metody začaly mít problémy s hledáním optimálního výsledku u 9 a 10 bodů. Jejich hodnoty odchylek, ale nepřesáhly 15%. Tedy až na metodu Greedy Search, která nejkratší cestu nalezla jen u nejmenšího datového vstupu (4 body). V dalších případech její odchylka rostla až k hranici 54% u devíti bodů. Z těchto závěrů vyplývá, že Greedy Search je naprosto nevhodná pro malý vzorek dat, neboť v této variantě očekáváme nalezení optimální cesty vždy.



Obrázek 4-15: Graf znázorňující velikost odchylky nejlepší hodnoty fitness funkce daného algoritmu na počtu vstupních dat. Čím je odchylka menší, tím je metoda efektivnější. Zdroj: vlastní.

4.2.2.2 Vliv nastavení parametrů na výpočet

Některé z implementovaných metod nabízí dodatečné možnosti nastavení výpočtu. Jedná se o algoritmy: Random Search, Hill Climbing, Ant Colony a Cuckoo Search. Tímto nastavitelným parametrem je počet iterací algoritmu. Kromě něj ještě dvě z metod, mravenčí kolonie a kukaččí algoritmus používají další nastavitelnou hodnotu a

tou je velikost populace. V následující části se budeme zabývat vlivem nastavení těchto parametrů s dopadem na časovou složitost a určení cesty grafem.

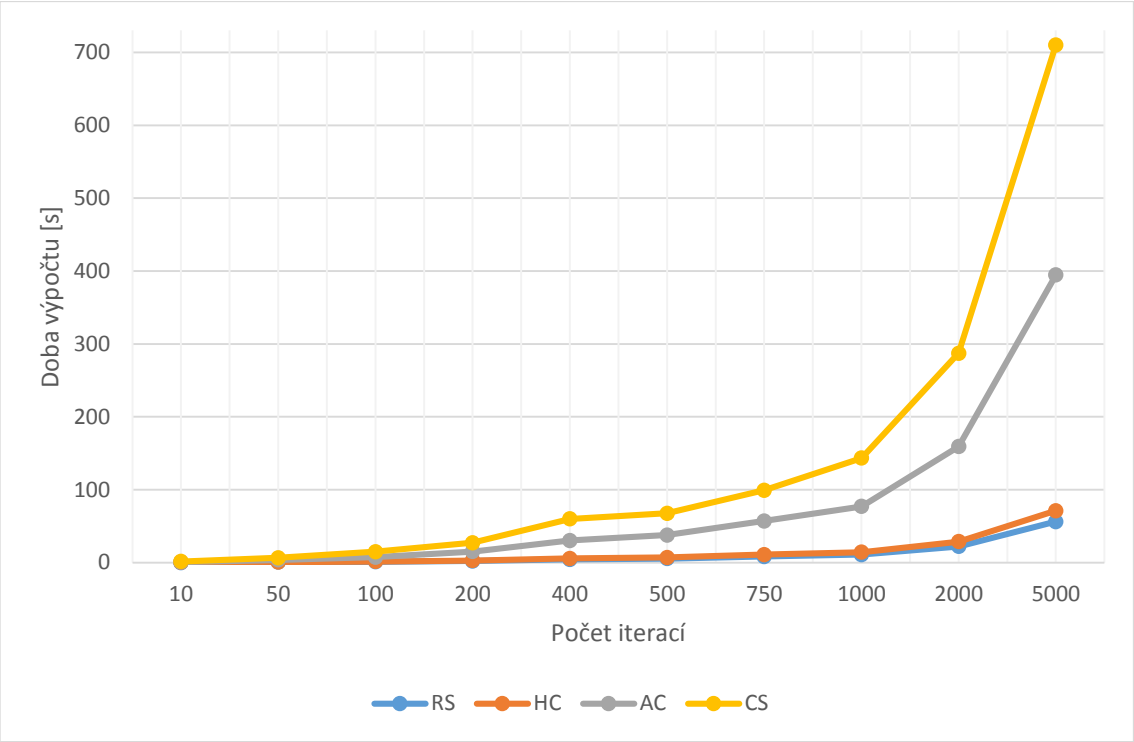
Počet iterací algoritmu

Závislost doby výpočtu jednotlivých metod na počtu iterací je znázorněna na obrázku 4-16. Z obrázku vyplývá, že do počtu iterací přibližně 200 pracují všechny algoritmy podobný čas. Od vyšších hodnot se pak již projevují rozdíly a to především u dvou metod, Ant Colony a Cuckoo Search. Jejich pomalejší činnost je způsobena závislostí na dalším nastavovaném parametru, kterým je velikost populace. Ta byla v průběhu měření nastavena na konstantní hodnotu 20, ale u vyšší hodnoty iterace se projevila jako zpomalující činitel. Důvod proč tento druhý parametr nebyl nastaven na minimální hodnotu byla snaha zachytit s měnícím se počtem iterací zlepšující trend výpočtu fitness funkce. Pokud by hodnoty populace byly na příliš nízké hodnotě, algoritmy CS a AC by dosahovaly podstatně horších výsledků než ostatní metody a mohlo by dojít k chybné interpretaci a hodnocení algoritmů.

Závislost změny fitness funkce na změně počtu iterací je zobrazena v tabulce 4-8. U všech sledovaných algoritmů je vidět pozitivní vliv vyššího počtu iterací, nejvíce pak u metody Hill Climbing.

Tabulka 4-8: Závislost nastavení počtu iterací na výsledku algoritmů. Zdrojem dat byl soubor o velikosti 100 bodů. U algoritmů s dalšími možnostmi nastavení, velikostí populace, byla tato položka nastavena na hodnotu 20. Toto nastavení je dostupné u metod Ant Colony a Cuckoo Search. Zdroj: vlastní.

Data:	data07-100.xlsx				
Počet iterací	**10**	**50**	**100**	**200**	**400**
RS Fitness	763,77	699,46	729,81	696,28	698,54
HC Fitness	624,12	401,39	357,04	332,83	288,23
AC Fitness	289,13	267,96	238,40	248,25	253,11
CS Fitness	638,82	561,76	602,85	528,89	565,72
Počet iterací	**500**	**750**	**1000**	**2000**	**5000**
RS Fitness	681,24	678,83	678,88	686,34	668,55
HC Fitness	245,42	240,30	251,97	218,63	211,88
AC Fitness	270,13	238,70	259,30	253,44	228,85
CS Fitness	574,23	535,61	547,75	530,28	520,78



Obrázek 4-16: Graf závislosti doby výpočtu jednotlivých algoritmů na počtu iterací. Měření bylo provedeno pro soubor vstupních dat o velikosti 100 bodů (soubor data07-100.xlsx). U metod, které mají ještě další nastavení, AC a CS, byla velikost populace nastavena na hodnotu 20. Zdroj: vlastní.

