Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

BAKALÁŘSKÁ PRÁCE

Optimalizace mravenčí kolonií na problému obchodního cestujícího

2021

Petr Jančár

Vedoucí práce: Mgr. Jiří Balun

Studijní obor: Aplikovaná informatika, prezenční forma

Bibliografické údaje

Autor:	Petr Jančár
Název práce:	Optimalizace mravenčí kolonií na problému obchodního cestujícího
Typ práce:	bakalářská práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2021
Studijní obor:	Aplikovaná informatika, prezenční forma
Vedoucí práce:	Mgr. Jiří Balun
Počet stran:	52
Přílohy:	1 CD
Jazyk práce:	český

Bibliograhic info

Author:	Petr Jančár
Title:	Ant colony optimization for the traveling salesman problem
Thesis type:	bachelor thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2021
Study field:	Applied Computer Science, full-time form
Supervisor:	Mgr. Jiří Balun
Page count:	52
Supplements:	1 CD
Thesis language:	Czech

Anotace

Optimalizace mravenčí kolonií je technika umělé inteligence, která vychází z pozorování skutečných mravenců a jejich schopnosti řešit obtížné optimalizační problémy. Tato práce se zabývá aplikací optimalizace mravenčí kolonií na problém obchodního cestujícího. Součástí práce je implementace a experimentální srovnání této techniky a dalších vybraných algoritmů.

Synopsis

Ant colony optimization is an artificial intelligence technique based on the observation of real ants and their ability to solve difficult optimization problems. This thesis deals with the application of ant colony optimization to the travelling salesman problem. Part of the thesis is the implementation and experimental comparison of this technique and other selected algorithms.

Klíčová slova: mravenčí kolonie; optimalizace; TSP; graf; algoritmus; C#

Keywords: ant colony; optimization; TSP; graph; algorithm; C#

Děkuji Mgr. Jiřímu Balunovi za cenné rady a trpělivost při vedení této práce. Poděkování patří také rodině za podporu.

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

datum odevzdání práce podpis autora

Obsah

Seznam obrázků

Seznam tabulek

Seznam definic a příkladů

Seznam zdrojových kódů

1 Úvod

Problém obchodního cestujícího je známá úloha, zobecňující úlohu nalezení nejkratší cesty, která prochází všemi zadanými městy na mapě a navrací se nazpět do výchozího města. Navzdory jednoduchému zadání úlohy se jedná o velmi obtížný problém. S rostoucím počtem měst na mapě totiž rychle narůstá počet cest mezi nimi. Již pro několik desítek měst je tak prakticky nemožné projít všechny možné cesty a vybrat z nich tu nejkratší. Jinými slovy – algoritmus pro nalezení nejkratší možné cesty je sice možné sestavit, avšak v praxi je pro větší počet měst nepoužitelný. Proto se v praxi obvykle používají algoritmy, které nezaručují nalezení nejkratší možné cesty. Tyto algoritmy lze obecně hodnotit na základě dvou důležitých faktorů. Jedná se o jejich časovou efektivitu a kvalitu řešení, které poskytují.

Zajímavou technikou, kterou lze využít pro řešení problému obchodního cestujícího, je optimalizace mravenčí kolonií. Tato technika vychází z pozorování skutečných mravenců, respektive kolonií mravenců. Navzdory absenci výraznější inteligence jednotlivých mravenců se mravenčí kolonie vyznačují pozoruhodnou schopností provádět obtížné úlohy, mnohdy výrazně převyšující individuální schopnosti jednotlivých jedinců. Složité chování a samoorganizaci mravenčích kolonií lze vysvětlit jistou formou nepřímé komunikace mezi jednotlivými mravenci skrze změny prostředí, ve kterém se nacházejí. Hlavní myšlenkou optimalizace mravenčí kolonií je vytvořit kolonii umělých mravenců, inspirovaných chováním skutečných mravenčích kolonií a využít je k řešení obtížný optimalizačních úloh, například problému obchodního cestujícího.

Jedna z existujících implementací algoritmů optimalizace mravenčí kolonií pro problém obchodního cestujícího se nachází na webových stránkách s adresou http://www.aco-metaheuristic.org/aco-code/. Jedná se o implementaci v jazyce ANSI C, šířenou pod licencí GPL.

Cílem této bakalářské práce je popsat techniku optimalizace mravenčí kolonií pro řešení problému obchodního cestujícího, implementovat ji v programovacím jazyce C# a experimentálně porovnat s dalšími vybranými algoritmy.

2 Grafy

Tato kapitola čerpá z literatury [1], [2].

V teorii grafů je graf abstraktní pojem, který je určen množinou vrcholů a množinou hran. Vrchol představuje základní objekt grafu. Hrana je určena některými dvěma vrcholy grafu a vyjadřuje skutečnost, že dané vrcholy mezi sebou mají nějaký vztah.

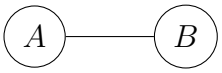
Grafy jsou užitečné pro zjednodušené modelování reálných objektů a vztahů mezi nimi. Pomocí grafu lze znázornit například mapu měst a silničních spojení mezi nimi, topologii počítačové sítě nebo modelovat letovou trajektorii letadel. Jako abstrakci lze grafy využít k řešení různých úloh, například k nalezení nejkratší cesty mezi dvěma městy.

Menší grafy lze zadat i ve formě obrázku. Pak mluvíme o vizualizaci, nebo také diagramu grafu. Vrcholy grafu jsou obvykle znázorněny kružnicemi a hrany čárami nebo oblouky mezi vrcholy, které tyto hrany spojují. Výhodou znázornění grafu diagramem je jeho přehlednost.

U grafů rozlišujeme dva typy hran:

- *neorientované*, které reprezentují symetrické vztahy mezi vrcholy,
- *orientované*, které reprezentují nesymetrické vztahy mezi vrcholy.

Jako příklad symetrického vztahu si můžeme představit vztah „být soused“. Pokud je osoba A sousedem osoby B, pak jistě i osoba B je sousedem osoby A. Vztah „být soused“ mezi osobami A a B bychom tedy v grafu označili neorientovanou hranou, viz obrázek 1.



Obrázek 1: Neorientovaná hrana

Příkladem nesymetrického vztahu je vztah „být nadřízený“. Skutečnost, že osoba A je nadřízeným osoby B, bychom v grafu označili orientovanou hranou, viz obrázek 2. V diagramu grafu se orientovaná hrana označuje malou šipkou nebo trojúhelníkem na jejím konci.

Obrázek 2: Orientovaná hrana

Rozdělení hran na orientované a neorientované přirozeně vede na rozdělení grafů do tří kategorií:

- *neorientované grafy*, které obsahují pouze neorientované hrany,
- *orientované grafy*, které obsahují pouze orientované hrany,
- *smíšené grafy*, které obsahují jak orientované, tak neorientované hrany.

Pro úplnost dodejme, že některé grafy mohou obsahovat hrany, jejichž oba koncové vrcholy jsou totožné (tzv. *smyčky*), nebo mít dva uzly spojeny více než jednou hranou (tzv. *násobné hrany).* Takovými grafy se v této práci nebudeme zabývat. Vystačíme si pouze s grafy, které neobsahují smyčky, ani násobné hrany. Takové grafy se nazývají *obyčejné grafy.*

Nyní zavedeme formální definice pojmů z teorie grafů, které se v textu budou dále vyskytovat.

2.1 Grafy formálně

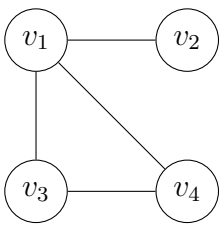
Definice 1 (Neorientovaný graf)
Neorientovaný graf G je dvojice $\langle V, E \rangle$, kde

- $V = \{v_1, v_2, \ldots, v_n\}$ je neprázdná množina *vrcholů* grafu G,
- $E \subseteq \{\{v_1, v_2\} \mid v_1 \neq v_2, v_1 \in V, v_2 \in V\}$ je množina *(neorientovaných) hran* grafu G.

Počet vrcholů grafu G značíme $|V|$, počet hran grafu G značíme $|E|$.

Hranou neorientovaného grafu tedy rozumíme dvouprvkovou množinu vrcholů $\{v_1, v_2\}$, kde $v_1, v_2 \in V$.

Příklad 2 (Neorientovaný graf)
Na obrázku 3 vidíme diagram neorientovaného grafu $G = \langle V, E \rangle$, kde $V = \{v_1, v_2, v_3, v_4\}$ a $E = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_3, v_4\}, \{v_1, v_4\}\}$.



Obrázek 3: Neorientovaný graf

Pro potřeby této práce si vystačíme s neorientovanými obyčejnými grafy. Proto budeme dále v textu pod pojmem graf rozumět neorientovaný obyčejný graf, pokud nebude uvedeno jinak.

Definice 3 (Podgraf)
Graf $\langle V_1, E_1 \rangle$ je podgrafem grafu $\langle V_2, E_2 \rangle$, právě když $V_1 \subseteq V_2$ a $E_1 \subseteq E_2$.

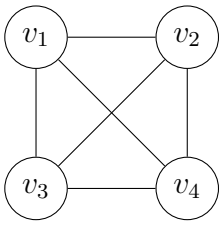
Definice 4 (Koncové vrcholy)
Vrcholy v_1, v_2 neorientované hrany $e = \{v_1, v_2\}$ nazveme *koncové vrcholy* hrany e, nebo také vrcholy *incidentní* s hranou e.

Definice 5 (Sousední vrcholy)
Mějme graf $G = \langle V, E \rangle$. Vrcholy $v_1, v_2 \in V$ nazveme *sousední vrcholy*, jestliže v E existuje hrana $\{v_1, v_2\}$.

Definice 6 (Stupeň vrcholu)
Mějme graf $G = \langle V, E \rangle$ a vrchol $v \in V$. Počtu hran z E, které mají vrchol v jako jeden z koncových vrcholů, říkáme *stupeň vrcholu* v.

Definice 7 (Úplný graf)
Mějme graf $G = \langle V, E \rangle$. Graf G nazveme *úplný graf*, pokud pro každé dva různé vrcholy $v_1, v_2 \in V$ platí, že jsou sousední.

Příklad 8 (Úplný graf)
Na obrázku 4 vidíme diagram úplného grafu, který obsahuje čtyři vrcholy.



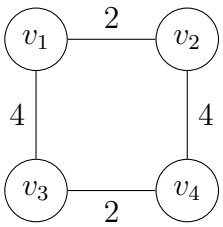
Obrázek 4: Úplný graf

Někdy může být užitečné, aby vrcholy nebo hrany daného grafu nesly doplňující informaci. Jako příklad můžeme uvést situaci, kdy budeme chtít grafem reprezentovat síť měst. Vrchol by v takovém grafu reprezentoval některé z měst. Hrana by pak odrážela skutečnost, že existuje silniční spojení mezi danými městy. Přirozeně bychom mohli požadovat, aby hrany reflektovaly skutečnou vzdálenost daných silničních spojení – tedy aby každá hrana měla doplňující informaci o vzdálenosti mezi městy. Takovéto doplňující informaci se v teorii grafů říká ohodnocení.

Definice 9 (Ohodnocení)
Mějme graf $G = \langle V, E \rangle$ a množinu hodnot D. Zobrazení $w : E \mapsto D$ se nazývá *hranové ohodnocení* grafu G s množinou hodnot D. Zobrazení $w : V \mapsto D$ se nazývá *vrcholové ohodnocení* grafu G s množinou hodnot D.

Grafu společně s ohodnocením(i) říkáme *hranově ohodnocený*, *vrcholově ohodnocený*, nebo jen *ohodnocený* graf. Množina hodnot D může obsahovat libovolné prvky nebo datové struktury. Nejčastěji se setkáme se situací, kdy D je číselná množina. Hodnota $w(e) \in D$ pak může představovat například délku hrany e. Hodnota $w(e)$ se někdy nazývá váha, délka nebo cena hrany e.

Příklad 10 (Ohodnocený graf)
Na obrázku 5 vidíme diagram ohodnoceného grafu. Hranové ohodnocení w je dáno předpisem $w(\{v_1, v_2\}) = 2$, $w(\{v_3, v_4\}) = 2$, $w(\{v_1, v_3\}) = 4$, $w(\{v_2, v_4\}) = 4$.



Obrázek 5: Ohodnocený graf

2.2 Cestování v grafech

Vraťme se nyní k situaci, kdy ohodnocený graf reprezentoval síť měst a silničních spojení mezi nimi. Hranové ohodnocení v tomto grafu odpovídá délce jednotlivých silničních spojení. Na takovém grafu je možné provádět různé typy úloh. Jako příklad uveďme úlohu, při které chceme nalézt nejkratší cestu z daného města A do města B. Cestu si lze intuitivně představit jako průchod mezi jednotlivými sousedními vrcholy po hranách, které tyto vrcholy spojují.

Definice 11 (Cestování)
Sled v neorientovaném grafu $\langle V, E \rangle$ je posloupnost $v_0, e_1, v_1, e_2, v_2, \ldots, e_n, v_n$, kde $v_i \in V$ jsou vrcholy, $e_j \in E$ jsou hrany a platí $e_i = \{v_{i-1}, v_i\}$ pro $i = 1, \ldots, n$. Číslo n se nazývá *délka sledu*. Sled $v_0, e_1, v_1, e_2, v_2, \ldots, e_n, v_n$ se nazývá

- *uzavřený*, je-li $v_0 = v_n$,
- *tah*, platí-li $e_i \neq e_j$ pro $i \neq j$,

- *cesta*, platí-li $v_i \neq v_j$ pro $i \neq j$,
- *kružnice*, je-li $v_0 = v_n$ a s výjimkou vrcholů v_0 a v_n jsou každé dva vrcholy různé.