Velikost populace

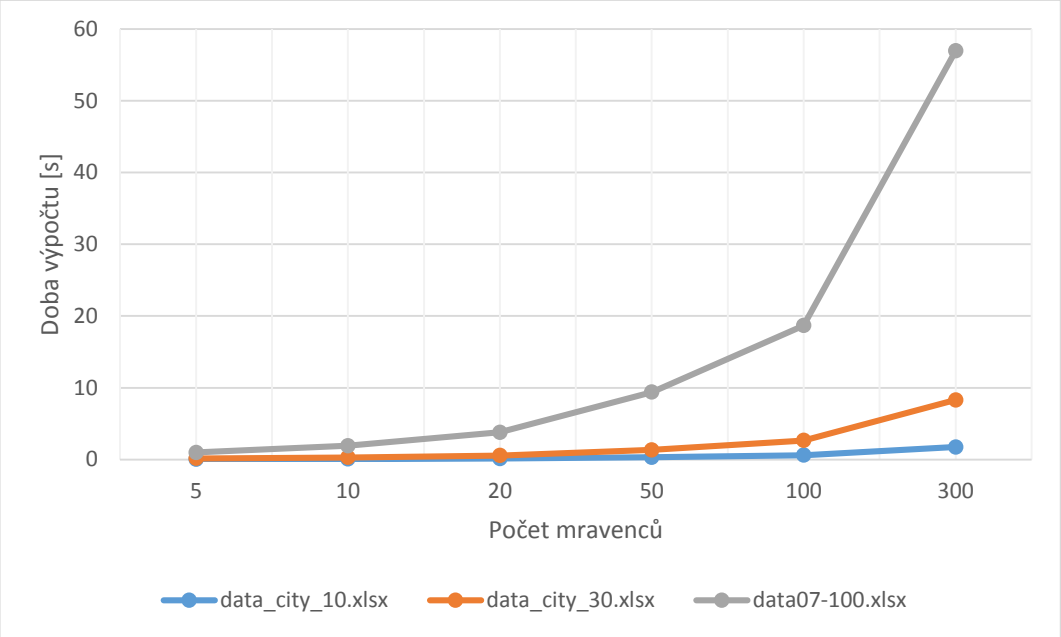
Algoritmus Ant Colony umožňuje nastavovat 2 parametry pro výpočet. Kromě počtu iterací i velikost populace. V případě této metody tedy počet „mravenců“. Na obrázku 4-17 je zachycen vliv nastavení tohoto parametru na dobu výpočtu. Počet iterací byl pro každý pokus nastaven na hodnotu 50. Měření probíhalo na 3 souborech o velikosti 10, 30 a 100 bodů. Největší časový nárůst proběhl mezi nastavenými hodnotami 100 a 300. V tabulce 4-9 jsou zachyceny hodnoty fitness funkce pro největší soubor dat (100 bodů). Z ní vyplývá, že zvyšování populace má kladný vliv na nalezení lepšího řešení. Pro provádění pokusů na větším počtu dat je pak optimální použít hodnotu populace okolo hodnoty 100. Při tomto nastavení podává algoritmus dostatečně kvalitní výsledek za relativně krátkou dobu.

Z naměřených výsledků dále pro algoritmus Ant Colony vyplývá, že na kvalitu výpočtu má větší vliv zvýšení velikosti populace než počet iterací, jak vyplývá z tabulek 4-8 a 4-10. Na velikost populace má ovšem vliv i počet bodů pro výpočet. Neplatí automaticky, že vždy větší populace algoritmu znamená lepší hodnotu fitness funkce. V tabulce 4-9 lze vidět, jak hodnota fitness klesá do nastavení populace 100, ale pro

hodnotu 300 se výsledek zhoršil. Důvodem je nižší počet bodů (30) než u testu se 100 body (tabulka 4-10). Proto je nutné volit tento parametr i s ohledem na velikost vstupních dat.

Tabulka 4-9: Vliv nastavení velikosti populace algoritmu Ant Colony při konstantní velikosti počtu iterací 50. Zdrojem dat je soubor obsahující 30 bodů. Zdroj: vlastní.

Data:	data_city_30.xlsx		Počet iterací		50	
Počet mravenců	5	10	20	50	100	300
čas výpočtu	0,14	0,28	0,54	1,33	2,66	8,30
Fitness	32,11	21,23	19,84	15,71	15,53	16,42



Obrázek 4-17: Vliv nastavení velikosti populace u algoritmu Ant Colony při konstantním počtu iterací 50 na dobu výpočtu. Měření provedeno na 3 souborech o velikosti 10,30 a 100 vstupních bodů. Zdroj: vlastní.

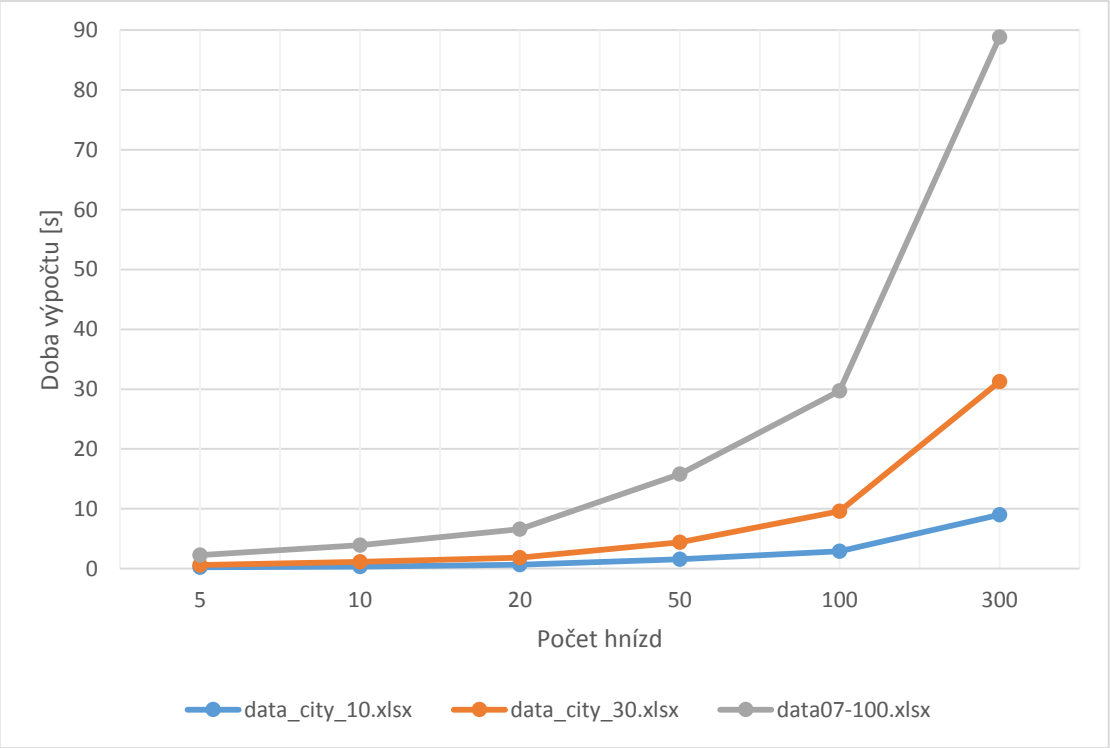
Tabulka 4-10: Vliv nastavení velikosti populace algoritmu Ant Colony při konstantní velikosti počtu iterací 50. Zdrojem dat je soubor obsahující 100 bodů. Zdroj: vlastní.