Délka sledu $v_0, e_1, v_1, e_2, v_2, \ldots, e_n, v_n$ *v hranově ohodnoceném grafu* je číslo $w(e_1) + \cdots + w(e_n)$. Délce sledu se také někdy říká váha, nebo cena sledu. *Vzdálenost* z vrcholu u do v je v hranově ohodnoceném grafu délka cesty z u do v, která má nejmenší délku ze všech cest z u do v vzhledem k hranovému ohodnocení.

Definice 12 (Souvislý graf)
Mějme graf $G = \langle V, E \rangle$. Graf G nazveme *souvislý graf*, právě když pro každé dva vrcholy $v_1, v_2 \in V$ existuje sled z v_1 do v_2.

Definice 13 (Hamiltonovská kružnice)
Kružnici nazveme *hamiltonovskou*, jestliže obsahuje všechny uzly grafu. Graf G nazveme *hamiltonovský*, pokud v něm existuje hamiltonovská kružnice.

Příklad 14 (Hamiltonovská kružnice)
Na obrázku 5 je posloupnost $v_1, \{v_1, v_2\}, v_2, \{v_2, v_4\}, v_4, \{v_4, v_3\}, v_3, \{v_3, v_1\}, v_1$ sled. Tento sled je také kružnice, a protože obsahuje všechny vrcholy grafu, je také hamiltonovská kružnice. Délka tohoto sledu je dána součtem $w(\{v_1, v_2\}) + w(\{v_2, v_4\}) + w(\{v_4, v_3\}) + w(\{v_3, v_1\})$ a je tedy rovna číslu 12.

Zjistit, zda je daný graf hamiltonovský, je obecně netriviální úloha. Platí, že každý úplný graf, který obsahuje nejméně tři vrcholy, je hamiltonovský. Této vlastnosti úplných grafů se při řešení různých grafových úloh často využívá, jak později uvidíme.

3 Problém obchodního cestujícího

Následující kapitoly předpokládají základní znalost pojmů používaných pro analýzu algoritmů.

Jednou z vůbec nejznámějších úloh v teorii grafů je úloha, která se nazývá *problém obchodního cestujícího* – dále jen *TSP* (z anglického *Travelling Salesman Problem*). Zadání úlohy je na první pohled jednoduché – v síti n měst, které jsou navzájem propojeny silničními spojeními o známých délkách, hledáme nejkratší cestu, která prochází každým z měst a končí v počátečním městě.

Navzdory poměrně jednoduchému zadání se TSP řadí mezi NP-těžké problémy. Není tedy znám algoritmus, který by nalezl optimální řešení dané instance TSP v polynomiálním čase. TSP lze také klasifikovat jako tzv. *optimalizační problém*.

Následující definice vycházejí z [3].

Definice 15 (Optimalizační problém)
Optimalizační problém Π je čtveřice $\Pi = \langle L, sol, cost, goal \rangle$, kde

- L je množina vstupních instancí problému,
- sol je zobrazení, které vstupní instanci přiřazuje množinu přípustných řešení,
- $cost$ je zobrazení, které vstupní instanci a přípustnému řešení přiřazuje cenu tohoto řešení,
- $goal$ je buď max, nebo min. Pokud je goal max, Π se nazývá *maximalizační problém*. Pokud je goal min, Π se nazývá *minimalizační problém*. Pro $x \in L$ se $y \in sol(x)$ nazývá *optimální řešení instance* x, pokud platí $cost(x, y) = goal\{cost(x, y') \mid y' \in sol(x)\}$. Cena optimálního řešení se značí $Opt_{\Pi}(x)$.

Každý z optimalizačních problému lze tedy podle definice popsat čtveřicí $\langle L, sol, cost, goal \rangle$. Definujme nyní tímto způsobem TSP.

Definice 16 (Problém obchodního cestujícího)
Problém obchodního cestujícího je optimalizační problém, kde

- L je dvojice $\langle G, c \rangle$, kde $G = \langle V, E \rangle$ je neorientovaný úplný graf a zobrazení $c : E \mapsto \mathbb{Q}$ je hranové ohodnocení, které se nazývá vzdálenostní funkce,
- $sol(G, c)$ je množina všech hamiltonovských kružnic grafu G,
- $cost(\langle G, c \rangle, h)$ je délka hamiltonovské kružnice h vzhledem ke vzdálenostní funkci c,
- $goal = min$.

Z definice je zřejmé, že podstatou TSP je nalézt nejkratší možnou hamiltonovskou kružnici daného grafu. Zjistit, zda daný graf G obsahuje hamiltonovskou kružnici, je obecně netriviální problém, jak již bylo uvedeno v kapitole 2.1. Proto se při řešení TSP využívá již zmíněné vlastnosti úplných grafů, která zaručuje, že každý úplný graf s více než třemi vrcholy obsahuje hamiltonovskou kružnici.

3.1 Podproblémy problému obchodního cestujícího

V závislosti na použité vzdálenostní funkci rozlišujeme dva hlavní podproblémy TSP – metrický a Euklidovský TSP.

Definice 17 (Metrický TSP)
Metrický TSP je podproblém TSP, jehož instance jsou dvojice $\langle\langle V, E\rangle, c\rangle$ takové, že vzdálenostní funkce c je metrika, tj. pro libovolné vrcholy $u, v, w \in V$ splňuje c následující axiomy:

1. $c(v, v) = 0$,
2. $c(u, v) = c(v, u)$,
3. $c(u, v) \leq c(u, w) + c(w, v)$.

První axiom v předchozí definici říká, že vzdálenost z vrcholu v do totožného vrcholu v se rovná nule. Podle druhého axiomu je vzdálenost z vrcholu v do vrcholu u stejná, jako vzdálenost z vrcholu u do vrcholu v. Třetí axiom se nazývá trojúhelníková nerovnost a tvrdí, že součet délek dvou stran v trojúhelníku nikdy není menší, než délka třetí strany.

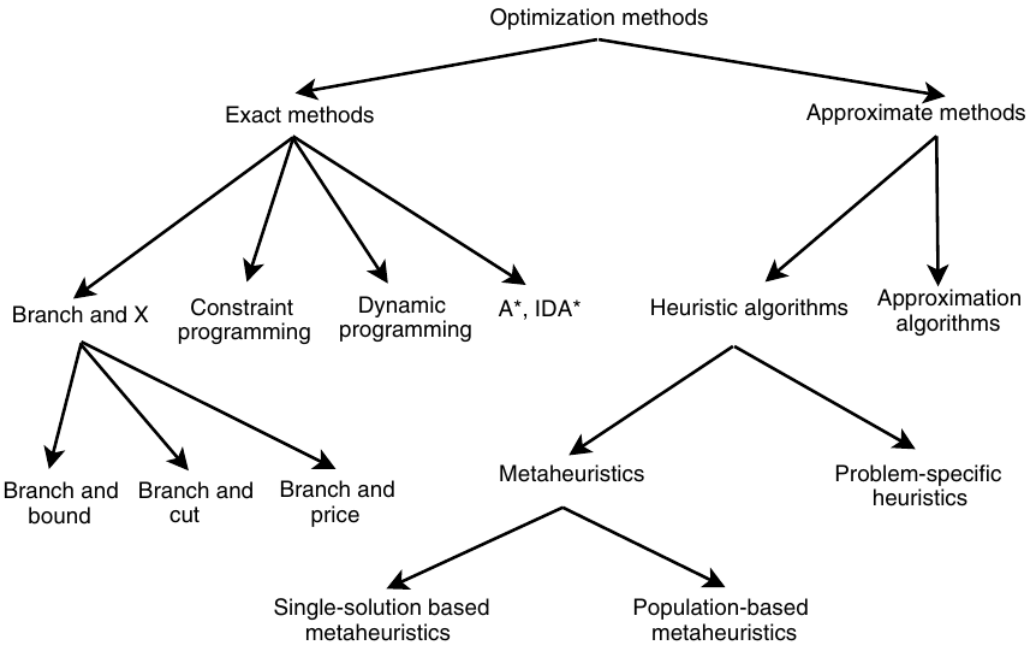
Definice 18 (Euklidovský TSP)
Euklidovský TSP je podproblém TSP, jehož instance jsou dvojice $\langle\langle V, E\rangle, c\rangle$ takové, že vrcholy z V mohou být položeny do roviny tak, že pro libovolné vrcholy $u = (u_1, u_2), v = (v_1, v_2), u, v \in V$, které chápeme jako dvojice souřadnic v rovině, je $c(\{u, v\})$ Euklidovská vzdálenost mezi u a v, tj. platí $c(\{u, v\}) = \sqrt{(v_1 - u_1)^2 + (v_2 - u_2)^2}$.

Metrický TSP je stále NP-těžká úloha, avšak díky axiomům, které splňuje, je snazší navrhnout efektivní aproximační algoritmus. Je zřejmé, že Euklidovský TSP je podproblémem metrického TSP. V následujícím textu budeme pod pojmem TSP rozumět výhradně Euklidovský TSP.

3.2 Metody řešení problému obchodního cestujícího

Tato kapitola vznikla především na základě poznatků ze zdroje [5]. Jejím účelem je ve stručnosti poskytnout ucelený přehled o možnostech řešení optimalizačních problémů a v tomto kontextu představit techniku optimalizace mravenčí kolonií.

Způsobů, kterými lze řešit optimalizační problémy, je poměrně velké množství. Pro snadnější orientaci v optimalizačních metodách je přiložen obrázek 6. Vybrané optimalizační metody dále stručně popíšeme.



Obrázek 6: Optimalizační metody [5]

Exaktní metody

Mezi exaktní metody se řadí algoritmy, které se vyznačují tím, že poskytují optimální řešení a garantují jeho optimalitu. Nevýhodou exaktních metod je skutečnost, že pro celou řadu optimalizačních problémů jsou použitelné jen pro malé instance. V případě řešení TSP exaktními metodami (například pomocí dynamického programování) se dostaneme nejlépe na exponenciální časovou složitost. Z tohoto důvodu se pro řešení větších instancí často přistupuje k algoritmům, které negarantují optimalitu nalezeného řešení, ale zato dokáží nalézt řešení podstatně rychleji než exaktní algoritmy.

Aproximační metody

Aproximační metody se dělí do dvou tříd – heuristické algoritmy a aproximační algoritmy. Pro obě třídy je společné, že negarantují nalezení optimálního řešení. Nalezení řešení je ale podstatně rychlejší, než v případě exaktních algoritmů.

Aproximační algoritmy

Nejvýznamnější vlastností, která charakterizuje třídu aproximačních algoritmů, je garance na mez kvality nalezeného řešení. ϵ-aproximační algoritmus garantuje

nalezení řešení, jehož cena není horší, než cena optimálního řešení vynásobená aproximačním faktorem ϵ.

Definice 19 (ϵ-aproximační algoritmus)
Algoritmus A řešící optimalizační problém Π má *aproximační faktor* ϵ, pokud jeho časová složitost je polynomiální a pro libovolnou vstupní instanci $x \in L$ vždy nalezne takové řešení $y \in sol(x)$, pro které (v kontextu minimalizace) platí:

- $cost(x, y) \leq \epsilon \cdot Opt_{\Pi}(x)$ pokud $\epsilon > 1$,
- $cost(x, y) \geq \epsilon \cdot Opt_{\Pi}(x)$ pokud $\epsilon < 1$,

kde ϵ je konstanta nebo funkce velikosti vstupní instance.

Mezi aproximační algoritmy řešící TSP se řadí kostrový 2-aproximační algoritmus (Double Tree) a Christofidesův 1,5-aproximační algoritmus.

Heuristické algoritmy

Heuristické algoritmy, na rozdíl od aproximačních algoritmů, neposkytují garanci na mez kvality nalezeného řešení. V praxi se často využívají z toho důvodu, že obvykle poskytují „dostatečně kvalitní“ řešení v „rozumném“ čase.

Rozlišujeme dvě hlavní třídy heuristických algoritmů. První z nich jsou algoritmy, které řeší konkrétní optimalizační problém. Tyto algoritmy jsou navrženy speciálně pro potřeby daného problému. Druhou třídou jsou *metaheuristiky*.

Metaheuristiky

Slovo *heuristika* má původ ve slovu *heuriskein* ze staré řečtiny, které znamená „umění objevovat nové strategie (pravidla) pro řešení problémů“. Předpona *meta*, také ze staré řečtiny, znamená „metodologie vyšší úrovně“. V informatice je *metaheuristika* metodologie (šablona) vyšší (obecnější) úrovně, která slouží jako řídící strategie pro návrh základních heuristik pro řešení konkrétních optimalizačních problémů [5].

Metaheuristiky jsou v posledních letech velmi populární přístup k řešení optimalizačních problémů. Lze je klasifikovat podle mnohých kritérií. Jedna z často používaných klasifikací je rozdělení metaheuristik na *metaheuristiky založené na jednom řešení* a *metaheuristiky založené na populaci*.

Princip metaheuristik založených na jednom řešení (například lokální prohledávání) lze velmi zjednodušeně popsat tak, že si po celou dobu běhu udržují jedno aktuální přípustné řešení. Počáteční přípustné řešení může být vypočítáno jiným heuristickým algoritmem. Modifikacemi aktuálního řešení tyto algoritmy iterativně přechází na nová, kvalitnější přípustná řešení.

Oproti tomu metaheuristiky založené na populaci se vyznačují iterativním zlepšováním na základě celé populace řešení. Díky tomu dokáží prohledat širší část množiny přípustných řešení instance daného problému. Do této kategorie metaheuristik spadají i metaheuristiky založené na *inteligenci hejna*.

Metaheuristiky založené na inteligenci hejna

Inteligence hejna je technika umělé inteligence, která se zabývá přírodními i umělými systémy, složenými z mnoha decentralizovaných a samoorganizujících jedinců. Zejména se tato technika zabývá kolektivním chováním, které je důsledkem lokálních interakcí jednotlivých jedinců a interakcí s prostředím, ve kterém se jedinci nacházejí [6].

Pro systémy inteligence hejna jsou typické následující vlastnosti [6]:

- jsou složeny z mnoha jedinců,
- jedinci jsou relativně homogenní (buď jsou identičtí, nebo spadají do některé z daných typologií),
- interakce mezi jedinci jsou založeny na jednoduchých pravidlech, které využívají pouze lokálních informací, které si jedinci předávají přímo, nebo skrze prostředí,
- celkové chování systému je dáno interakcemi jednotlivých jedinců a interakcemi jedinců s prostředím – to znamená, že skupinové chování je samoorganizující.

Navenek složité chování systému tedy není dáno centralizovaným řízením, ale pouze lokálními interakcemi jednotlivých jedinců a interakcemi jedinců s prostředím, ve kterém se nacházejí.

Jednou z významných rodin metaheuristik jsou *metaheuristiky založené na inteligenci hejna*. Často jsou inspirovány přírodními systémy – například systémy včel, ryb, světlušek nebo mravenců. Mezi nejúspěšnější metaheuristiky založené na inteligenci hejna, které byly aplikovány na velký počet optimalizačních problémů, se řadí *optimalizace hejnem částic* a *optimalizace mravenčí kolonií*.

4 Optimalizace mravenčí kolonií

Tato kapitola vznikla na základě poznatků z knihy [7]. Jedním ze dvou autorů této knihy je Marco Dorigo, italský informatik, který ve své disertační práci „Optimization, learning and natural algorithms“ v roce 1992 poprvé představil optimalizaci mravenčí kolonií. Jak Dorigo uvádí ve knize [7], algoritmy založené na optimalizaci mravenčí kolonií jsou nejúspěšnější a nejrozšířenější algoritmy založené na chování mravenců. Úspěch optimalizace mravenčí kolonií dokazuje široký počet problémů, na které byla tato technika aplikována a především fakt, že pro mnohé problémy se algoritmy založené na optimalizaci mravenčí kolonií řadí mezi ty, které vykazují nejlepší výsledky.

4.1 Inspirace v přírodě

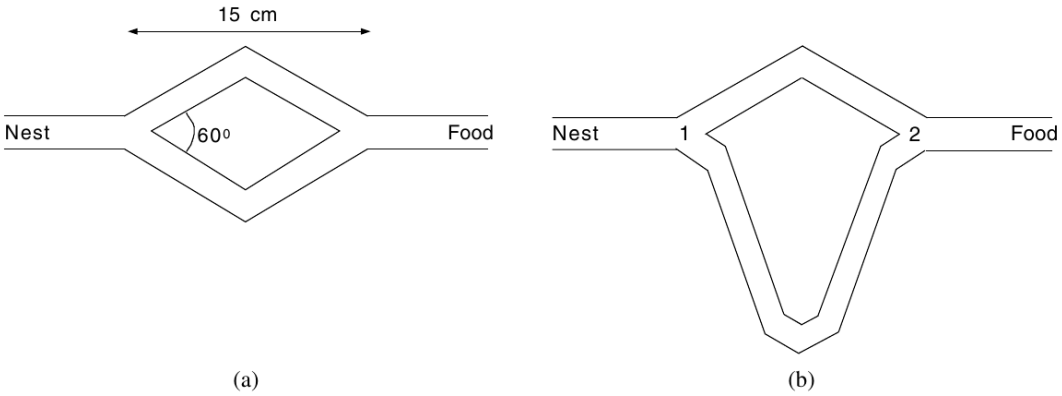
Mravenci jsou sociální hmyz žijící v koloniích. I přes absenci výraznější inteligence jsou mravenci velmi úspěšný druh hmyzu, což je přikládáno především jejich důmyslnému sociálnímu uspořádání. Na základě tohoto uspořádání jsou mravenčí kolonie schopny provádět úlohy, které v mnoha případech výrazně převyšují individuální schopnosti jednotlivých mravenců. Pozorováním systémů sociálního hmyzu biologové zjistili, že složité chování a samoorganizaci těchto systémů lze vysvětlit jistou formou nepřímé komunikace mezi jednotlivými jedinci skrze změny prostředí, ve kterém se jedinci nacházejí. Tato forma nepřímé komunikace se nazývá *stigmergie*. V případě kolonií mravenců se jedná o nepřímou komunikaci skrze *feromony* – chemické látky, které mravenci vypouštějí do prostředí.

4.1.1 Experiment dvojitého mostu

Chování mnoha druhů mravenců hledajících potravu, například *Iridomyrmex humilis*, je založeno na nepřímé komunikaci zprostředkované feromony. Mravenci při cestě od zdroje jídla do mraveniště a obráceně vypouštějí do prostředí feromon. Tímto způsobem tvoří na své cestě feromonovou stopu. Jiní mravenci, kteří také hledají potravu, mohou feromon vycítit. Na základě feromonu v prostředí pak mají tendenci vybírat cesty, které obsahují silnou koncentraci feromonu [7].

Chováním mravenců hledajících potravu se zabývalo hned několik vědců. Jedním z experimentů, který stojí za zmínku, je experiment provedený Deneubourgem a jeho kolegy v roce 1989. Mezi mraveništěm mravence argentinského (*Iridomyrmex humilis*) a zdrojem jídla byl v tomto experimentu umístěn most s dvěma rameny, neboli dvojitý most. Poměr $r = \frac{l_l}{l_s}$, kde l_l je délka delšího ramena mostu a l_s je délka kratšího ramena mostu, značil poměr mezi délkami obou ramen mostu.

V prvním experimentu (viz (a) na obrázku 7) byl poměr $r = 1$, tedy obě ramena dvojitého mostu měly stejnou délku. Na začátku experimentu byl mravencům umožněn pohyb mezi mraveništěm a zdrojem jídla a v průběhu času bylo monitorováno, kolik procent mravenců si vybralo jedno či druhé rameno



Obrázek 7: Experiment dvojitého mostu. (a) Ramena mají stejnou délku. (b) Ramena mají rozdílnou délku [7]

dvojitého mostu. V tomto experimentu vědci zjistili, že ačkoliv se na začátku objevovaly náhodné volby jednoho či druhého ramene, ve výsledku téměř všichni mravenci po určité době používali pouze jedno z ramen mostu. Vysvětlení tohoto jevu je následující. Skutečnost, že se na začátku experimentu objevovaly náhodné volby mravenců, lze vysvětlit tím, že na ramenech mostu zatím nebyl přítomen vůbec žádný feromon. Mravenci si tedy náhodně vybírali jedno z ramen mostu. Postupem času se nicméně vlivem náhody stalo více frekventované jedno z ramen mostu, a proto se na něj ukládalo více feromonu, který zase přilákal více mravenců, kteří ukládali zase další feromon, a tak dále. Tímto autokatalytickým procesem se jedno z ramen mostu stalo preferovaným a naprostá většina mravenců tak postupem času začala využívat pouze jedno z ramen dvojitého mostu.

Ve druhém experimentu (viz (b) na obrázku 7) byl poměr $r = 2$, tedy delší z ramen dvojitého mostu bylo dvakrát delší než kratší z ramen. Na začátku experimentu byl mravencům umožněn pohyb mezi mraveništěm a zdrojem jídla, podobně jako v prvním experimentu. Protože obě ramena mostu vypadala pro mravence identicky, dalo se očekávat, že při cestě za potravou bude přibližně polovina mravenců volit jedno rameno mostu a druhá polovina mravenců druhé rameno mostu. V průběhu času se ovšem ukázalo, že naprosto drtivá většina většina mravenců si při své cestě zvolila kratší rameno mostu. Tento jev lze vysvětlit následovně. Podobně jako v prvním experimentu, náhodné volby mravenců na začátku lze vysvětlit nulovou koncentrací feromonu na ramenech dvojitého mostu. Nicméně mravenci, kteří si vybrali kratší z ramen, se dostali za potravou a zpátky do mraveniště rychleji, než mravenci, kteří si vybrali delší z ramen. Z tohoto důvodu se na kratším ramenu dvojitého mostu začal feromon hromadit rychleji, než na delším ramenu a v důsledku autokatalytického procesu, již známého z prvního experimentu, se kratší rameno rychle stalo preferovaným. Jak dále uvádí Dorigo, v porovnání s prvním experimentem byl v druhém experi-

mentu vliv počátečních náhodných voleb mnohem menší, více se uplatnil vliv stigmergie, autokatalytického procesu a rozdílných délek ramen [7].

Experiment dvojitého mostu tedy prokázal, že mravenci mají schopnost nalézt nejkratší cestu mezi mraveništěm a zdrojem potravy na základě stigmergie – formy nepřímé komunikace, v případě mravenců zprostředkované feromony. Právě popsané chování mravenců hledajících potravu se stalo inspirací pro třídu mravenčích algoritmů, známou jako *optimalizace mravenčí kolonií* – dále jen ACO (z anglického *Ant Colony Optimization*). Stěžejní myšlenkou ACO je vytvořit kolonii umělých mravenců, inspirovaných skutečnými mravenci a na základě umělé stigmergie je využít k řešení obtížných optimalizačních úloh.

4.2 Metaheuristika optimalizace mravenčí kolonií

Jak již bylo uvedeno, ACO je metaheuristika, která pro řešení obtížných diskrétních optimalizačních úloh využívá kolonie umělých mravenců. Umělý mravenec je v ACO stochastická konstruktivní[1] procedura. Proto lze ACO využít pro řešení všech optimalizačních problémů, pro které je možné sestavit konstruktivní heuristický algoritmus. ACO již byla aplikována na široké množství problémů, mezi nimi také problém batohu, množinového pokrytí, barvení grafu, nebo TSP. Právě TSP byl vůbec první problém, na který byla ACO aplikována. Protože aplikace ACO na TSP je hlavním tématem této práce, nebudeme detailně rozebírat obecný popis metaheuristiky ACO a způsoby, jakými ji lze aplikovat na jiné optimalizační problémy. Namísto toho rovnou představíme ACO aplikovanou na TSP.

[1] Konstruktivní algoritmy jsou typ heuristických algoritmů. Do počátečního prázdného řešení iterativně přidávají komponenty řešení až do té doby, než sestaví kompletní přípustné řešení.

5 Optimalizace mravenčí kolonií na problému obchodního cestujícího

Tato kapitola čerpá informace z knihy [7]. Následující pojmy budou uváděny v kontextu aplikace ACO na TSP.

Jak již bylo uvedeno v kapitole 4.2, umělý mravenec je v ACO stochastická konstruktivní procedura. Představme nyní umělé mravence podrobněji. Mějme instanci TSP $\langle\langle V, E\rangle, d\rangle$. Označme jako e_{ij} hranu $e = \{i, j\} \in E$. Označme dále jako d_{ij} hodnotu $d(e_{ij})$, kde d je vzdálenostní funkce (z anglického *distance*). Ke každé hraně e_{ij} je přidružena hodnota τ_{ij}, která se nazývá *feromonová stopa*, a dále hodnota η_{ij}, která se nazývá *heuristická informace*. Feromonová stopa je upravována samotnými mravenci a představuje jakousi dlouhodobou komplexní paměť mravenčí kolonie o procesu hledání optimálního řešení daného problému. V případě aplikace ACO na TSP představuje pro mravence τ_{ij} informaci o tom, jak moc je žádoucí (výhodné) přejít z vrcholu i do vrcholu j po hraně e_{ij}. Heuristická informace reprezentuje určitou apriorní informaci o instanci daného problému. V případě aplikace ACO na TSP je zvolena jako $\eta_{ij} = 1/d_{ij}$. To znamená, že čím kratší je délka hrany d_{ij}, tím vyšší je hodnota heuristické informace η_{ij}. Feromonová stopa a heuristická informace jsou důležité hodnoty, které mravenci používají při pravděpodobnostním výběru dalšího vrcholu, který navštíví. Každý mravenec má dále paměť $\mathcal{M}$, která má hned několik funkcí. Jednou z nich je zapamatování si vrcholů, které mravenec v průběhu sestavování řešení navštívil. Konstruktivní proceduru umělého mravence lze popsat následovně:

1. je zvolen počáteční vrchol, do kterého je mravenec umístěn,
2. na základě feromonové stopy a heuristické informace je pravděpodobnostním přidáváním vrcholů, které mravenec ještě nenavštívil, postupně sestavováno přípustné řešení dané instance problému,
3. na závěr se mravenec vrátí do počátečního vrcholu.

Poté, co mravenec tímto způsobem sestaví přípustné řešení, může aktualizovat feromon na hranách, které v průběhu sestavování řešení navštívil. Hrany, na kterých má být feromon aktualizován, nalezne mravenec pomocí své paměti $\mathcal{M}$. V případě některých mravenčích algoritmů může být ještě před aktualizací feromonu aplikováno lokální prohledávání na řešení, které mravenec nalezl.