Data:	data07-100.xlsx		Počet iterací		50	
Počet mravenců	5	10	20	50	100	300
čas výpočtu [s]	0,97	1,91	3,78	9,39	18,68	56,97
Fitness	346,29	286,47	258,90	223,16	207,18	192,75

Druhým algoritmem, který umožňuje nastavovat velikost populace je kukaččí algoritmus (Cuckoo Search). Na obrázku 4-18 je zachycen graf závislosti doby výpočtu

na počtu hnízd. Průběh je podobný jako u předchozí metody AC, ale zde je vidět větší časová náročnost u prostředního souboru pro 30 bodů. Celkově vychází tento algoritmus jako pomalejší než Ant Colony a i jeho dosažené výsledky fitness funkce jsou horší.

Z tabulky 4-11 vidíme, že na kvalitu výsledku nemá nastavení populace takový vliv jako u předchozí metody. Pro větší množství bodů, tak není nutné používat maximální hodnoty, ale optimální hladina je mezi 20 – 50 hnízdy. Tím je možné snížit dobu výpočtu a naopak přidat počet iterací nebo provést výpočet opakovaně pro získání optimální cesty.



Obrázek 4-18: Vliv nastavení velikosti populace u algoritmu Cuckoo Search při konstantním počtu iterací 50 na dobu výpočtu. Měření provedeno na 3 souborech o velikosti 10,30 a 100 vstupních bodů. Zdroj: vlastní.

Tabulka 4-11: Vliv nastavení velikosti populace algoritmu Cuckoo Search při konstantní velikosti počtu iterací 50. Zdrojem dat je soubor obsahující 100 bodů. Zdroj: vlastní.

Data:	data07-100.xlsx		**Počet iterací**		50	
Počet hnízd	5	10	20	50	100	300
čas výpočtu [s]	2,24	3,91	6,57	15,78	29,69	88,83
Fitness	601,32	567,06	561,76	567,02	551,23	552,19

4.2.2.3 Velký vzorek dat

Jak již bylo zmíněno, algoritmus Exhaustive Search má omezení na použití do 11 bodů, proto v testu velkého vzorku dat není tato metoda zmíněna ve výsledcích. Díky její absenci také nemáme referenční hodnotu s nejkratší možnou cestou. Výsledky tak budou porovnávány mezi jednotlivými algoritmy. Dodatečné nastavení jednotlivých metod, počet iterací a velikost populace, byla nastavena podle poznatků získaných testováním v kapitole 4.2.2.2.

Měření bude probíhat na třech typech dat. Prvním je soubor o velikosti 45 bodů, který představuje relativně náhodné body v prostoru. Data prezentují velká města České republiky. Druhým typem dat je soubor o velikosti 50 bodů rozmístěných do útvaru podobnému kruhu. Těmito daty otestujeme algoritmy, zda se dokáží přizpůsobit danému prostoru a budou cestu hledat v navazujících bodech, nebo bude výpočet probíhat napříč přes velkou prázdnou oblast. Poslední soubor je největším z této trojice. Obsahuje 250 bodů a otestujeme, zda je program schopný v rozumném čase pracovat s tak velkými daty.

Pro každý typ testu bude měření opakováno desetkrát. Do tabulek bude zaznamenána průměrná doba výpočtu, nastavení jednotlivých algoritmů a hodnoty fitness funkce (nejlepší, průměrná a nejhorší hodnota).

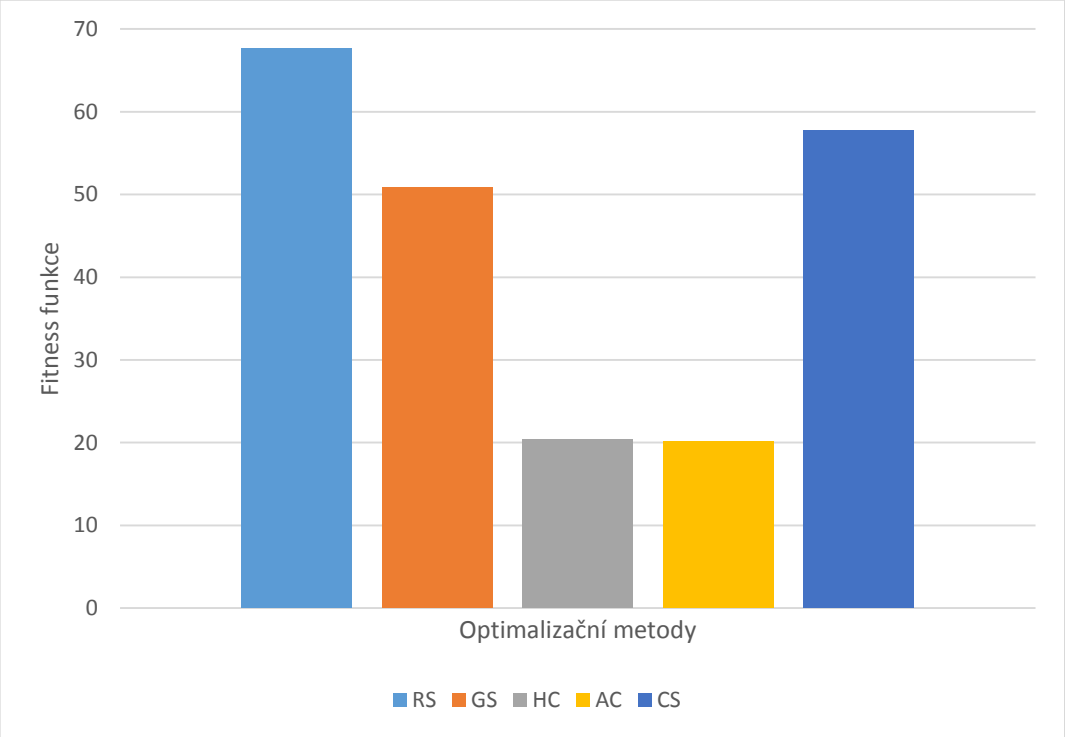
Města (45 bodů)

Na datovém souboru obsahujícím 45 bodů dosáhly nejlepšího výsledku algoritmy Hill Climbing a Ant Colony, jak je zobrazeno v tabulce 4-12. Metoda HC měla širší variabilitu dosažených hodnot fitness funkce. Pohybovala se v rozmezí hodnot od 35,87 do 20,36 s průměrnou hodnotou 28,5. Druhá metoda AC měla menší rozsah hodnot pohybující se okolo hodnot 23 až 20. Z časového hlediska, ale byla 5 krát pomalejší než horolezecký algoritmus.