Po bližším představení funkce umělých mravenců můžeme nyní v pseudokódu ACOMetaheuristicTSP ukázat obecný vysokoúrovňový popis ACO aplikované na TSP. V prvním kroku se nastaví parametry potřebné pro běh algoritmu a inicializují feromonové stopy. Poté se ve smyčce, dokud není splněna některá z ukončovacích podmínek algoritmu, opakují tři procedury – `ConstructAntSolutions`, `ApplyLocalSearch` a `UpdatePheromones`. V proceduře `ConstructAntSolutions` každý z mravenců sestaví přípustné řešení způsobem, který byl popsán výše. V závislosti na implementaci algoritmu může být sestavování řešení mravenci prováděno paralelně. V dalším kroku lze volitelně provést zlepšení nalezených

řešení lokálním prohledáváním v rámci procedury `ApplyLocalSearch`. V posledním kroku jsou v proceduře `UpdatePheromones` aktualizovány feromonové stopy. V závislosti na konkrétním algoritmu je některým mravencům umožněno na hrany uložit množství feromonu, které závisí na kvalitě (ceně) řešení, které daný mravenec nalezl. Platí, že čím lepší řešení mravenec nalezne, tím více feromonu může na hranu uložit. Tímto způsobem jsou v dalších iteracích algoritmu při pravděpodobnostním výběru hran mravenci zvýhodněny hrany, které se často objevovaly v kvalitních řešeních. Je tomu tak proto, že čím vyšší je intenzita feromonové stopy na hraně, tím vyšší je pravděpodobnost, že si mravenec při sestavování řešení tuto hranu vybere, jak již bylo uvedeno výše. Do aktualizace feromonových stop kromě samotných mravenců ještě zasahuje proces odpařování feromonu. V důsledku tohoto procesu v průběhu času postupně klesá intenzita feromonových stop. To je výhodné, protože tímto způsobem je kolonii umožněno „zapomenout“ neoptimální cesty, které mravenci zpravidla nacházejí zejména v prvních iteracích algoritmu.

Procedure ACOMetaheuristicTSP

```
1 nastav parametry, inicializuj feromonové stopy
2 while ukončovací podmínka neplatí do
3     ConstructAntsSolutions
4     ApplyLocalSearch          % volitelně
5     UpdatePheromones
6 end
```

5.1 Algoritmy optimalizace mravenčí kolonií aplikované na problém obchodního cestujícího

Jak uvádí Dorigo v [7], vůbec prvním ACO algoritmem byl *Ant System* (dále jen AS), který byl demonstrován právě na TSP. Autorem AS je sám Marco Dorigo a vznikl v roce 1992. Dorigo dále uvádí, že AS dosáhl povzbudivých výsledků, leč ne tak dobrých, jako poskytovaly nejlepší známé algoritmy řešící TSP. Důležitost AS tak spočívá především v inspiraci pro další ACO algoritmy, které často AS rozšiřují a poskytují lepší výsledky. Přehled vybraných důležitých ACO algoritmů aplikovaných na TSP se nachází v tabulce 1. Některá rozšíření AS, jako je EAS, MAX-MIN AS nebo Rank-Based AS zachovávají stejnou proceduru, ve které mravenci sestavují řešení a proceduru odpařování feromonu, jako v původním AS. Uvedená rozšíření AS a samotné AS se tak liší především ve způsobu, kterým jsou aktualizovány feromonové stopy. Některé další ACO algoritmy se od AS liší podstatně více. Mezi ně patří algoritmus Ant-Q a jeho nástupce ACS.

Nyní si podrobněji představíme základní mravenčí algoritmus AS a jeho dvě rozšíření – Elitist AS a Rank-Based AS.

Tabulka 1: Vybrané ACO algoritmy [7]

ACO algoritmus	Autoři
Ant System (AS)	Dorigo (1992)
Elitist AS	Dorigo (1992)
Ant Colony System	Dorigo & Gambardella (1997)
Max-Min AS	Stützle & Hoos (1996)
Rank-Based AS	Bullnheimer, Hartl, & Strauss (1997)

5.2 Ant System

Popis algoritmu AS a jeho rozšíření vychází z popisu Marca Doriga v knize [7].

Algoritmus AS z vysokoúrovňového pohledu odpovídá popisu z pseudokódu ACOMetaheuristicTSP. Algoritmus má dvě stěžejní fáze – fázi, ve které mravenci sestavují přípustná řešení a fázi, ve které se aktualizují feromonové stopy. Lokální prohledávání v tomto algoritmu není použito.

Na začátku běhu algoritmu musí být vhodně nastaveny počáteční hodnoty feromonových stop. Dorigo v [7] uvádí, že pro tento algoritmus je vhodné feromonové stopy inicializovat na hodnoty o trochu vyšší, než je očekávané množství feromonu, který mravenci uloží na hrany v jedné iteraci. Je tomu tak proto, že pokud jsou počáteční hodnoty feromonových stop příliš nízké, tak je hledání optimálního řešení příliš ovlivněno prvními vygenerovanými řešeními, což obecně vede k tomu, že je prohledávána užší část množiny přípustných řešení (tzv. prohledávacího prostoru). Nicméně, pokud jsou počáteční hodnoty feromonových stop příliš vysoké, tak bude trvat nějaký čas, než se feromon odpaří do stavu, kdy feromon ukládaný mravenci bude moct efektivně ovlivnit hledání optimálního řešení. Vhodné počáteční nastavení feromonových stop, které budeme značit τ_0, je dáno vztahem

$$\forall\{i,j\} \in E : \tau_{ij} = \tau_0 = \frac{m}{C^{nn}}, \tag{1}$$

kde m je počet mravenců a C^{nn} je cena přípustného řešení, vypočítaného některým jiným konstruktivním heuristickým algoritmem. Tento algoritmus je vhodné zvolit tak, aby rychle nalezl přípustné řešení. V praxi je často využíván například algoritmus *Nearest Neighbor*.

5.2.1 Fáze sestavování přípustných řešení mravenci

V AS může všech m mravenců sestavovat přípustné řešení současně. Na začátku je každý z mravenců umístěn do náhodně zvoleného počátečního vrcholu. Poté každý z mravenců sestaví kompletní přípustné řešení. V každém kroku sestavování řešení si mravenec k musí na základě pravděpodobnostního výběru zvolit další vrchol, do kterého se přemístí. Pravděpodobnost, se kterou si mravenec k,

který se aktuálně nachází ve vrcholu i vybere vrchol j, lze vyjádřit jako:

$$p_{ij}^k = \frac{[\tau_{ij}]^\alpha [\eta_{ij}]^\beta}{\sum_{l \in \mathcal{N}_i^k} [\tau_{il}]^\alpha [\eta_{il}]^\beta}, \text{ pokud } j \in \mathcal{N}_i^k, \tag{2}$$

kde $\eta_{ij} = 1 \,/\, d_{ij}$ je heuristická informace, α, β jsou parametry, které určují, jak velký vliv má feromonová stopa a heuristická informace, a $\mathcal{N}_i^k$ je dosažitelné okolí mravence k umístěného ve vrcholu i, neboli množina vrcholů, do kterých mravenec může z vrcholu i přejít a které ještě nenavštívil.

Z rovnice (2) vyplývá, že pravděpodobnost, se kterou si mravenec k umístěný ve vrcholu i vybere jako další vrchol j, je tím vyšší, čím vyšší jsou hodnoty τ_{ij} (množství feromonu) a η_{ij} (apriorní heuristická informace). Parametry α a β ovlivňují, jak velký vliv mají na výběr hodnoty τ_{ij} a η_{ij}. Je-li $\alpha = 0$, pak se vliv feromonové stopy vůbec neuplatňuje – ztrácí se vliv stigmergie a „kolektivní paměti mravenčí kolonie o hledání optimálního řešení“, kterou feromonová stopa představuje. To znamená, že si mravenec vybírá následující vrchol pouze na základě vzdálenosti mezi tímto vrcholem a vrcholem, ve kterém se právě nachází. To odpovídá chování hladových algoritmů. Při hodnotě $\alpha > 1$ naopak rychle dochází k situaci, při které všichni mravenci následují stejnou cestu a sestavují stejné řešení, které je obecně neoptimální. Podobné je to s parametrem β – pokud je jeho hodnota rovna nule, ztrácí se vliv heuristické informace, což vede k neoptimálním výsledkům. Je proto velmi důležité tyto parametry vhodně nastavit tak, aby algoritmus poskytoval co nejoptimálnější výsledky. Obecně neexistuje optimální nastavení parametrů pro každou vstupní instanci TSP. Dorigo v knize [7] popisuje nastavení parametrů, které poskytuje dobré výsledky pro většinu vstupních instancí TSP. Přehled nastavení parametrů pro vybrané ACO algoritmy (bez lokálního prohledávání) se nachází v tabulce 2. Význam parametrů ρ, e a w bude dále v textu vysvětlen.

Tabulka 2: Nastavení parametrů pro vybrané ACO algoritmy [7]

ACO algoritmus	α	β	ρ	m	τ_0
AS	1	2 až 5	0,5	n	$m \,/\, C^{nn}$
EAS	1	2 až 5	0,5	n	$(e+m) \,/\, \rho C^{nn}$
AS_{rank}	1	2 až 5	0,1	n	$0,5r(r-1) \,/\, \rho C^{nn}$

n je počet vrcholů v dané instanci TSP. Některé ACO algoritmy využívají dalších parametrů. Pro algoritmus EAS je to parametr $e = n$. Pro algoritmus AS_{rank} je to parametr $w = 6$.

Množinu vrcholů $\mathcal{N}_i^k$ mravenec k určí za pomoci své paměti $\mathcal{M}^k$. Tato paměť mravence obsahuje vrcholy, které již mravenec v průběhu sestavování navštívil, a to v pořadí, ve kterém je navštívil. Díky tomu je mravenec po sestavení kompletního přípustného řešení T^k schopen vypočítat jeho cenu a posléze uložit feromon na příslušné hrany.

5.2.2 Fáze aktualizování feromonových stop

Fáze aktualizování feromonových stop začíná ve chvíli, kdy je dokončena fáze sestavování přípustných řešení mravenci. Tedy ve chvíli, kdy každý z mravenců sestavil přípustné řešení. Fáze aktualizování feromonových stop probíhá ve dvou krocích. V prvním kroku se uplatňuje mechanismus odpařování feromonu. Tento mechanismus lze popsat jako:

$$\forall\{i,j\} \in E : \tau_{ij} = (1-\rho)\tau_{ij}, \tag{3}$$

kde $0 < \rho \leq 1$ je míra odpařování feromonu. Dorigo v [7] uvádí, že v důsledku tohoto mechanismu hodnota feromonu na hraně, která mravenci opakovaně není vybrána, exponenciálně klesá s počtem iterací algoritmu. Pravděpodobnost, že si takovou hranu mravenci vyberou, tedy klesá s počtem iterací algoritmu. Zvýhodňovány jsou naopak hrany, které byly mravenci vybírány častěji.

Poté, co je ze všech hran vypařen feromon, následuje krok, ve kterém všichni mravenci ukládají feromon na hrany, které navštívili v průběhu sestavování přípustného řešení. Ukládání feromonu mravenci lze popsat jako:

$$\forall\{i,j\} \in E : \tau_{ij} = \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k}, \tag{4}$$

kde $\Delta\tau_{ij}^{k}$ je množství feromonu, které mravenec k uloží na hranu $\{i,j\}$:

$$\Delta\tau_{ij}^{k} = \begin{cases} 1 \,/\, C^k, & \text{pokud hrana } \{i,j\} \text{ náleží do } T^k, \\ 0, & \text{jinak.} \end{cases} \tag{5}$$

kde C^k je cena přípustného řešení T^k sestaveného mravencem k a je vypočítána jako součet délek všech hran, které náleží do T^k. Je zřejmé, že čím lepší je řešení, které mravenec nalezl, tím vyšší je množství feromonu, které mravenec uloží na hrany, které náleží do daného řešení. Tímto způsobem jsou v dalších iteracích algoritmu při výběru hran mravenci zvýhodněny hrany, které se nacházely v kvalitních řešeních.

5.3 Rozšíření algoritmu Ant System

Jak již bylo uvedeno, algoritmus AS byl první algoritmus optimalizace mravenčí kolonií, který byl aplikován na TSP. Výzkum možností tohoto algoritmu ale nadále pokračoval, a tak brzy vznikly algoritmy, které AS upravily a které posléze dosahovaly lepších výsledků než původní AS. Představíme si nyní dvě rozšíření AS, které jsou jeho přímými následníky – algoritmy Elitist AS a Rank-Based AS. Ty se od AS liší v podstatě jen způsobem, kterým jsou aktualizovány feromonové stopy.