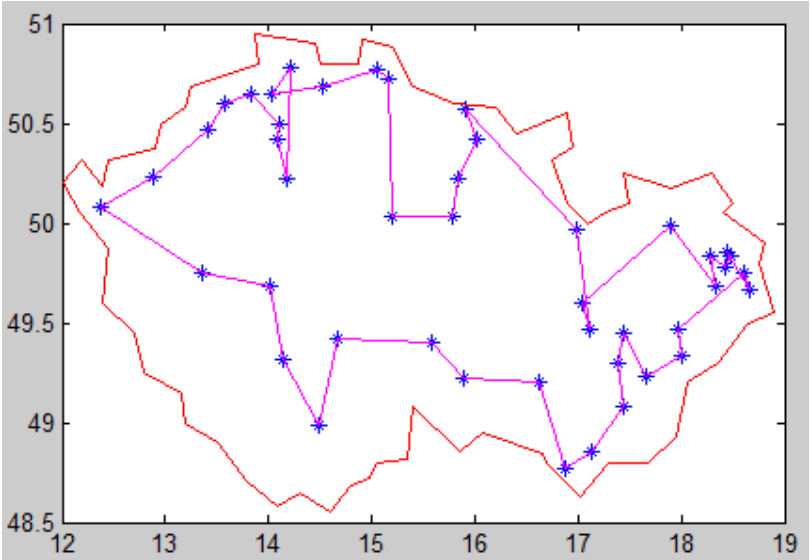
Tabulka 4-12: Výsledky testu pro soubor data_city_45.xlsx (45 bodů). Zdroj: vlastní.

Algoritmus		RS	GS	HC	AC	CS
Dodatečné nastavení	iterace	1000	-	1000	300	200
	populace	-	-	-	80	10
Počet bodů		45	45	45	45	45
fitness funkce	nejlepší	67,65	50,81	20,36	20,09	57,69
	průměrná	75,64		28,52	21,75	70,37
	nejhorší	80,04		35,87	23,04	79,15
průměrná doba výpočtu [s]		4,80	0,02	4,53	23,66	6,79

Na obrázku 4-19 je zobrazen sloupcový graf s nejlepšími hodnotami fitness funkce získané během měření. Z rozdílů mezi jednotlivými metodami můžeme určit za nejvhodnější k použití metody Hill Climbing a Ant Colony. Druhou jmenovanou metodou byl proveden test s cílem získat co nejlepší výsledek. Grafický výsledek se zobrazenou cestou je uveden na obrázku 4-20. Výsledná nejlepší získaná hodnota fitness byla 18,33 s nastavením metody na počet iterací 200 a populací 100. Z obrázku 4-19 je vidět, že se zvoleným nastavením metody podle výsledků z kapitoly 4.2.2.2 se nejideálnějšímu výsledku přiblížila i metoda Hill Climbing s odchylkou hodnoty fitness 11%.



Obrázek 4-19: Porovnání optimalizačních algoritmů na datech o velikosti 45 bodů. Graf obsahuje nejlepší hodnoty fitness funkcí dosažených během 10 opakování výpočtu. Zdroj: vlastní.



Obrázek 4-20: Nejlepší dosažený výsledek pro 45 bodů metodou Ant Colony s hodnotou fitness 18,33 a s nastavením počtu iterací na 200 a velikostí populace 100. Zdroj: vlastní.

Kruhový test (50 bodů)

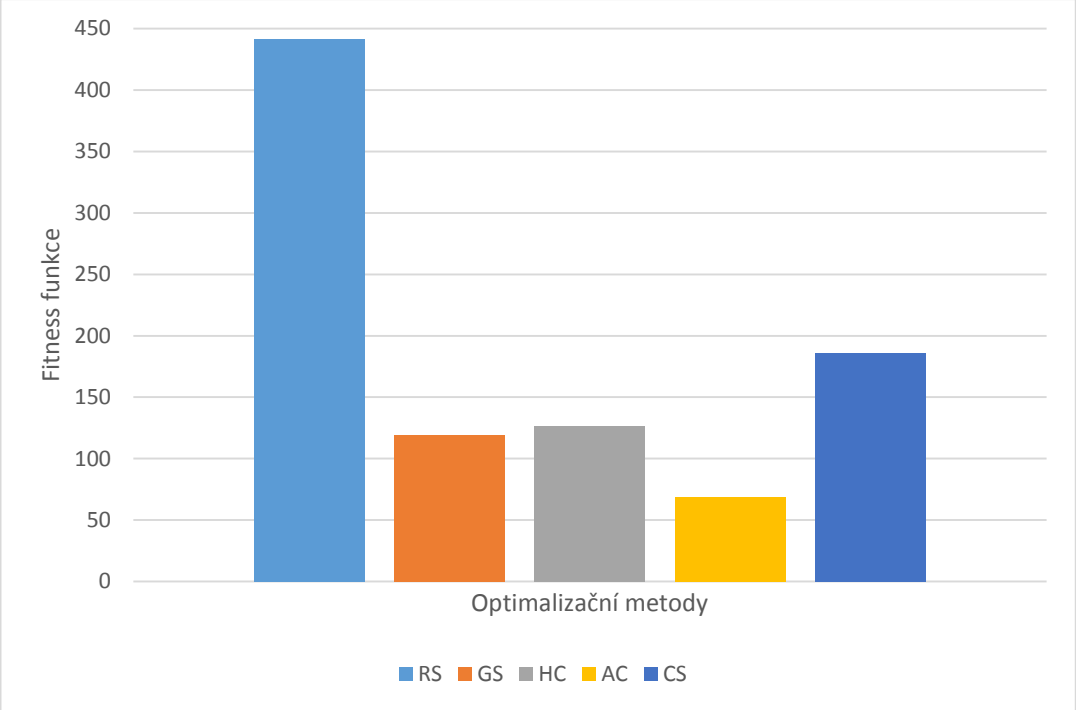
Co se velikosti vstupních dat týče, je tento test s 50 body podobný předchozímu se 45. Rozdílem je, že v předchozím případě byly body víceméně náhodně rozmístěny v prostoru, ale u tohoto typu souboru (50 bodů) jsou body rozmístěny do útvaru podobnému kruhu.

Tabulka 4-13 znázorňuje výsledky měření. Jako nejlepší se umístil opět algoritmus Ant Colony, ale zajímavější je vyhodnocení dalších metod, zvláště Greedy Search. I když oproti nejlepší metodě dosáhla odchylka fitness funkce výše 73%, je tento výsledek pořád dostatečně dobrý ve vztahu k výsledkům dalších metod. Tento rozdíl je ještě lépe vidět díky sloupcovému grafu na obrázku 4-21, který zobrazuje nejlepší dosažené hodnoty fitness. Jako nejhorší algoritmus se umístil opět Random Search, který s rostoucím počtem bodů dosahuje stále horších výsledků k poměru k ostatním metodám.

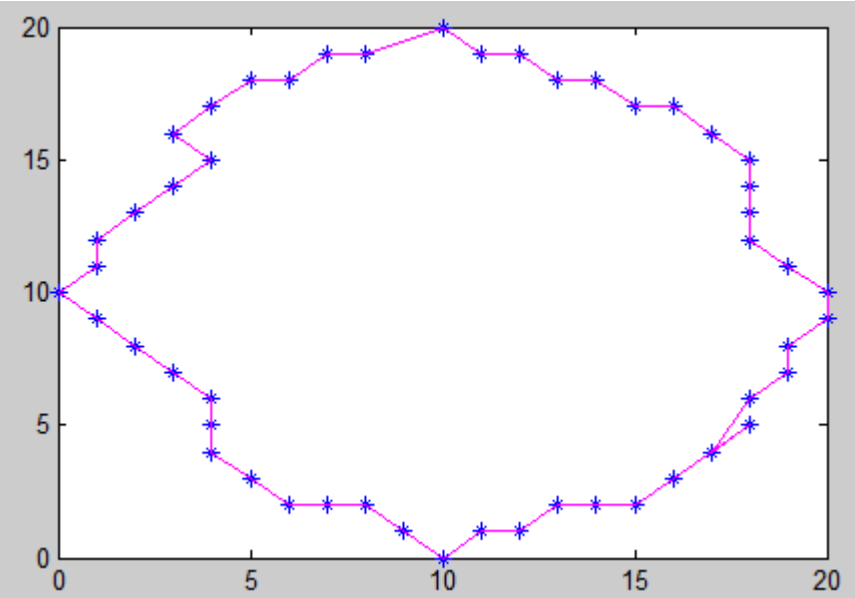
Tabulka 4-13: Výsledky testu pro soubor data10.xlsx (50 bodů). Zdroj: vlastní.