Elitist Ant System

Elitist Ant System (EAS) je vůbec prvním publikovaným rozšířením algoritmu AS. Hlavní myšlenka EAS je si po celou dobu běhu algoritmu udržovat v paměti dosavadní nejlepší nalezené řešení (označováno jako T^{bs}) a v každé iteraci při aktualizování feromonových stop přidat dodatečný feromon na hrany, které do T^{bs} náleží. Množství dodatečného feromonu lze vyjádřit jako $e \,/\, C^{bs}$, kde C^{bs} je cena dosavadního nejlepšího nalezeného řešení T^{bs} a e je parametr, kterým lze ovlivnit váhu, kterou přikládáme řešení T^{bs}. Rovnice (4) popisující ukládání feromonu mravenci se tedy pro algoritmus EAS upraví na tvar:

$$\forall\{i,j\} \in E : \tau_{ij} = \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k} + e\Delta\tau_{ij}^{bs}, \tag{6}$$

kde $\Delta\tau_{ij}^{k}$ má stejný význam, jako v rovnici (4) a $\Delta\tau_{ij}^{bs}$ je definováno jako:

$$\Delta\tau_{ij}^{bs} = \begin{cases} e \,/\, C^{bs}, & \text{pokud hrana } \{i,j\} \text{ náleží do } T^{bs}, \\ 0, & \text{jinak.} \end{cases} \tag{7}$$

Rank-Based Ant System

Jedním z dalších algoritmů, který rozšiřuje původní AS, je algoritmus Rank-Based AS (AS_{rank}). V AS_{rank}, na rozdíl od AS, není dovoleno každému mravenci vypouštět feromon. Poté, co je dokončena fáze sestavování přípustných řešení, jsou mravenci seřazeni vzestupně dle ceny řešení, které nalezli. Každému mravenci je tak přiřazena hodnota r, která odpovídá pořadí mravence. V každé iteraci je pouze $w-1$ mravencům dovoleno, aby uložili feromon. Množství feromonu, které mohou mravenci uložit, je ovlivněno hodnotou r, která jim byla přiřazena. Podobně jako v EAS se navíc přidává dodatečný feromon na hrany, které náleží do dosavadního nejlepšího nalezeného řešení T^{bs}. Rovnice (4) popisující ukládání feromonu mravenci je tedy pro algoritmus AS_{rank} upravena na tvar:

$$\forall\{i,j\} \in E : \tau_{ij} = \tau_{ij} + \sum_{r=1}^{w-1} (w-r)\Delta\tau_{ij}^{r} + w\Delta\tau_{ij}^{bs}, \tag{8}$$

kde $\Delta\tau_{ij}^{r}$ je množství feromonu, které mravenec s pořadím r uloží na hranu $\{i,j\}$:

$$\Delta\tau_{ij}^{r} = \begin{cases} 1 \,/\, C^{r}, & \text{pokud hrana } \{i,j\} \text{ náleží do } T^{r}, \\ 0, & \text{jinak.} \end{cases} \tag{9}$$

a $\Delta\tau_{ij}^{bs}$ je definováno jako:

$$\Delta\tau_{ij}^{bs} = \begin{cases} 1 \,/\, C^{bs}, & \text{pokud hrana } \{i,j\} \text{ náleží do } T^{bs}, \\ 0, & \text{jinak.} \end{cases} \tag{10}$$

Z rovnice (8) je tedy patrné, že množství feromonu, které mohou mravenci vypustit, klesá s jejich pořadím r. Největší množství feromonu je přidáno na hrany, které náleží do řešení T^{bs}.

5.4 Implementace algoritmů

Účelem této kapitoly je pomocí pseudokódu ukázat, jak lze implementovat optimalizaci mravenčí kolonií na problému obchodního cestujícího. Popisován je základní algoritmus AS (bez volitelného lokálního prohledávání). Popis vychází z knihy [7].

5.4.1 Použité datové struktury

Na začátku je potřeba zmínit datové struktury, ve kterých jsou ukládány informace o instanci TSP, mravencích, feromonových stopách a dalších.

Způsobů, kterými je možné v počítači reprezentovat graf, je hned několik. V ACO se používá reprezentace, které se říká matice vzdálenosti.

Definice 20 (Matice vzdálenosti)
Mějme graf $G = \langle V, E \rangle$ společně s ohodnocením $w : E \mapsto \mathbb{R}$ a nechť $|V| = n$. Dále mějme bijektivní zobrazení $u : V \mapsto \{1, \ldots, n\}$. Matice vzdálenosti A je čtvercová symetrická matice typu $n \times n$ nad $\mathbb{R}$, pro kterou platí: $\forall i, j \in V : a_{u(i),u(j)} = w(\{i, j\})$.

Matice vzdálenosti tedy při grafu čítajícím n vrcholů má n^2 prvků. Vzhledem k tomu, že je matice symetrická, zabírá v paměti více místa, než by bylo nezbytně nutné. Výhodou nicméně je, že nemusíme kontrolovat, zda se hodnota $w(\{i, j\})$ nalézá v matici vzdálenosti na pozici $a_{u(i),u(j)}$, nebo $a_{u(j),u(i)}$. Pro jednoduchost si vrcholy z V označme hodnotami $0, \ldots, n-1$. V programu matici vzdálenosti reprezentujeme dvojrozměrným polem `real dist[n][n]`. Hodnota $w(\{i, j\})$ je tedy v poli uložena na pozicích `dist[i][j]` a `dist[j][i]`.

Pro každou hranu $\{i, j\} \in E$ dále musíme v paměti uchovat informaci o hodnotě τ_{ij}, tedy o množství feromonu, který se na hraně $\{i, j\}$ nachází. Informace o feromonových stopách v programu reprezentujeme dvojrozměrným polem `real pheromone[n][n]`. Toto pole je opět symetrické, hodnota τ_{ij} je tudíž v poli uložena na pozicích `pheromone[i][j]` a `pheromone[j][i]`.

V průběhu sestavování řešení musí každý z mravenců vypočítat hodnotu $[\tau_{ij}]^\alpha [\eta_{ij}]^\beta$ z rovnice (2). To je velmi neefektivní. Pro urychlení výpočtu je vhodné zavést dodatečné dvojrozměrné pole `real choice_info[n][n]`. Toto pole je podobně jako v předchozích případech symetrické, hodnota $[\tau_{ij}]^\alpha [\eta_{ij}]^\beta$ je tak v daném poli uložena na pozicích `choice_info[i][j]` a `choice_info[j][i]`.

Poslední strukturou, kterou je potřeba zmínit, je struktura reprezentující samotného umělého mravence. Připomeňme, že mravenec k musí být schopen:

- uložit neúplné řešení, které prozatím sestavil,
- určit dosažitelné okolí $\mathcal{N}_i^k$,
- vypočítat cenu kompletního sestaveného řešení.

K ukládání sestavovaného řešení mravenci slouží pole `tour` velikosti $n + 1$. Jeho prvky jsou jednotlivé vrcholy z V. Na pozici `tour[n]` je uložen stejný vrchol, jako na pozici `tour[0]`. To je výhodné pro procedury, které s tímto polem dále pracují.

Ačkoliv by pro určení dosažitelnosti vrcholu j z vrcholu i bylo dostatečné pole `tour`, pro urychlení výpočtu je použito dodatečné pole `visited` velikosti n. Pokud vrchol j již byl navštíven, je `visited[j] == true`. V opačném případě je `visited[j] == false`.

Cena řešení, kterou mravenec vypočítá na základě pole `tour`, je uložena v proměnné `tour_length`.

Jednotliví mravenci jsou uloženi v poli `ant ants[m]` velikosti m.

```
1  structure ant
2  {
3    real tour_length
4    integer tour[n+1]
5    boolean visited[n]
6  }
```

5.4.2 Implementace algoritmu Ant System

Na pseudokódu AntSystem vidíme vysokoúrovňový popis algoritmu AS. Jednotlivé části, ze kterých se skládá, dále podrobně popíšeme.

Procedure AntSystem

```
1 InitializeData
2 while not terminate do
3     ConstructSolutions
4     UpdateStatistics
5     UpdatePheromoneTrails
6 end
```

Inicializace dat

Procedura InitializeData je zavolána na samotném začátku běhu algoritmu a slouží k inicializaci dat potřebných pro běh algoritmu. Je zapotřebí:

1. přečíst instanci TSP,
2. sestavit matici vzdálenosti `dist`,
3. sestavit matici `choice_info`,
4. inicializovat jednotlivé mravence,

5. inicializovat parametry potřebné pro běh algoritmu,
6. inicializovat proměnné, které ukládají statistiky o běhu algoritmu.

Procedure InitializeData

```
1 ReadInstance
2 ComputeDistances
3 ComputeChoiceInfo
4 InitializeAnts
5 InitializeParameters
6 InitializeStatistics
```

Ukončovací podmínka

Ukončovacích podmínek pro algoritmus může být více, záleží na konkrétní implementaci. Běžné ukončovací podmínky jsou:

- byl dosažen maximální počet iterací algoritmu,
- vypršel čas, který byl pro běh algoritmu určen,
- algoritmus vykazuje stagnující chování,
- algoritmus nalezl přijatelné řešení.

Sestavování řešení

Sestavování řešení mravenci probíhá v proceduře ConstructSolutions. Nejprve jsou na řádcích 1 až 5 nastaveny hodnoty v poli `visited` každého z mravenců na `false`. Každý z mravenců tak má všechny vrcholy označeny jako nenavštívené. Na řádcích 7 až 11 se každému mravenci vybere náhodný[2] počáteční vrchol sestavovaného řešení a tento vrchol se označí jako navštívený. Dále na řádcích 12 až 17 probíhá v jednotlivých krocích přidávání dalšího vrcholu do sestavovaného řešení každého z mravenců, až dokud řešení nejsou kompletní. Výběr dalšího vrcholu do sestavovaného řešení každého jednotlivého mravence obstarává procedura `ASDecisionRule`, která je volána na řádku 15. V poslední fázi se na řádku 19 každý mravenec vrátí do počátečního vrcholu a na řádku 20 je v proceduře `ComputeTourLength` vypočítána cena sestaveného řešení každého z mravenců a uložena do jeho paměti.

[2]Náhodný vrchol je vygenerován voláním procedury random{0, . . . , n-1}, která vrací číslo z množiny $\{0, \ldots, n-1\}$.

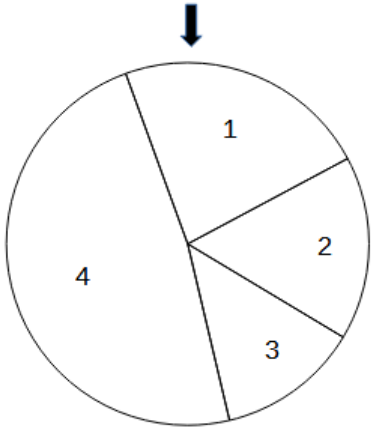
```
Procedure ConstructSolutions
1  for k=0 to m-1 do
2      for i=0 to n-1 do
3          ants[k].visited[i] = false
4      end
5  end
6  step = 0
7  for k=0 to m-1 do
8      r = random{0, ..., n-1}
9      ants[k].tour[step] = r
10     ants[k].visited[r] = true
11 end
12 while step < n do
13     step = step + 1
14     for k=0 to m-1 do
15         ASDecisionRule(k, step)
16     end
17 end
18 for k=0 to m-1 do
19     ant[k].tour[n] = ant[k].tour[0]
20     ant[k].tour_length = ComputeTourLength(k)
21 end
```

Výběr následujícího vrcholu

Výběr následujícího vrcholu, do kterého se mravenec přesune z vrcholu, ve kterém se právě nachází, probíhá v proceduře ASDecisionRule. Nejprve je na řádku 1 do proměnné `c` uložen vrchol, ve kterém se mravenec k aktuálně nachází. Na řádcích 3 až 19 je implementována rovnice (2). Implementace rovnice vychází z techniky *výběru ruletovým kolem*, používané v genetických algoritmech. Představme si pomyslné kolo. Toto kolo rozdělíme na x dílů, odpovídající počtu vrcholů, do kterých může mravenec k přejít z vrcholu c, ve kterém se právě nachází ($x = |\mathcal{N}_c^k|$). Velikost každého dílu je přímo úměrná hodnotě `choice_info[c][j]`, kde $j \in \mathcal{N}_c^k$. Je zřejmé, že čím větší část kola díl zabírá, tím vyšší je je pravděpodobnost, že se na daném dílu kolo zastaví. Toto pomyslné kolo následně roztočíme. Díl, respektive vrchol, na kterém se kolo zastaví, je vybrán jako vrchol, do kterého se mravenec přemístí.

Výběr ruletovým kolem je implementován následovně. Na řádcích 3 až 10 jsou do proměnné `sum_probabilities` sečteny hodnoty `choice_info[c][j]` pro všechny $j \in \mathcal{N}_c^k$. Tím je vytvořeno pomyslné ruletové kolo. Na řádku 11 je do proměnné `r` vygenerováno náhodné číslo z intervalu $(0, sum\_probabilities)$. To je analogie roztočení kola. Poté se na řádcích 14 až 17 do proměnné `p` sčítají pravděpodobnosti výběru jednotlivých vrcholů, dokud není $p >= r$. V ten moment se pomyslné kolo zastaví a příslušný vrchol je vybrán jako následující vrchol, do

kterého se mravenec k přesune.



Obrázek 8: Příklad ruletového kola. S velikostí dílu se zvyšuje pravděpodobnost, že se na tomto dílu kolo zastaví

Procedure ASDecisionRule(k, i)

```
   Input: k    % identifikátor mravence
   Input: i    % aktuální konstrukční krok
 1 c = ants[k].tour[i - 1]
 2 sum_probabilities = 0.0
 3 for j=0 to n-1 do
 4     if ant[k].visited[j] then
 5         selection_probability[j] = 0.0
 6     else
 7         selection_probability[j] = choice_info[c][j]
 8         sum_probabilities = sum_probabilities + selection_probability[j]
 9     end
10 end
11 r = random[0, sum_probabilities]
12 j = 0
13 p = selection_probability[j]
14 while p < r do
15     j = j + 1
16     p = p + selection_probability[j]
17 end
18 ant[k].tour[i] = j
19 ant[k].visited[j] = true
```

Výpočet ceny řešení

Výpočet ceny řešení mravence k probíhá v rámci procedury ComputeTourLength. Cena řešení je vypočítána jednoduše jako součet ohodnocení hran mezi jednotlivými sousedními vrcholy z řešení mravence.