Algoritmus		RS	GS	HC	AC	CS
Dodatečné nastavení	iterace	1000	-	1000	300	200
	populace	-	-	-	80	10
Počet bodů		50	50	50	50	50
fitness funkce	nejlepší	441,81	119,06	126,57	68,80	186,38
	průměrná	460,82		155,61	82,23	214,21
	nejhorší	476,86		185,00	96,92	259,28
průměrná doba výpočtu [s]		5,20	0,02	4,74	26,45	7,34

Pro nejlepší algoritmus z testu, Ant Colony, byla provedena další série pokusů se snahou o dosažení co nejlepšího výsledku. Toho se podařilo dosáhnout s následujícím nastavením: počet iterací 200 a velikost populace 100. Obrázek s nalezenou cestou nalezneme na obrázku 4-22. S tímto nastavením bylo dosaženo hodnoty fitness funkce 66,31.



Obrázek 4-21: Porovnání optimalizačních algoritmů na datech o velikosti 50 bodů. Graf obsahuje nejlepší hodnoty fitness funkcí dosažených během 10 opakování výpočtu. Zdroj: vlastní.



Obrázek 4-22: Nejlepší dosažený výsledek pro 50 bodů (na souboru data10.xlsx) metodou Ant Colony s hodnotou fitness 66,31 a s nastavením počtu iterací na 200 a velikostí populace 100. Zdroj: vlastní.

Velké množství dat (250 bodů)

Poslední test je proveden na největším souboru dat, 250 bodech. Nastavení jednotlivých algoritmů zůstalo stejně, až na metodu Ant Colony, která pro hodnoty iterace 300 a populace 80, které byly použity v předchozích případech, nebyla vhodná z časového hlediska. Jelikož se měření provádělo opakovaně, tak se přistoupilo ke snížení obou parametrů. Výsledky testu jsou zachyceny tabulkou 4-14. Nejlepšího výsledku dosáhla metoda Greedy Search a po ní Ant Colony.

Tabulka 4-14: Výsledky testu pro soubor data08-250.xlsx (250 bodů). Zdroj: vlastní.

Algoritmus		RS	GS	HC	AC	CS
Dodatečné nastavení	iterace	1000	-	1000	100	200
	populace	-	-	-	40	10
Počet bodů		250	250	250	250	250
fitness funkce	nejlepší	4546,31	800,7718	1188,43	880,07	1362,82
	průměrná	4714,109		1251,544	931,405	1417,849
	nejhorší	4799,89		1358,76	958,09	1446,16
průměrná doba výpočtu [s]		27,936	0,36	60,485	84,624	39,405

U tak velkého souboru bodů už nás nemusí tolik zajímat časové hledisko, ale důraz může být kladen na kvalitu fitness funkce. Proto byl proveden i pokus s maximálním možným nastavením jednotlivých algoritmů.

V části věnující se vlivu nastavení algoritmů na kvalitu výpočtu se prováděla měření na souborech maximálně se 100 body. V tomto případě s ještě většími daty se ukázalo, že vhodné parametry pro menší soubory zde úplně neplatí. Proto se přistoupilo k tomuto extrémnímu testu, který měl ukázat, zda pro velká data je dosavadní nastavení vhodné či nikoliv. Výsledky jsou zobrazeny v tabulce 4-15. Z tabulky je patrné, že změna nastavení se na lepším výsledku vůbec neprojevila u metod Random Search a Cuckoo Search. Lepších výsledků naopak dosáhly metody Hill Climbing a Ant Colony. Druhá jmenovaná metoda dosáhla celkově nejlepšího výsledku fitness funkce.

Tabulka 4-15: Výsledky pokusu s maximálním možným nastavením algoritmů bez ohledu na dobu výpočtu. Provedeno pro soubor data08-250.xlsx (250 bodů). Zdroj: vlastní.

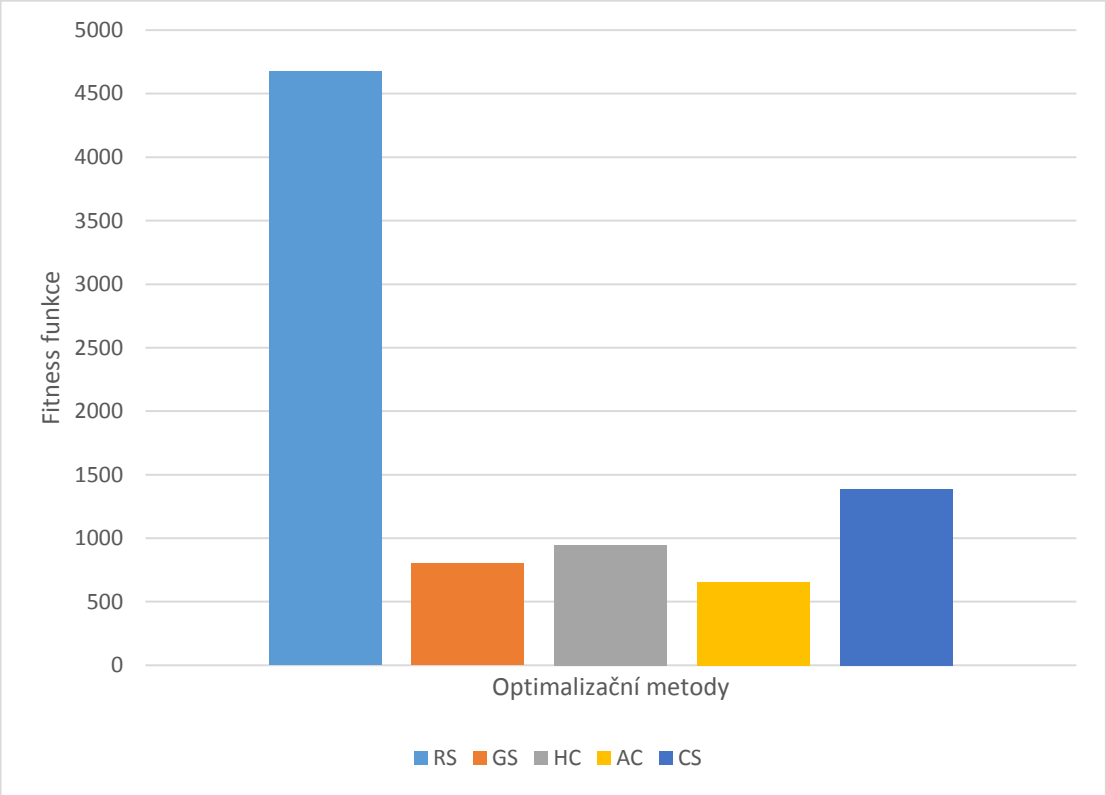
Algoritmus		RS	GS	HC	AC	CS
Dodatečné nastavení	iterace	5000	-	5000	600	1000
	populace	-	-	-	300	50
Fitness funkce		4675,53	800,77	947,56	661,34	1389,30
Doba výpočtu [s]		142,02	0,36	302,00	3695,97	866,80

Hodnoty fitness všech metod tohoto testu s maximálním nastavením jsou zobrazeny na obrázku 4-23. Z něj vyplývá, že pro daný soubor dat můžeme za vhodný výsledek považovat výstup tří algoritmů: Ant Colony, Hill Climbing a Greedy Search. I když mluvíme o testu s maximálním nastavením, neplatí to tak úplně pro metody AC a CS.

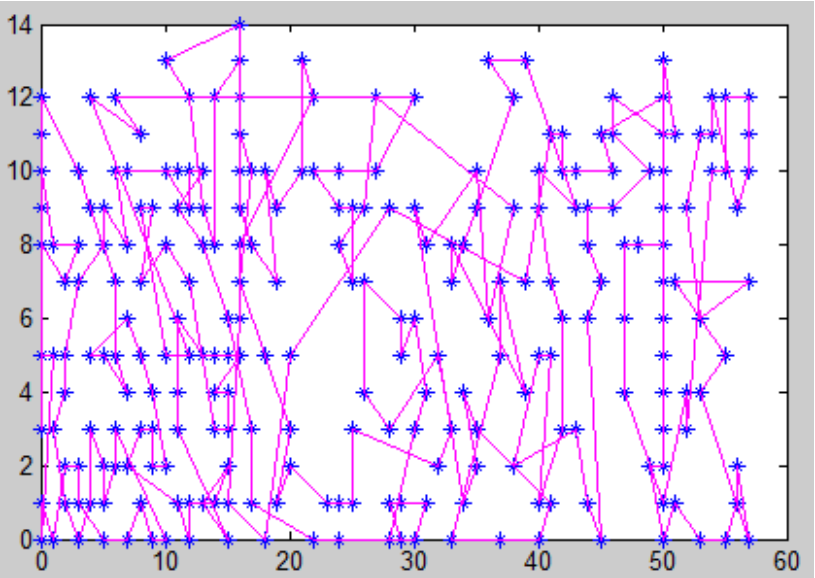
Pro algoritmus Ant Colony bylo v předchozích testech dokázáno, že na kvalitu má vliv více velikost populace, která v tomto testu byla nastavena na maximální hodnotu, kterou program podporuje a tím je číslo 300. Počet iterací byl nastaven na 600. Při dalším pokusu s vyšším počtem iterací, konkrétně 2500, bylo dosaženo lepší hodnoty fitness, ale za cenu mnohem delší doby výpočtu, přičemž zlepšení výsledku již nebylo příliš výrazné (nová hodnota fitness byla 654,8 a stará 661,3). Výsledek tohoto pokusu se zobrazenou cestou je vidět na obrázku 4-24. Z něj je patrné, že u takto vysokého počtu bodů ještě zdaleka nejde o nejkratší možnou cestu, ale zvyšováním počtu iterací se výsledek již o mnoho nezlepší.

U metody Cuckoo Search bylo testy zjištěno, že příliš vysoká hodnota populace vede naopak spíše k horším výsledkům, proto v tomto testu zaměřeném na maximálně kvalitní výsledek fitness funkce byl tento parametr nastaven pouze na hodnotu 50 a i počet iterací nebyl nastaven na maximální možnou velikost 5000, ale pouze na 1000.

Porovnáním výsledku s hodnotou z tabulky 4-14 je vidět, že výsledek se nezlepšil, naopak byl o něco horší, ale i tak ho můžeme pokládat za srovnatelný v rámci určité náhodnosti výpočtu algoritmu, který využívá Lévyho rozložení.



Obrázek 4-23: Porovnání optimalizačních algoritmů na datech o velikosti 250 bodů. Graf obsahuje nejlepší hodnoty fitness funkcí dosažených během testu s maximálním nastavením jednotlivých algoritmů. Zdroj: vlastní.



Obrázek 4-24: Nejlepší dosažený výsledek pro 250 bodů (na souboru data08-250.xlsx) metodou Ant Colony s hodnotou fitness 654,87 a s nastavením počtu iterací na 2500 a velikostí populace 300. Zdroj: vlastní.

Vyhodnocení velkého vzorku dat

Pro velký vzorek dat se ukázalo, že metoda Random Search je téměř nepoužitelná. Již pro menší počet bodů (30) dosahovala mezi ostatními metodami nejhorších výsledků a se zvětšujícími se vstupními daty se rozdíl v kvalitě výsledku mezi tímto algoritmem a ostatními dále zvětšoval.

U hladového algoritmu (Greedy Search) je situace opačná než u RS. S rostoucím počtem bodů, poskytuje tato metoda oproti ostatním lepší výsledky. Výsledek sice není nikdy optimální a nedosahuje nejkratší cesty, ale výhodou je velmi rychlý výpočet algoritmu. Tato metoda se proto jeví jako vhodná k prvnímu otestování dat a přibližnému zjištění hodnoty fitness.

Horolezecký algoritmus (HC) dosahoval dobrých výsledků a řadí se mezi druhou až třetí nejvhodnější metodu pro velký vzorek dat. U vyššího počtu bodů je ale nutné u metody zvýšit počet iterací výpočtu pro dosažení adekvátního výsledku srovnatelného s lepšími metodami.

Algoritmus Ant Colony podává na velkém vzorku dat nejlepší výsledky hodnot fitness funkce, ale je nutné správně nastavit parametry iterace a velikosti populace.

Poslední metoda Cuckoo Search s velkým počtem bodů nepodává příliš dobré výsledky. Řadí se tak na předposlední místo před metodu Random Search. U testu

zaměřeného na maximálně kvalitní výsledek fitness funkce byla výsledná hodnota CS asi dvojnásobně větší než u nejlepší metody AC.

4.2.2.4 Porovnání výkonu na různých počítačích

Všechny předchozí tesy byly prováděny na testovacím počítači č. 1, jehož konfigurace je popsána v tabulce 4-1. Nyní provedeme několik testů na jiném počítači, abychom zjistili, zda má například výkon procesoru vliv na rychlost běhu výpočtu a v jaké míře. Pro tyto testy byl použit počítač, jehož konfigurace je popsána tabulkou 4-16.

Tabulka 4-16: Konfigurace testovacího počítače číslo 2. Zdroj: vlastní.

Procesor	Intel Core i5-4570 (3,2 GHz)
Operační paměť	8 GB
Operační systém	Microsoft Windows 8.1 Professional (64 bit)

Kromě testování na různých počítačích bylo zkoumáno, jaký vliv má na rychlost běhu aplikace různá verze prostředí Matlabu. Z testů bylo zjištěno, že testovací počítač číslo 2 zvládal tu samou úlohu se shodným nastavením jednotlivých algoritmů asi 3 krát rychleji než testovací počítač číslo 1. Kompletní výsledky pro největší soubor dat (250 bodů) je zobrazen v tabulce 4-17. V další části, která porovnávala různé verze Matlabu, konkrétně verze 2008 a 2013, se testovalo na počítači číslo 2. Ukázalo se, že v novější verzi programu Matlab, je testovaná aplikace v průměru o 8% rychlejší než ve starší verzi 2008. Výpočet programu v kompilované verzi, která nepotřebuje ke svému běhu prostředí Matlab, je průměrně o 15% rychlejší než ve verzi 2008.