Procedure ComputeTourLength(k)

```
   Input: k    % identifikátor mravence
   Output: res    % cena řešení mravence k
1  res = 0.0
2  for j=0 to n-1 do
3      res = res + dist[ants[k].tour[j]][ants[k].tour[j+1]]
4  end
```

Aktualizace statistik

Poté, co v rámci procedury `ConstructSolutions` všichni mravenci sestaví řešení, je zavolána procedura `UpdateStatistics`. Ta slouží k aktualizaci proměnných, které ukládají statistiky o běhu programu. Může se jednat například o uložení nového nejlepšího nalezeného řešení, aktualizaci počtu iterací algoritmu, které již proběhly, a další.

Aktualizace feromonových stop

Poslední krok v každé iteraci algoritmu AS je aktualizace feromonových stop. Ta probíhá v proceduře UpdatePheromoneTrails. Připomeňme, že aktualizace feromonových stop probíhá ve dvou krocích:

1. odpařování feromonu,
2. ukládání feromonu.

Odpařování feromonu, popsané v rovnici (3), je implementováno v proceduře Evaporate. Ukládání feromonu mravenci pro algoritmus AS je popsáno v rovnici (4) a implementováno v proceduře DepositPheromone. Po aktualizaci feromonových stop je navíc zapotřebí aktualizovat hodnoty v poli `choice_info`. Aktualizace pole `choice_info` probíhá v proceduře ComputeChoiceInformation.

Procedure UpdatePheromoneTrails

```
1  Evaporate
2  for k=0 to m-1 do
3      DepositPheromone(k)
4  end
5  ComputeChoiceInformation
```

Pro úplnost k algoritmu AS dodejme, že vhodným využitím dodatečných datových struktur lze výpočet (zejména pak proceduru `ASDecisionRule`) urychlit za cenu vyšší paměťové náročnosti.

Procedure Evaporate

```
1 for i=0 to n-1 do
2     for j=i to n-1 do
3         pheromone[i][j] = (1 - ρ) * pheromone[i][j]
4         pheromone[j][i] = pheromone[i][j]
5     end
6 end
```

Procedure DepositPheromone(k)

```
   Input: k    % identifikátor mravence
1 Δτ = 1 / ant[k].tour_length
2 for i=0 to n-1 do
3     j = ant[k].tour[i]
4     l = ant[k].tour[i+1]
5     pheromone[j][l] = pheromone[j][l] + Δτ
6     pheromone[l][j] = pheromone[j][l]
7 end
```

Procedure ComputeChoiceInformation

```
1 for i=0 to n-1 do
2     for j=i to n-1 do
3         τ = pheromone[i][j]
4         η = 1 / dist[i][j]
5         choice_info[i][j] = τ^α * η^β
6         choice_info[j][i] = choice_info[i][j]
7     end
8 end
```

5.4.3 Implementace rozšíření algoritmu Ant System

Jak bylo uvedeno v kapitole 5.3, rozšíření algoritmu AS, konkrétně pak EAS a AS_{rank}, se od AS liší pouze způsobem, jakým jsou aktualizovány feromonové stopy. Při implementaci algoritmů EAS a AS_{rank} je tak pouze zapotřebí upravit proceduru `UpdatePheromoneTrails` tak, aby odpovídala rovnicím (6), resp. (8).

6 Experimentální porovnání algoritmů

Tato kapitola představuje výstup praktické části této práce. Výše popsané algoritmy optimalizace mravenčí kolonií – Ant System, Elitist Ant System a Rank-Based Ant System byly implementovány a experimentálně porovnány s vybranými algoritmy řešícími TSP.

6.1 Vybrané algoritmy

Pro experimentální porovnání byly vybrány následující algoritmy: *Nearest Neighbor*, *Double Tree* a *2-opt*. Tyto algoritmy nyní stručně představíme.

Nearest Neighbor

Nearest Neighbor je jedním z prvních algoritmů řešících TSP. Nalezení přípustného řešení dané instance TSP je při použití tohoto algoritmu velmi rychlé, obvykle však nalezené řešení není optimální. Jedná se o heuristický hladový algoritmus – v každém kroku je do konstruovaného řešení „hladově“ přidán vrchol, který se zrovna nachází nejblíže. Podrobněji lze algoritmus popsat takto [9]:

1. Označ všechny vrcholy jako nenavštívené.
2. Vyber libovolný vrchol, nastav jej jako aktuální vrchol u. Označ vrchol u jako navštívený.
3. Nalezni nejkratší hranu spojující vrchol u a nenavštívený vrchol v.
4. Nastav vrchol v jako aktuální vrchol u. Označ vrchol v jako navštívený.
5. Pokud existuje některý vrchol, který je označený jako nenavštívený, pokračuj krokem č.3. Jinak skonči.

Double Tree

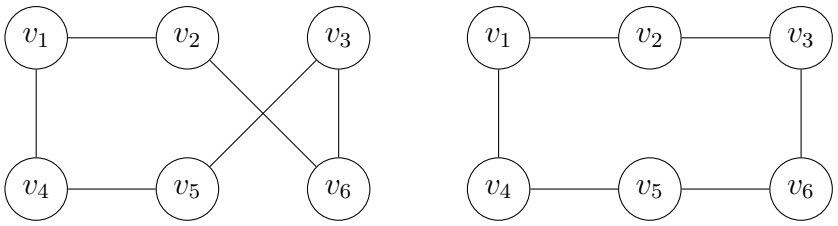
Double Tree se řadí mezi aproximační algoritmy řešící metrický TSP. Lze dokázat, že se jedná o 2-aproximační algoritmus. Je tedy garantováno, že cena řešení nalezeného tímto algoritmem je v nejhorším případě dvakrát vyšší, než cena řešení optimálního. Pro nalezení řešení využívá algoritmus Double Tree minimální kostru grafu. Kostrou grafu $G = \langle V, E \rangle$ rozumíme podgraf grafu G na množině všech jeho vrcholů, který je souvislý a neobsahuje žádnou kružnici. Minimální kostra grafu G s hranovým ohodnocením w je taková kostra grafu G, pro kterou platí, že součet ohodnocení hran této kostry je nejmenší ze všech možných koster. Pro implementaci v této práci byl pro nalezení minimální kostry grafu využit Kruskalův algoritmus. Algoritmus Double Tree lze podrobněji popsat takto:

1. Nalezni minimální kostru T grafu G.
2. Proveď prohledávání T do hloubky, začínající libovolným vrcholem v. Nechť $S = v_1, \ldots, v_n$ je pořadí, v jakém je přistupováno k vrcholům.
3. Opakuj, dokud se v S nachází některý vrchol u více než jednou: odstraň z S první vrchol u v pořadí.

2-opt

2-opt se řadí mezi heuristické algoritmy řešící metrický TSP, konkrétněji spadá do třídy algoritmů lokálního prohledávání. Algoritmus 2-opt si tedy po celou dobu běhu udržuje právě jedno přípustné řešení dané instance TSP, které iterativně vylepšuje lokálními modifikacemi.

Způsob, kterým 2-opt modifikuje řešení, lze zjednodušeně popsat takto. V aktuálním řešení jsou vyhledávány hrany, které se protínají. Řešení je následně modifikováno tak, aby se hrany neprotínaly. Uveďme jednoduchý příklad. Na obrázku 9 (a) vidíme řešení TSP na grafu o šesti vrcholech. Na první pohled je zřejmé, že hrany $\{v_2, v_6\}$ a $\{v_3, v_5\}$ se protínají. Algoritmus 2-opt tedy řešení upraví tak, že odebere hrany $\{v_2, v_6\}$ a $\{v_3, v_5\}$ a namísto nich do řešení přidá hrany $\{v_2, v_3\}$ a $\{v_5, v_6\}$, viz (b) na obrázku 9. Díky tomu, že metrický TSP splňuje trojúhelníkovou nerovnost, má takto modifikované řešení menší cenu, než řešení před modifikací.



(a) (b)

Obrázek 9: Modifikace řešení TSP pomocí 2-opt. (a) Řešení před modifikací. (b) Řešení po modifikaci

Počáteční přípustné řešení může být vypočítáno jiným heuristickým algoritmem. Pro implementaci v této práci byl vybrán algoritmus Nearest Neighbor.

6.2 Knihovna TSPLIB

TSPLIB[3] je knihovna instancí pro TSP a příbuzné problémy. Obsahuje desítky různých instancí, z nichž je u některých známo optimální řešení[4]. Instance jsou

[3] http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/index.html
[4] Viz dokumentace: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp95.pdf

uloženy v souborech s příponou .tsp. Každý soubor je rozdělen do dvou částí – specifikace instance a datové části.

Specifikace instance se nachází na začátku souboru a obsahuje údaje ve tvaru `klíč:hodnota`. Jedná se o informace o dané instanci TSP. Pro experimentální srovnání byly vybrány soubory, které mají ve své specifikaci hodnoty `TYPE: TSP` a `EDGE_WEIGHT_TYPE:EUC_2D`. `TYPE:TSP` znamená, že typ problému je TSP a `EDGE_WEIGHT_TYPE:EUC_2D` značí, že použitá vzdálenostní funkce je Euklidovská vzdálenost (v dvojrozměrném prostoru). K vzdálenostní funkci je potřeba dodat, že vzdálenost mezi dvěma body je vždy zaokrouhlena na nejbližší celé číslo. Je tomu tak z historických důvodů – pro uložení vzdáleností v paměti počítače se používal téměř výhradně datový typ `integer`.

Datová část, která obsahuje instanci TSP, následuje hned po specifikaci instance. Pro experimentální srovnání algoritmů byly vybrány soubory, jejichž datová část začíná řádkem `NODE COORD SECTION`. Vrcholy z instancí TSP, které se v těchto souborech nachází, jsou položeny do roviny a zadány pomocí souřadnic na ose x a ose y. Každý řádek za řádkem `NODE COORD SECTION` popisuje právě jeden vrchol ve tvaru `c sx sy`, kde `c` je číslo (název) vrcholu datového typu `integer`, `sx` je souřadnice vrcholu na ose x datového typu `real` a `sy` je souřadnice vrcholu na ose y datového typu `real`.

Soubor knihovny TSPLIB může být volitelně ukončen řádkem `EOF`. Ukázka instance TSP ve formátu zavedeném knihovnou TSPLIB se nachází na obrázku 10.

```
NAME: example
TYPE: TSP
COMMENT: 8 locations in Berlin
DIMENSION: 8
EDGE_WEIGHT_TYPE: EUC_2D
NODE_COORD_SECTION
1 565.0 575.0
2 25.0 185.0
3 345.0 750.0
4 945.0 685.0
5 845.0 655.0
6 880.0 660.0
7 25.0 230.0
8 525.0 1000.0
EOF
```

Obrázek 10: Ukázka instance TSP ve formátu TSPLIB

6.3 Podrobnosti k měření

Experimentální porovnání výše popsaných algoritmů proběhlo na přenosném počítači s CPU Intel Core i5-8250U a 8 GB RAM. Algoritmy byly implementovány

v programovacím jazyce C#. Implementace je blíže popsána v kapitole 7. Pro experimentální srovnání algoritmů byly z knihovny TSPLIB vybrány instance TSP o různém počtu vrcholů. Přehled vybraných instancí TSP se nachází v tabulce 3.

Tabulka 3: Testovací soubory z knihovny TSPLIB

Soubor	Počet vrcholů	Cena optimálního řešení
berlin52.tsp	52	7542
bier127.tsp	127	118282
pr264.tsp	264	49135
pr439.tsp	439	107217

Doba běhu jednotlivých algoritmů je vždy počítána až od chvíle, kdy jsou vytvořeny potřebné dodatečné struktury, matice vzdálenosti atd. a začíná konstrukce samotného řešení.

Parametry algoritmů optimalizace mravenčí kolonií byly zvoleny dle tabulky 2 (pro parametr β byla zvolena hodnota 2).

6.4 Experimentální výsledky

Porovnání vybraných algoritmů

V tabulce 4 se nachází porovnání algoritmů NN, DT a 2-opt na vybraných instancích TSP. Každý algoritmus byl spuštěn 20krát. Tabulka ukazuje průměrnou dobu běhu algoritmů (v milisekundách) a průměrnou odchylku ceny nalezených řešení od ceny optimálního řešení (v procentech).

Tabulka 4: Porovnání vybraných algoritmů

	NN		DT		2-opt	
	Čas	Odchylka	Čas	Odchylka	Čas	Odchylka
berlin52	0,15 ms	23,5 %	1,2 ms	38 %	39 ms	7 %
bier127	0,17 ms	23,7 %	18 ms	30 %	1330 ms	5 %
pr264	0,26 ms	18,4 %	183 ms	30 %	15613 ms	8,4 %
pr439	0,29 ms	28,4 %	840 ms	36,5 %	153306 ms	7 %

Nejkvalitnější řešení byla nalezena algoritmem 2-opt. Čas potřebný pro nalezení řešení nicméně u algoritmu 2-opt roste nejrychleji ze srovnávaných algoritmů. Algoritmus NN nedosahuje kvality řešení nalezených algoritmem 2-opt, čas potřebný pro nalezení řešení ale s rostoucím počtem vrcholů roste jen velmi pomalu. Nejméně kvalitní řešení byla nalezena algoritmem DT.