Tabulka 4-17: Porovnání doby běhu mezi testovacím počítačem č. 1, 2 a mezi různými verzemi programu Matlab. Vstupní data: data08-250.xlsx. Konfigurace počítačů je popsána v tabulkách 3-1 a 3-16. Zdroj: vlastní.

Algoritmus		RS	GS	HC	AC	CS
Dodatečné nastavení	iterace	1000	-	1000	100	200
	populace	-	-	-	40	10
Počet bodů		250	250	250	250	250
průměrná doba výpočtu [s] na PC01 (Matlab R2008b)		27,93	0,36	60,48	84,62	39,40
průměrná doba výpočtu [s] na PC02 (Matlab R2008b)		8,34	0,12	16,03	45,82	11,63
průměrná doba výpočtu [s] na PC02 (Matlab R2013a)		8,15	0,10	15,10	41,60	10,93
průměrná doba výpočtu [s] na PC02 (kompilovaná verze)		7,47	0,09	13,24	41,36	10,41

4.3 Využití výsledků

Na základě provedených testů můžeme konstatovat, že použití evolučních algoritmů na problému obchodního cestujícího je možné a vhodné. Kukaččí algoritmus (Cuckoo Search) podával průměrné hodnoty výsledků a u velkého množství vstupních dat byl v porovnání s ostatními implementovanými algoritmy výrazně horší, ale druhý evoluční algoritmus, mravenčí kolonie (Ant Colony) podával nejlepší výsledky ve všech testech. Ovšem je nutné nastavit správně vstupní parametry metody (počet iterací a velikost populace). Jinak je výpočet časově neúměrně dlouhý nebo nemusí být výsledek fitness funkce dostatečně kvalitní. Nevýhodou této metody je tak vyšší čas výpočtu, celkově vyšla tato metoda jako druhá nejpomalejší (pomalejší je pouze Cuckoo Search).

Vytvořená aplikace v prostředí Matlabu může být použita pro konkrétní potřeby některé firmy při řešení jejich problému obchodního cestujícího. Jako nejvhodnější metoda je doporučována Ant Colony. Pro případnou kontrolu výsledku je pak vhodné zkusit výpočet i s jiným algoritmem.

Pro velké množství vstupů už nejlepší algoritmy nepodávají vždy adekvátní výsledky. Zde je prostor pro další úpravu aplikace do budoucna. Mezi tyto úpravy patří například detekce křížení hran při určování cesty a jejich odstranění. Dalším rozšířením může být automatický odhad a nastavení správných parametrů pro metody, které další nastavení používají.

Z ekonomického hlediska můžeme výhody použití evolučních algoritmů na problému obchodního cestujícího rozdělit na finanční a nefinanční.

Mezi přímé přínosy, ty které jsou finančně vyčíslitelné, můžeme zařadit úsporu pracovní síly. Pracovník nemusí plánovat trasu ručně nebo poloautomaticky pomocí dalších nástrojů, ale určí ji pomocí vytvořené aplikace a může se tak věnovat jiné pracovní činnosti. Při využití v prostředí nabídky produktů vede ke snížení dodacích termínů, protože zaměstnanec doveze k zákazníkovy zboží v kratším čase. Úspora finančních nákladů – při použití motorového vozidla úspora benzínu a při menším používání v konečném důsledku i prostředků na údržbu vozidla.

Mezi nepřímé přínosy, které nemůžeme finančně vyčíslit, můžeme zahrnout zvýšení konkurenceschopnosti podniku, efektivnější dosažení podnikových cílů a lepší informovanost řídících pracovníků, kteří mohou na základě údajů aplikace o ujeté vzdálenosti lépe a efektivněji úkolovat zaměstnance.

Závěr

Na základě teoretického prozkoumání řady optimalizačních metod, byla vybrána řada základních metod včetně zástupců evolučních algoritmů, které byli zpracovány v programu Matlab do podoby aplikace s grafickým uživatelským rozhraním řešící problém obchodního cestujícího. Většina algoritmů byla implementována autorem této práce, kromě metody Ant Colony Optimization[4], která byla pouze uzpůsobena pro potřeby vytvořeného programu.

Výsledná aplikace byla testována ve verzích Matlabu 2008 a 2013, pro jiné verze není zaručen bezproblémový chod programu. Dále byla vytvořena i kompilovaná verze, která je samostatně spustitelná i na počítačích bez instalovaného Matlabu.

Testování jednotlivých metod probíhalo na malém a velkém vzorku dat. Toto rozdělení bylo nutné vzhledem k jedné z metod (Exhaustive Search), která pracuje metodou hrubé síly a zkoumá všechny možné stavy problému, ze kterých následně vybírá nejlepší variantu. Díky výstupu této metody, tak byly k dispozici referenční hodnoty pro ostatní metody. U velkého vzorku dat nešlo z časových důvodů použít tuto metodu, a proto byly jednotlivé algoritmy hodnoceny podle vzájemných výsledků. Z implementovaných evolučních algoritmů se metoda Cuckoo search ukázala jako průměrná až podprůměrná podle dosažených výsledků. To může být způsobeno i variantou implementace letu kukačky, která se řídí Lévyho rozložením. V rámci dalšího vývoje aplikace by bylo vhodné vyzkoušet i jiné přístupy řešení. Další z evolučních algoritmů, Ant Colony se ukázala jako nejlepší metoda, která dosahovala nejmenších hodnot fitness funkce. Její nevýhodou je delší doba výpočtu a taky nutnost správně nastavit její parametry, počet iterací, ale především velikost populace, která má největší vliv na kvalitu výpočtu. Celkově lze konstatovat, že evoluční algoritmy jsou velmi vhodné pro řešení problému obchodního cestujícího.

Další vývoj aplikace se může ubírat směrem k dalšímu vylepšení algoritmu Ant Colony. Jednou z možností, je detekce a odstranění křižování cest, které ve většině případů vede celkově k horšímu výsledku fitness funkce.

---

[4] Odkaz na zdrojové soubory metody:
http://www.sourcecodeonline.com/details/solving_tsp_with_ant_colony_system.html

Výstup této práce, vytvořená aplikace, může sloužit jako podklad pro program vytvořený na míru konkrétní firmě, která řeší problém TSP ve své činnosti. Taková aplikace může firmě snížit náklady spojené s kratší cestou a menším časem potřebným cestovat, které vedou ke zvýšení zisku firmy. Úspora se může projevit časem pracovníka, který objíždí jednotlivá města nebo zákazníky, ať už v rámci například celé republiky nebo konkrétního jednoho města. Ušetřený čas může poté věnovat jiné práci. Pokud při cestách využívá motorové vozidlo, díky kratší cestě dochází k úsporám paliva, které opět vede ke snížení nákladů firmy.