Porovnání algoritmů optimalize mravenčí kolonií

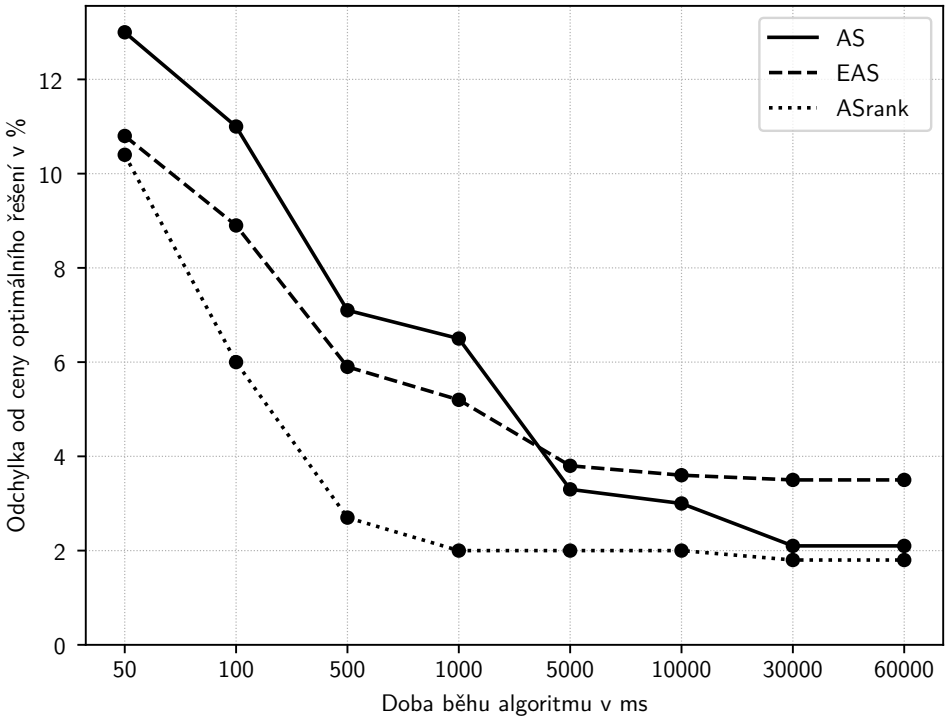
Nyní mezi sebou porovnáme algoritmy AS, EAS a AS_{rank} a výsledky dále porovnáme s výsledky algoritmu 2-opt, který v předchozím porovnání vybraných algoritmů dosahoval nejlepších vý sledků. V případě ACO algoritmů byly algoritmy spuštěny vždy 10krát a výsledky opět zprůměrovány. Protože všechny tři ACO algoritmy jsou implementačně velmi podobné, je u těchto algoritmů podobná i doba jedné iterace. Z tohoto důvodu dále nebude uváděna doba jedné iterace pro každý z algoritmů zvlášť.

Porovnání ACO algoritmů na instanci TSP berlin52 zachycuje graf na obrázku 11. Zde dosáhl nejlepších výsledků algoritmus AS_{rank}, který také nejrychleji zlepšoval kvalitu nalezeného řešení. Cena řešení nalezeného algoritmem AS se přiblížila ceně řešení nalezeného algoritmem AS_{rank}, nicméně až po delším časovém úseku 30000 ms. Algoritmus EAS nalezl kvalitnější řešení než AS jen v prvních sekundách. Již po 1000 ms nalezly všechny mravenčí algoritmy lepší řešení, než algoritmus 2-opt. Průměrná doba, za kterou ACO algoritmy vykonaly jednu iteraci, byla 7,8 ms.

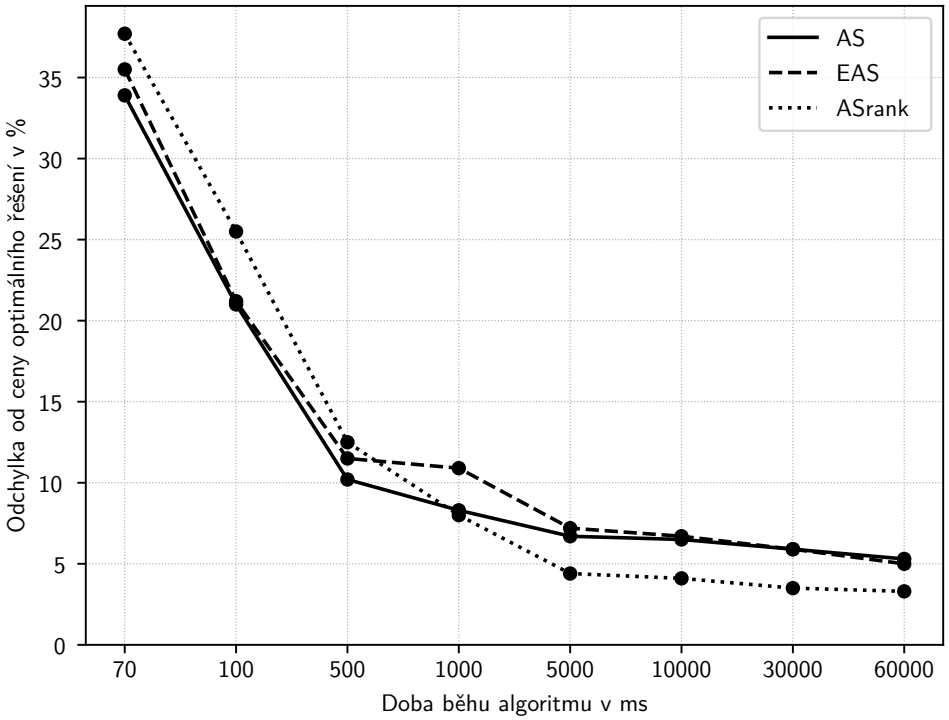
Na obrázku 12 se nachází graf porovnání ACO algoritmů na instanci TSP bier127. Po uplynutí 60000 ms opět nalezl nejkratší cestu algoritmus AS_{rank}. Algoritmy AS a EAS dosáhly téměř stejných výsledků, srovnatelných s algoritmem 2-opt. Avšak algoritmus 2-opt nalezl řešení rychleji, v průměru již po 1330 ms. Průměrná doba jedné iterace ACO algoritmů byla 37,2 ms.

Graf na obrázku 13 ukazuje porovnání ACO algoritmů na instanci TSP pr264. Výsledky jsou podobné, jako při testování na instanci bier127. Ačkoliv v počátečních iteracích vykazoval algoritmus AS_{rank} nejhorší výsledky, po uplynutí 100000 ms nalezl nejlepší řešení. Algoritmy AS a EAS nalezly řešení podobné ceny, srovnatelné s řešením nalezeným algoritmem 2-opt. Průměrná doba, za kterou ACO algoritmy vykonaly jednu iteraci, byla 157 ms.

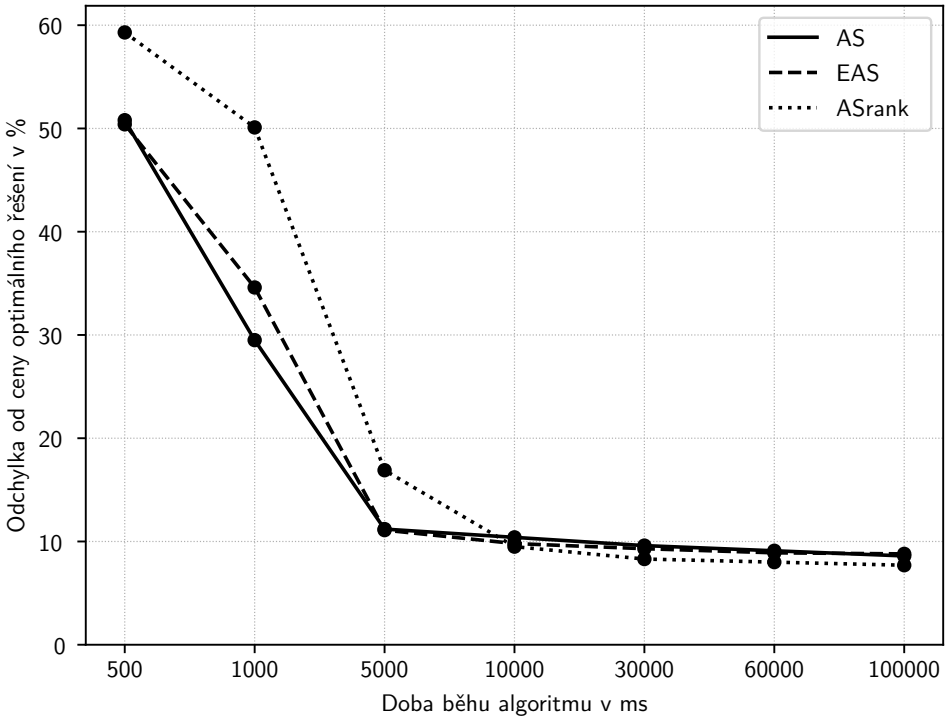
Porovnání ACO algoritmů na poslední vybrané instanci TSP pr439 zachycuje graf na obrázku 14. Algoritmus AS na této instanci po uplynutí 150000 ms nalezl řešení, jehož procentuální odchylka od ceny optimálního řešení byla dvakrát větší, než v případě algoritmu 2-opt. Doba potřebná k nalezení řešení algoritmem 2-opt přitom byla taktéž 150000 ms. Algoritmus EAS dosahoval podobných výsledků jako algoritmus AS. Algoritmus AS_{rank} opět vykazoval v prvních iteracích nejhorší výsledky, avšak po uplynutí 150000 ms nalezl zdaleka nejlepší řešení – procentuální odchylka od ceny optimálního řešení byla průměrně 6,3 %. Průměrná doba jedné iterace ACO algoritmů byla 443,5 ms.



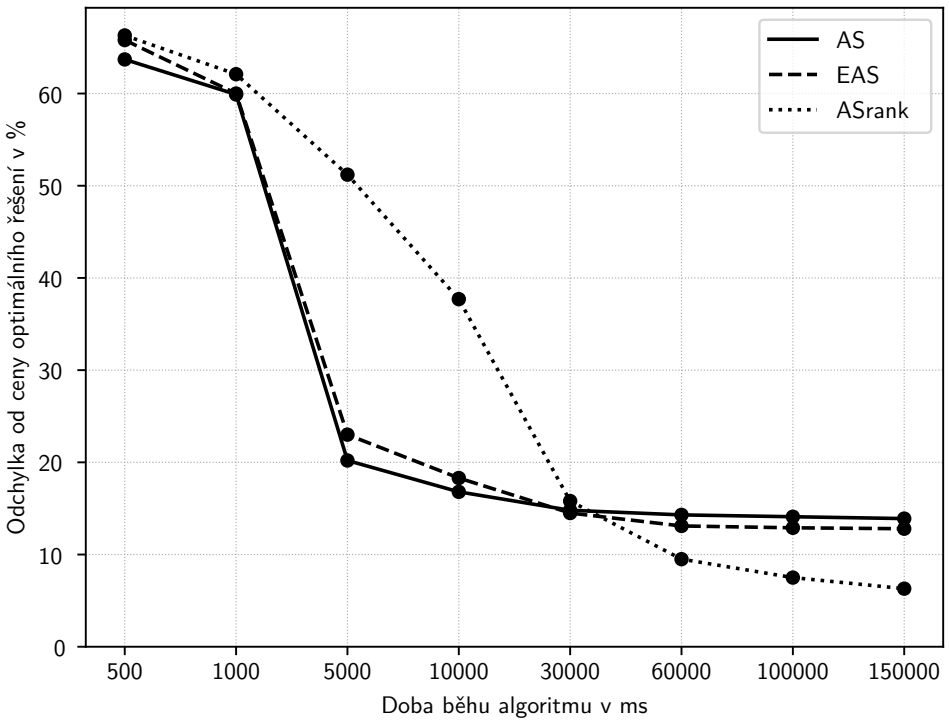
Obrázek 11: Porovnání algoritmů optimalizace mravenčí kolonií na instanci TSP berlin52



Obrázek 12: Porovnání algoritmů optimalizace mravenčí kolonií na instanci TSP bier127



Obrázek 13: Porovnání algoritmů optimalizace mravenčí kolonií na instanci TSP pr264



Obrázek 14: Porovnání algoritmů optimalizace mravenčí kolonií na instanci TSP pr439

7 Programátorská dokumentace

Implementace algoritmů vznikla v integrovaném vývojovém prostředí Visual Studio 2019 v programovacím jazyce C#. Jedná se o sadu tříd, jejichž hlavní funkcionalita představuje vytvoření grafů, na kterých je možné spustit algoritmy řešící TSP. Pro flexibilitu potřebnou k experimentálnímu porovnávání algoritmů byla zvolena forma konzolového projektu. Z důvodu velkého množství tříd zde bude popsána pouze základní funkcionalita a příklady použití. Podrobná dokumentace všech veřejně viditelných tříd, vlastností tříd a metod je přiložena ke zdrojovým kódům na CD v souboru *src/TSP/_site/index.html*. Po spuštění této statické stránky je potřeba vybrat v horním menu položku *API Documentation*.

7.1 Popis tříd

Vertex reprezentuje vrchol grafu.

Edge reprezentuje hranu grafu.

Point reprezentuje bod v rovině o souřadnicích x a y. Vzdálenost od daného bodu k bodu `point` lze zjistit pomocí metody `DistanceTo(Point point)`.

Graph reprezentuje neorientovaný graf. Je možné jej inicializovat seznamem vrcholů třídy `Vertex` a seznamem hran třídy `Edge`, dále seznamem bodů třídy `Point`, zadáním cesty k souboru .tsp, nebo voláním metod třídy `GraphGenerator`. Pomocí metody `IsSimple()` lze ověřit, zda je daný graf obyčejný. Pomocí metody `IsComplete()` lze ověřit, zda je daný graf úplný. Žádné dva vrcholy v grafu nesmí mít stejný název.

GraphGenerator je statická třída, poskytující metody k vygenerování náhodného úplného grafu. Voláním metody `Generate()` je vytvořen graf, který má nejméně 10 vrcholů a nejvíce 100 vrcholů. Počet vrcholů generovaného grafu lze zvolit voláním metody `Generate(int vertexCount)` (nejvíce 1000 vrcholů).

Walk reprezentuje sled. Ve vlastnosti `VerticesSequence` jsou uloženy vrcholy sledu v pořadí, ve kterém byly navštíveny. Délku sledu lze zjistit voláním metody `Cost()`. Pomocí metody `IsHamiltonianCycle()` lze ověřit, zda je sled hamiltonovskou kružnicí.

ParametersAS reprezentuje parametry pro algoritmus Ant System.

ParametersEAS reprezentuje parametry pro algoritmus Elitist Ant System.

ParametersASRank reprezentuje parametry pro algoritmus Rank-Based Ant System.

Solvers.NNSolver je třída, která řeší TSP algoritmem Nearest Neighbor. V konstruktoru je potřeba předat graf třídy `Graph`, na kterém bude řešen TSP. Algoritmus je spuštěn voláním metody `SolveTSP()`. Vlastnost `Solution` uchovává řešení reprezentované objektem třídy `Walk`. Dobu běhu algoritmu (v milisekundách) uchovává vlastnost `Runtime`.

Solvers.DTSolver je třída, která řeší TSP algoritmem Double Tree. V konstruktoru je potřeba předat graf třídy `Graph`, na kterém bude řešen TSP. Algoritmus je spuštěn voláním metody `SolveTSP()`. Vlastnost `Solution` uchovává řešení reprezentované objektem třídy `Walk`. Dobu běhu algoritmu (v milisekundách) uchovává vlastnost `Runtime`.

Solvers.NN2OPTSolver je třída, která řeší TSP algoritmem 2-opt. V konstruktoru je potřeba předat graf třídy `Graph`, na kterém bude řešen TSP. Algoritmus je spuštěn voláním metody `SolveTSP()`. Vlastnost `Solution` uchovává řešení reprezentované objektem třídy `Walk`. Dobu běhu algoritmu (v milisekundách) uchovává vlastnost `Runtime`.

Solvers.ASSolver je třída, která řeší TSP algoritmem Ant System. V konstruktoru je potřeba předat graf třídy `Graph`, na kterém bude řešen TSP. Volitelně je možné v konstruktoru předat také objekt třídy `ParametersAS`. Pokud v konstruktoru není předán objekty třídy `ParametersAS`, jsou použity výchozí parametry. Algoritmus je spuštěn voláním metody `SolveTSP()`. Vlastnost `Solution` uchovává řešení reprezentované objektem třídy `Walk`. Dobu běhu algoritmu (v milisekundách) uchovává vlastnost `Runtime`. Ve vlastnosti `BestSolutionIteration` je uloženo, v kolikáté iteraci algoritmu bylo nalezeno nejlepší řešení. Pomocí vlastnosti `MaxRuntime` lze určit maximální dobu běhu algoritmu v (milisekundách). Pomocí vlastnosti `MaxIterationCount` lze určit maximální počet iterací algoritmu. Vlastnost `Parameters` uchovává parametry algoritmu, reprezentované objektem třídy `ParametersAS`.

Solvers.EASSolver je třída, která řeší TSP algoritmem Elitist Ant System. V konstruktoru je potřeba předat graf třídy `Graph`, na kterém bude řešen TSP. Volitelně je možné v konstruktoru předat také objekt třídy `ParametersEAS`. Pokud konstruktoru není předán objekty třídy `ParametersEAS`, jsou použity výchozí parametry. Algoritmus je spuštěn voláním metody `SolveTSP()`. Vlastnost `Solution` uchovává řešení reprezentované objektem třídy `Walk`. Dobu běhu algoritmu (v milisekundách) uchovává vlastnost `Runtime`. Ve vlastnosti `BestSolutionIteration` je uloženo, v kolikáté iteraci algoritmu bylo nalezeno nejlepší řešení. Pomocí vlastnosti `MaxRuntime` lze určit maximální dobu běhu algoritmu v (milisekundách). Pomocí vlastnosti `MaxIterationCount` lze určit maximální počet iterací algoritmu. Vlastnost `Parameters` uchovává parametry algoritmu, reprezentované objektem třídy `ParametersEAS`.

Solvers.ASRankSolver je třída, která řeší TSP algoritmem Rank-Based Ant System. V konstruktoru je potřeba předat graf třídy `Graph`, na kterém bude řešen TSP. Volitelně lze v konstruktoru předat objekt třídy `ParametersASRank`. Pokud v konstruktoru není předán objekty třídy `ParametersASRank`, jsou použity výchozí parametry. Algoritmus je spuštěn voláním metody `SolveTSP()`. Vlastnost `Solution` uchovává řešení reprezentované objektem třídy `Walk`. Dobu běhu algoritmu (v milisekundách) uchovává vlastnost `Runtime`. Ve vlastnosti `BestSolutionIteration` je uloženo, v kolikáté iteraci algoritmu bylo nalezeno nejlepší řešení. Pomocí vlastnosti `MaxRuntime` lze určit maximální dobu běhu

algoritmu v (milisekundách). Pomocí vlastnosti `MaxIterationCount` lze určit maximální počet iterací algoritmu. Vlastnost `Parameters` uchovává parametry algoritmu, reprezentované objektem třídy `ParametersASRank`.

Všechny třídy řešící TSP přepokládají, že je jim předán úplný graf. Zda je graf úplný, lze ověřit metodou `IsComplete()`.

7.2 Příklady použití

Následují jednoduché ukázky použití. Ve zdrojovém kódu 1 je ukázka vytvoření grafů. Zdrojový kód 2 ukazuje spuštění algoritmu Nearest Neighbor. Ve zdrojovém kódu 3 je ukázka spuštění algoritmu Ant System. Tyto ukázky se nachází také ve zdrojovém kódu algoritmů uvnitř metody `main()` třídy `Program`.

```
1  // vytvoreni grafu seznamem vrcholu a seznamem hran
2  Vertex v0 = new Vertex("v0");
3  Vertex v1 = new Vertex("v1");
4  Vertex v2 = new Vertex("v2");
5  Edge e0 = new Edge(3, v0, v1);
6  Edge e1 = new Edge(4, v1, v2);
7  Edge e2 = new Edge(5, v2, v0);
8  List<Vertex> vertices = new List<Vertex>() { v0, v1, v2 };
9  List<Edge> edges = new List<Edge>() { e0, e1, e2 };
10 Graph graph1 = new Graph(vertices, edges);
11
12 // vytvoreni grafu seznamem bodu
13 Point c0 = new Point("c0", 1, 1);
14 Point c1 = new Point("c1", 4, 1);
15 Point c2 = new Point("c2", 4, 5);
16 Point c3 = new Point("c3", 1, 5);
17 Graph graph2 = new Graph(new List<Point>() { c0, c1, c2, c3 });
18
19 // vytvoreni nahodneho grafu pomoci tridy GraphGenerator
20 Graph graph3 = GraphGenerator.Generate(25);
21
22 // vytvoreni grafu zadanim cesty k souboru .tsp
23 Graph graph4 = new Graph("berlin52.tsp");
```

Zdrojový kód 1: Ukázka vytvoření grafů

```
Graph graph = new Graph("berlin52.tsp");
bool isComplete = graph.IsComplete();

NNSolver solver = new NNSolver(graph);
solver.SolveTSP();
bool isHamCycle = solver.Solution.IsHamiltonianCycle(graph);
double solutionCost = solver.Solution.Cost;
double solverRuntime = solver.Runtime;
```

Zdrojový kód 2: Ukázka spuštění algoritmu Nearest Neighbor

```
Graph graph = new Graph("berlin52.tsp");
bool isComplete = graph.IsComplete();

ParametersAS parameters = new ParametersAS(1, 5, 0.5,
  graph.Vertices.Count);
ASSolver solver = new ASSolver(graph, parameters);
solver.MaxIterationsCount = 100;
solver.MaxRuntime = 1000;
solver.SolveTSP();
bool isHamCycle = solver.Solution.IsHamiltonianCycle(graph);
double solutionCost = solver.Solution.Cost;
double solverRuntime = solver.Runtime;
```

Zdrojový kód 3: Ukázka spuštění algoritmu Ant System

Závěr

Výsledkem této práce je popis techniky optimalizace mravenčí kolonií na problému obchodního cestujícího, implementace algoritmů v programovacím jazyce C# a experimentální porovnání této techniky a dalších vybraných algoritmů.

Experimentální porovnání algoritmů prokázalo, že technika optimalizace mravenčí kolonií není vhodná, pokud je nutné nalézt řešení v co nejkratším čase. V takovém případě je vhodnější využít například algoritmus NN. Pokud je ale na výpočet řešení dostatek času, ACO algoritmy dokáží nalézt výrazně kvalitnější řešení, než všechny ostatní srovnávané algoritmy, včetně algoritmu 2-opt. Tato skutečnost se projevila zejména při pokusech na instancích TSP s menším počtem vrcholů, kde byl algoritmům poskytnut dostatečný čas. Porovnání algoritmu AS s jeho rozšířeními – algoritmy EAS a AS$rank$ ukázalo, že algoritmus EAS dosahuje o trochu lepších výsledků než algoritmus AS. I přes horší výsledky v prvních iteracích se ukázalo, že nejlepších výsledků dosahuje algoritmus AS$rank$.

Možné rozšíření této práce by mohlo spočívat například v implementaci dalších ACO algoritmů nebo rozšíření stávajících algoritmů o lokální prohledávání.

Conclusions

The result of this work is a description of the ant colony optimization technique for the travelling salesman problem, the implementation of algorithms in the programming language C# and an experimental comparison of this technique and other selected algorithms.

Experimental comparison of algorithms has shown that the technique of ant colony optimization is not suitable if it is necessary to find a solution in the shortest possible time. In this case, it is more appropriate to use, for example, the NN algorithm. However, if there is enough time to find the solution, ACO algorithms can find a significantly better solution than all other compared algorithms, including the 2-opt algorithm. This fact manifested itself especially in experiments on TSP instances with a smaller number of vertices, where the algorithms were given sufficient time. A comparison of the AS algorithm with its extensions - EAS and AS*rank* algorithms showed that the EAS algorithm achieves slightly better results than the AS algorithm. Despite the worse results in the first iterations, the AS*rank* algorithm showed the best results.

A possible extension of this work could consist, for example, in the implementation of other ACO algorithms or the extension of current algorithms by local search.

A Obsah přiloženého CD

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

`src/`
: Kompletní zdrojové texty implementace techniky optimalizace mravenčí kolonií na problému obchodního cestujícího a dalších vybraných algoritmů.

`readme.txt`
: Instrukce pro spuštění algoritmů, včetně všech požadavků pro jejich bezproblémový provoz.

`data/`
: Ukázková a testovací data použitá v práci při porovnávání algoritmů.

Literatura

[1] BĚLOHLÁVEK, Radim, VYCHODIL, Vilém. *Diskrétní matematika pro informatiky II.* Univerzita Palackého v Olomouci, 2006. Dostupné z: http://belohlavek.inf.upol.cz/vyuka/dm2.pdf

[2] VEČERKA, Arnošt. *Grafy a grafové algoritmy.* Univerzita Palackého v Olomouci, 2007. Dostupné z: https://phoenix.inf.upol.cz/esf/ucebni/Grafy_a_grafove_algoritmy.pdf

[3] OSIČKA, Petr. Učební text k předmětu Algoritmy pro těžké problémy. Univerzita Palackého v Olomouci.

[4] Wikipedia contributors. Wikipedia, The Free Encyclopedia: *Travelling salesman problem* [online]. [cit. 2021-03-18] Dostupné z: https://en.wikipedia.org/wiki/Travelling_salesman_problem

[5] TALBI, El-Ghazali. *Metaheuristics: from design to implementation.* Hoboken: Wiley, c2009. ISBN 978-0-470-27858-1.

[6] DORIGO, Marco, BIRATTARI, Mauro. Scholarpedia: *Swarm intelligence* [online]. [cit. 2021-03-18] Dostupné z: http://www.scholarpedia.org/article/Swarm_intelligence

[7] DORIGO, Marco, STÜTZLE Thomas. *Ant colony optimization.* Cambridge: MIT Press, 2004. ISBN 0-262-04219-3.

[8] Wikipedia contributors. Wikipedia, The Free Encyclopedia: *Ant* [online]. [cit. 2021-03-18] Dostupné z: https://en.wikipedia.org/w/index.php?title=Ant&oldid=1028441447

[9] Wikipedia contributors. Wikipedia, The Free Encyclopedia: *Nearest neighbour algorithm* [online]. [cit. 2021-06-23] Dostupné z: https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm

[10] Wikipedia contributors. Wikipedia, The Free Encyclopedia: *2-opt* [online]. [cit. 2021-06-23] Dostupné z: https://en.wikipedia.org/wiki/2-opt