Literatura

[1] DOSTÁL, P. *Pokročilé metody rozhodování v podnikatelství a veřejné správě.* 1. vyd. Brno: CERM, 2012. 720 s. ISBN 978-80-7204-799-4.

[2] DOSTÁL, P. *Advanced Decision Making in Business and Public Services*. Brno : CERM, 2011. 168 s., ISBN 978-80-7204-747-5.

[3] ZELINKA, I., et al. *Evoluční výpočetní techniky – principy a aplikace*. 1. Vydání, Ben – technická literatura, Praha 2009. ISBN 978-80-7300-218-3.

[4] DOSTÁL, P. *The Use of Soft Computing in Management. In Vasant, P. Handbook of Research on Novel Soft Computing Intelligent Algorithms: Theory and Practical Applications.* USA: IGI Globe, 2013. DOI: 10.4018/978-1-4666-4450-2, ISBN13: 9781466644502, ISBN10: 1466644508, EISBN13: 9781466644519.

[5] THE MATHWORKS. *MATLAB – User's Guide.* The MathWorks, Inc. 2013.

[6] ZAPLATÍLEK, K., DOŇAR, B. *MATLAB – tvorba uživatelských aplikací*. 1. vyd. BEN – technická literatura, 2004. 215s. ISBN 80-7300-133-0.

[7] MIČKA, P. *Problém obchodního cestujícího* [online]. [cit. 2014-01-23]. Dostupné z www: <http://www.algoritmy.net/article/5407/Obchodni-cestujici>.

[8] HLINĚNÝ, P. *Základy teorie grafů pro (nejen) informatiky*. Výukový text Teorie grafů. Masarykova univerzita. Fakulta Informatiky. [vid. 2011-10-20]. [cit. 2014-01-23]. 145s.

[9] ČADA, R., KAISER, T., RYJÁČEK, Z. *Diskrétní matematika*. Studijní opora. Západočeská univerzita v Plzni. Katedra matematiky FAV. Plzeň, 2004. 176s.

[10] VANĚK, T. *Teorie složitosti*. Teorie výpočetní složitosti. Přednáška. Praha, ČVUT: Fakulta elektrotechnická. Dostupné z www: <www.comtel.cz/files/download.php?id=4794>.

[11] VOLNÁ, E. *Evoluční algoritmy a neuronové sítě*. 1. vyd. Ostravská univerzita v Ostravě, 2012. 152s. Dostupné z www: <http://www1.osu.cz/~volna/Evolucni_algoritmy_a_neuronove_site.pdf>.

[12] HELLER, P. MATOUŠEK, V. *Umělá inteligence a rozpoznávání*. Umělý život (úvod do evolučních algoritmů a evolučních strategií). Přednáška. Západočeská univerzita v Plzni. Plzeň, 6. – 13. března 2013. Dostupné z www: <http://www.kiv.zcu.cz/studies/predmety/uir/predn/P3/FThema3.pdf>.

[13] DOSTÁL, M. Evoluční výpočetní techniky. Univerzita Palackého v Olomouci. Katedra informatiky. Olomouc, 2007. Dostupné z www: <http://phoenix.inf.upol.cz/esf/ucebni/evt.pdf>.

[14] DOSTÁL, P. *Pokročilé metody analýz a modelování v podnikatelství a veřejné správě.* 1. vyd. Brno: CERM, 2008. 340 s. ISBN 978-80-7204-605-8.

[15] LIDBERG, S. Evolving cuckoo search. From single-objective to multi-objective. Master Degree Project in Automation Engineering. University of Skövde. 2011. Dostupné z www: < http://his.diva-portal.org/smash/get/diva2:445763/FULLTEXT01.pdf>.

[16] WONG, E., et al. *Ant Colony Optimization*. 2011 [online]. [cit. 2014-04-29]. Dostupné z www: <http://www.math.ucla.edu/~wittman/10c.1.11s/Lectures/Raids/ACO.pdf>.

[17] HONZÍK, J. *Algoritmy*. Studijní opora. Vysoké učení technické v Brně. Fakulta informačních technologií. Brno, verze 4, 2007, 262s.

[18] TESAŘ, K., MAREŠ, M. *Složitost*. KSP – Korespondenční seminář z programování [online]. [cit. 2014-05-11]. Dostupné z www: <https://ksp.mff.cuni.cz/tasks/25/cook1.html>.

[19] ČEŠKA, M., VOJNAR, T., SMRČKA, A. *Teoretická informatika*. Studijní opora. Vysoké učení technické v Brně. Fakulta informačních technologií. Brno, 2013, 168s.

[20] DEMLOVÁ, M. *Teorie algoritmů*. Přednášky. České vysoké učení technické v Praze. Fakulta elektrotechnická. 12.3.2013.

[21] DEMLOVÁ, M. *Turingův stroj*. Přednášky. České vysoké učení technické v Praze. Fakulta elektrotechnická. 5.3.2013.

[22] WONG, L., LOW, M., CHONG, CH. Bee Colony Optimization with Local Search for Travelling Salesman Problem. School of Computer Engineering, Nanyang Technological University, Singapore. [cit. 2014-05-12]. Dostupné z www: <http://web.mysites.ntu.edu.sg/yhlow/public/Shared%20Documents/papers/tsp-indin08.pdf>.

Seznam obrázků

Seznam programů

Seznam tabulek

Příloha A

Obsah přiloženého DVD:

/compile – zkompilovaná verze programu

- TSP.exe

/data – data, na kterých bylo prováděno testování

- data_city_04.xlsx
- data_city_05.xlsx
- data_city_06.xlsx
- data_city_07.xlsx
- data_city_08.xlsx
- data_city_09.xlsx
- data_city_10.xlsx
- data_city_15.xlsx
- data_city_20.xlsx
- data_city_25.xlsx
- data_city_30.xlsx
- data_city_35.xlsx
- data_city_40.xlsx
- data_city_45.xlsx
- data01.xlsx
- data01.xlsx
- data02.xlsx
- data03.xlsx
- data04.xlsx
- data05.xlsx
- data06-50.xlsx
- data07-100.xlsx
- data08-250.xlsx
- data09.xlsx

- data10.xlsx
- „odkaz – zdroj dat pro data09.txt“
- „odkazy – zdroje dat.txt“

/results – výsledky uložené pomocí programu a záznamy provedené při testování v souborech *.xlsx

- „AC_300_80_data10.txt“
- Results01.xlsx
- Results02.xlsx
- Results03.xlsx
- Results04.xlsx

/src – zdrojové soubory programu v podobě *m* souborů

- antColony
  - antMain.m
  - ants_cost.m
  - ants_cycle.m
  - ants_information.m
  - ants_primaryplacing.m
  - ants_traceupdating.m
  - readme.txt
- aboutDialog.m
- clearResult.m
- cs.m
- deactivateElem.m
- displayResult.m
- dist.m
- drawBorder.m
- drawRoute.m
- errMsg.m
- ES.m
- getAlgName.m
- greedySearch.m

- Gui.m
- hillClimbing.m
- levyFlight.m
- mouseClick.m
- mouseMove.m
- randomSearch.m
- saveResult.m
- totalDist.m
- writeResult.m

/main.m – hlavní *m* soubor pro spuštění celé aplikace